

**ÉCOLE DOCTORALE SCIENCES ET MÉTIERS DE L'INGÉNIEUR**  
**[Laboratoire de recherche - Campus d'Angers]**

**THÈSE**

*présentée par :* **Baptiste DAVID**  
*soutenue le :* **07 décembre 2021**

*pour obtenir le grade de :* **Docteur d'HESAM Université**

*préparée à :* **École Nationale Supérieure d'Arts et Métiers**  
*Spécialité :* **Informatique**

**Nouvelles approches de la sécurité informatique reposant sur  
la vision offensive et bas-niveau des systèmes.**

**New trends in offensive, low level-based computer security.**

**THÈSE dirigée par :**

**[Monsieur FILIOL Eric]**

**Jury**

<b>Mme Mirna DZAMONJA</b>	Associate Professor, Institut de Recherche en Informatique Fondamentale (CNRS-Université de Paris)	Rapporteur
<b>Mme Antonella SANTONE</b>	Associate Professor, Department of Engineering (SSD ING-INF/05), University of Sannio	Rapporteur
<b>M. Johann BARBIER</b>	Directeur d'études, Alten	Examineur
<b>Mme Samia BOUZEFRANE</b>	Professeure des universités, Laboratoire CEDRIC, Conservatoire National des Arts et Métiers	Examinatrice
<b>M. Maroun CHAMOUN</b>	Full professor, Université Saint Joseph	Examineur
<b>M. Éric FILIOL</b>	Full professor, ENSIBS, Vannes & National Research University High School of Economics, Moscou Lomonossov	Examineur
<b>M. Bimal ROY</b>	Full professor, Indian Statistical Institute	Examineur
<b>M. Igor ZDOBNOV</b>	Engineer, Doctor Web, Ltd.	Invité



*À ma famille, à vous qui m'avez tant donné et que j'aime tant.  
Ces quelques mots dérisoires pour vous exprimer toute ma gratitude.  
Plus que du bonheur, c'est un privilège que de vous avoir à mes côtés.*

*"Et puis, il y a ceux que l'on croise, que l'on connaît à peine, qui vous disent un mot, une phrase, vous accordent une minute, une demi-heure et changent le cours de votre vie." — Victor Hugo*

---

THIS PAGE INTENTIONALLY LEFT BLANK

---



# Remerciements

Il n'en est à l'aventure aucune plus expresse que de vivre des rencontres dans sa vie. Car de ces rencontres se construisent des idées, des projets, des amitiés et qui parfois, changent une vie. Ce sont ces rencontres qui m'ont permis de réaliser ce travail de recherche qui n'aurait pas été possible sans leur plein concours et soutien. Qu'ils en soient ici remerciés pour celles et ceux qu'il m'est donné de citer.

En tout premier lieu, je tiens à remercier mon directeur de thèse, le professeur Eric Filiol, pour la confiance et l'honneur qu'il m'accorda en acceptant d'encadrer ce travail doctoral. Pour votre temps, pour votre disponibilité, pour votre liberté d'esprit, pour votre dynamisme, pour votre rigueur, pour votre bienveillance malgré tous mes défauts (et ils sont inénarrables) et plus simplement pour tout ce que vous avez su faire naître en moi. Il y a eu des moments uniques de bonheur, de satisfaction, de doute, de rire et de joie. Des moments simples et des moments exceptionnels qui font que la vie d'un homme bascule à jamais. Il y a eu aussi ces moments si difficiles qu'ils vous brisent le cœur. C'est dans ces moments que vous avez su montrer ce que veulent dire les mots de dignité, de grandeur et d'humanisme. Je n'aurais pas voulu un autre directeur de thèse que vous. Vous m'êtes à jamais une source d'inspiration insatiable.

Je remercie également Igor Zdobnov et plus généralement l'entreprise Dr Web pour m'avoir accueilli en séjour doctoral à Saint-Petersbourg. Outre l'activité technique et scientifique qui fut particulièrement dense, ce séjour doit beaucoup à votre accueil chaleureux. Il est rare de rencontrer des personnes aussi talentueuses dans tant de domaines. Nos discussions à bâton rompu et cet humour si particulier bercent mes souvenirs d'une certaine nostalgie. Merci pour ces moments.

Il m'est impossible d'oublier Afianian Amir avec qui nous avons si souvent échangé pour l'écriture de notre article. Merci d'avoir partagé avec moi cette aventure bien que nous ne nous soyons jamais rencontrés.

Je souhaite aussi exprimer toute ma gratitude à l'ensemble de mes collègues qui ont su m'entourer avec bienveillance. À Maxence Delong pour tous ces moments partagés. Que ce soit dans nos travaux communs de recherche ou dans la lutte à dépasser les fatalités inhérentes à la médiocrité... Notre amitié c'est forgée dans le pire, mais surtout pour le meilleur.

Plus directement, je veux remercier Pierre-François Maillard, François Plumerault et Mathilde Venault avec lesquels j'ai eu l'insigne honneur de partager certains de mes travaux de recherche. Pour le temps passé ensemble, votre soutien, pour l'ensemble de votre travail et les relectures nocturnes de Mathilde. Savoir que j'ai pu contribuer à vous aider à devenir ce que vous êtes aujourd'hui est un honneur sans pareil que vous m'avez fait. Vous êtes — et de loin — ma plus grande fierté au sein de ce doctorat.

Aussi, ces travaux n'auraient pu exister sans le soutien affectif et moral de tant de mes amis. Qu'il me soit ici possible de citer Alain, Amir, Aurélien, Bruno, Cédric, Clarisse, Elizabeth, Guillaume, Jean-Michel, Kévin, Laurence, Laurent, Matthieu, Michel, Morgane, Nicole, Patrice, Paul, Philippe, Pierre, Stéphanie, So' et Susan...

Il me faut enfin citer l'ESIEA dans le cadre de laquelle cette thèse fut réalisée, malgré tout. En ce sens, il convient de saluer ici Monsieur Da Rugna pour avoir apporté, *in fine*, le quantum de sérénité nécessaire pour l'achèvement de ces travaux.

Plus généralement, ce travail est dédié à tous ces enseignants qui m'ont appris que seul le savoir pouvait triompher contre l'obscurantisme. J'espère humblement que ce travail saura rendre hommage à ce qu'ils ont pu m'apporter.

THIS PAGE INTENTIONALLY LEFT BLANK

---

# Contents

<b>Remerciements</b>	<b>5</b>
<b>List of Tables</b>	<b>12</b>
<b>List of Figures</b>	<b>19</b>
<b>Résumé</b>	<b>21</b>
<b>Abstract</b>	<b>23</b>
<b>1 Introduction</b>	<b>25</b>
1 General presentation	25
2 Context of this research work	25
3 Problem and Significance	27
4 Roadmap	29
4.1 Structure of the thesis	29
4.2 Illustrated representation of the thesis	31
4.3 Reading improvements	33
4.4 Publication proceedings	34
<b>2 Protection at development level: backdoor in compilers</b>	<b>37</b>
1 Introduction	37
2 State of the art	39
2.1 Different instances of backdoor	39
2.2 Definition of a software backdoor	40
2.3 Backdoor at development time	42
3 First approach	51
4 Macro assembly and ml mistake with Boolean negation operator	54
4.1 Context	54
4.2 MASM assembler	54
4.3 Operators from MASM compiler	56
4.4 Use of NOT operator with MASM compiler	58
4.5 Bug in MASM compiler	58
4.6 Explanation of the bug in MASM compiler	60
5 How to build a sneaky backdoor with ml compiler bug?	62
5.1 Context of the backdoor	62
5.2 Description of bug consequences in order to insert a backdoor	63
5.3 Creation of the backdoor	63
6 Correction about the bug in MASM	67
6.1 Reporting of the bug	67
6.2 Potential corrections of the bug	69
6.3 Effective solution deployed by Microsoft	70
7 Conclusion	72
7.1 Impacts and achievements	72

7.2	Postmortem documentation . . . . .	73
7.3	Future of this work . . . . .	74
7.4	Research contributions . . . . .	75
<b>3</b>	<b>Protection of analyzed executable files: malware</b>	<b>77</b>
1	Introduction . . . . .	77
2	State of the art . . . . .	79
2.1	Preliminaries . . . . .	79
2.2	Manual Dynamic Analysis Evasion . . . . .	81
2.3	Automated Dynamic Analysis Evasion . . . . .	89
2.4	A Brief Survey on Countering Malware Evasion . . . . .	95
2.5	Conclusion about the state-of-the-art . . . . .	96
3	New manual dynamic analysis evasion technique on debuggers . . . . .	97
3.1	INT 3 mishandling exploitation . . . . .	98
3.2	Wrong jump interpretation . . . . .	104
3.3	Partial instruction prefix handling . . . . .	106
3.4	Unsupported instruction . . . . .	108
3.5	Conclusion about exploiting of Windbg flaws . . . . .	109
4	New universal dynamic analysis evasion technique . . . . .	111
4.1	Introduction . . . . .	111
4.2	Preliminaries . . . . .	111
4.3	Method of detection . . . . .	112
4.4	Detection of analysis environment . . . . .	119
4.5	Improvement of the test campaign and reproducibility of results . . . . .	128
4.6	Limitations and further improvements . . . . .	137
5	Future work and broader approach . . . . .	143
6	Conclusion . . . . .	148
6.1	Reminder of the achievements . . . . .	148
6.2	Research contributions . . . . .	149
<b>4</b>	<b>State of the art about Windows keyboard management</b>	<b>151</b>
1	General introduction . . . . .	151
1.1	About keylogger threat and the organization of the following chapters . . . . .	151
1.2	Introduction about keyboard technology . . . . .	154
2	Keystroke from hardware keyboard . . . . .	155
3	PS/2 technology . . . . .	160
3.1	Presentation of PS/2 technology . . . . .	160
3.2	Kernel interface with a device and scan code sets . . . . .	161
3.3	Handling PS/2 by Windows . . . . .	164
4	USB and HID technology . . . . .	168
4.1	USB protocol . . . . .	169
4.2	HID protocol . . . . .	189
4.3	Research Contributions . . . . .	231
5	Kbdclass and Windows subsystem . . . . .	232
5.1	Transition from kernel to user mode architecture . . . . .	232
5.2	Broadcast of keystrokes by the system with Window Messages . . . . .	275
5.3	Other means to access keyboard . . . . .	312
5.4	Miscellaneous about keyboard . . . . .	332
5.5	Research contributions . . . . .	341
<b>5</b>	<b>Keyloggers and existing anti-keylogger solutions</b>	<b>343</b>
1	Keyloggers history . . . . .	343
2	Hardware keyloggers . . . . .	344
2.1	Direct access hardware keylogger devices . . . . .	344
2.2	Indirect access hardware keylogger devices . . . . .	349
2.3	Protection against hardware keylogger . . . . .	350

3	Software keyloggers . . . . .	352
3.1	Firmware keylogger . . . . .	352
3.2	Kernel-mode keylogger . . . . .	354
3.3	User-mode keylogger . . . . .	364
4	Anti-keylogger solutions . . . . .	369
4.1	Passive solutions . . . . .	370
4.2	Active solutions . . . . .	372
4.3	Industrial solutions . . . . .	393
4.4	Conclusion about anti-keylogger solutions . . . . .	414
5	Research contributions . . . . .	417
<b>6</b>	<b>Gostxboard solution</b> . . . . .	<b>419</b>
1	Problem and definition of the need . . . . .	419
1.1	Objectives sought . . . . .	419
1.2	About to secure administrator applications requirement . . . . .	420
1.3	General considerations about our solution . . . . .	421
2	General architecture of the solution . . . . .	422
3	Genesis of the project . . . . .	424
4	Detailed architecture of GostBoard solution . . . . .	425
4.1	GostBoard WDM driver . . . . .	425
4.2	Ciphering scan codes for keystrokes . . . . .	442
4.3	Ciphering keys and exchange procedure . . . . .	452
4.4	Protecting the protected application and its cipher keys . . . . .	458
4.5	GostBoard Dll . . . . .	467
4.6	Self-defense mechanisms . . . . .	475
5	Going further current limitations . . . . .	491
5.1	Improving cipher key protection against crash-dump . . . . .	491
5.2	Multi-processes management . . . . .	492
5.3	Crash of the protected process . . . . .	493
5.4	Protection at deeper level in the device stack . . . . .	493
5.5	Limitation with low level keyboard hook procedure . . . . .	494
5.6	Original protection based on HID source driver . . . . .	497
6	Conclusion . . . . .	502
6.1	General conclusion about GostxBoard solution and the research work produced to secure keyboard . . . . .	502
6.2	Limits and future work . . . . .	507
6.3	Research contributions . . . . .	509
<b>7</b>	<b>Miscellaneous projects</b> . . . . .	<b>511</b>
1	Introduction . . . . .	511
2	Doctoral stay achievements . . . . .	511
2.1	Published projects . . . . .	512
2.2	Unpublished projects during the doctoral stay . . . . .	518
3	Third party productions . . . . .	527
3.1	Superfetch documentation . . . . .	527
3.2	UEFI ciphering system . . . . .	534
3.3	Detection of Crawler Traps based on new metric distances for data-mining . . . . .	539
4	Research contributions . . . . .	548
<b>8</b>	<b>Conclusion &amp; Future research work</b> . . . . .	<b>549</b>
1	Conclusion to the methodology used in this study . . . . .	549
2	Contribution and significance carried out by our study . . . . .	550
2.1	Improving security at the software compilation level . . . . .	550
2.2	Piece of news about backdoors nowadays . . . . .	551
2.3	Improving automatic malware analysis by preventing evasion . . . . .	552
2.4	Improving anti-keylogger security . . . . .	553

---

2.5	General conclusion about the methodology presented in this research work	555
Bibliography		617

# List of Tables

2.1	Results of compilation of an undefined code with two different compilers. . . . .	52
2.2	Different use of <b>NOT</b> operator with MASM compiler. . . . .	57
2.3	View of compiled code when using <b>NOT</b> operator with MASM compiler. . . . .	58
2.4	Trust table of the final condition in code 2.6. . . . .	65
2.5	Trust table of the trapped condition from source code point of view. . . . .	66
2.6	Trust table of the compiled trapped condition with <b>ml</b> compiler. . . . .	66
2.7	Initial code to correct. . . . .	70
3.1	Classification and comparison of malware anti-debugging techniques. . . . .	88
3.2	Classification and comparison of malware sandbox evasion techniques. . . . .	95
3.3	Classification and comparison of countermeasure tactics against evasive malwares. . . . .	96
3.4	Results of DBI framework detection test. . . . .	123
3.5	Results of debuggers detection test. . . . .	126
3.6	Results of hypervisor detection test. . . . .	127
3.7	Test with two host environments, <i>unstressed</i> . . . . .	134
3.8	Test with two host environments, in <i>stressed</i> conditions. . . . .	135
3.9	Test with two guest environments, <i>unstressed</i> . . . . .	135
3.10	Test with two guest environments, in <i>stressed</i> conditions. . . . .	136
3.11	Test in the guest environments, fully <i>stressed</i> with only 100 tests per instance of CPUID instruction. . . . .	136
3.12	Result from test with our measures of dispersion based on the median for host machines. . . . .	142
3.13	Result from test with our measures of dispersion based on the median for guest machines. . . . .	142
4.1	List of different scan codes from all different scan code sets — IBM PS/2 Model 50 and 60 Technical Reference. . . . .	163
4.2	Speeds of USB devices among different norms. . . . .	169
4.3	PID Types extracted from USB documentation. . . . .	173
4.4	Standard Device Descriptor. . . . .	179
4.5	USB Configuration Descriptor. . . . .	179
4.6	USB Interface Descriptor. . . . .	180
4.7	USB Endpoint Descriptor. . . . .	181
4.8	Format of Setup Data. . . . .	182
4.9	Standard Device Requests codes. . . . .	183
4.10	Descriptor Types codes. . . . .	183
4.11	HID Descriptor. . . . .	195
4.12	Interface 0 HID Report Descriptor Keyboard. . . . .	198
4.13	Structure used as Input and Output from HID report given in table 4.12. . . . .	198
4.14	HID class-specific requests. . . . .	199
4.15	List of functions used to update the foreground threads. . . . .	277
4.16	Numeric pad codes translations. . . . .	304
4.17	Value used by MapVirtualKeyEx function to perform translation. . . . .	305
4.18	List of hooks types with their scope associated (from [1, 2]). . . . .	322
5.1	Evaluation of SpyShelter software as an anti-keylogger solution. . . . .	397
5.2	Evaluation of KeyScrambler software as an anti-keylogger solution. . . . .	402

5.3	Evaluation of Zemana Keystrokes Encryption SDK as an anti-keylogger solution. . . . .	407
5.4	Evaluation of LMT Anti Logger solution. . . . .	408
5.5	Evaluation of GuardedID as an anti-keylogger solution. . . . .	413
5.6	General resume of the different industrial solutions. . . . .	413
6.1	Summary of the drawbacks of the different proposed solutions. . . . .	451
6.2	General resume of the GostxBoard solution compared with similar requirements from Table 5.6. . . . .	506
7.1	Function initials and their meaning in SysMain (Superfetch). . . . .	529



# List of Figures

1.1	Representation of the thesis architecture (1/2).	31
1.2	Representation of the thesis architecture (2/2).	32
2.1	Detection of the backdoor by the Linux kernel team.	43
2.2	Modification of compiler by Ken Thompson.	45
2.3	Insertion of a bug each time a pattern matches in the source code.	46
2.4	Csmith's finding bugs procedure.	48
2.5	Resume of Bauer's strategy to patch sudo program in order to insert the backdoor thank to LLVM compiler.	49
2.6	Decompilation of the code generated from Visual Studio 2018.	52
2.7	Decompilation of the code generated from GCC.	53
2.8	Equivalent code in C and in assembly x86 language.	55
2.9	Disassembly of code in figure 2.2.	55
2.10	List of symbols used in MASM officially supported.	56
2.11	Evaluation of complex statements with MASM operators.	57
2.12	Documentation about the precedence of the <b>NOT</b> operator.	57
2.13	Disassembly of code compiled in figure 2.3.	59
2.14	Decompiled code from the one compiled in figure 2.4.	61
2.15	Decompiled correction for the proposed solution.	71
2.16	MASM updated version is now using error A2154 to prevent such bug to be exploited.	71
2.17	Timeline about compiler's backdoor evolution in history.	72
3.1	Representation of the sandbox concept with different technical tools.	80
3.2	Interpretation of the debug break instruction before execution.	99
3.3	Interpretation of the debug break instruction after execution.	99
3.4	Illustration of the correct disassembling of "int 3h".	101
3.5	Illustration of the incorrect disassembling of "int 3h" by Windbg.	101
3.6	Correction to cancel side effect of the misinterpretation from Windbg.	101
3.7	View of the assembly code executed where there is no Windbg.	102
3.8	Graphical view of the Windbg's detection procedure.	104
3.9	Correct interpretation under AMD CPU but misinterpretation on Intel CPU due to the prefix used.	106
3.10	What will be executed on an Intel CPU with the provided opcodes.	106
3.11	Illustration of the assembly semantic by Intel [3].	107
3.12	Illustration of undocumented use of REX prefix in assembly semantic.	107
3.13	Misinterpretation of code by Windbg due to REX prefix.	108
3.14	The code provided in figure 3.13 should be interpreted like this one.	108
3.15	Unsupported instruction should be interpreted as a nop.	108
3.16	Unsupported instruction is not correctly interpreted by Windbg which tries to provide an irrelevant meaning to it.	108
3.17	Instance of unsupported CPU instruction interpreted as nop.	109
3.18	Extract from Intel documentation [4] explaining how to perform cross-modifying code between two threads.	113

3.19	Intel’s cross-modifying code procedure.	114
3.20	Modified cross-modifying code procedure.	115
3.21	Basic detection mechanism	115
3.22	Regular execution of our method without analysis environment.	116
3.23	Execution of our method on an analysis environment.	117
3.24	Debugger detection mechanism	125
3.25	Virtual machine detection mechanism	127
3.26	Comparison of the ROC curves for the different CPUs.	129
3.27	Detection rate of the method with server hardware configuration.	130
3.28	Detection of Docker under Windows.	131
3.29	Detection of Docker under MacOS.	131
3.30	Detection of Docker under Linux.	131
3.31	Detection rates according to the host/Guest operating system used.	132
3.32	Windows host machine.	133
3.33	Windows guest on Windows host.	133
3.34	Linux Host.	133
3.35	Linux on Linux.	133
3.36	Windows on Linux.	133
3.37	3 logical cores on 4 are overloaded at 100 % of activity.	134
3.38	Tests from host machine.	139
3.39	Tests from guest machine.	139
3.40	Example of test on a host machine which has a correlation coefficient close to zero (and thus seen as a guest).	140
3.41	Illustration of simple procedure about how to protect code in an executable file.	145
3.42	Illustration of a chain of different source of noise, each opening the access to the next when the environmental key is correct.	146
3.43	Timeline of the evolution of malware protections, taken from [5].	147
4.1	Top malware families - extracted from [6].	152
4.2	Plan of the next chapters dealing with keylogger threat management.	153
4.3	IBM PC Model F Type 1 keyboard device.	155
4.4	Keyboard IBM-PC model F type 1 mechanism from top without key.	156
4.5	Bottom barrel plate with hammers.	156
4.6	Bottom cover removed to get access to the metal bar that makes contact between the hammers and the PCB underneath.	156
4.7	Capacitive PCB representing the electronic matrix of keys.	156
4.8	Keyboard matrix integrated circuit manager.	157
4.9	Intel 8048 keyboard controller.	157
4.10	Logical diagram representing the circuit given in Figure 4.8.	158
4.11	Logical diagram representing the circuit given in Figure 4.9.	159
4.12	Representation of DIN connector.	160
4.13	Presentation of the architecture between the keyboard and the host’s motherboard.	161
4.14	Interaction between keyboard and host with interruption ports.	165
4.15	PS/2 keyboard device stack.	167
4.16	Rubber ducky dongle used to emulate a keyboard with pre-recorded sequences of keystrokes. The technologies behind this type of device are direct applications of USB and HID protocols.	168
4.17	Different types of formats used by USB devices.	169
4.18	USB three definitional areas.	170
4.19	USB bus topology.	171
4.20	PID Format as defined in USB [7]	172
4.21	Shape of USB packets.	174
4.22	Scheme of USB host and devices interactions.	174
4.23	USB descriptors hierarchy.	178
4.24	Illustration of a configuration descriptor with a single interface but two alternate settings.	180
4.25	Illustration of a configuration descriptor with a single interface and three endpoints.	181

4.26	WinUsb architecture on Windows Operating System. . . . .	184
4.27	Screenshot from the device manager about keyboard's information. . . . .	186
4.28	Current architecture of USB drivers stack in Windows. . . . .	188
4.29	From [8], direction of the exchanges on HID pipes. . . . .	191
4.30	From [8], all USB descriptor structures and HID descriptors. . . . .	192
4.31	Generic Item Format. . . . .	193
4.32	Illustration of the right-handed coordinate system recommended in HID (with mouse, for instance). . . . .	195
4.33	Hierarchy between different HID descriptors. . . . .	195
4.34	Detailed view of hierarchy between different components in HID descriptor classes. . . . .	197
4.35	Simplified view of HID architecture on Windows. . . . .	200
4.36	Content of HidRegisterMinidriver shows us how registration is performed by hooking original IRP callback routines of caller mini-driver. . . . .	202
4.37	Driver stack for a generic HIDClass device with optional and required vendor-supplied components. . . . .	203
4.38	Types of WDM Drivers given in the device call stack. The lowest is the number on the picture, the closer to hardware the driver is. . . . .	204
4.39	Illustration of a link collections array. . . . .	205
4.40	Calls to HidP_TranslateUsageAndPagesToI8042ScanCodes routine in kbdhid.sys driver. . . . .	209
4.41	Pseudo-code of KbdHid_AutoRepeat routine from kbdhid.sys driver. . . . .	209
4.42	Pseudo-code of KbdHid_AddDevice routine from kbdhid.sys driver. . . . .	210
4.43	Pseudo-code of KbdHid_StartRead routine from kbdhid.sys driver. . . . .	212
4.44	Pseudo-code of the beginning of KbdHid_ReadComplete routine from kbdhid.sys driver. . . . .	213
4.45	Pseudo-code about debug purposes in KbdHid_ReadComplete routine from kbdhid.sys driver. . . . .	213
4.46	Pseudo-code about handling changes in keystrokes for KbdHid_ReadComplete routine from kbdhid.sys driver. . . . .	214
4.47	Pseudo-code about auto-repeat in KbdHid_ReadComplete routine from kbdhid.sys driver. . . . .	215
4.48	Pseudo-code about talkative keyboards in KbdHid_ReadComplete routine from kbdhid.sys driver. . . . .	215
4.49	Pseudo-code of HidP_TranslateUsageAndPagesToI8042ScanCodes from hidparse.sys driver. . . . .	217
4.50	Pseudo-code of HidP_TranslateUsage from hidparse.sys driver. . . . .	218
4.51	Pseudo code of HidP_AssociativeLookup routine. . . . .	218
4.52	Pseudo code of HidP_StraightLookup routine. . . . .	218
4.53	Beginning of the content of HidP_KeyboardToScanCodeTable. . . . .	219
4.54	Content of the sub-table HidP_KeyboardSubTables. . . . .	219
4.55	Pseudo-code of HidP_KeyboardKeypadCode routine from hidparse.sys. . . . .	220
4.56	Pseudo-code of HidP_KbdPutKey routine from hidparse.sys. . . . .	220
4.57	Representation of the internal routines notifications in the keyboard HID stack for a pressed key. . . . .	222
4.58	Representation of the architecture of HID keyboard drivers with exported API. . . . .	223
4.59	Hardware identifiers linked with the keyboard device. . . . .	224
4.60	Physical device name associated by the keyboard. . . . .	224
4.61	View from WinObj software where our one of our HID keyboard interface is selected. . . . .	225
4.62	Illustration of the 16-bit Windows synchronous message system. . . . .	233
4.63	Illustration of the Win32 asynchronous message system. . . . .	234
4.64	Entry point of csrss.exe application on Windows 10. . . . .	237
4.65	Entry point called for winsrv.dll module by csrss. . . . .	239
4.66	Part of UserServerDllInitialization in winsrvext.dll, initialization of the provided parameter with useful callbacks. . . . .	240
4.67	Callback SrvCreateSystemThreads from winsrvext.dll and registered by UserServerDllInitialization. . . . .	241
4.68	Function StartCreateSystemThreads from winsrvext.dll. . . . .	241
4.69	Part of NtUserCallNoParam from win32kfull.dll. . . . .	242
4.70	List of routines which can be used via NtUserCallXxx routines selected via an index provided as first parameter. . . . .	242
4.71	Extract from the code of xxxCreateSystemThreads routine in win32kbase.sys. . . . .	243
4.72	Beginning of RawInputThread routine from win32kfull.sys — Initialization and wait for notification events. . . . .	245
4.73	Content of SetWinlogonHotKeys routine. . . . .	246
4.74	Content of SetPenHotKeys routine. . . . .	246

4.75	Setup of <code>RawInputThread</code> routine from <code>win32kfull.sys</code> — Initialization of keyboard devices and desktop. . . . .	247
4.76	Winlogon desktop used to handle UAC. . . . .	248
4.77	Simplified pseudo-code of <code>RIMRegisterForInputWithCallbacks</code> routine (from <code>win32kbase.sys</code> ). . . . .	251
4.78	Simplified pseudo-code of <code>CBaseInput::RIMCallBack</code> routine (from <code>win32kbase.sys</code> ). . . . .	252
4.79	Code from <code>CBaseInput::Read</code> routine (from <code>win32kbase.sys</code> ). . . . .	253
4.80	Simplified pseudo-code of <code>RIMReadInput</code> routine (from <code>win32kbase.sys</code> ). . . . .	254
4.81	Content of <code>RIMStartDeviceRead</code> routine from <code>win32kbase.sys</code> . . . . .	255
4.82	Simplified pseudo-code of <code>rimInputApc</code> routine (from <code>win32kbase.sys</code> ). . . . .	257
4.83	General illustration of read operation with a HID keyboard device. Note the double IRP procedure engaged from each part of <code>kdbclass.sys</code> . . . . .	258
4.84	Beginning of the <code>xxxRegisterForDeviceClassNotifications</code> routine in <code>win32kfull.sys</code> . . . . .	260
4.85	Pseudo-code of the <code>win32kbaseAccessProceduresStream</code> routine in <code>win32kbase.sys</code> . . . . .	267
4.86	End of <code>RawInputThread</code> routine from <code>win32kfull.sys</code> — legacy procedure. . . . .	269
4.87	Pseudo-code of the <code>RitTakeOver</code> routine in <code>win32kfull.sys</code> . . . . .	271
4.88	Pseudo-code of the <code>PrepareForMasterInputThreadTakingOver</code> routine in <code>win32kfull.sys</code> . . . . .	272
4.89	List of routines held by <code>gapfnScSendMessage</code> value in <code>win32kfull.sys</code> . . . . .	273
4.90	List of routines held by <code>ServerHandlers</code> value in <code>win32kfull.sys</code> . . . . .	273
4.91	View of the different routines able to call <code>xxxActivateWindowWithOptions</code> in <code>win32kfull.sys</code> . . . . .	276
4.92	Part of pseudo code of <code>OnReadNotificationsValidGuiThreadContext</code> routine in <code>win32kfull.sys</code> . . . . .	278
4.93	Tree view of calling routines to <code>CitProcessForegroundChange</code> from <code>win32kbase.sys</code> . . . . .	279
4.94	Locations of the flags and values used in the <code>lParam</code> parameter when a key down message is received (from [9]). . . . .	284
4.95	Beginning of the <code>HidpRegisterDeviceInterface</code> routine in <code>hidclass.sys</code> . . . . .	289
4.96	Beginning of the <code>KbdHidCreate</code> routine in <code>kbdhid.sys</code> . . . . .	289
4.97	Pseudo-code of the <code>KeyboardClassCreate</code> routine in <code>kdbclass.sys</code> . . . . .	290
4.98	Pseudo-code of the <code>NtUserSendInput</code> routine in <code>win32kfull.sys</code> . . . . .	297
4.99	Extract from the pseudo-code of the <code>EditionHandleSonarKeyEvent</code> routine in <code>win32kfull.sys</code> . . . . .	298
4.100	Pseudo-codes of <code>zzzStartSonar</code> and <code>DrawSonar</code> routines in <code>win32kfull.sys</code> . . . . .	299
4.101	Illustration about the translation performed in <code>xxxNumpadCursor</code> routine with <code>ausNumPadCvt</code> table from <code>win32kbase.sys</code> . . . . .	300
4.102	List of routines held in <code>aNLSVKFProc</code> value from <code>win32kbase.sys</code> . . . . .	300
4.103	Different phases of translation with the corresponding API to go from one phase to another. . . . .	303
4.104	End of the pseudo-code of <code>win32kbaseNtUserMapVirtualKeyEx</code> routine from <code>win32kbase.sys</code> . . . . .	303
4.105	Content of the <i>keyboard layout file</i> internal structure retrieve for the call to <code>InternalMapVirtualKeyEx</code> routine. . . . .	303
4.106	Representation of the scan code content for the numeric pad. . . . .	304
4.107	Dump of the tables layout structure provided to <code>InternalMapVirtualKeyEx</code> routine. . . . .	306
4.108	Assembly code from a portion of <code>InternalMapVirtualKeyEx</code> routine in the case where the translation is performed with <code>MAPVK_VK_TO_VSC</code> flag. . . . .	307
4.109	Extract of the pseudo-code from <code>InternalMapVirtualKeyEx</code> routine called with <code>MAPVK_VK_TO_VSC</code> flag as a final search in the numeric pad. . . . .	308
4.110	Assembly code from a portion of <code>InternalMapVirtualKeyEx</code> routine in the case where the translation is performed with <code>MAPVK_VK_TO_CHAR</code> flag. . . . .	309
4.111	Extract from the pseudo-code of <code>NtUserGetKeyboardState</code> routine in <code>win32kfull.sys</code> to report the current content of the keyboard for an input thread. . . . .	314
4.112	Pseudo-code of <code>IsKeyStateCached</code> routine in <code>win32base.sys</code> . . . . .	315
4.113	Pseudo-code of <code>GetKeyState</code> function in <code>user32.dll</code> . . . . .	316
4.114	Pseudo-code of <code>GetAsyncKeyState</code> function in <code>win32kbase.sys</code> . . . . .	318
4.115	The <code>aDeviceTemplate</code> structure holding all information to communicate with a given device type. . . . .	339
4.116	Extract from <code>InputApc</code> routine used to notify the read callback routine associated with the device in <code>aDeviceTemplate</code> global structure. . . . .	339
5.1	Summary of the tree architecture of the different keylogger threats. . . . .	344
5.2	Usual connection between PS/2 keyboard and computer. . . . .	345

5.3	Keylogger lies between PS/2 keyboard and computer.	345
5.4	KeyGrabber USB device.	346
5.5	Evolution of the product range at Keelog company. Note the reduction in length.	346
5.6	Plugged version of KeyGrabber Forensic Keylogger.	346
5.7	KeyGrabber Forensic Keylogger length.	346
5.8	Usual USB keyboard view.	346
5.9	Hardware keylogger impact in the USB device tree organization.	346
5.10	Illustration of KeyGrabber Forensic Keylogger Cable from Keelog.	348
5.11	Dedicated circuit to be inserted in a targeted keyboard.	348
5.12	Already trapped keyboard with an embedded malicious printed circuit board.	348
5.13	Example of contact less keylogger from [10].	350
5.14	Commercial electronics SCRC50 cable contact less keylogger.	350
5.15	Various possibilities of infection from the UEFI.	354
5.16	Job proposal for a keylogger development.	363
5.17	Illustration of a decoy architecture to fool keyloggers and preserving the whole system.	374
5.18	General architecture of NoisyKey project.	375
5.19	Architecture of niosykey.dll.	375
5.20	Integration of keyboard shadowing with the driver stack of an HID keyboard from [11].	378
5.21	Illustration of the sequences between user's keystrokes and decoyed ones. It is not hard to find user's relevant information inside log files for keylogger's managers.	381
5.22	Example of random layout from [12].	384
5.23	Example of random layout from [13].	384
5.24	Example of random layout from [14].	384
5.25	Illustration of a random keyboard with hidden key's content from [15].	384
5.26	Interaction of a Virtual-Machine Monitor and Guests (from [16]).	388
5.27	Windows 10's general architecture with VBS security activated (from [17]).	392
5.28	General SpyShelter anti keylogger rules manager.	395
5.29	SpyShelter single application filter.	395
5.30	Setup procedure from SpyShelter proposes to install the Keystroke Encryption driver which is marked as an experimental feature.	395
5.31	KeyScrambler is available in several versions. The difference belongs in the list of software to protect supported.	398
5.32	KeyScrambler uses a specific bar to display the content of ciphered keyboard data.	398
5.33	Illustration of the possible architecture used by KeyScrambler.	400
5.34	Pseudo code of the function recording the KeyCrypt's window messages handler.	405
5.35	List of functions targeted for hook procedure.	405
5.36	Part of the trampoline hook procedure used.	405
5.37	Slides from the technical documentation of the <i>Zemana KeyCrypt SDK</i> project.	406
5.38	Hook function in the Dll called by the original function.	407
5.39	Post-callback used after original hooked function has been called.	407
5.40	Illustration of LMT Anti Logger extracted from a youtube video.	408
5.41	What keyloggers see.	410
5.42	Architecture used by GuardedID.	410
5.43	How GuardedID works with its own self-monitoring capability.	411
6.1	General architecture of our protection solution.	423
6.2	System-supplied driver stacks for USB keyboard and mouse/touchpad devices.	437
6.3	Filtering architecture as given where the <i>UpperFilter</i> value for keyboard is defined with "GostxBoard, kbdclass, Ctrl2Caps".	439
6.4	View from OSR's DeviceTree software on our virtual machine, configured with two keyboards (PS/2 & USB/HID) running Ctrl2Caps and GostxBoard drivers.	440
6.5	Simplified view of the cipher transformation on scan codes used by our solution.	444
6.6	Keys that are kept as cryptosystem output are in white. Keys which are in grey should be avoided.	444
6.7	Transposition from scan code to index.	445
6.8	Our ciphering system for scan codes based on a random shuffled permutation.	446

6.9	Basic principle of the keyboard key cryptographic system with <i>over-ciphering</i> .	447
6.10	Illustration of the algorithm used for the cryptographic chain implementation.	447
6.11	Illustration of the cipher and the decipher procedures. Note that intermediate ciphered values are all deciphered until we have a value from the input set.	448
6.12	Use of the cryptographic system as a pseudo-random generator to protect scan codes.	448
6.13	Transform the original allowed output set of values to only draw allowed values whatever is the random value index.	450
6.14	Illustration of a regular driver object with dispatch routines set.	454
6.15	Hooked dispatch routines for monitoring purpose by a third party driver (monitoring dispatch routines are in red).	454
6.16	Possible system power state transitions, extracting from [18].	465
6.17	Pseudo code illustration of <code>GostxBoardInitiateSession</code> function to initiate protection with the driver.	469
6.18	Illustration of steal focus when the driver still ciphers.	473
6.19	Illustration of steal focus with protection stopped.	474
6.20	Windows boot process consists of several phases (extracted from [19]).	476
6.21	Tree architecture of the EFI partition.	479
6.22	Secure Boot and Trusted Boot illustrates the boot process (freely inspired from [19]).	481
6.23	Bootling procedure before Windows 8.	489
6.24	Bootling procedure with secure boot and ELAM technology able to control third party drivers and ensures that antivirus driver has been correctly loaded.	489
6.25	Device tree representing the drivers and their associated device objects with a HID source driver (from [20]).	498
6.26	Illustration of the proposed solution.	500
7.1	Illustration of the different steps to perform a <i>classical</i> Dll injection procedure.	513
7.2	Code written by the developer.	513
7.3	Equivalent code the first time the function is called.	513
7.4	Simplified view of the interactions present in <code>IMAGE_DELAYLOAD_DESCRIPTOR</code> structure.	514
7.5	Replacement of the original <code>IMAGE_DELAYLOAD_DESCRIPTOR</code> structure by the one used for injection purpose. Note that the generic exported function by the process is resolved in blue on the figure when resolved by the process.	516
7.6	Injection procedure based on the generic function used to interface the first delayed function call.	517
7.7	General representation of the protected folders against unexpected software trying to access files.	519
7.8	General resume of the anti-ransomware directory protection solution.	520
7.9	Regular access provided to Word.	521
7.10	Access denied to third party software.	521
7.11	<b>Packing</b> process with an <b>original</b> executable file packed with a <b>packer</b> producing the <b>packed</b> executable file.	521
7.12	General view of a packed file. This one uses the unpacking payload to extract the original file in memory before executing it. That way, the original file is hidden in the packed file.	522
7.13	General view of our tool able to manage the classification of polymorphic packers.	524
7.14	Classification of a new element in our unsupervised data-mining clustering system.	525
7.15	SysMain global operation.	530
7.16	SysMain internal databases construction process.	531
7.17	SysMain internal databases reading process.	531
7.18	Content of the cache concerning WinRAR software, as referenced by SysMain in the Cache Manager.	532
7.19	Illustration of the UEFI boot procedure in two stages. The first from the ship on the motherboard and the second a boot partition on the hard drive disk.	535
7.20	PI Architecture Firmware Phases (extracted from [21]).	536
7.21	Basic architecture of Intel computer with AHCI interface.	537
7.22	Different layers of the SATA protocol model.	537
7.23	Intel Minnowboard Turbot computer.	538
7.24	Access to the mother board on a laptop.	538
7.25	Electronic setup for flashing the contents of the machine's BIOS/UEFI.	538
7.26	General resume of the procedure used to cluster web-pages in different families.	544

7.27 Overlapping rates computed between regular web-sites vs trapped ones. It helps to measure the impact of bad detection of cluster by a given distance. . . . . 547

THIS PAGE INTENTIONALLY LEFT BLANK

---



# Résumé

La compréhension de la sécurité informatique passe nécessairement par une réelle maîtrise des briques de technologies élémentaires qui constituent son socle et par une compréhension des dynamiques qui motivent l'émergence de nouvelles menaces.

C'est dans cet objectif que les travaux de recherche de cette thèse ont été menés. Il nous tenait à cœur d'intégrer une vision duale, c'est-à-dire autant offensive que défensive, pour faire face aux menaces actuelles et à venir. Tant par une connaissance très technique que par la maîtrise des dynamiques liées aux contraintes de l'attaquant, il a été ici possible de concevoir des outils à la fois offensifs et défensifs.

Les présents travaux visent à améliorer la défense de plusieurs systèmes après une phase préalable de recherche des failles dans ces derniers. Ainsi nous avons travaillé sur plusieurs axes pour offrir une solide défense dans la profondeur. Dans un premier temps, nous avons amélioré la sécurité du compilateur MASM en découvrant une faille présente depuis plus de 20 ans, qui permettait jusqu'alors à un attaquant d'introduire silencieusement des backdoors lors de la compilation des programmes. Nous avons également dévoilé de nouvelles techniques d'évasion pour les malwares dans l'objectif de mieux anticiper ces dernières. Une de ces techniques permet de détecter n'importe quel type d'environnement d'analyse automatique utilisé de nos jours. Au meilleur de notre connaissance, aucune autre technique ne propose de telles capacités opérationnelles. Enfin, nous nous sommes penchés sur le fonctionnement des keyloggers, grâce à un travail inédit de documentation des mécanismes internes de Windows 10 par rétro-conception.

Cette étude nous a permis d'élaborer une solution contre les keyloggers, qui est plus performante que l'ensemble de celles existant à ce jour. De la correction de vulnérabilités trouvées dans un compilateur à la conception de nouvelles techniques d'évasion, nous nous sommes assurés d'apporter des solutions innovantes pour améliorer la sécurité des systèmes à long terme. Cela par le biais d'outils que nous avons construits pour neutraliser les keyloggers, par la conception de méthodes de forensic basées sur l'analyse du service Superfetch, par la découverte de nouvelles techniques de détection de crawler-traps, par l'étude de système de chiffrement total basés sur l'UEFI et enfin par la création d'algorithmes de classification de malware. Ces différentes recherches ont abouti sur de nombreux résultats, dont la parution de la CVE-2018-8232, l'édition d'articles scientifiques et de publication dans des conférences internationales académiques et de *hacking* telles que Defcon ou Black Hat US.

En conclusion, nous avons cherché à privilégier des travaux qui visent à analyser, évaluer et améliorer la sécurité d'un projet à différents niveaux. C'est pourquoi beaucoup de domaines ont été abordés car la sécurité informatique est un domaine global et finalement multidisciplinaire, qui nécessite bien souvent des compétences transverses. Ce qui fait que nos travaux œuvrent avant tout à sécuriser l'information traitée par un système automatisé, de sa collecte, de son traitement à son stockage tout en s'assurant qu'aucune menace ne puisse agir contre des programmes s'exécutent conformément à ce qui est attendu.

Mots-clés : sécurité informatique, programmation bas niveau, rétro-conception, malware, vision offensive, keylogger.

THIS PAGE INTENTIONALLY LEFT BLANK

---

# Abstract

Understanding computer security requires a strong knowledge of the underlying technologies and a deep awareness of the origin of today's threats. To help the community facing the new challenges of computer security, our research is based on these fundamentals.

One of our goal was to include in our work, both of the offensive and defensive approaches, to best meet the requirements of the fight against cyber threats. Starting with a technical background and knowing the attacker's point of view allowed us to design both offensive and defensive tools.

Our work aims at enhancing the defense of several systems by first looking for vulnerabilities in them. Thus, we worked on several axes to provide a strong defense in depth. At first, we improved the security of MASM compiler by exploiting a vulnerability that has been present for more than 20 years. It had allowed to silently introduce backdoors at compilation time. On the other hand, we developed new evasion techniques for malware, in order to better manage them. One of these techniques allows to detect any type of automatic analysis environment used nowadays. At the best of our knowledge, there is no other technique able to produce similar results with such operational consequences. Finally, we managed keyloggers threat thanks to an extensive and unpublished documentation of Windows 10 internal mechanisms, achieved through a reverse engineering process. Within this context, we proved that it is possible to produce a solution above those existing at the moment.

From the correction of the vulnerability found in the compiler to the design of new evasion techniques, we made sure to bring innovative architectures to improve the security of the systems in the long run. This has been achieved through the tool we built to prevent users from being victims of keyloggers, through the new forensic methods we elaborated based on the Superfetch service, though the new techniques we discovered to detect web crawler-traps, though the studies done about UEFI full encryption system and though the algorithms created to classify malicious programs.

All this research work has been published in different academic and hacking international conferences, including DefCon and Black Hat USA in addition to several scientific articles and CVE-2018-8232. In conclusion, we have privileged works that aim to analyze, evaluate and enhance the security of a project at different levels. Many domains have been covered because computer security is a global and ultimately multidisciplinary field, which often requires cross-disciplinary skills. The idea is to always secure the information processed by an automated system, from its collection, its processing to its storage, while ensuring there is no threat acting against programs which are running as expected.

Keywords: cyber security, kernel programming, reverse engineering, malware, offensive, keylogger.

THIS PAGE INTENTIONALLY LEFT BLANK

---

# Chapter 1

## Introduction

### 1 General presentation

Why do we need security in computer science? After all, what we call computers nowadays are nothing more than power supplied machines made of metal, cables and plastic. What would security have to do with what are, in the end, only high-performance calculators? Asking such a question forgets the preponderant usage that we have of these machines. They durably changed the world since the second half of the twentieth century. Connected through Internet networks and the miniaturization leading part of their development make it difficult to imagine to pass one day without a smart-phone, a laptop, a computer... We entrust them with the management of our lives, that is to say our finances, our habits, our contacts, our exchanges our colleagues, friends, our most intimate secrets (professional, medical, intimate), our time... What would be the consequences if it would be possible to deliberately tamper with these devices? Who can predict the magnitude of the disaster if it would be possible for an entity to collect all the data stored on these machines? This is the role of security: to prevent these painful questions from being asked, because there is no good answer to them.

If the interest about security is understood, the question to what can be done to improve it arises. Not a week goes by without news of a hacking incident. And this is only the tip of the iceberg. McAfee Labs claimed in a report to detect 419 threats per minute in Q2 2020, an increase of almost 12% over the previous quarter [22]. This observation is nothing but new since malware and threat landscape was already in a growth in 2000s [23] and projections about nowadays are not better [24]. Regarding vulnerabilities exploited by malware, 18,352 have been identified in 2020 (where some of them could be exploited as backdoors or 0-days) [25]. If a proof was needed, these observations on the evolution of the threat from an antivirus company show that the issue of security is far from being over. And it is precisely on the topic to see how it is possible to improve the security of a computer system that this thesis was written.

The question is therefore to know how it is possible to secure or more directly to improve the security of a computer system. On the one hand, we are convinced that security is intimately linked to what rules the technology in its most intimate aspects. Building something safe is impossible if it is not perfectly understood. More directly, we cover our problems by a technical low level aspect, in the sense that we try to be as close as possible to the hardware used by computer. On the other hand, with an original approach used in this study, we took the attacker point of view to better attempt to improve the security of a system. From the offensive point of view, we propose to better understand the causes and consequences that allow a potential attack a system. From the offensive point of view, it is possible to test the existing systems. From the offensive point of view, we can find new attacks and thus anticipate by correcting or proposing adapted defenses. Because the final objective is to provide better security.

### 2 Context of this research work

The protection of a computer system can be done in two different ways. Either we deal with the causes that can potentially cause this system to no longer act as expected, or we deal with the possible consequences resulting

from these causes on the system.

Dealing with the causes that may contribute to compromising the security of a system can be done in different ways. On the one hand, simply by avoiding introducing a flaw in the system at conception or, on the other hand, at implementation time. This is easier said than done. Systems are designed by men and women who are inherently fallible. Of course, it is possible to design tools or procedures to automate controls and increase security. But these tools are not silver bullets since they are created by humans and therefore, they are also prone to be imperfect. Errors can come from negligence or be intentional (in the case of *backdoor*). Of course, this situation is not hopeless. Between an imperfect system and no system at all, there is an obvious choice. Moreover, while perfection is desirable, it is not always necessary to reach it to get good results.

When designing a security system, it is always better to address the causes of a potential problem than their consequences. For the same reason that it is better to avoid fire than having to extinguish it, this guarantees a greater reliability of the system over time and the possibility of continuing to build on solid foundations. In our IT context, this means being interested in security design, code quality, secure programming, code evaluation procedures, unit testing and more generally what we can call DevSecOps [26, 27]. Guarantee that what is developed does what it is supposed to do, without adding flaws or doing more than what is expected.

In addition to ensuring the security of software, security mechanisms can be used to prevent the execution of potentially malicious code. This is commonly known as antiviral detection. The objective is to analyze any program hosting on a computer before it is executed. In practice, antivirus is trying to guess by analyzing a given program whether there are potentially dangerous behaviors. There are several strategies for doing this [28, 29]. On the one hand, old fashion style, by performing a static analysis by examining the program, looking for occurrences of malware pattern without running any code from the analyzed program. There are several ways of doing this, but a distinction is generally made between deterministic models (which look for a specific and known element from malware samples) and heuristic models, which determine probabilistically the susceptibility of the target to be malicious. In this last case, it provides the possibility to find known or *unknown* malware by looking for pieces of code that look to be "malware-like," instead of looking for specific known signatures from former samples.

On the other hand, it is possible to perform dynamic detection methods by running the code and observing its behavior. By checking the activity of the evaluated program, antivirus is trying to model the program's behavior by making a distinction between allowed actions from those that are suspected or forbidden [30]. In practice, antivirus are not logging all instruction executed by a process (it would be too long and too complex). Instead of, it focuses on all relevant actions such as access to file system, registry, network, devices, other processes and other relevant resources. By comparing the observed behavior from the one previously recorded of known malware programs (or malicious strategies), it is possible to detect and even to prevent the malicious action from occurring by reacting in time. Such an operation can be performed in two different ways: either manually with a human piloting a tool, or automatically with a dedicated tool. Each solution has its advantages and disadvantages. Of course, malware samples are not passive facing such dynamic analysis and they try to probe such analysis environment in order to stop their malicious behavior before detection or to evade analysis.

When it is no longer possible to prevent an attack before it occurs, then care must be taken to reduce its scope. In practice, threats do not always take the form of an executable program on a hard disk, downloaded and executed by the user. In many cases, the exploitation of a vulnerability allows to execute code remotely [31, 32] and thus to perform malicious actions on the target. This way, programs that are analyzed and considered to be safe can become the vector of attacks. In such a case, there is a design flaw or programming error in the targeted process and it was not possible to detect it before execution. Therefore, the consequences must be addressed by reducing malicious opportunities.

We come to the point where we have to deal with the potential consequences of an illegitimate action in the system. Ideally, we should be able to permanently contain all malicious actions undertaken. But unfortunately this is not possible for at least two reasons. On the one hand, if the malicious action comes from a legitimate process but is vulnerable to a remote takeover (by any means), then the malicious code will act with the rights of the process and it will inevitably be able to initiate the same actions as the latter (but perhaps with illegitimate

---

intentions). On the other hand, in case an attacker manages to see malicious code executed in its own dedicated process, then everything depends on the rights inherent to this process.

It is precisely with the objective of limiting the threat and its consequences that security can be conceived. For this purpose, two approaches are possible. The first one is based on a global approach by limiting the scope of the processes running on the machine. In a way, this corresponds to the real-time action provided by an antivirus software that monitors file access or network traffic. In a more specific way, it could be a sandbox or containerization solution, that is to say the process runs in a controlled environment where its actions are observed in order to neutralize the dangerous ones.

The other approach consists in securing a particular action or a process. The idea here is to protect a particular resource to allow access only to trusted elements and to act with maximum security. This can be technologies like the *Trusted Platform Module* (TPM) [33] with hardware support, but also a software approach with security-enhanced programs. We oscillate here between the general protection of the system to avoid malicious actions and the individual protection which reinforces the access to the resources of a given process. This approach aims to mitigate the consequences of a possible security breach.

When we combine these various modes of defense, we observe that they overlap, acting in layers, like a continuous mesh. The fact of designing the security of a system as a series of sub-systems, where the possible failure of one is handled by another sub-system, and so on, is what is called *defense in depth*. The protection of the whole system is provided through several independent methods. The main idea is to provide different layers of security at different levels where potential issues could be. It is precisely with this idea that this thesis was built.

### 3 Problem and Significance

The general question we are trying to answer in this thesis is to know how, through the offensive approach, it is possible not only to make rise new threats, but also to know how it is possible, by having a deep knowledge of the system, to neutralize old threats and potentially new ones. The problem exposed here is to attempt to improve the knowledge in computer security. By finding (and fixing) any kind of vulnerabilities in software. By creating protection tools. By improving knowledge on the subject, both by studying the existing threat and by anticipating what it could be.

#### Problem 1: General problem covered by this study.

How to propose more efficient security solutions through an offensive and low-level approach in computer security?

Obviously, dealing with computer security, it is not possible to cover such a large topic in a single thesis. It is too vast a field. That is why we chose to illustrate our approach by selecting an example of defense from offensive approach in three different cases. The idea is to deal with different sub-problems in order to bring a better understanding of computer systems, but also to counter and prevent current and future threats. Moreover, it seemed coherent to us to concentrate our efforts on a single environment, i.e. the latest operating system from Microsoft: Windows 10. According to diverse statistical sources, it is the most widely used operating system for desktop PCs worldwide (with approximately 78 % market share) [34, 35].

A first problem is the security of software development. As explained, in addition to the absence of flaws written inadvertently in a program, it is necessary to ensure that the code programmed is the one that will be executed on the machine. In practice, to create a program, the developer writes the source-code in a given programming language and it is then processed to produce an executable file. This processing phase is called *compilation*. What is not well understood today is whether it is possible to exploit a bug in the compilation phase to successfully introduce a vulnerability into the compiled code. There are academic studies that tend to show that this is theoretically possible or constructions that come close to this, but without making strong assumptions to justify certain characteristics. These assumptions are significant because such attacks remain

quite theatrical. In addition, they are considered as minor since they have always been promptly corrected when very few had happened.

**Problem 2: Improving security at the software compilation level.**

Is it possible to deliberately introduce a backdoor into software at compile time in a stealthy and effective way?

Knowing a real attack exploiting real vulnerabilities present for a long time will provide a benefit since it will prove two points. The first is that compilers, as any other software, are prone to error stealthy exploitable for nasty goals. More directly, an error could be exploited directly in an operational context. The second is that such issue should be taken more seriously because security is not only about what has been written in the original source-code.

The second problem is to focus on malware evasion methods to avoid analysis. Indeed, as explained, in order to prevent a malicious program from running, it is still necessary to be able to analyze it. And it goes without saying that malware samples do their best to avoid this. In this area, there are several gaps. Firstly, the techniques used are varied and depend on the analysis tools used and the method of analysis beyond (manual or automatic). In practice, surveys can be focus only on a subset of the techniques used [36] or depicted in a trivial way without any detailed overview [37, 38]. There is a real lack for a comprehensive classification regrouping all different known evasion techniques. Secondly, there is a lack in anticipating new trends in evasion techniques. There is a real need to portray the current trends and evaluating new evasion methods in a synthetic way. And finally, there is no *generic method* of escape today. By generic, we mean the ability to act on any analysis environment used. And by *method*, we mean a functional and reliable technique with a very high probability rate of success.

**Problem 3: Improving automatic malware analysis by preventing evasion.**

Is it possible to efficiently detect and evade automatic malware analysis environments?

In all cases, this will allow us to have a synthetic vision on the state of the threat, but also to potentially produce a taxonomy more likely to help the industry and academia. In this end, being able to produce new evasions would not only address potential future evasions (or current unknowingly ones) but would also disarm malware from trying to use these methods again. Let us note finally that a generic method of escape would have a scientific interest in the sense that it would constitute, from a conceptual point of view, a kind of "*holy grail*" for malware. Being concerned with such methods (and especially by the way these techniques can be designed) can help to better design the defenses implemented by the analysis systems.

Finally, our third problem was to deal with a particular threat to mitigate its nasty consequences. In our case, we focused on keyloggers. Why keyloggers? Because keyloggers are threats that have been used since a long time and their actions are still embedded in the most modern malware. Can they be easily detected? The answer is no [39, 11, 40]. As we will see in this study, a keylogger is ultimately nothing more than a program that interfaces with the keyboard. The difference with a legitimate program is what it does with the received data. Basing a detection on a behavior close to the one from legitimate software programs is a hard task. May be too hard. This is why there is an interest in neutralizing the threat, rather than trying to characterize it. This is an area that has been studied both academically [41, 42, 43, 44] and industrially (Chapter 5, section 4.3). If each of the presented approaches is interesting, the proposed solutions (when they go beyond the concept stage) are sometimes approximate or unable to really neutralize the threat. Several points are misunderstood today. On the one hand, by the nature of the threat, its capabilities and means of action, it is hard to completely defeat it. On the other hand, for some cases, technical concepts specific to operating system make these solutions either useless or dangerous for the user.



**Problem 4: Improving anti-keylogger security.**

Is it possible to neutralize keylogger threat on a Windows 10 operating system at running time?

The interest of research in this area is very clear: make systems free from the threat of keyloggers. If it is not possible to remove these *parasites* because they are hard to detect, it is nevertheless possible to improve the symbiosis with them by preventing them from collecting sensitive information. The direct interest of this type of study is to allow the design of software programs that are keylogger-safe. In addition, the study of existing solutions, sometimes presented in a critical light, also helps to point out weaknesses and potential solutions to provide improved solutions. Finally, the complete study of the functioning of the keyboard within a computer brings an unprecedented understanding of the internal mechanisms of the Windows 10 operating system.

Generally speaking, the interest of this thesis is twofold. On the one hand, it suits our idea of defense in depth. Thus, we conceive security since the design of the software (by practicing quality development, secure, especially fed by the sometimes critical analysis that we could carry on the third party projects observed) including all different steps of the development (from source-code to the compilation phase). This design takes also care about performance requirements, user's experience, guaranteeing privacy and everything that makes a quality software. Then, we are dealing with the possibilities for malware to evade analysis. The idea is always to prevent the execution of the potential threat. And besides an exhaustive survey on the state about evasion techniques (which can help to detect them), there is also an interest in anticipating the future threats in order to better understand it and eventually to defeat it. Finally, assuming that the detection of the threat may have failed or that it is simply out of reach, we try to neutralize it anyway. We attempt to highlight a global approach against cyber threats.

On the other hand, the second interest of this thesis is the documentation work about computers and technology. A better understanding of closed systems like Windows or some malware helps to design better defenses, to design better tools and — with a great humility — to try to modestly improve the meaning about very complex or unknown concepts. This study is a unique opportunity to detail unpublished and highly specialized knowledge.

## 4 Roadmap

### 4.1 Structure of the thesis

The construction of this thesis is in line with the problem and sub-problems explained previously. It is essentially composed of four main parts, sometimes broken down into several chapters.

The first part is composed of by Chapter 2 which aims to introduce of a backdoor by the operational exploitation of a vulnerability within the MASM compiler. The idea in this chapter is to show that it is quite possible to exploit a decades-old vulnerability in a compiler used in the industry to introduce a backdoor. More directly, we will show how to introduce a mechanism specifically written (but harmless) in a source code that allows a third party to gain a remote access to the system driven by the program compiled from this source code. First, we will show an unknown vulnerability in the MASM compiler that allows to silently change the meaning of the generated source code in some situations. Then, we will expose the proof of feasibility of the introduction of a backdoor by explaining step by step how to proceed. Finally, we will explain how the flaw was fixed by Microsoft and the implications that such a disclosure could potentially have.

The second part is focused on protection used by malware against analysis. In France, in 2019, nearly 45 % of small and medium-sized companies had suffered a cyber attack according to Daniel Benabou [45]. And the consequences can be dramatic. This is why malware analyzing matters, to prevent such dramatic consequences. This part is composed by Chapter 3 only. By first exposing an exhaustive state-of-the-art about evasion techniques used by known malware, we propose a classification of different evasion techniques to make a distinction between manual and automatic dynamic analysis evasion. After a brief survey on countering malware evasion, we will provide two new evasion techniques, one against manual and one against automatic dynamic analysis evasion. In the first case, we will be interested in the analysis tools called *debuggers* to fault them. By various

original methods, we will see how to deceive a human analyst in front of such a tool and in the case of the Windbg debugger, we will be able to detect the use of a debugger at runtime. In the second case, we will see a generic method able to evade all dynamic analysis environment tools with a high rate of success. In this section, discussing the exploitation of undocumented mechanisms on Intel processors, we crafted a new evasion method for debuggers, virtual machines and Dynamic Binary Instrumentation tools. Finally, we will discuss the evaluation (through a test campaign), the reliability and possible improvements of this method.

The third part aims to deal with the case of keyloggers. It is divided into three chapters. In Chapter 4, we will discuss about technical background behind keyboard technology. The latter exposes the technical characteristics of a keyboard from the physical pressure of a key to the reception of the signal sent from the device by an application under Windows. For this, we will talk about the communication protocols used by keyboards, namely the PS/2 protocol and the most modern USB/HID protocol. We will then see in detail how the information received from the physical device is processed by the kernel of Windows operating system, according to each protocol. From there, we will continue our study by explaining how Windows gathers information from two different protocols and then processes received keystrokes to be exploitable by any applications (including keyboard layout management, translation from different languages, special character generations, special system signals such as CTRL+ALT+DEL and so on). Finally, we will see in an exhaustive and detailed way how (user-mode) applications can recover data from the keyboard, whether they are directly displayed on the screen or not. This long chapter covers all the technical background in order to be able to deal with keylogger technology (which relies on the keyboard) and solutions to neutralize them.

Chapter 5, we will establish the state-of-the-art of the keylogger threat which can be located at different levels (hardware or software). For various reasons, we have chosen to focus on the software threats, because it is the easiest one to broadcast. From the latter, we will try to present the various possible methods of action for keyloggers, based on explanation given in chapter 4. Once the knowledge of the different threats has been established, we will propose a state-of-the-art of the solutions that exist today to counter the keylogger threat. Starting from academic solutions, we will make the difference between active and passive solutions. The first ones aim at detecting keyloggers (as a traditional antivirus would do) and the second ones aim at providing a proactive action on the threat by neutralizing it at runtime. This is such passive solutions that we are trying to improve here. This is why we finish the state-of-the-art by studying the existing industrial solutions. In this study, we take a critical look at existing solutions, emphasizing advantages of certain methods as noting the weaknesses of others. For the sake of consistency, our approach is based mainly on research papers in the academic case and also on statements from software vendors. It is not as precise as reverse engineering, but our objective is not to technically document existing solutions, but rather to analyze the strategies in order to propose a solution that synthesizes all their advantages while limiting the weaknesses.

Once the technical knowledge of the underlying system has been acquired and once the knowledge of the threat and the existing solutions, it remains for us to propose our own solution in Chapter 6. The latter starts by expressing the specifications of our solution. The goal is not to deal with all possible software keylogger threats (this would not be technically possible, which we will be showed), but to deal with the most possible ones. In practice, our objective is to protect a given software equipped with our protection solution. This one is guaranteed to receive the contents of the keyboard in a secure manner, without restricting either the user experience or performances and especially without any possibility for a user-mode keylogger to gain access to it. Our solution is described in its architecture and with relevant implementation details. Then, we will presents the limits of our solution. That way, we will propose improved possibilities to push the limits and the associated price with these improvements. Finally, it will be an opportunity to propose ideas for different solutions that could potentially present interesting results.

Chapter 7 is a prelude to the conclusion given in Chapter 8. In Chapter 7, we will briefly present all the projects carried out in the framework of this thesis and which have not been presented in the framework of this document. For the sake of space mainly, for the sake of preserving some intellectual property sometimes, and for the sake of coherency also. A thesis is often the occasion to see many subjects and our researches allowed us to explore different possibilities and different directions during our experiments. Nevertheless, the work presented here has sometimes been published and we will refer to the reader, whenever necessary, to the appropriate references for more details.

Finally, Chapter 8 will resume all our work, including the different answer of the different problems exposed in section 3. The objective is also to step back and discuss the research methodology employed and the potential implications of the research presented in this document.

### 4.2 Illustrated representation of the thesis

For the sake of readability, we propose to give in Figures 1.1 and 1.2 a representation of the plan of the thesis in the shape of a flowchart. This aims to show the sequence of the different chapters, the main documentation, reverse engineering and research work done by ourselves. We also highlight the different stages of publications during this thesis.

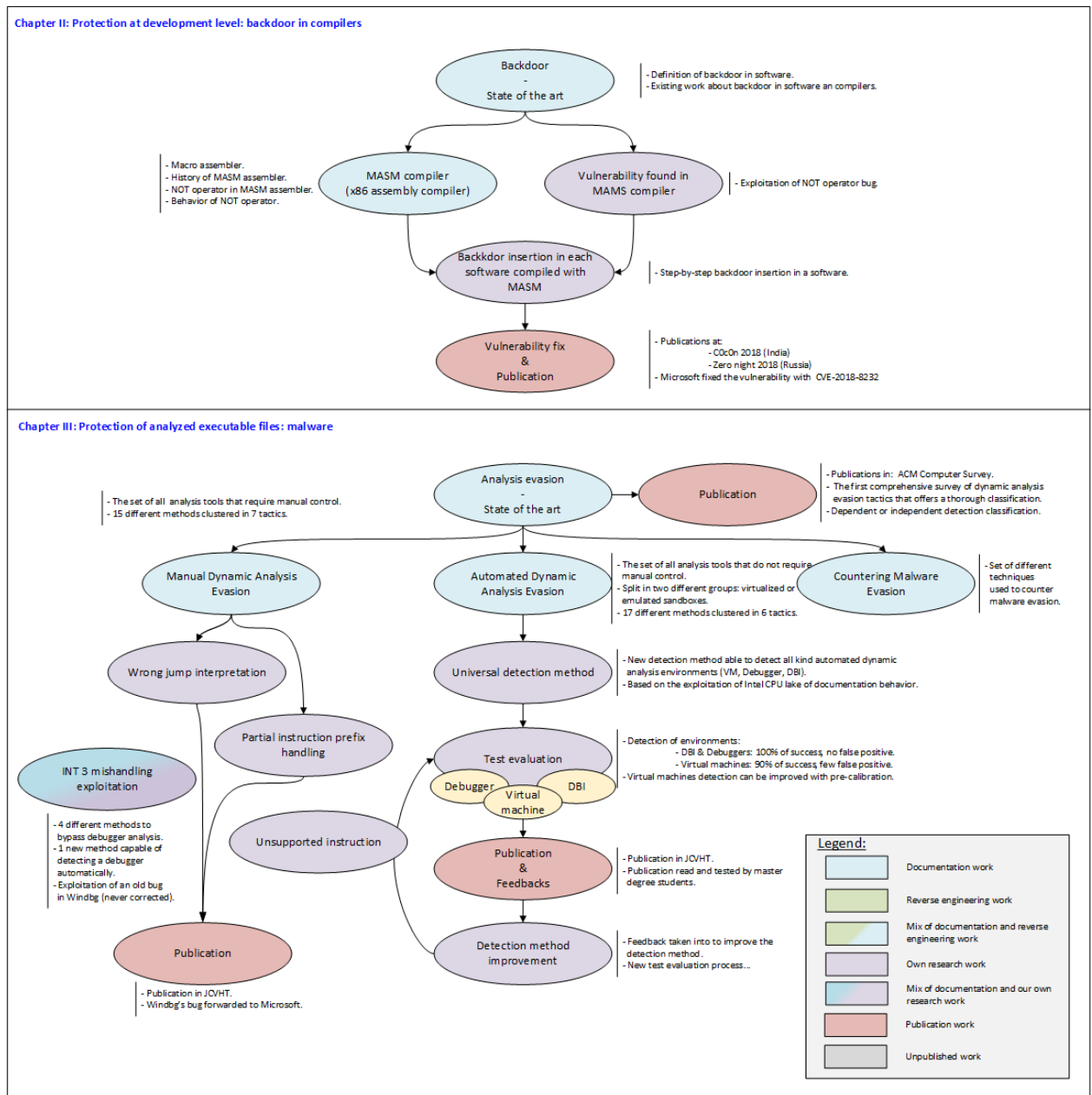


Figure 1.1: Representation of the thesis architecture (1/2).

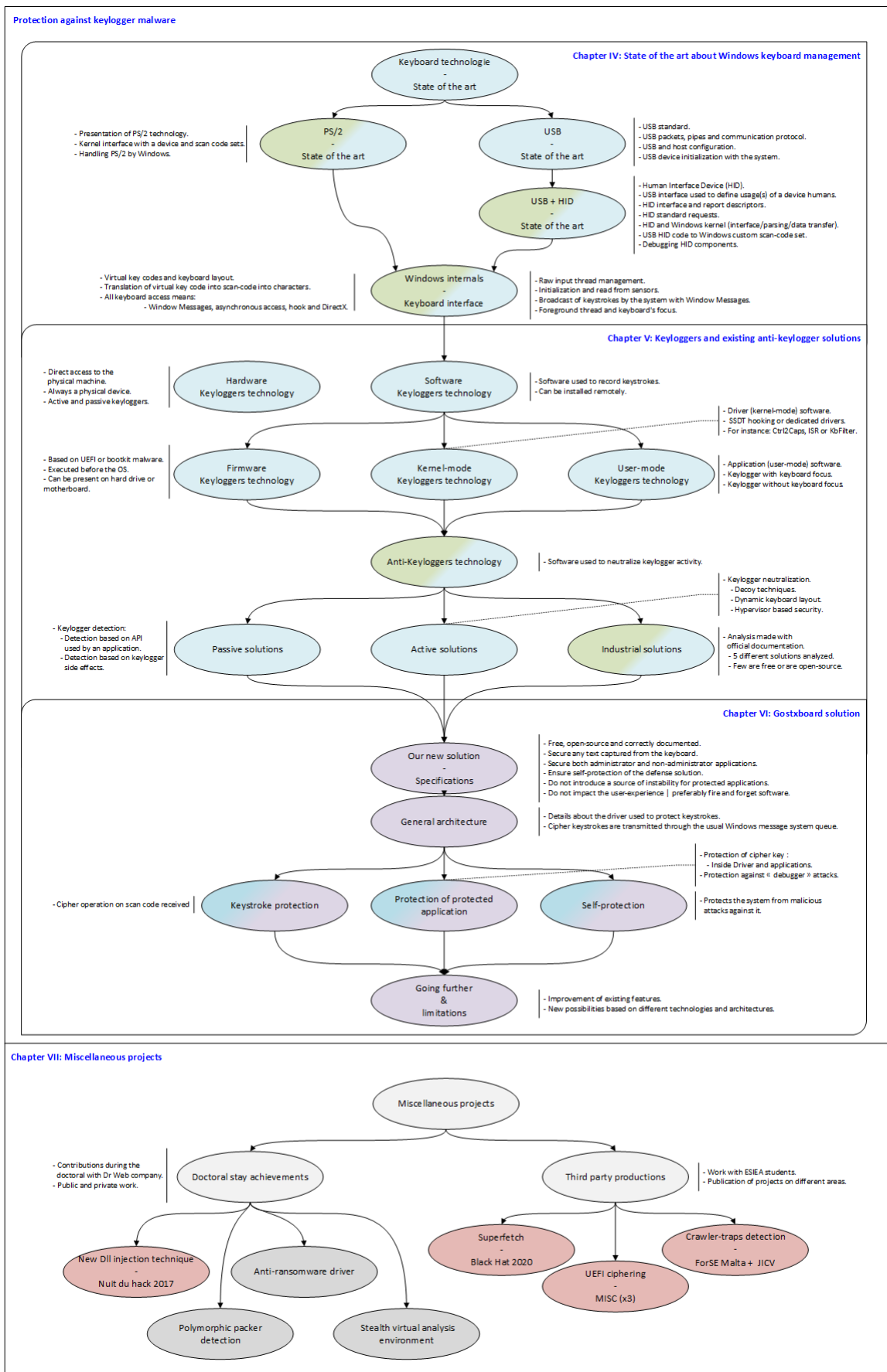


Figure 1.2: Representation of the thesis architecture (2/2).

### 4.3 Reading improvements

This thesis is relatively dense and lengthy. The reason is mainly due to the high level of technical details that is sometimes given in some chapters. Of course, it would have been possible to synthesize most of these details, but it is also necessary to see that these technical details are subtleties that sometimes allow to orient a choice in a given direction. This helps to justify why something is done that way rather than the other way. From our point of view, on key point of research is precisely to allow to justify technical choices and thus to allow develop optimal solutions with respect to the specifications. It is also, for those who would be passionate, the way to dive into the internal mechanics of the Windows 10 operating system.

Nevertheless, it can be long and boring. This thesis also takes this point into account. Chapter 4 is particularly responsible for the length of this thesis. It gathers the technical details of the keyboard operation in Windows 10. These details matter and they represent a singular contribution of the thesis because to the best of our knowledge, there is no publication of any kind that explains at this level of detail the internal functioning of a keyboard under Windows.

In order not to make the reading of the manuscript too fastidious, the reader can choose to omit Chapter 4 that could be considered as superfluous without altering the understanding of the text. In order to understand Chapters 5 and 6, the reader must have a minimum knowledge about the Windows operating system and the functioning of standards that control the interaction between devices and operating systems in general. The assimilation of this minimum becomes easier thanks to the additional information that we thought necessary to include in Chapter 4 in order to avoid additional references to numerous sources, when they exist, since some details given in this study are unpublished.

In order to allow the initiated reader to be able to omit Chapter 4 without inconvenience, we propose to write boxes within this chapter resuming relevant points. These boxes are numbered and referenced within the text of the chapter, generally at the beginning of a section or sub-section or directly in the text whenever required. Each box holds key points written in a concise way, either to resume what is presented in the following lines of text (a green *Resume* box) or to the technical detail about how a specific procedure is implemented (a red *Key-Point* box). These boxes are referred to in the following chapters whenever a specific point is mentioned. It aims to resume a position or the main points described in the following text. For instance, if we need information to know how the translation between scan-codes and virtual key codes is performed, we can refer to Key-Point 4.45. Reading the Key-Point is supposed to be enough to understand the main points explained in this document. Nevertheless, the reader willing to extend his or her knowledge on a given box is invited to read the rest of the section referencing this particular point, for more details.

This reading improvement is mostly used with Chapter 4, but not only. Indeed, for the sake of coherency and hoping it could help to read our long study, this system of boxes has been used in the whole document. That way, it helps to understand the main arguments developed in this study while keeping details and explanations in the main corpus of text.

In addition, for ease of reading, it is possible to read each part independently. The four parts have been written with the assumption that if the reader has a minimum of knowledge in computer science, we will provide the necessary references to help the understanding of the topics that could not be developed in each part. This also explains why the bibliography is important here. Since we are basing ourselves on the Windows operating system, the MSDN's documentation is our only reliable source of documentation. Then, it is necessary to refer to specific points in its wide documentation rather than referencing it globally. Doing it globally would prevent the reader from being able to identify the exact page from which a given statement is taken in our study.

Finally, at the end of each section and chapter, there is another box that summarizes the key points provided. The idea is to highlight the important contributions coming from us. Our contributions to what did not exist or our contribution to situations that were imperfectly understood before. It allows the reader to make the difference between what already existed and what our work brings. Finally and whenever possible, we take the liberty of describing the significance of our research in certain cases.

---

## 4.4 Publication proceedings

### Patents and industrial contributions

- Baptiste David, CVE-2018-8232, A Tampering vulnerability exists when Microsoft Macro Assembler improperly validates code, aka "Microsoft Macro Assembler Tampering Vulnerability.", This affects Microsoft Visual Studio, July 10th 2018, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-8232>.

### International peer-reviewed journals

- Venault, Mathilde and David, Baptiste, "Superfetch: the famous unknown spy", Journal of Computer Virology and Hacking Techniques, June 2021, 17. 1-14. 10.1007/s11416-020-00370-y.
- Baptiste David and Maxence Delong and Éric Filiol. "Detection of Crawler Traps: Formalization and Implementation Defeating Protection on Internet and on the TOR Network (extended version)". Journal of Computer Virology and Hacking Techniques, volume 17, issue 3, August 2020.
- François Plumerault and Baptiste David, "DBI, debuggers, VM: gotta catch them all". Journal of Computer Virology and Hacking Techniques, July 2020. <https://doi.org/10.1007/s11416-020-00371-x>
- François Plumerault and Baptiste David. "Exploiting flaws in Windbg: how to escape or fool debuggers from existing flaws", Doi: 10.1007/s11416-020-00347-x, Journal in Computer Virology and Hacking Techniques, volume 16, Issue 2, January 2020, pp. 173-183.
- Amir Afianian and Salman Niksefat and Babak Sadeghiyan and David Baptiste, "Malware Dynamic Analysis Evasion Techniques: A Survey", ACM Computing Surveys, Volume 52, Issue 6, November 2019, <https://doi.org/10.1145/3365001>
- Baptiste David and Éric Filiol and Kévin Gallienne. "Structural Analysis of Binary Executable Headers for Malware Detection Optimization". Journal in Computer Virology and Hacking Techniques, volume 13, Issue 2, April 2016, pp. 87-93.

### Books and chapters

- Éric Filiol and Baptiste David and Paul Irolla, "Les virus informatiques", Chapitre des Techniques de l'Ingénieur, H5440, October 2017 (extended and updated version from 2007).

### ArXiv publication

- Baptiste David, "How a simple bug in ML compiler could be exploited for backdoors?", November 27th, 2018, arXiv:1811.10851, <https://arxiv.org/abs/1811.10851>

### International conference with committee and proceedings

- Maxence Delong and Baptiste David and Éric Filiol, "Detection of Crawler Traps: Formalization and Implementation - Defeating Protection on Internet and on the TOR Network", 4th International Workshop on FORmal methods for Security Engineering - ForSE 2020, February 25-27th 2020, Valletta, Malta.
  - Maxence Delong and Éric Filiol and Baptiste David. "Investigation and surveillance on the Darknet: an architecture to reconcile legal aspects with technology". ECCWS 2019 18th European Conference on Cyber Warfare and Security, July 4-5th, 2019, Coimbra, Portugal.
  - Baptiste David and Éric Filiol and Kévin Gallienne and Olivier Ferrand, "Heuristic and Proactive IAT/EAT-based Detection Module of Unknown Malware", 15th European Conference on Cyber Warfare and Security (ECCWS) 2016, ACPI, pp. 84-93, Bundeswehr University, July 7-8th, 2016, Munich, Germany.
-



### International conference with committee

- Mathilde Venault and Baptiste David, "*Superfetch: Everything you need to know about privacy*", C0c0n 2020, September 18th 2020, Kochi, India.
- Mathilde Venault and Baptiste David, "*Fooling Windows through Superfetch*", Black hat USA 2020, August 2020, Las Vegas, USA.
- Baptiste David, "*Vulnerability in compiler leads to stealth backdoor in software (extended version)*", Zero night 2018 conference, November 20th, 2018, Saint-Petersburg, Russia, <https://2018.zeronights.ru/en/reports/vulnerability-in-compiler-leads-to-stealth-backdoor-in-software/>
- Baptiste David, "*Vulnerability in compiler leads to perfect stealth backdoor in software*", International Cyber Security and Policing Conference C0c0n, October 5-6th, 2018, Kochi, India. <https://is-ra.org/c0c0n/2018/speakers/agenda/>
- Baptiste David, "*Shall We Play a Game: How to Fool Antivirus Software*", 15th Nuit du Hack, June 2017, Eurodisney, Marne-la-Vallée, France.
- Baptiste David and Paul Amicelli, "*How to Secure the Keyboard Chain II*", C0c0n 2015 (CyOps Con), International Cyber Security and Policing Conference 2015, August 20th-21st, 2015, Kochi, India.
- Paul Amicelli and Baptiste David, "*How to Secure the Keyboard Chain*", DefCon conference 23, August 6-9th, 2015, Las-Vegas. <https://www.youtube.com/watch?v=W5B-zjaDzFU>

### Technical press articles

- Pierre-François MAILLARD and Armand ITO and Solène SPRENGER and Baptiste DAVID, ANALYSE UEFI AVEC WINDBG. Multi-System & Internet Security Cookbook (MISC), number 108, March 2020, pages 22-30.
- Pierre-François MAILLARD and Armand ITO and Solène SPRENGER and Baptiste DAVID, L'UEFI, PROGRAMMATION UEFI. Multi-System & Internet Security Cookbook (MISC), number 107, January 2020, pages 68-74.
- Pierre-François MAILLARD and Armand ITO and Solène SPRENGER and Baptiste DAVID, L'UEFI, AU CŒUR DU SYSTÈME. Multi-System & Internet Security Cookbook (MISC), number 105, November 2019, pages 68-75.

### Workshop and teaching activities during seminars

- David Baptiste and Pierre-François Maillard, "*Reverse engineering: Reverse engineering and ROP gadget, how to take the control of the system*", Cocon 12, Pre-conference workshops, 10-11 November, 2021, Kochi, India. [https://india.c0c0n.org/ws\\_11](https://india.c0c0n.org/ws_11)
  - David Baptiste, "*Reverse engineering: how to break (badly used) cryptography*", Cocon 12, Pre-conference workshops, 25-26th September, 2019, Kochi, India. [https://india.c0c0n.org/ws\\_11](https://india.c0c0n.org/ws_11)
  - Baptiste David and François Plumerault, "*Reverse engineering under Windows 10 with malware analysis*", Cocon 11, Pre-conference workshops, October 03-04th, 2018, Kochi, India. <https://is-ra.org/c0c0n/2018/workshop/pre-conference-workshop/>
-

THIS PAGE INTENTIONALLY LEFT BLANK

---



## Chapter 2

# Protection at development level: backdoor in compilers

### 1 Introduction

Secure development of software involves to master the entire software creation process and toolchain. Generally, security starts at the time codes are written. Of course, there are many sources for good practices in programming [46, 47]. To err is human. This is why there are a lot of tools [48, 49] able to provide an automatic analysis about security of the source code written (without executing it). Such tools are called *static code analysis* tool [50] and they all have qualities and drawbacks [51, 52, 53, 54] without being able to be perfect [55]. Technically speaking, static analysis is performed on the source code of the software, looking for specific patterns responsible for potential bugs which could lead to security issues. It is a kind of *development companion* able to find bugs automatically which are nowadays automatically integrated in software of *Integrated Development Environment* (IDE), such as the one of Visual Studio 2019 [56]. But the security provided by static analysis is not perfect and this is why the security cannot rely only on static analysis.

To improve the security, *dynamic code analysis* is possible. Such analysis is performed on a compiled executable file which is tested as execution time (hence the term "*dynamic*"). The analysis can be done under two different assumptions. The first is within a white box context with access to the source code of the tested application. The other assumption is without the source code. In such a case, we analyze the execution trace and the interactions with the environment. Note the use of *fuzzing technology* [57, 58, 59] in this case. At the opposite of static code analysis, this one does not necessarily need the source code since it only requires the executable files holding the software. There are numerous tools able to provide dynamic analysis [60] but most of them are based on *Dynamic Binary Instrumentation* (DBI) where instrumentation refers to the possibility to control, at execution time, different sections of the code executed. The main idea of such a tool lies in the possibility to decompile<sup>1</sup> original code before and during execution. The goal is to modify the code in order to inject specific code between execution of two sections of the original code. More details and further information can be found here [61].

Usually DBI frameworks are hard to develop [62]. This is why a lot of tools are based on a small number of existing DBI engines [60], customized for different purposes [63, 64, 65, 66]. Among the best known DBI framework [67], we have Pin from Intel [68], DynamoRIO [69], Valgrind [70], QEMU [71] and FRIDA [72]. We can also find other tools such as ANaConDA [73] and QBDI [74]. All of these tools have different performances [75], depending on the codes tested and architecture choices made by each of these DBI. Note that it is possible to have mixed approaches [62] and per programming language tools [76]. In our case, DBI can be used in conjunction of *fuzzers* [77] that generate input data to the software and where DBI framework provides the possibility to observe how the software reacts to the provided input.

---

<sup>1</sup>Decompiling an executable file can be seen as a way to get a *certain* version of the original source code. Of course, this version is complete but less easy to analyze. Note that if a code is executable, this one is always readable at least with the elementary instructions composing it by the CPU, giving us a sort of "source code".

Generally, a mix of static and dynamic analysis can be considered as enough to guarantee the security of developed software [78] even if it is not perfect [79]. Indeed, static analysis allows us to verify that there is no known dangerous pattern in the written code. Dynamic analysis allows us to test, empirically, that the program does not react badly to certain inputs given. These methodologies are generally integrated in the DevSecOps [80, 81] process. But, between writing the source code (which can be statically analyzed) and the executable program (which can be dynamically analyzed), there is one step that is not mentioned: compilation. This procedure is generally not audited and therein lies the rub. Indeed, debuggers are software as any others. It means they are written from source code, by humans, who sometimes, make mistakes. These mistakes result in bugs and unexpected behaviors which can be exploited to introduce a malicious code or a *backdoor*.

In this chapter, we are going to focus on the security of the process of compilation. To illustrate our research work, we propose to exploit a flaw in an existing debugger to insert an operational backdoor during the development a given software. Even if it has been discussed as theoretically possible before by different authors, this technical result has never been shown as technically feasible before us, at least publicly. After this general introduction, we are defining the notion of backdoor and we are presenting a state of the art in section 2 about different types of backdoor inserted during development. Then, we are going to focus in section 4 on MASM compiler which presented a flaw when compiling a specific code of macro assembler. This will allow us present in section 5 how to exploit this flaw when writing specific assembly code. Finally in section 6, we will present the patch provided by Microsoft to avoid MASM compiler to be exploited by our technique.

## 2 State of the art

### Resume 1:

- ☞ We propose a survey on different definitions about backdoor in compilers to finally propose that synthesizes all of them.
  - 👉 There is no universal definition of what a backdoor is (it depends on the operational context).
  - 👉 There is a difference between a *vulnerability* in a software (a bug that can be exploited for specific and potentially malicious purposes) and a *backdoor* (a mechanism that provides privileged access to an attacker).
- ☞ We present an historical and technical state-of-the-art of different backdoor techniques within compilers.
  - 👉 We propose to define 4 possible levels where to introduce a backdoor: source-code, compiler, installation and running time.
  - 👉 We examine advantages and disadvantages of each technique.
  - 👉 We show that in the case of compilers, there have never been really operational release.

### 2.1 Different instances of backdoor

#### Key Point 2.1:

- ☞ A "backdoor" does not have a universal and formal definition in literature.
  - 👉 But several terms are regularly used: obtain privileged access, intentionally hidden, unintended bugs, exploitable, deniable or unexpected or arbitrary computation...

We must be direct and recognize that the notion of *backdoor*<sup>2</sup> does not have a universal and formal definition. There is no consensus. A backdoor can be realized by many actors, which can be powerful as a state [82] or coming from industrial world such as device manufacturers [83, 84]. Overall, the purpose of a backdoor is to obtain any privileged access to a given system. A backdoor differs from a classic vulnerability that is exploited to achieve the same end by the developers' own willingness to leave a code voluntarily vulnerable (where a vulnerability can generally be considered involuntary from the developer). It is the intentional dimension that makes the difference.

It must be acknowledged that the blatant lack of real-world backdoor examples is not to help to define precisely this type of object. As explained in [85], documented real-world backdoors are generally simplistic, relying on intentionally hard-coded credentials (a password composed of static data or hard-coded credentials is compared with `strcmp` function) [86], intentionally hidden authentication interface<sup>3</sup> [84], or "unintended" bugs easily exploitable [87, 88]. More example about such real world backdoors can be found in [89, 90].

In the academic literature, despite their relative scarcity, there are several definitions of what a *backdoor* could be. The work from Thomas and Francillon [85] summarizes all that could be defined as relevant. In [91], authors aims to show how to design a deniable *bugdoors* for microcontroller firmware. Their goal is to modify an interrupt in TinyOS system to make its handler function vulnerable to a given attack. To increase the stealthiness of their action, they are manipulating run-time state and by using several exploits which can be chained from the interrupt, they are going to perform arbitrary computations.

Another solution is to design a trigger-based malware to remain undetected over the long term [92]. And to prevent detection, authors are hiding their code by embedding it in unaligned instructions. Indeed, Intel

<sup>2</sup>And not a *trap door* which is specific to cryptography.

<sup>3</sup>Examples could be found on: <https://pwnies.com/previous/2016/best-backdoor/> or <https://pwnies.com/previous/2017/best-backdoor/>.

x86 assembly supports variable-length instruction sets. It means that assembly instructions can be encoded on different a variable number of bytes, which makes starting point of code highly relevant. That way, disassembly of codes depends where the analysis starts in memory. Shifted from one bytes, the disassembly shows a totally different code. With this stealth technique, using a crafted trigger bugs to implement covert control transfers to this code is enough to finish the backdoor.

In [93], authors compromise the firmware of a commercial off-the-shelf hard-drive to design a stealth rootkit that replaces arbitrary blocks from the disk while they are written, providing a *data replacement backdoor*. With a performance overhead less than 1 %, authors shows that it is possible to establish a communication channel with the *backdoored* disk in order to infiltrate commands and to ex-filtrate data. Then it is possible to link this communication channel with others channel (such as internet connection) to allow a full remote and easy access to the malicious system. Such design is close to some rootkit technologies, for instance one based on PCI exploitation [94].

Still about technical backdoor mechanisms, [85] authors talk about "NOBUS" (i.e., *NObody But US*) vulnerabilities by the NSA [95] and those associated with APT actors [96] which belong outside of the literature... These are usually advanced solutions implemented by highly competent states or criminal groups. The objectives are diverse and it is difficult to attribute authorship.

Another approach, more academic, is to see the backdoor as a system of *weird machines* and *finite-state machines*. Both [97] and [98] present ways of affecting the flow of computation in order to find new means to perform and to drive unexpected or arbitrary computation. For short, they define models to force *normal systems* to execute programs written in those models as a mean to implement backdoor-like functionality. In [99], Dullien Thomas provide many clear definition and formal definitions that help to better understand the concepts of *exploit*, *wired machine*, and how programming of a weird machine leads to exploitation. According to the authors of [85], much of [99] serves as inspiration for their work.

## 2.2 Definition of a software backdoor

### Resume 2:

✎ We propose our own definition based on previous literature work and operational requirements.

From these various approaches, it is possible to start building a definition of what a backdoor can be. To proceed, we will start from the approach given in [85]. In this last paper, authors first present the notion of backdoor as given in [100]. For them, "*a backdoor is a mechanism surreptitiously introduced into a computer system to facilitate unauthorized access to the system*". This definition is interesting because it describes the expected goal (privileged access - the "why") but it says nothing about the means to achieve it (the "how"). In [101], a high level classification of backdoors is provided to classify backdoors in three groups: *system backdoors* (compromise the underlying operating system), *application backdoor* (legitimate software modified to bypass security mechanisms under certain conditions) and *crypto backdoors* (intentionally designed weaknesses in a cryptosystem for particular keys or clear-text messages access). If the difference between *system backdoors* and *application backdoor* may appear a bit artificial (one can be used in the case of the other and vice versa), the distinction can be seen as the difference between conceptual (crypto, mathematical) and technical (computer, programmed) backdoors.

In a synthetic way, [85] presents a definition which is intended to be at the junction of the various papers previously mentioned. In an approach that is intended to be very formal (and which goes beyond the scope of our chapter, since the authors of [85] are determined to propose detection models), we can try to summarize their approach by retaining four criteria necessary for a backdoor. To be a defined as a backdoor, as code must be composed of:

- A pivotal component able to active the backdoor: its *trigger mechanism* ;
- To activate the trigger, a backdoor must present a type of *input source* ;

- Since the goal is to switch from *normal system state* to the *backdoor-activated state*, a code is responsible to handle the backdoor behavior: the *backdoor payload* ;
- The goal of the payload is to get an escalated privileges, privilege abuse or unauthenticated access, i.e., a *privileged state*.

That way, it is possible to define a backdoor. The definition given in [85] as been rewritten to avoid the specific vocabulary driven by the analysis of state machines, specific to the research of authors and beyond the scope of ours.

An intentional construct contained within a system that serves to compromise its expected security by facilitating access to otherwise privileged functionality or information. Its implementation is identifiable by its decomposition into four components: *input source*, *trigger*, *payload*, and *privileged state*, and the intention of that implementation is reflected in its complete or partial (e.g., in the case of bug-based backdoors) presence within expected system, but not the observed system containing it.

This definition is interesting because it highlights essential aspects that characterize a backdoor. However, it does not focus on important aspects of a backdoor, especially in modern attacks, nowadays. Especially, we can talk about the secrecy that surrounds the backdoor and the possibility of denying the real involvement of the perpetrator. In addition, operationally, a backdoor does not need to be triggerable. The example of [93] where the hard disk is trapped shows the perpetual activity of the malicious mechanism, even if it remains also possible to control it on command. This is why, perhaps, such a definition could be further improved with these notions.

In our case, we are going to focus on inserting such a malicious feature in a software at development time. It means that we suppose that it is possible for the attacker to manipulate directly the source code of the targeted software. We also suppose that the source code can be audited by a perfectly capable human or software. It means, for our attacker, that this one has to hide the real purpose of the code he is writing. Hiding from the eyes of any auditor but not from the *eyes* of the computer which is about to execute its software.

According to our context and for the sake of simplicity, we propose to keep as definition of a backdoor the following definition.

In a software, a backdoor is a feature, unknown from the original user, introduced deliberately in order to provide over a long period of time a third party access to a privileged part in a given software.

The goal is to allow the designer of the backdoor to get an access which is not supposed to hold. It can be performed through the access to a resource used by the software (data, devices, etc). Or it can be done via the ability to deactivate the software remotely (denial of service). Finally, it could change silently the behavior of this one in an unexpected way (code update). In the case of a software backdoor, one can draw the four the main objectives of a backdoor to add to the original definition.

1. Be persistence: Must survive to reboot and re-installation.
2. Be secret: Once it is known, this one is no more usable.
3. Be deniable: Allow the attacker to pretend it is not his fault (possibility to legitimately apologize with "it's a bug").
4. Be stealth: Hard to detect at analysis time and hard to detect at running (exploitation) time.

Historically, in 1974, the US Air Force published [102] the results of a vulnerability analysis of Multics<sup>4</sup> operating system which was consider as the most secure one at that time. In the section 3.4.5.1 of USAF's

---

<sup>4</sup><https://multicians.org/history.html>

document, authors are defining different classes of backdoors as different possibilities to insert a backdoor in a system. Technically, they are considering four possibilities to insert a backdoor during the implementation of a software during the life-cycle of that one. The life-cycle of a software is as given in the following list.

1. Writing the source code ;
2. Compiling the source code ;
3. Broadcast and download the compiled software ;
4. Installing the compiled software.

Even if it is not present (because it would not be possible to do it such a way in the past), it might also be interesting to add in-memory execution techniques to manipulate the operation of targeted software [103]. In the idea, one executable running code modifies another in memory before the latter is executed. Thus, at execution, the subsequent actions performed differ from those originally programmed.

As part of our state-of-the-art, we are going to focus on the two first steps. The two last steps can be covered quicker [102] since they are not directly part of the development (and out of our scope). But for the sake of completeness, we can say that the introduction of backdoor at broadcast or downloads phase requires to modify the software between the server where the software (or its installer) lies and the computer of the victim. This can be done by controlling intermediate router where the software transit or by obtaining the possibility to redirect the distribution channel through which the software passes. Of course, in case of use of secure technologies for file transfer (HTTPS, SSL, TLS, IPSec, etc.), it is mandatory to bypass the security provided by them. Note that this principle can be extended to software update, especially if the HTTPS connection is not checked with certificate pinning techniques [104].

The last case supposes that the system of the victim is already compromised. During installation of the software, the corrupted system recognizes the software to introduce, at running time, a backdoor in the installed software. This action can be performed once for all during installation where executable files are directly modified on the disk. Or it can be performed at execution time (in memory) each time the software is about to be running. In such a case, the system injects code directly in the memory of the targeted process. In this chapter, we focus on the second case only, during the compiling step.

## 2.3 Backdoor at development time

A backdoor can be inserted at development time using different techniques. We are going to cover many of them, keeping the historical point of view as a guideline. Technically, it is hard to point the first backdoor in software world (it could be Easter eggs<sup>5</sup>) and it is out of scope of this manuscript. To keep things simple we are going to cover some of the most well-known types of backdoor at development level. The goal is not to be exhaustive but to illustrate the pros and cons of each example we present.

### 2.3.1 Source code backdoor

#### Key Point 2.2:

- ☞ It is possible to insert a backdoor in a software during its development.
  - ☞ This approach is done directly within the source code of the targeted software.
  - ☞ The operation can be done with closed or open source code.
- ☞ Historically, there are some public examples of such situations.

---

<sup>5</sup>Easter eggs are not really a backdoor but rather a joke or an unexpected feature. But even though the goal is different, the technical design is similar to a backdoor.

The first type of backdoors presented are may be the most easiest one to implement. It is about inserting a trap door at source code level. It means an attacker is modifying or writing a source code to insert a vulnerability in the software. Famous real examples exist about such situation. The case of the database Interbase [105] which had in its source code a shadow user (username: "politically" with password: "correct") from 1994 to 2001, when the source code of the Interbase project had been released by its editor Borland. The fact that the project was close-source at that time and the lack of control allowed such a vulnerability to remain for years.

Another famous case lies in the NSAKEY plot [106] which appears as a symbol leak from Windows NT 4.0 operating system. If Microsoft's operating system remains close-source, it is possible to retrieve symbols from the compiled code for debug purposes [107]. In a version of these symbols, it appears a value called "\_NSAKEY" which has given rise to quite a few fantasies. Some explanations have been given by Microsoft [108] and experts [109]. Whatever is the reality of that cold case, this plot illustrates that leaks of information are still possible even if the source code is not available.

The most famous source code backdoor is maybe the attempt tried on the source code of the Linux's kernel in 2003 [110]. Indeed, in the file kernel/exit.c, an attempt to change the condition "current->uid == 0" to an assignation "current->uid = 0" was enough to allow a call of the function sys\_wait4() with specific arguments to gain root access. If the trick was nice, it remains it has been promptly detected by the developers team. It is the direct application of the Linux community's source code control that validates the modifications. An illustration of mail exchanges after detection of the backdoor is illustrating in figure 2.1. Note that this attempt would never had been able to corrupt the mainline kernel, since the repository where the code has been modified may not be the one used by the kernel at that time [110].

[\[prev in list\]](#) [\[next in list\]](#) [\[prev in thread\]](#) [\[next in thread\]](#)

List: [linux-kernel](#)  
 Subject: [RE: BK2CVS problem](#)  
 From: ["Chad Kitching" <CKitching \(\) powerlandcomputers ! com>](#)  
 Date: [2003-11-05 22:48:09](#)  
[\[Download message RAW\]](#)

```
From: Zwane Mwaikambo
> > +      if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
> > +          retval = -EINVAL;
>
> That looks odd
>
```

Setting current->uid to zero when options \_\_WCLONE and \_\_WALL are set? The retval is dead code because of the next line, but it looks like (an attempt) to backdoor the kernel, does it not?

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@vger.kernel.org

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>

[\[prev in list\]](#) [\[next in list\]](#) [\[prev in thread\]](#) [\[next in thread\]](#)

Figure 2.1: Detection of the backdoor by the Linux kernel team.

One last example is about Cisco Unified Videoconferencing product. In 2010, Florent Daigniere discovers many security vulnerabilities in this CISCO project [111], allowing remote access in FTP or SSH by pre-registered user accounts which cannot be modified or removed. Similar in consequences from the backdoor in Interbase, the backdoor lies in the configuration of the Unix system embedded on CISCO products and not directly in the



source code modification. Sometimes, modifying the environment or configuration files is enough to introduce a backdoor.

Of course, many other examples could be presented here. Generally speaking, all critical flaws in software could be interpreted as potential backdoors. But it is forgetting that to be qualified as backdoor, it is still necessary to be able to prove the real will of the developer to leave or to introduce a given flaw. From the famous rule of thumb that states *“never attribute to malice that which is adequately explained by stupidity”*, a lot of flaws could fall under this statement. A good example of *bad bugs which could be interpreted as potential backdoor* may lie in the context of the *Pwnie Awards* ceremony<sup>6</sup>. Taking place during the Blackhat USA conference, this is an annual awards ceremony celebrating the achievements and failures of security researchers and the security community. Listing all the rewarded failures<sup>7</sup>, there is a special award for the *“Lamest Vendor Response”*. Some of them, by the delay<sup>8</sup> [113] to fix a bug or by the bad faith<sup>9</sup> of the software developers could be interpreted as potential backdoor if unfortunately incompetence was not a sadly more credible option.

Recently, cyber-criminals tried to corrupt an open-source library used by a crypto-money application to steal customers’ Bitcoins [114]. Technically, the BitPay company uses a third-party NodeJS package used in its Copay and BitPay applications. This NodeJS package had been modified to load malicious code which could be used to capture users’ private keys. The problem has of course been detected and fixed. But this shows the ingenuity of the attackers, who do not necessarily attack companies’ projects (to which they do not have access since they are only modified by companies) but to their projects’ dependencies. It also shows the need to take a broad look at the notion of secure programming for a given project.

But it is possible to draw the main consequences of a try to backdoor a software by directly manipulating its source code. The first lies in the observation of how easy it is to modify the source code directly. In case of lack of control or with the help of complicity, it is possible to introduce backdoor in software. But, it requires to take care of side effects and soon or later, the backdoor will be discovered. Anyone not in the conspiracy reading the source code or by a release in future time is enough to break secrecy. Then, taking into account open-source world, it becomes even more complex to hide a code modification which would result in a vulnerability to the eyes of experimented developers.

### 2.3.2 Historical compilers backdoor

#### Key Point 2.3:

- ☞ A compiler backdoor aims to introduce a flaw automatically and silently in a software during the compilation phase.
  - 👉 Historically, US Air Force is the first the publish report [102] about such a threat — but the report was confidential.
  - 👉 Ken Thompson is the author of the first publicly published paper [115] about such a threat.
  - 👉 Ken Thompson presupposes that a compiler has already been trapped and used by the victim. This is a strong assumption from an operational point of view.
  - 👉 In both cases, the threat is quite theoretical and mentioned as a possibility.

We must explain first what means a backdoor in a compiler. Technically speaking and as explained before at length, a backdoor is a mechanism which allows to bypass the security mechanisms in a software. Since the compiler is not a security component in itself, it does not make sense to try to introduce a backdoor in the

<sup>6</sup><https://pwnies.com/about/>

<sup>7</sup><https://pwnies.com/previous/>

<sup>8</sup>According to [112], in 2005, three vulnerabilities were discovered in *gmail* but were never fixed because they were believed to be unexploitable in a default installation. In 2020, Qualys Security team re-discovered these vulnerabilities and was able to exploit one of them remotely in a default installation. <https://pwnies.com/previous/2020/lamest-vendor-response/>

<sup>9</sup>Western Digital’s MyPassword Drive user a cipher system to protect data stored on the driver. The cipher keys used are actually just redundant copies of a 32bit rand value repeated over and over, making the keys impossible ... to lose <https://pwnies.com/previous/2016/lamest-vendor-response/>



compiler. But it makes sense to trap the compiler so that it is going to introduce automatically a backdoor in the software it is about to compile. By abuse of language, we might talk about backdoor in compiler to refer to the fact that the compiler is trapped in order to trap the software it builds.

Compiler backdoor is the first time publicly described in the document coming from the USAF [102] about the security of Multics operating system. It starts from the observation that the system could be compromised by the provider of the operating system if any patches would be applied to binary object files of the system. The countermeasure proposed was to recompile the system from source code, since that one was owned by the Pentagon. However, if there is a backdoor in the compiler of the system (which was at the time a PL/I compiler for Multics operating system), it means a backdoor could have been maintained in the system despite compilation of a secure and audited source code. Even worst, even if PL/I compiler source code was available to the US department of defense, to compile it a first time, it would have required a compiled version of the compiler. This one would possibly include a backdoor and the ability to maintain its own existence by recompiling itself while keeping the original backdoor. Thirty years later, authors of the USAF report published a new paper [116] where they reported lessons from that evaluation. According to the authors, observations from compilers security would lead to the *Trusted Computer System Evaluation Criteria* (TCSEC) [117] Class A1 requirement for "generation of new versions from source using a compiler maintained under strict configuration control".

What makes the compiler backdoor famous is the paper of Ken Thompson [115] who took his inspiration from the rapport written by USAF. At that time, Thompson was not able to correctly cite that document, but according to [116], he updated his paper [118] once original authors from USAF provided him the real references. Thompson's idea is simply to put into practice what he read. From a given C compiler, he modified an existing pattern from the C parser to produce another compiled code than the expected one. The figure 2.2 is a recomposition from his original paper [115].

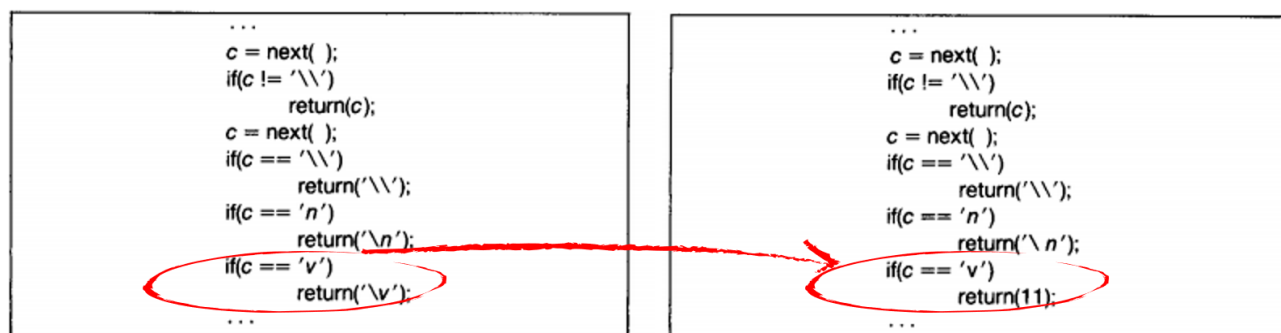


Figure 2.2: Modification of compiler by Ken Thompson.

From the possibility of modifying the output of the compiler when a given pattern matches in the provided source code, it is possible for Thompson to introduce what he called a "bug" each time such situation occurs. This is illustrated in figure 2.3 extracted from Thompson's original paper.

The figures provided are a bit different from explanations provided by Thompson in its paper. Indeed, he considers what he calls a bug if it has been introduced not deliberately, since it is deliberate in his case, it should be called a "Trojan horse". Then, Thompson claimed that he modified the Unix C compiler to inject his Trojan horse to a program. His goal was to modify the program responsible to manage user's login to secretly give him root access. In addition, he talked about the possibility to add a new pattern to trap the compiler itself. Since compilers are compiled from compilers' generated executable, he proposed to compile a modified version of the compiler with a non-trapped compiler. The compiled compiler is supposed to have a recon pattern able to match in its own source code. In such a case, compilation of the compiler from its source code (removed from the inserted bug) with the backdoored compiled version of the compiler will perpetuate insertion of the backdoor in the newly compiled compiler. Undetectable, even if the source code of the compiler is open. On sentence from his conclusion is: "No amount of source-level verification or security will protect you from using untrusted code".

```
compile(s)
char *s;
{
    if(match(s, "pattern1")) {
        compile ("bug1");
        return;
    }
    if(match(s, "pattern 2")) {
        compile ("bug 2");
        return;
    }
    ...
}
```

Figure 2.3: Insertion of a bug each time a pattern matches in the source code.

One legitimate criticism can be done on Thompson’s paper. The author announces a lot of results and exploits but he does not explain how he gets them technically. The figures are naive and they do not explain how to reproduce the results. While legitimate security considerations may be taken into account to explain this lack of details, this paper should also be seen as a theoretical presentation of what could be done when it could be potentially applied to real-life cases.

David Wheeler in his PhD thesis [119] is giving the answer about the credibility of Thompson’s paper. In a conversation on a public forum, Jonathan Thornburg [120] explains by publishing extracts of mail exchanges the reality of Thompson’s work. Even if Thompson wrote a compiler which was able to compile backdoored software, this compiler never left Bell Labs. Thompson’s exploit has always been limited by a laboratory environment and it has never been used operationally, at least in what Wheeler says.

If Thompson’s idea is a good one, it is not less complex to put it into practice. To do so, the victim must use a compiler that has already been compromised. There is nothing obvious about distributing a modified compiler. Moreover, given the rather sensitive nature of this type of software, it’s inconceivable that the identity of the distributor would be not known. This reduces the capacity for denial in case of discovery. Finally, directly modifying compiler to introduce a backdoor might seem to be an operational dead end.

### 2.3.3 Modern compilers backdoor

#### Key Point 2.4:

- ☞ We introduce the notion of "*modern compilers backdoor*" to make the difference with the historical ones (Key-Point 2.3).
  - 👉 Historical backdoor in compilers assume that a compiler has already been trapped and that the user uses it already.
  - 👉 This is equivalent to install a malware (i.e. the *backdoored* compiler) on the user's machine.
  - 👉 A malware could already modify generated executables from a developer's machine without needing to infect the compiler.
  - 👉 A more operational (and modern) version aims at using a compiler as it is.
- ☞ Modern compiler backdoor uses a vulnerability already present in the compiler used by everyone to introduce a backdoor in a compiled software.
  - 👉 Compilers are software like any other: they have bugs too.
  - 👉 This presupposes to find a vulnerability that can be exploited in the compiler.
  - 👉 The vulnerability allows to *miscompile* some very specific lines of code and thus to introduce a malicious behavior.
- ☞ If few examples that have seldom been published, they are far from being fully operational.
  - 👉 They use bugs already fixed in the compilers.
  - 👉 They often require writing abstruse source code to achieve their goals.

If it is not desirable to modify a compiler to deliberately introduce a flaw, perhaps it is possible to exploit an already existing one in the compiler. As surprising as it may seem, compilers are not perfect and they can have bugs. In the mind of most developers, the compiler must not perform any error since these ones are very complicated to find. Indeed, when a bug occurs, the first reflex is about to search any error from source code and not blaming the compiler. Source code defines the logic the created program is supposed to follow. And error coming from a correct source code which would result in an unexpected behavior could be very hard to find. Compilers are supposed to be reliable since software logic depends on them.

But compilers are just regular software which implies they can hold bugs. But compiler bugs are semi-mythical, not because they do not exist but because they are hard to find [121]. There are different types of bugs for a compiler. The firsts are those about to make crash the compiler. They are not relevant for us since they do not produce any executable file. The second type is about to force the compiler to *silently miscompile a program*. As explained in [122], such behavior can result in an incorrect execution and even security vulnerabilities in the miscompiled program. Bugs from compilers are supposed hard to detect since they can only be triggered under specific circumstances, they may go unnoticed during software development and they surface only after deployment.

Bugs in compilers are documented [121, 123, 124] and there are even statistics about them [125]. Most of the time, bugs come from aggressive compiler optimization or bad assumptions they make to achieve them [123]. According to [126], bugs come from optimization safety checks which are inadequate [124], static analyses which are unsound, or transformations which are flawed. This can be due to strange conditions written or side effects of programming language incorrectly managed. To find codes which are able to produce miscompilations, it is possible to write fuzzer tools, such as Csmith [126]. This fuzzer found hundreds of bugs in production C compilers such as GCC and LLVM, including ones which had been corrected with a top priority by compilers' authors. Also, there are fuzzers for different programming languages, such as jsfunfuzz [127] which is popular JavaScript fuzzer. According to the creators of Csmith [126], the bugs they are tracking are out of reach for current and future automated program-verification tools because the specifications that need to be checked were

never written down in a precise way, if they were written down at all. The design of Csmith lies on automatically generating randomized C program and then performs a test harness to check the C code produced is valid before compiling the program using several compilers, each providing an executable file. Then, Csmith runs the different executable files and it compares the outputs. Knowing the source code is the same for all compilers, if there is a difference in the output, it means that a compiler does not generate the expected code. The process used by Csmith can be resumed in figure 2.4 extracted from [126].

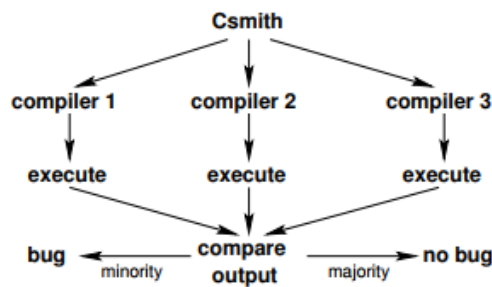


Figure 2.4: Csmith’s finding bugs procedure.

Csmith is not the first tool but the evolution of an already existing tools [128, 129, 130, 131, 132, 133] which has been driven by adding the possibility to contain complex code using many C language features while ensuring that the expected output of generated code is expected to be unique. From the Csmith project, other developments have also emerged, such as Orion [122] known as *Athena tool*. Based on a method called *Equivalence Modulo Inputs* (EMI) [134], the objectives was to use *Markov Chain Monte Carlo* (MCMC) techniques with the help of effective samples source code projects to allow a generation of various programs. According to authors’ paper [122], their results show that this approach is very effective in finding deep bugs that require long sequences of sophisticated mutations on the seed program. Athena tool found 72 new bugs in GCC and LLVM with most of them fixed, a large proportion with a top priority. Research keeps going on that field with recent publications [135].

A relevant point is to get the ability to generate bugs in compilers but to keep them able to produce code without neither any error code nor warning. It means that the compiler silently miscompiled the given program. This is what [126] calls a “*silent wrong-code error*”. Note that there are studies about inaccurate warnings and lacks of warnings generated from compilers [136]. It is precisely this type of silent bug, which produces a valid code without warning that is relevant for us, especially about designing a backdoor. Indeed, such bugs able to silently generate wrong-code is perfect to be introduced in the source code at development time. This is that idea which drove the work of Bauer & Co in [137] and at a lesser degree [138] (which is quite specific).

Bauer [137] decided to reuse the idea of Ken Thompson [115] (however without ever citing him) to modify the program responsible to manage login of users so that an error in a compiler will guarantee a privilege escalation bug. The targeted program was *sudo* version 1.8.13 on Unix system. To proceed, he used the bug number 15940 in Clang/LLVM 3.3, released in June 2013 thank to a fuzzing tool used by Ishiura Lab Compiler Team [139]. The bug is as follows:

```

1 int x = 1;
2 int main (void) {
3   if ( 5 % ( 3 * x ) + 2 != 4 )
4     __builtin_abort();
5   return 0;
6 }

```

Code:

If we follow the logic written in that C code, the output’s program is supposed to exit normally and to return zero. But compiled with the bugged version of LLVM compiler, it aborts, meaning it enters in the condition, which should not happen. No warning is given at compilation time as explained in the bug’s report [139]. From that observation, Bauer proposes to patch *sudo* source code so that when it is compiled using Clang/LLVM 3.3,

the sudoers file is bypassed and any user can become root. His code is inserted at `plugins/sudoers/parse.c:220`. The main issue he is facing is that the code able to trigger the compiler's bug is anything but discreet. Who would really write such a confuse code? Even if author hopes it will not pass a real code review, he proposes a strategy to try to proceed whatsoever. Applying different patches in different part of the project during a patient campaign over time. A resume of the strategy of code proposed by Bauer [137] is provided in figure 2.5.

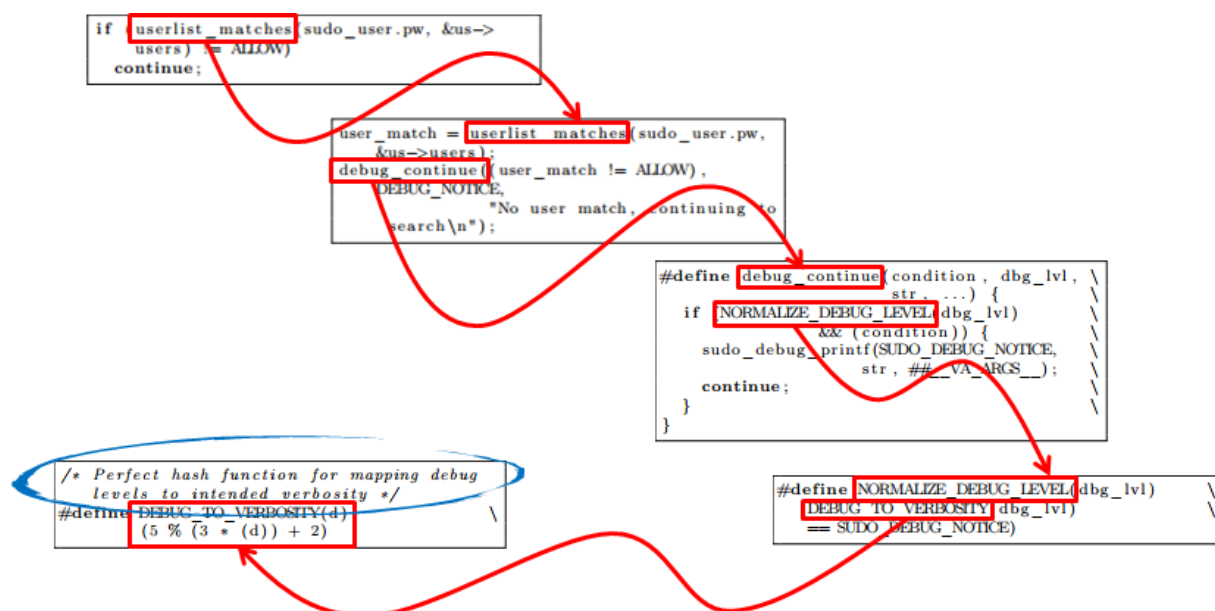


Figure 2.5: Resume of Bauer's strategy to patch sudo program in order to insert the backdoor thank to LLVM compiler.

Bauer claims that if his plan is correctly executed, it *"would surely succeed sometimes"*. It should be noted that there are no further arguments to support Bauer's proposal. Even if it is possible to *"hide"* few lines of code within a large project, the fact remains that the incriminated lines are still indeed there. In addition, the question arises to how the code lines are inserted.

Even if the author tries to be convincing, there may be a matter of doubt about the proposed logic... Who can believe that deliberately obfuscating code in an open source project, especially on a critical security part of the code, has any chance of being accepted by any reviewer? And even worse, who would believe that a dubious comment (circled in blue in figure 2.5) could be enough to dull the vigilance of experts who usually deal with such a program? Who can legitimize the insertion of code defined by several obfuscated macros in a code that was basically clear and unambiguous? And how can we deny or plead error in good faith if the scam is one day discovered? It will be difficult to deny the facts after having put so much energy into hiding the trick. And as explained in section 2.2, a backdoor must be deniable as a last resort. The problem is to use a bug — certainly exploitable — but which cannot be integrated into a code under at least a plausible reason.

This observation rises an interesting question about the efficiency and the operability of compilers' bugs found with the help of fuzzer techniques. In 2019, an article from Marcozzi & Co. [140] explains that if fuzzing tools have been proven to find hundreds of errors in most widely-used compilers such as GCC and LLVM [141], little attention has been given to know if the bugs found by fuzzing tools are present in real-world applications.

From a sample of compilers' bugs, they showed that almost half of the bugs could have an impact on generated executable files from real-world applications. But the impact is only limited to a small number of functions which lead to a small number of test suite failures. In addition, sources of bugs reported to compilers' authors are shared between those coming from fuzzing tools and from real developers since they are impacted during development. Such observation mitigates the impact about the number of bugs found in studies presenting results from fuzzing campaigns.

Finally, they explain that consequences of compilers' bugs have no semantic impact or that they would require very specific runtime circumstances to trigger an execution divergence. More directly, the fact that the bugs identified by fuzzers only concern a marginal number of cases in real-world applications since major bugs — probably written by real developers — are quickly reported to the compilers' authors, this means that there are few really exploitable bugs left. Moreover, the exploitable ones often require such special conditions during execution that it is very difficult to trigger them (which is why they might have escaped the attention of application authors). Not to mention the fact that these specific conditions require *dubious tricks* to insert vulnerable instructions into the source code [137].

This leaves us with the conclusion that if fuzzing tools are useful to find existing bugs in compilers, they are nowadays used by some researchers to insert backdoors at compilation time in software. Recent works about compiler fuzzing is the main evidence of this. But this method is not perfect and far from being operational. On the first hand, it requires a bug which is exploitable, which means to deal only with bugs resulting in silent wrong-code generation. Such class of bugs is far from being common. On the other hand, even if such bugs exist, they are far from being naturally present in real-world software and hard to exploit at runtime since they are present in marginal part of applications, as [140] explained it.

### 2.3.4 Wrapping everything up

#### Resume 3:

☞ In this sub-section, we have taken up all the elements about compilers backdoor to define a gap present in the research today.

✍ In practice, this means using an 0-day vulnerability in a compiler in order to silently and efficiently introduce a backdoor in a compiled software.

The present state of the art, without trying to be perfectly exhaustive, gives us the main developments and ideas underlying the insertion of backdoor when writing or compiling source code. If the case of insertion during writing was quickly dismissed as quite difficult to be correctly achieved in practice nowadays, especially to insert a backdoor for long-term purpose, the one about the compilation brings higher hopes. From the USAF report's predictions to ideas popularized by Thompson, a path has been opened. The emergence of fuzzing techniques is a blatant illustration of this. Even if fuzzing techniques are initially focused on the objective of correcting the bugs they found, some researchers are beginning to present works oriented towards attacking and exploiting these bugs to introduce backdoors.

However, there are always some capacity shortages. The first is that the proposed solutions are far from being operational. Ken Thompson supposed the victim already uses a backdoored compiler which is far from being obvious, except considering that we can control the broadcast of such backdoored software. But in such a case, it is equivalent to directly executing code on the victim's machine. With such assumption, it is possible to do much better than just exploiting a backdoored compiler... More directly, this hypothesis violates one the fundamental law in security which says: *"if we can persuade you to run something, it is not your machine anymore"* [142]. It is not surprising that the security cannot be maintained under such conditions. This is why using an already existing flaw in a compiler is more desirable. But then again, there are a few subtleties. Notwithstanding the fact that not all existing bugs are so easily exploitable, this also implies that we have got unknown bugs from developers' compiler. Otherwise bugs could be patched and not exploitable anymore. As a consequence, this means that the approaches about using already known bugs are irrelevant. And, all the researches about compilers' backdoors use known and already fixed bugs. We can always assume that the victim is using a compiler that is not up to date but it is an assumption that goes beyond of the scope of an operational context.

Finally, all the attacks presented here have one thing in common: they never got beyond the stage of an idea on paper or a proof of concept locked up in a laboratory. And it is finally from this observation that our work starts. The objective is to achieve a backdoor that maximizes the benefits of the various techniques while bypassing the operational limits observed to date. More directly, we are looking to create a backdoor, as defined in section 2.2, with the following properties:

- Exploit a bug from a compiler to insert a backdoor ;
- The bug must be unknown (0-day vulnerability) ;
- It must be from the class of bugs which silently generates wrong-code ;
- It must be usable for quite a long time (not being too obvious or too suspiciously obfuscated) ;
- It must be possible to legitimately justify the code inserted in the source code to generate the bug.

### 3 First approach

#### Resume 4:

- ☞ We will show in this subsection that it is possible to exploit a flaw in any C compiler to produce a different behavior.
  - ✍ We use a lack of precision in the C language standard to exploit the free interpretation left to the different compilers.
  - ✍ It is possible to find code constructions in C that allow to produce, for the same code compiled with two different compilers, two different results.

One approach to have a flaw in a debugger which can survive for a long time is to exploit not a bug but a lack of explanation in the programming language. A solution is to abuse of existing undefined behavior in compilers. For instance, with C language, it is not hard to find examples of such constructions [143, 144]. To take an example, one might wonder what would be the value of  $i = 0$  after executing that line of code:

```
i = i++ + 1; // Undefined behavior.
```

Code 2.1: Example of undefined behavior with C language when compiled.

Of course, the previous code is not important enough to introduce a backdoor. Let us consider a more important one in the following code:

```
1 int i = 0, j = 0;
2 char *tab = NULL;
3
4 tab = (char *)calloc(10, sizeof(char));
5 if (tab == NULL) {
6     __abort_operation();
7     return;
8 }
9
10 // Undefined operation.
11 tab[++i] = ++i;
12
13 printf("Value of i: 0n%02d.\n", i);
14 for (j = 0; j < 10; j++) {
15     printf("\t0x%02x -> 0x%02x.\n", j, tab[j]);
16 }
```

Code:

We can imagine that `tab` value is used to drive execution of another array full of pointers of code. For the sake of simplicity, we only kept the relevant part of the code. In such a case, the function pointer at index  $i$  is executed if and only if the  $i$ -th value is initialized in the `tab` array. We propose to compile this code with two different compilers. The first is the one coming from Microsoft Visual Studio [145, 146] and the second one is GCC<sup>10</sup>. Compilation happens without any warning and the result of execution is given in table 2.1.

The first observation is that  $i$  is equal to 2 in both cases. This makes sense since at the end, two incremental operations have been performed on that value. The difference lies in the content of the array `tab`. In the case

<sup>10</sup><https://gcc.gnu.org/>



<pre>gcc version 4.9.2 gcc main.c -o main.exe main.exe [i] Value of i : 0n02. 0x00   0x00 0x01   0x02 0x02   0x00 0x03   0x00 0x04   0x00</pre> <p>Code compiled with GCC</p>	<pre>[i] Value of i : 0n02. 0x00   0x00 0x01   0x00 0x02   0x02 0x03   0x00 0x04   0x00</pre> <p>Code compiled with Visual Studio 2018</p>
---	--

Table 2.1: Results of compilation of an undefined code with two different compilers.

of GCC, the value at index 1 is equal to 2. And in the case of Visual Studio, the value at index 2 is equal to 2. Explanation of the difference comes from the way the two compilers are interpreting the order of execution for the given instructions.

If we examine the compiled code generated from Visual Studio code with IDA<sup>11</sup> software, we show in figure 2.6 that the procedure has been divided in three parts. The first part is about allocation with the call to the `calloc` function. The return value (the address of the allocated buffer) is returned through RAX register. This one is tested to check the allocation succeeded. Then, in the right block of code designating the case where the allocated would have been a success, a direct access is done in memory to store at offset 2 the value equals to 2 with `mov byte ptr [rax+2], 2`. This is due to an optimization at compilation time which pre-compute the expected value when dealing with constants.

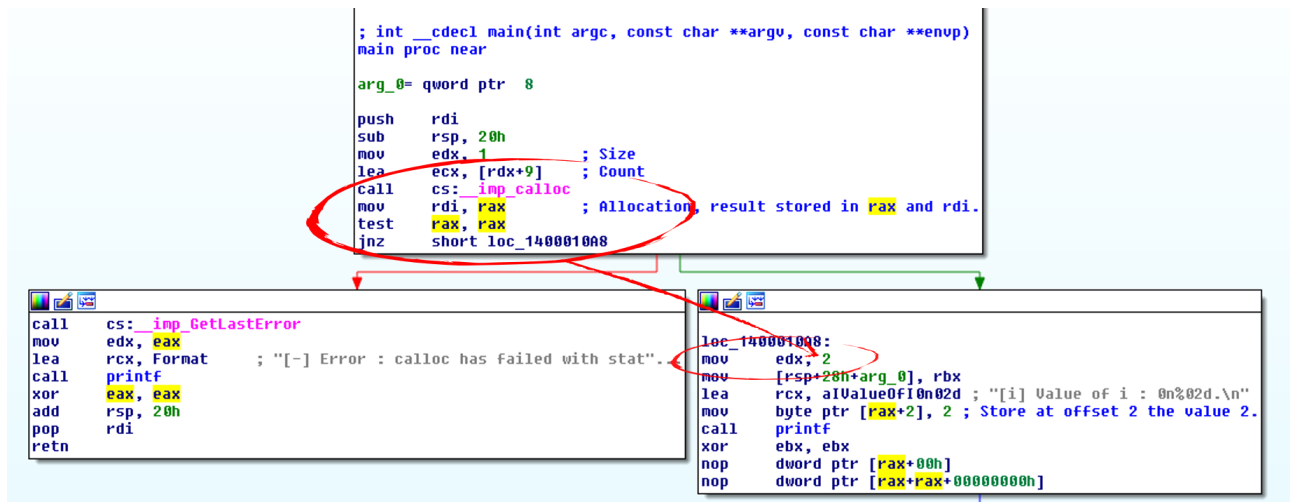


Figure 2.6: Decompilation of the code generated from Visual Studio 2018.

The case of GCC compiled code is a bit more complex. The code generated is represented in figure 2.7 and it has been split in four different steps. After the allocation of the buffer with a call to `calloc` function, if this one succeeded, we execute the step one. This step can be seen as `++i` instruction which means that `i` is equal to one. The step two aims computing the base address of the buffer allocated plus the current value of `i` (ie: `i = 1` at that point). The step three corresponds to the second incremental operation, which means `++i` thus `i = 2`. Finally, in step four, we are storing the current value of `i` in the base address computed in step two. That explains why at offset one the value stored is equal to 2 with GCC compiled code.

<sup>11</sup><https://www.hex-rays.com/products/ida/>



```

call    calloc
mov     [rbp+array], rax
cmp     [rbp+array], 0 ; Test allocation.
jnz     short __allocation_success ; Compute ++i (set i to one here).

__allocation_success: ; Compute ++i (set i to one here).
1 add     [rbp+i], 1
2 mov     eax, [rbp+i] ; Set the content of i to eax.
3 movsxd rdx, eax ; Copy the content of rax (ie: i) to rdx.
4 mov     rax, [rbp+array] ; Rax is now the base address of the array.
add     rax, rdx ; Add to the base address of array the offset store in i (ie: 1).
add     [rbp+i], 1 ; Add one to i.
mov     edx, [rbp+i] ; Get the content of i to rdx.
mov     [rax], dl ; Store the content of the new i value to array[1].
mov     eax, [rbp+i]
mov     edx, eax
lea     rcx, aIValue0fI0n02d ; "[i] Value of i : 0n%02d.\n"
call    printf
mov     [rbp+var_4], 0
jmp     short loc_4015AD

```

Figure 2.7: Decompilation of the code generated from GCC.

By adopting two different strategies to compile the same source code, two different compilers can produce two different outputs. The good point with this technique lies in the possibility to exploit such a trick for a long time and for legitimate purposes. No warning of any kind is displayed by default and operations written are legitimate. But this choice is not perfect.

On the first hand, if C norm does not define precisely execution order of such undefined constructions, it does not means the strategy taken by one compiler is not going to change from versions to versions of the compiler. Moreover, undefined constructs are inevitably strange for an experienced developer who will always prefer simpler and more efficient code. It is unlikely that in a serious project such codes would have a chance of being inserted in this way. Projects such as *Open Source Security Foundation*<sup>12</sup> are good to illustrate the standards and seriousness that some companies integrate in the development of their projects.

On the second hand, it is necessary to potentially play on the differentiation between several types of compilers. Changing from one compiler to one other can potentially cause a software to stop working and thus reveal the backdoor introduced in the code. This is normally a limited risk (there is usually no change of compiler for a given project), but an evolution in certain compiler design choices or any new features (especially about optimization) may produce different results and highlight the backdoor mechanism (or at least remove it by correcting what appears to be a *wired bug*).

This solution hybrid between source code backdoor and compiler is not sufficient enough to produce an acceptable result. But it rises the ability of compilers to make arbitrary choices in compiling code. Nevertheless, such point can be potentially interesting.

<sup>12</sup><https://openssf.org/>

## 4 Macro assembly and ml mistake with Boolean negation operator

### Key Point 2.5:

- ☞ MASM (*Microsoft Macro Assembler*) compiler is used since decades to develop assembly code in Microsoft environment.
  - ✍ In particular, it can be used part of development of critical programs such as firmware, drivers, mathematical libraries...
- ☞ The MASM compiler offers a facility for writing assembly code called "*Macro Assembler*".
  - ✍ This is one of the key reasons for MASM's success.
- ☞ MASM compiler does not correctly handle a conditional operator in assembler.
  - ✍ We found this bug in the MASM compiler with the **NOT** operator.
  - ✍ The construction is perfectly valid (according to MASM's documentation) and it produces neither any error nor warning.
  - ✍ In practice, MASM "forgets" the **NOT** operator in a specifically crafted condition.
  - ✍ This is a bug in MASM that can be exploited for malicious purposes (section 5).

### 4.1 Context

During our researches, we have been brought to work with different compilers, including assembly ones. Assembly language is not unique and there are plenty of different assembly languages (one can think about x86, x64, ARM, PowerPC, Alpha AXP, MISC, etc) which refer to different architectures and different CPU vendors. This is due to the fact that each CPU manufacturer is designing its own specifications which forces to get one assembly language per CPU family. The main advantage of the C language, for instance, is to be able to provide a large possibility to act at low level while keeping a sufficient level of abstraction to support multiple CPU architectures. But, we need sometimes to deal with assembly languages directly. This is the case whenever we have to review firmware codes, some drivers, UEFI framework, reverse-engineering operations, etc... And it could happen to translate some project from an assembly language into another, from time to time, when we need to port one application on another CPU family. The bug presented in the following section might not be unrelated to this situation...

### 4.2 MASM assembler

#### Resume 5:

- ☞ In this subsection, we explain some of the points that made the success of MASM compiler.
  - ✍ One of these important points is the use of operators that make assembly code look like C code.

Writing assembly code directly is reputed to be hard and complicated. Facilities provided by more abstracted languages are not present. It means that implementing simple algorithms can be a complicated task for developers. But it is easy to find documentation to know how to program in assembly language, especially on popular architectures such as x86 assembly family from Intel [147]. In the 32 bits CPU's family, compatible with Intel, there are different compilers. This is due to historical reasons. The same way, there are different assembly syntaxes (AT&T and Intel for the most famous ones). And, since there are (old) different compilers of assembly, it means there are different languages, such as GoAsm, NASM or MASM. The last one, MASM [148] which stands for *Microsoft Macro Assembler* [149], is famous by the ease it provides, since 1981, to implement programs thanks to the use of its macro code [150, 151]. This compiler has been developed by Microsoft and it is still present in the last versions of Visual Studio. It can be used internally for optimization purposes or directly when compiling code directly written in assembly language. One can note that MASM compiler did

not have major change from decades and it could have been used directly in projects such as UEFI, firmware, drivers, hypervisor, antivirus, operating system, optimized libraries used for calculation, etc...

Assembly language is less complicated than people usually think. Not far from C language, this one handles for us the management of the stack, the different calling conventions of functions and program's control flow by conditions and loops. These are the main differences from a technical point of view. By experience, beginners in assembly programming have difficulties in handling conditions. In C, writing conditions is a quite simple task and the same can be performed with assembly language, as displayed in Figures 2.8.

```

//                                     mov eax, [ebp+08h] ; Get 'a' value.
// It's a condition.                   cmp [ebp+0Ch], eax ; Check content of 'b' from 'a'.
//                                     jnz __out_of_cond ; Jump out of condition it is not equal.
if(a == b && c != 26){                 ; Otherwise, continue.
    ... // (...)                       cmp [ebp+10h], 26 ; Check the content of 'b' with 26.
    ...                                 je __out_of_cond ; If it does not match, go out of condition.
}                                       ; (...) ; Code if condition is true.
                                     __out_of_cond: ; Code after the condition's content.
                                     nop

```

Figure 2.8: Equivalent code in C and in assembly x86 language.

To understand the assembly code, the reader must take into account that local values are stored on the stack in x86 assembly and, in our example, are referenced from EBP register used as base pointer. The instruction `cmp` can be viewed as the subtraction between two values with the content of each variable preserved (only the resulting EFLAG register is modified, according to the logic of the operation). In case of equality, the difference is zero, otherwise it is not, which explains why `jne` (jump not equal) instruction is used there.

For obvious reasons, it is more interesting to write algorithms and complex structures than purely technical codes. Macro assembly such as MASM provides facilities for developers to help developers in conditions writing. Something equivalent, close to C language, can be written as given in code 2.2 below. The code is compiled with the help of `ml` [152] tool. The command `ml` is the short name of MASM compiler used as executable name to access the MASM compiler in Visual Studio [151].

```

1 mov eax, dword ptr [ebp+08h]
2 .if (eax == dword ptr [ebp+0Ch]) && (dword ptr [ebp+10h] != 26)
3     nop ; Relevant code when condition is met.
4 .endif

```

Code 2.2: "Rewritten of code in Macro Assembly (MASM)."

Easier to read, useful for long development, this code is halfway between C and assembly code (note the use of brackets does not change complexity here). Compiled with `ml` and disassembled with a debugger such as `Windbg`, the result is provided in the figure 2.9.

```

main!main+0x26:
010b1036 8b4508      mov     eax,dword ptr [ebp+8]
010b1039 3b450c      cmp     eax,dword ptr [ebp+0Ch]
010b103c 7507       jne    main!main+0x35 (010b1045) Branch

main!main+0x2e:
010b103e 837d101a   cmp     dword ptr [ebp+10h],1Ah
010b1042 7401       je     main!main+0x35 (010b1045) Branch

main!main+0x34:
010b1044 90        nop

main!main+0x35:
010b1045 90        nop
010b1046 90        nop

```

Figure 2.9: Disassembly of code in figure 2.2.

The code is built in order to force the program to jump at `0x010b1045` if one of the condition is not met. Otherwise, the program continues its normal flow to reach `0x010b1044`.

### 4.3 Operators from MASM compiler

#### Key Point 2.6:

- ☞ MASM proposes a **NOT** operator (symbol `!`) uses to *negate* a condition or a value.
  - ☞ In practice, what is wrong (i.e. 0) becomes true (i.e. 1) and *vice versa*.
  - ☞ Its use is correctly documented with many examples of use in MASM's documentation.

According to MASM's official documentation [153], in addition to **AND** conditions, MASM compiler supports many more operators such as **OR** or **NOT**. It is easy to find different examples of code illustrating the use of these operators in official MASM's documentation. The figure 2.10 resumes all of these symbols.

### Expression Operators

The binary relational operators in MASM 6.1 are the same binary operators used in C. These operators generate MASM compare, test, and conditional jump instructions. High-level control instructions include:

Operator	Meaning
<code>==</code>	Equal
<code>!=</code>	Not equal
<code>&gt;</code>	Greater than
<code>&gt;=</code>	Greater than or equal to
<code>&lt;</code>	Less than
<code>&lt;=</code>	Less than or equal to
<code>&amp;</code>	Bit test
<code>!</code>	Logical NOT
<code>&amp;&amp;</code>	Logical AND
<code>  </code>	Logical OR

A condition without operators (other than `!`) tests for nonzero as it does in C. For example, `.WHILE (xx)` is the same as `.WHILE (xx != 0)`, and `.WHILE (!xx)` is the same as `.WHILE (xx == 0)`.

Figure 2.10: List of symbols used in MASM officially supported.

These symbols can be used with registers or values. Moreover, these ones can also be used in conditions or complex expressions. Here again, official documentation is providing details about. An extract from documentation is provided in figure 2.11.

In our case, we are going to focus on **NOT** operator noted with the symbol `!`. This operator has a high precedence from other operators. This is directly documented from the documentation and reported in figure 2.12.

Finally, it comes different examples using the **NOT** operator. These ones are extracted in table 2.2 from official documentation and they illustrate the different possibilities to use the **NOT** operator with values, flags and registers. This last point matters since it will allow us to define the bug we exploit in MASM compiler.

## Expression Evaluation

The assembler evaluates conditions created with high-level control structures according to short-circuit evaluation. If the evaluation of a particular condition automatically determines the final result (such as a condition that evaluates to false in a compound statement concatenated with **AND**), the evaluation does not continue.

For example, in this **.WHILE** statement,

```
.WHILE (ax > 0) && (WORD PTR [bx] == 0)
```

the assembler evaluates the first condition. If this condition is false (that is, if AX is less than or equal to 0), the evaluation is finished. The second condition is not checked and the loop does not execute, because a compound condition containing **&&** requires both expressions to be true for the entire condition to be true.

Figure 2.11: Evaluation of complex statements with MASM operators.

## Precedence Level

As with C, you can concatenate conditions with the **&&** operator for **AND**, the **||** operator for **OR**, and the **!** operator for negate. The precedence level is **!**, **&&**, and **||** with **!** having the highest priority. Like expressions in high-level languages, precedence is evaluated left to right.

Figure 2.12: Documentation about the precedence of the **NOT** operator.

<pre>.REPEAT mov  bx, es:[4Ch]      ; Now repeatedly poll clock mov  ax, es:[4Bh]      ; count until the target sub  bx, dx            ; time is reached sbb  ax, cx <b>UNTIL !carry?</b></pre>	<pre>; If the current byte in the buffer doesn't exist in the ; search token, increment buffer pointer to current position ; +1 and start over. This can skip up to 'EDX' ; bytes and reduce search time <b>IF !rgbInTok[ebx]</b> add  edi, ebx          ; Initialize ECX with inc  edi              ; length of szTok mov  ecx, edx</pre>	<pre>; Finally, if we have searched all szTok characters, ; and land here, we have a mismatch and we increment ; our pointer into rgbSearch by one and start over. <b>ELSEIF (!ecx)</b> inc  edi mov  ecx, edx .ENDIF</pre>
---	--	---

Table 2.2: Different use of **NOT** operator with MASM compiler.

## 4.4 Use of NOT operator with MASM compiler

### Key Point 2.7:

☞ The **NOT** operator works as expected for simple samples.

In the following example, we build two codes to illustrate the action of the **NOT** operators from macro assembly to generated opcodes by the **ml** compiler. On the left column, we test whether the value stored in **eax** is different from zero or not. If it is not, instruction **mov edx, 1** is executed. Otherwise, we go to the following **nop** instruction. On the right column, the condition is negated, it means if the value stored in **eax** is zero, then, the instruction **mov edx, 1** is now executed. This is represented in table 2.3 where each source code and compiled code are represented per column.

No negation				Negation			
<pre>mov eax, dword ptr [ebp+04h] .if eax     mov edx, 1 ; Condition is true. .endif nop</pre>				<pre>mov eax, dword ptr [ebp+04h] .if !eax     mov edx, 1 ; Condition is true. .endif nop</pre>			
0040103c	8b4504	mov	eax,dword ptr [ebp+4]	0040103c	8b4504	MOV	eax,dword ptr [ebp+4]
0040103f	0bc0	or	eax,edx	0040103f	0bc0	OR	eax,edx
00401041	7405	je	main!main+0x10 (00401048)	00401041	7505	jne	main!main+0x10 (00401048)
00401043	ba01000000	mov	edx,1	00401043	ba01000000	MOV	edx,1
00401048	90	nop		00401048	90	nop	

Table 2.3: View of compiled code when using **NOT** operator with MASM compiler.

The test operation is performed by **or eax, eax** instruction in both cases. This specific operation could look like a *no operation* since it is not supposed to perform any modification on **eax** content. That is the goal in a sense it preserves the content of the value but set specific flags in **EFLAG** register to allow conditional instruction to work according to the content of **eax**, especially if **eax** is zero or not.

The difference comes in the conditional jump instruction which is performed right after. In the first case, **je** which stands for "jump near if equal" means that the jump operation is performed when flag zero is set to one (**ZF=1**). In other word, if the result of **or eax, eax** would set **ZF** to one if **eax** was zero. In such a case, the jump will be executed and the instruction provided if condition is met would be skipped (going to **nop** instruction). Or course, if content of **eax** is different from zero, the instructions inside the if statement are executed.

For the second case, the conditional jump is now **jne** which is the opposite version of **je**. The condition is met when the zero flag is set to zero (**ZF=0**). From a logic point of view, it is a negation of the first case. Until that point, everything works correctly as expected. This point is important since it proves that code written for MASM compiler can legitimately uses **NOT** operator since the compiler provides the logic written in source code.

## 4.5 Bug in MASM compiler

### Key Point 2.8:

☞ Unfortunately, when used in the context of a multiple condition, the **NOT** operator is no longer interpreted correctly.

☞ It is ignored by the compiler which skips and ignores it.

Issues come when we raise the level of complexity in conditions. Consider the following conditions:

```
|| .if eax == !ebx
```

```

2 |   mov eax, 2
   | .endif
4 |
6 |   .if (!eax) == ebx
   |     mov eax, 3
   | .endif
8 |
10 |  .if eax == ebx
   |     mov eax, 4
   | .endif

```

Code 2.3: "Use of NOT operator in different conditions."

For obvious reasons, all of them are supposed to produce different results, according to the values stored in `eax` and `ebx`. Technically speaking, a condition such as `(eax == !ebx)` could be rewritten as `(eax == (ebx == 0))` to be more understandable. The same applies for `((!eax) == ebx)` and the use or the lack of brackets do not change anything for the compiler (according to the precedence of **NOT** operator defined in MASM official documentation [153]). Because all conditions are different, they all should be computed differently (use of different op-codes or conditional jumps, for instance) since each describes a different condition to be met. Compilation with `ml` does not give any warning nor error. In consequence, the main question stands in the following disassembly (Figure 2.3) of these conditions where all conditions are compiled in the exactly same way.

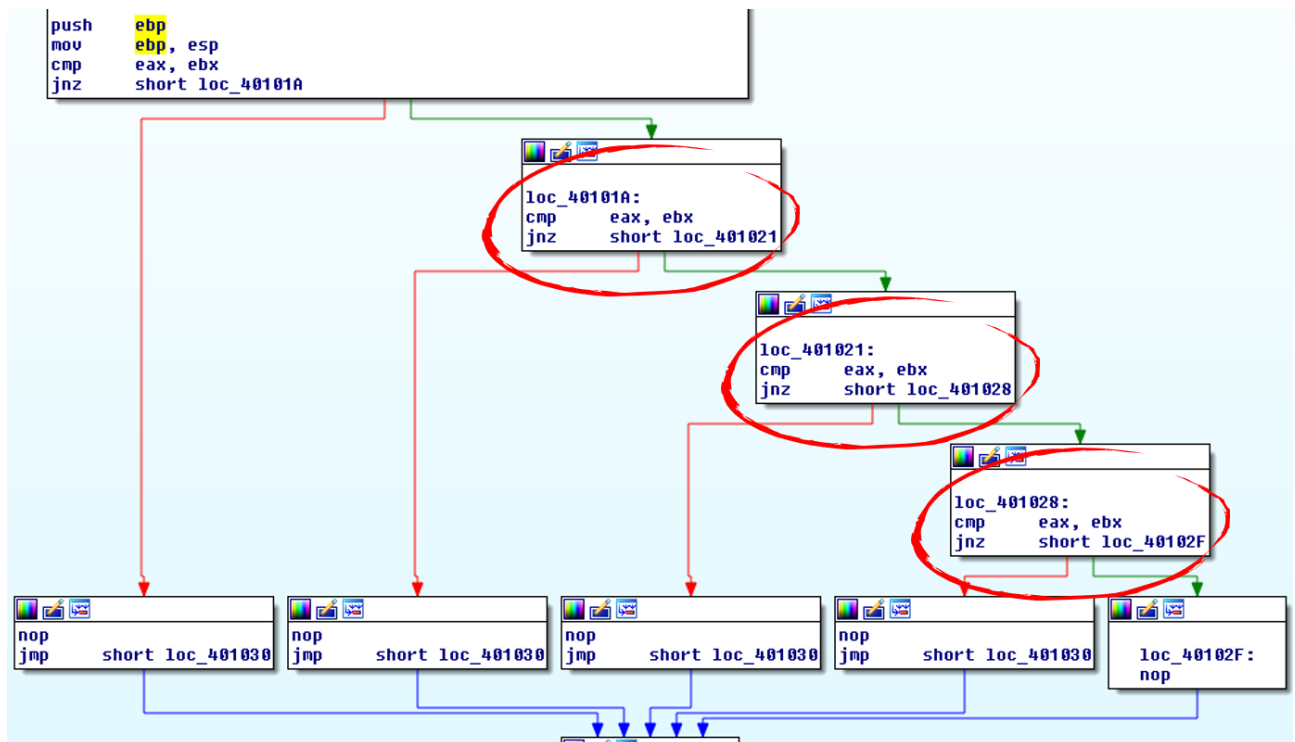


Figure 2.13: Disassembly of code compiled in figure 2.3.

Each condition is built in the same manner (`cmp eax, ebx` followed by a conditional jump if not equal — `jne` — to the next condition) despite the fact they are differently written in source code and thus all different from a logic point of view. From the `ml` compiler point of view, `eax == ebx` is the same than `(!eax) == ebx`, which is obviously wrong. Any debugging session is enough to confirm the logic written originally in the source code is not respected at generation time by the compiler.



## 4.6 Explanation of the bug in MASM compiler

To understand which error has been made by **ml** compiler, we wrote an equivalent code in C, compiled it with visual studio compiler and reversed it with IDA software. The C equivalent code 2.4 is written with the constraint to retrieve two values from the user. This design is implemented to prevent the compiler to perform any optimization in the backyard which could remove the two variables.

```

1 #include <tchar.h>
2 #include <Windows.h>
3
4 int _tmain(int argc, LPTSTR argv[]) {
5     int a = 0;
6     int b = 0;
7
8     //
9     // Ask two numbers to prevent compiler's optimization with constants.
10    //
11    _tprintf(_T("Get number a ? "));
12    scanf_s("%d", &a);
13
14    _tprintf(_T("Get number b ? "));
15    scanf_s("%d", &b);
16
17    //
18    // Perform the same test than those written in assembly.
19    //
20    if(a == !b){
21        _tprintf(_T("First!\n"));
22    }
23    if(!a == b){
24        _tprintf(_T("Second!\n"));
25    }
26    if(a == b){
27        _tprintf(_T("Third!\n"));
28    }
29
30    return 0;
31 }

```

Code 2.4: "C code supposed to test NOT condition."

Decompiled code 2.4 is a piece of cake to check how the Visual Studio compiler built this source code. Given in figure 2.14, difference with the code compiled with **ml** is blatant. The first block of code is dedicated to retrieve values from the user. Then, the first condition ( $a == !b$ ) is computed by the use of a temporary value stored on the stack. This one is computed in blocs 2 and 3 by the use of instruction "mov [ebp+tmp], number" where number corresponds to the boolean result of the check performed on  $b$  value with instruction "cmp [ebp+b], 0". Indeed, according to the value stored in  $b$ , a temporary local value is set to one or zero (second and third block) to be finally compared to a value at the end in the fourth bloc. In other words, the negation of  $b$  is stored in a *shadow* local value (stored on the stack) to be used in the comparison next. The same applies for the next conditions (in such a case, the temporary value is defined from value  $a$ , according to the source code).

Comparing with what the C compiler did, we can conclude where is the issue with **ml** compiler. For short, C compiler uses a temporary value to compute the negation of a given local value, what **ml** compiler does not. An explanation could sit in the lack of capacity of **ml** to build its own local values. Even if MASM is perfectly capable to handle local values in a function [154], it does not do it with operators. The guessed reason would be to not use the stack behind the developer's back. Indeed, in critical situation, the value of the head of the stack (stored in ESP register) could be at its limit address. In such a case, using the stack to store a shadow value would result in a memory invalid address access, rising an exception, which is not exactly the behavior that a developer could expect.

In brief, **ml**'s bug resides in the use of negation operator in an equality (or inequality) test. Every code which follows the pattern " $! <register/value/memory>$ " is eligible to be incorrectly compiled. This bug is introduced



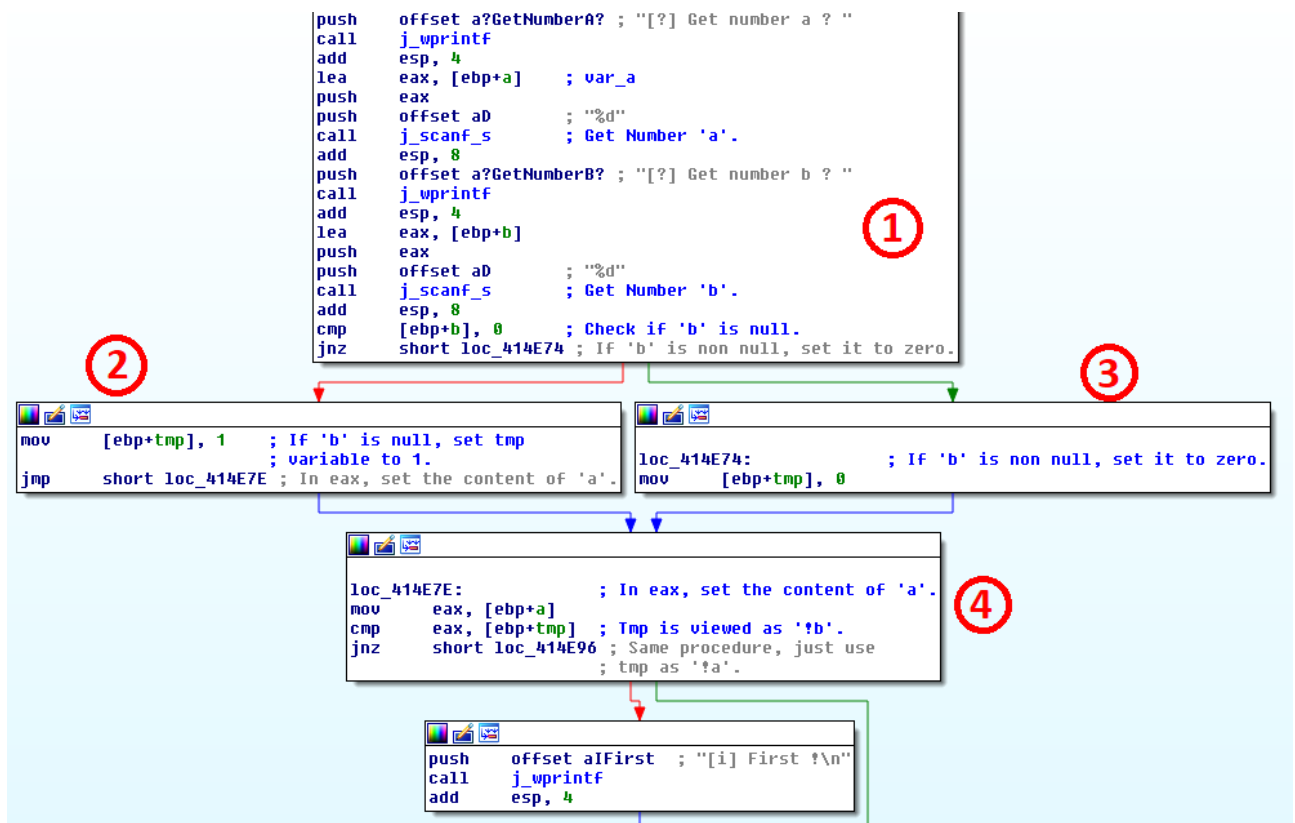


Figure 2.14: Decompiled code from the one compiled in figure 2.4.

by a bad management of the value negated. Indeed, the use of **NOT** operator changes the content of the value *inside* the condition. Out of the condition, the negated value is not supposed to be altered. The lack of use of temporary variable to hold the result of the operator leads to misinterpret the condition, resulting that a wrong-code is silently generated. Note that regular negation from a single value (for instance: ".if !eax") or a full condition (such as: ".if !(condition)") is perfectly fine since it is the test instruction or the conditional jump which is impacted in that case.

## 5 How to build a sneaky backdoor with ml compiler bug?

### Resume 6:

- ☞ We exploit the vulnerability in MASM to produce valid source code that implements a backdoor.
  - ✍ We explain how to transform the vulnerability in MASM to silently introduce a backdoor in a program compiled with it.
  - ✍ We reuse the context proposed by Ken Thompson [115] to trick a user authentication program as [137] did.
  - ✍ We explain the step-by-step methodology to be used to successfully implement a backdoor without breaking the original program.

### 5.1 Context of the backdoor

#### Key Point 2.9:

- ☞ We exploit a vulnerability in MASM that allows us to silently change the logic of written conditions.
  - ✍ Triggering the vulnerability is done via a simple code written (and using perfectly documented code examples).
  - ✍ We go beyond the gap (defined in Resume 3) established from literature review.

If it is possible to exploit the bug to execute unexpected code despite the apparent logic of the source code, a backdoor is writable. The goal is to get an official source code which should pass a code review from anyone (expert or not). Of course, reverse engineering compiled code from application would be enough to find the *mistake* we included in the logic flow of the process. Even if we still need to know where and what to look for... But we are acting in a position where we have access to the source of the targeted application. An application that we can compile by ourselves. Open-source software are perfect target for that, but not only. In the case of a company, any frustrated, dishonest, corrupted employee is enough to perform malicious action on the source code — especially with the guarantee that action taken remains stealthy. One can think about a consultant hired as freelance by a company which outsources its development. This happens in areas of highly specialized expertise, such as, surprisingly, kernel or firmware programming, where assembly language may be needed since such expertise is not a mass sport...

As described in [102], two plots are available to build the backdoor. First, we own the project we want to backdoor. On the positive side, it makes it easy to introduce a backdoor since we are the main leader developer. On the negative side, if the backdoor is found, we lose and here comes consequences on our reputation, regardless of possible lawsuits. To reduce the possible cost, the second plot is about to propose a patch for an existing software. It allows us to patch *several* open-source projects since we can act as an occasional contributor. The case of the consultant is equivalent. This is stealthier in the sense that we are no more on the first line. In addition, we get the benefit to possibly argue it was a plausible mistake instead of a malicious intent by design. In all cases, attack is performed through a modification of an existing software to detour it from its original procedure.

To stay within the context of past research, we propose to introduce a backdoor at compilation time in a program equivalent to *sudo*, as Thompson [115] and Bauer [137] did. This example could be applied to other program such as *runas* [155] on Windows operating system. Since these programs are not written in assembly code, we are going to assume that we are in the case of trying to backdoor a firmware program authenticating its users, the same way *sudo* would have done it. The program makes the difference between simple users and administrators. Both populations authenticate themselves with a password specific to each user. Of course, the program code is written in assembly language. As in any authentication program, there is always a final condition that guarantees (or denies) access to a protected resource. It is this condition that we are going to target.

The main part, about the modification we submit, is about to not raise a security issue. First, no analysis (coming from human or software with formal verification) of code would be able to detect a security breach in the code since the official security logic is respected. This is an essential condition for the backdoor to remain stealthy and to be present over a long period of time. Then, the bug we exploit must avoid to raise questions about the legitimacy of the inserted code. One should not rely on an obfuscation or a dubious comment as in [137]. After all, we are trying to fool the logic of conditions in a sneaky way (a condition supposed to be false is going to be interpreted finally, as true) so that code, which is supposed to reject something, now, accepts it.

Even in the case where the code would be detected, it still remains possible to deny any responsibility in the vulnerability introduced. Indeed, the logic of checks in the original code is supposed to be true and we can always claim that we were not supposed to know that the debugger used had a bug resulting in such security flaw in the generated code. At worst, one can blame us not to have tested our software enough on side effects. Of course, with an example as simple as the one provided in code 2.5, it can be complicated to argue we did not test it. But more complex cases can be implemented.

## 5.2 Description of bug consequences in order to insert a backdoor

### Resume 7:

✍ We explain in this subsection how change the logic of a condition with the bugged **NOT** operator.

Formally speaking, the bug in the compiler does not take into account the **NOT** operator in context of condition checks. In our case, condition is limited to equal or difference. The legitimate use of **NOT** operator with equal (or difference) test written can be modeled with the help of Boolean values (meaning that  $a \in \mathbb{F}_2$  with  $\mathbb{F}_2 = \{0, 1\}$ ). This makes sense since conditions manipulate Boolean expression to jump on a specific code, according to the validity of the condition written. With the compiler's bug, we have an equivalence between all the following codes:  $\forall a, b \in \mathbb{F}_2$ , we have  $a == b \iff !a == b \iff a == !b \iff !a == !b$  which, obviously, does not respect mathematical logic. A more complex example can be illustrated with:  $!(a == b)$ . In a non bogus environment, we have the equivalent conditions:

$$\begin{aligned} !(a == b) &\iff !(a \neq b) \\ &\iff a == b \end{aligned}$$

In the bogus environment of **ml** compiler, truth equations are changing. It is due to the **NOT** operator which is not interpreted correctly. The first **NOT** is interpreted on the whole condition while it changes the jump instruction. Starting with an equality, it is now considered as an inequality. Because of the bug, the second **NOT** (in the brackets) is not interpreted at all, which leads to the following conditions:

$$\begin{aligned} !(a == b) &\iff !(a == b) \\ &\iff a \neq b \end{aligned}$$

With this change in logic in conditions of the program, we can replace the cases where a condition is normally not executed into one which is now executed. Note the opposite operation is also correct: it is possible to change a condition which is executed to one which is no more executed. This is the main point to build the plot of our backdoor.

## 5.3 Creation of the backdoor

### Resume 8:

✍ We explain step-by-step how to design a backdoor in a source code, preserving the original behavior while introducing a new one.

✍ We reuse the same plot (i.e. *sudo* software to provide admin rights) used in literature.

Using logic bug to change behavior of a targeted application is quite simple. We illustrate things with the use of access check to a resource reserved to administrator. The goal is to introduce a flaw in a function supposed to check access rights of the calling process before continuing potential privileged resources. The simplest condition could look like the one provided in code 2.5.

```

1 | ; If the caller is not an administrator, it means it is a simple user.
2 | if !caller_context.isAdmin
3 |     mov eax, 0C0000022h      ; ERROR_ACCESS_DENIED
4 |     goto __end
5 | .endif
6 | nop      ; Continue execution only for administrators.

```

Code 2.5: "Simple backdoor using the compiler's bug."

With the previous code, if a regular user calls the function, it gains access to the privileged function. This is due to the fact that the **NOT** operator is not correctly interpreted by the compiler. Finally, **ml** compiles a condition close to ".if caller\_context.isAdmin == 1" which triggers the condition, so that the function is stopped. But the code 2.5 is wrong for an operational backdoor. Indeed, if an administrator calls the function, because of the bug, the legitimate access is now refused. In this case, even the simplest unit testing would be enough to see that something is wrong. If we insert a backdoor, this one must keep the original behavior of the modified code. Inserting a new *backdoor functionality* does not mean to remove a feature that is already present.

The solution comes with a small raise of complexity about the original code to backdoor. Continuing on the context provided by Thompson [115], this one can check, first, if the user belongs to a group, then if the couple user-name and password is correct. At that time, if the user belongs to the group of administrator, access can be guaranteed. A correct implementation would lead to check all these operations ones after the others. But, for optimizing things up, we propose to check if the provided user-name is administrator or just a regular user (implicitly supposing there is no other group except administrator or regular user). Finally, the check of the provided password can be performed. The targeted code can be written as the one given in code 2.6.

```

1 | ; Check if the provided user belongs to users
2 | ; group and save result in esi.
3 | push offset CurrentUserName
4 | call IsUserRegularUser
5 | mov esi, eax
6 |
7 | ; Check if the provided user belongs to
8 | ; administrator group and save result in edi.
9 | push offset CurrentUserName
10 | call IsUserAdministrator
11 | mov edi, eax
12 |
13 | ; Check password provided with the user name.
14 | push offset PasswordProvided
15 | push offset CurrentUserName
16 | call CheckIfPasswordIsValid
17 |
18 | ; Check if the user is administrator and
19 | ; the password is correct.
20 | .if edi == 1 && esi == 0 && eax == 1
21 |     jmp __allow_admin
22 | .endif

```

Code 2.6: "Test procedure to give access only to administrator with the right login and password."

The *double* check of the user affiliation can be justified by an overzealous attitude or the fact that non administrator user could perform some unprivileged tasks in the code following the condition. The code in Figure 2.6 is the original code to backdoor. Of course, the code could be optimized by directly call the different functions in the condition directly. But for illustration purposes, we propose to keep them away from the condition, since it does not change the logic of that one. Furthermore, the main working space is in the condition for us. If we

study the condition with a truth table, we can map all possible outputs of the condition regarding its inputs. Such truth table from condition written in code 2.6 is given by table 2.4.

Administrator	User	Password correct	Result
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	?
1	1	1	?

Table 2.4: Trust table of the final condition in code 2.6.

Note the last two possibilities referenced in table 2.4 should never happen since they correspond to impossible cases. Indeed, by definition, a user cannot be both in administrator group and in the one of regular users (ie: non administrator). And even if it would have been possible, there is no real definitive answer here about knowing if we need to provide or not the access. Since they are impossible by definition, these undefined states could be an interesting spot to exploit with our compiler's bug... Technically, our goal is to get a regular user (ie: non administrator), with the correct password of its user account, being executed as an administrator, in addition to regular administrator users whose execution is still guaranteed. First, we are going to modify the condition in a manner that it does not change the logic of the condition but it introduce our **NOT** operator. One can write the code 2.7.

```

1 | .if (edi == 1 && !esi == 1 && eax == 1
2 |     jmp __allow_admin
3 | .endif

```

Code 2.7: "Change the condition (but not the logic) to introduce NOT."

The code 2.7 introduces the **NOT** operator but it does not change the logic of the condition from a formal point of view. Note that there are many ways, with the compiler's bug, to introduce this operator (for instance, `!esi == edi` since EDI is supposed to be equal to one in case where we would deal with an administrator user). Our modification of the condition is targeted on the ESI register. This is due to the fact that ESI is supposed to contain the Boolean value representing the belonging to the group of regular users. This modification introduces the bugged operator in order to misinterpret the real membership of the provided user in the executed condition.

The logic of the condition is to only accept what is defined as true to let privileged functionalities being executed. With just the last modification, no user (administrator or not) would be allowed. Indeed, the condition is now only valid if the user is identified as being membership of both administrator and non administrator group. This is due to the use of **AND** operators in the condition. To remove this obstacle, we can rewrite the condition with a new order and by the use of *negative logic*. It means that everything which is identified not to be an authenticated user will be rejected, allowing the execution to continue otherwise. Of course, undefined states where a user would belong to both groups are not supposed to happen in theory. This is why we can pretend it is not necessary to implement additional conditions that are redundant and not required. A possible backdoored condition is given in the code 2.8.

```

1 | .if (eax == 1 && (!esi == 1 || edi == 1))
2 |     jmp __allow_admin
3 | .endif

```

Code 2.8: "Final trapped condition."

Condition written in code 2.8 is conform with the logic of the original condition given in code 2.6. It first checks whether the password is correct or not. On the first hand, if the password is incorrect, the code does

not execute the content of the condition without evaluating the rest of the condition. The justification of this first check lies in the need to avoid unnecessary checks if the password is invalid. Direct call of functions in the condition could be a way to optimize login performances by only evaluating user groups when password is correct. On the other hand, if the password is correct, the condition checks if the provided user is either an administrator or a not regular user. This double check is to *officially* avoid *undefined* membership status of users, allowing only truly authenticated administrators to get access in case of a user would belong to both group would have happened...

The trust table of the condition written in code 2.8, when only considering the trust table and ignoring the bug in MASM compiler is given in Table 2.5.

Password correct	User	Administrator	Result
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1 - Imp
1	0	1	1
1	1	0	0
1	1	1	1 - Imp

Table 2.5: Trust table of the trapped condition from source code point of view.

Reading the trust table given in Table 2.5 simply shows that the written condition is consistent with what can be expected from it. Only administrators with the correct password are authorized by the condition. The two possible cases where an authorization would be possible is when a user belong to both or none of the groups, which is by definition, impossible.

Of course, thank to the compiler's bug, the compiled condition is slightly different from the trust Table 2.5. Actually, the sub-condition (`!esi == 1`) is interpreted as (`esi == 1`). Under this condition, it allows an administrator or a regular user, provided the password linked to its account, to get access to privileged functionalities. Table 2.6 resumes the truth table really compiled by `ml`.

Password correct	User	Administrator	Result
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1 - Imp

Table 2.6: Trust table of the compiled trapped condition with `ml` compiler.

The antepenultimate and the penultimate lines define the expected result. Either an administrator or a regular user — correctly authenticated — check the condition. This is the design of the backdoor we wanted to introduce. It allows one more *hidden* users to get access to privileged functionalities while keeping the original behavior from the condition. About the case of the last line, this one is not relevant, since it is theoretically not possible that a single user can be a member of two groups at the same time. And, even if it could happen, officially, the condition would give access if the user that belongs to administrator group. To some degree, condition made a choice to privilege admin users, whatever they belong to another group at the same time... A perfectly sustainable choice from security point of view.

At the difference of [137], our code does not appear as *needlessly complicated*. It is perfectly justifiable without betray its unavowable design. One malicious developer could insert such a code as a patch in a code. Justification would be about optimization purposes, as we did. Another difference lies in the flexibility of our backdoor. It can be introduced in any condition, changing the logic flow of this one without removing the original behavior of the condition. To perform the modification, one can follow these steps:

1. Define the original truth table of the condition ;
2. Write the expected truth table for the new condition ;
3. Build a legitimate condition which follows the last truth table by the use of **NOT** operator in an equal test condition to exploit the bug.

## 6 Correction about the bug in MASM

### Key Point 2.10:

- ☞ The bug has been reported to Microsoft, which has registered and corrected it under the number CVE-2018-8232.
  - ☞ This is such an original bug that Microsoft did not really anticipate that it could exist.
  - ☞ There have been heated debates about what this type of vulnerability really was and how to fix it.
  - ☞ The bug has been present in the Microsoft compiler for decades (at least 30 years).
  - ☞ It has been corrected with the concern of not breaking the existing software compiled with MASM.
  - ☞ MASM's error code **A2154** has been given to a construction which would use the vulnerability.
  - ☞ Microsoft chooses to break compatibility with its documentation for a build that it nevertheless considers unlikely.
- ☞ However, the correction remains partial (only in the Visual Studio compiler).

### 6.1 Reporting of the bug

Once the flaw has been discovered, and because of its criticality, it has been transferred to Microsoft at the end of February 2018. It was a long story before the bug got fixed. This is kind of the social part of the thesis... It was complex to explain the issue and the impact of such a bug in Microsoft's compiler. Why? Simply because there is no critical vulnerability in the compiler itself. After all, there is no elevation of privileges, nor is there any bypassing of any security within the system. This is obviously the main problem posed by a vulnerability in a compiler. As explained in section 2.3.2, it is not directly in the compiler where the security flaw is, but the objects potentially produced by the latter.

To suggest a metaphor, we can consider a manufacturing line that has a latent defect which insidiously impacts from time to time produced objects from that line. In itself, this does not prevent objects from being produced on the production line, especially when we are ignoring existence of this latent defect. Nevertheless, the defect in the production chain has insidious repercussions on products that come out of that one. The defect induced in the produced objects is not present for most or it is minimal and marginal for few because it requires some very specific conditions to be present. But there is a non-zero probability that it may be present and consequences can be serious for users of that object. At worst, when such situation occurs in industrial world, if a serious defect is discovered, we can always recall the objects produced and check or fix them. But in our case, the production line is neither central nor easily accessible. In fact, people can import and copy by themselves their own production line, conform to the original one, latent defect included. Therefore, it is neither possible to know who used that chain, nor which product has been produced with, nor even whether



these products may be affected by the induced defect. And of course, the older the original production line is, the greater the quantity of objects produced is. By consequence of statistics laws, greater is the number of potentially impacted objects.

Metaphor from the precedent paragraph is transparent if we substitute "production line" by "compiler", "produced object" by "compiled software" and "latent defect" by "silent bug in the compiler". This is literally the problem of compilers' bugs which results in incorrectly generated software. Of course, this raises the question of responsibility and how to respond to that case. The central point is the compiler and preventing future compiled programs from being impacted by the defect is a minimum that stands to reason. In addition to this first point, it comes the question of already produced products and potentially impacted by the compiler's bug. This means we could sought to warn compiler's users, tell them to check their source codes, possibly recompile them, update the executable files compiled on the system, etc. The stain is huge and this is exactly the problem Microsoft had to deal with when they were informed about the bug.

At the beginning, Microsoft answered this was not a bug for them and thought about closing the case. After further explanations, they admitted the problem was serious in the case of user of the compiler would have been exposed to the bug. Confronted with their own documentation that references numerous examples of illustration and use of the operator in question (Figures in Table 2.2), it was getting hard to deny the problem. The 13<sup>th</sup> March, CVE-2018-0984<sup>13</sup> number has been temporarily assigned to the problem. One month latter, Microsoft contacted us to ask more details about the flaw in the compiler since they had an "*internal heated debate*" about this bug. After providing them as many details as we had, we asked them about a potential correction of the bug and the strategy they had about managing already compiled code. Their response was quite surprising. Indeed, they admit they were still debating internally to know whether this should be a functional bug (which would be fix in a *vnext* — next version) versus a security issue (where they would assign a CVE and push a security update).

On that note, after a month of silence, Microsoft contacted us again to announce us that their debate is over: it is not a vulnerability, assigned CVE number is removed and a fix might be written one day in a future version of the compiler. MASM compilers exists since 1981 and it is not the software that is known to be regularly updated... This response strongly resembles a "no" response and the desire to get rid of the problem rather than fixing it. But we decided to bow to Microsoft's decision. After all, if it is not a so-called vulnerability, then we can publish everything the next day and let the scientific and cyber community deal directly with it. It is a classic research process and the CVE number withdrawn, the question of guaranteeing the precedence of our searches arises directly...

Surprisingly, Microsoft was not comfortable at all with the idea that we would publish anything about the subject. As a result, what is not a vulnerability and does not deserve to be quickly corrected does not need to be particularly public for them either. Apart from this strange ability to cultivate the art of palinody, they confess that "*this issue is atypical*" and the decision of publication makes the "*product team to revisit the decision*". Notwithstanding we had already informed them of our willingness to publish once the problem would have been fixed, we agreed about the phone call they proposed to directly discuss together. Of course, this issue is atypical and the goal for us was just about fixing this security hole and raising any alarm bell on the subject of backdoor potentially introduced at compilation time.

The phone call has been a great moment to discuss the issue. From Microsoft point of view, the problem is indescribable. Nobody has touched this compiler for 15 years and the problem is not just about to fix the bugged operator. Their primary concern is that the modification does not specifically break any existing code compilation. Indeed, compilers are sensitive tools and the slightest modification could have potentially more serious consequences than the bug itself. Apart from the clients potentially impacted by code that would no longer compile or that would have to be modified, the code generated by MASM should not change too much to keep some backward compatibility. And it is of course difficult to estimate how many people could be impacted by this change, for the reasons explained above. Note that we are dealing here with a compiler containing a bug that we have been able to successfully observe since the very first versions. The bug is about 30 years old, which allows a large number of potential silently exploitation. It remains that since Microsoft has no idea how

---

<sup>13</sup><https://cve.mitre.org/cgi-bin/cvename.cgi?name=2018-0984>



many software would have been potentially impacted or who to contact or to notify about an update. Thus, they decided to just update the compiler without any more publicity. A strategy that could be summarized as *“every man for himself, and God for us all”*.

Finally, Microsoft accepted to fix the bug and recognize a certain severity to the problem. The 16<sup>th</sup> May, just following the phone call, CVE-2018-8232<sup>14</sup> number was assigned to the bug. The 7<sup>th</sup> July 2018, Microsoft published the correction of the compiler in Visual Studio update. Other sources of supply for the compiler will not be fixed<sup>15</sup> since the product is not anymore really supported as a project by itself [156].

The story could be finished at that point. Even if one could also wonder about Microsoft’s attitude. Of course, we must not fall into conspiracy theory. Always consider only the facts that can be proven and verified. Definitely, a lot of time has been spent discussing the nature of this bug. According to the Microsoft team, it was not even considered that a bug could be reported on this product. Surely, such a bug, exploited that way, is far from being common. But it leaves one wondering about the fact that it took about more than 6 months to fix a bug that, all in all, is not very complex. Attempting to bury the affair at the beginning is not in line with Microsoft’s classic attitude. Indeed, they are always prompt and transparent in fixing bugs. It is simply a question about their public image. Whatever it is, we can appreciate some of the specialized press review about this update of Microsoft, especially the one from Zero Day Initiative journal [157]. The latter seems to have a very strong opinion on the question of the operability of a compiler flaw by qualifying it as *“sounding like a plot device in a Mission Impossible movie”*. Either he was obviously very (too?) well informed on the subject, or he was obviously not informed at all since at that time, officially Microsoft and our services were only informed about technical details and operability from this bug... Nevertheless, it is questionable to know if everything possible has not been done to reduce the public impact of this vulnerability.

It must be seen that the case is original, that clumsiness may have been in done on both sides and, on this case, the problem is a quite original (if not potentially sensitive) and frankly uncommon. There may not be any specific procedure to deal with that and this could explain the twists and turns of this story. It must also be seen that such subjects sometimes go beyond the purely scientific and technical aspects to go on more political ones. This is also an aspect that is part of scientific research and it should also be noted.

## 6.2 Potential corrections of the bug

Correction of the bug has been proposed to Microsoft. This was with the initial report we transferred to them by mail. For the sake of clarifying what is possible, we have reproduced here the corrective actions that we proposed at that time.

Since the bug is clearly identified, detection of specific patterns in conditions is enough to detect potential issues. From that point, many possibilities are doable. First, the compiler could display warning or error messages to prevent compilation of such code. If it does not fix the problem, it allows developers to be informed about it. In case of the correction would lead to consider the patten as an error, it would avoid developer to use what is considered now as *partially unsupported operator*. Indeed, the operator is still present but only allowed in clear and simple situations (on a single register only, for instance). Whatever the choice between warning or error is, in both cases it would be an invitation for developers to review their code if they are using such construction.

The second possibility is to solve the problem by improving the code generated with **NOT** operator. For short, it is about allowing the compiler to correctly manage the **NOT** operator. But this solution would increase the volume of generated code. As presented in section 4.6, the main problem comes from the lack of a temporary variable. Choosing this solution would require to use the stack or another memory spot (register, heap, thread local storage, ...) to store the variable. Starting from the code presented in Table 2.7 with its decompiled view, we are looking construction of output opcodes able to manage the operator correctly.

---

<sup>14</sup><https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2018-8232>

<sup>15</sup><https://www.masm32.com/>

Source code	Compiled code
<pre> 1   .if !eax == 0 2       mov edx, 1 ; Condition is true. 3   .endif 4   nop </pre>	<pre> 0040103c 0bc0          or     eax, eax 0040103e 7505          jne   main!main+0xd (00401045) 00401040 ba01000000    mov   edx, 1 00401045 90           nop </pre>
Code:	

Table 2.7: Initial code to correct.

From a technical point of view, we can design the correction the same way the C compiler did it in figure 2.14. By saving original value on the stack, we can apply after the **NOT** operator on the register representing the value. That way, the compiled code could be generically corrected to something close to the code displayed in figure 2.9.

```

1 | .if eax == 0 ; Test first to negate eax.
2 |     push 1 ; Use stack as temporary value.
3 | .else
4 |     push 0 ; If eax is 1 it becomes 0.
5 | .endif
6 | .if DWORD PTR [esp] == 0 ; The original test by itself.
7 |     add esp, 4 ; Clean the stack from the temporary value used for negation.
8 |     xor eax, eax
9 | .endif
10| add esp, 4 ; Clean the stack, whatever the result of the test is.

```

Code 2.9: "Proposition of solution to correct the bug."

The main idea is to save the negated value into the stack in order to use it directly in the original condition. Using stack is quite convenience since this is the regular procedure to create local values for functions. But it could be critical for some architecture where the stack size is limited. Note that temporary value used must be removed from the stack whatever the result of the condition is. This is why we add "add esp, 4" to clean the stack at the end of our procedure. It is in order to keep consistency with previous values already present in the stack. Add operation is justified since the stack is architecturally *expand down*.

The decompiled version of the code in Table 2.7 follows requirement of the condition originally defined by the developer. The complexity cost is relatively small since operations on stack are something common. Of course, this proposal could be optimized for different purposes, but the idea of using assembly code is to preserve the original instruction written by developers.

### 6.3 Effective solution deployed by Microsoft

The solution adopted by Microsoft is to consider the construction of a complex expression with the **NOT** operator in a condition as an error. This one is referenced under the MASM's error code **A2154** which is specific to a *syntax error in control-flow directive*. The assembly code using such construction is no more generated. It means that it is no more possible to exploit the bug in the compiler to silently insert a backdoor in a compiled application. The error message is directly displayed by MASM when trying to compiling a code using such construction in conditions, as shown in figure 2.16.

More than correcting the bug in the compiler by Microsoft, a large audit of applications developed with that compiler should be required. Indeed, a compiler is a software responsible to generate other software. It means that other source codes have been compiled with it and these ones should be audited in order to find bogus constructions in conditions, on the first hand. If a software would use a condition using the specific pattern able to trigger the bug, this one must be updated.

Taking into account that assembly language is used by some firmware builder, drivers developers or just for tenuous part of critical code, it can be complicated to update these software since some code can potentially not

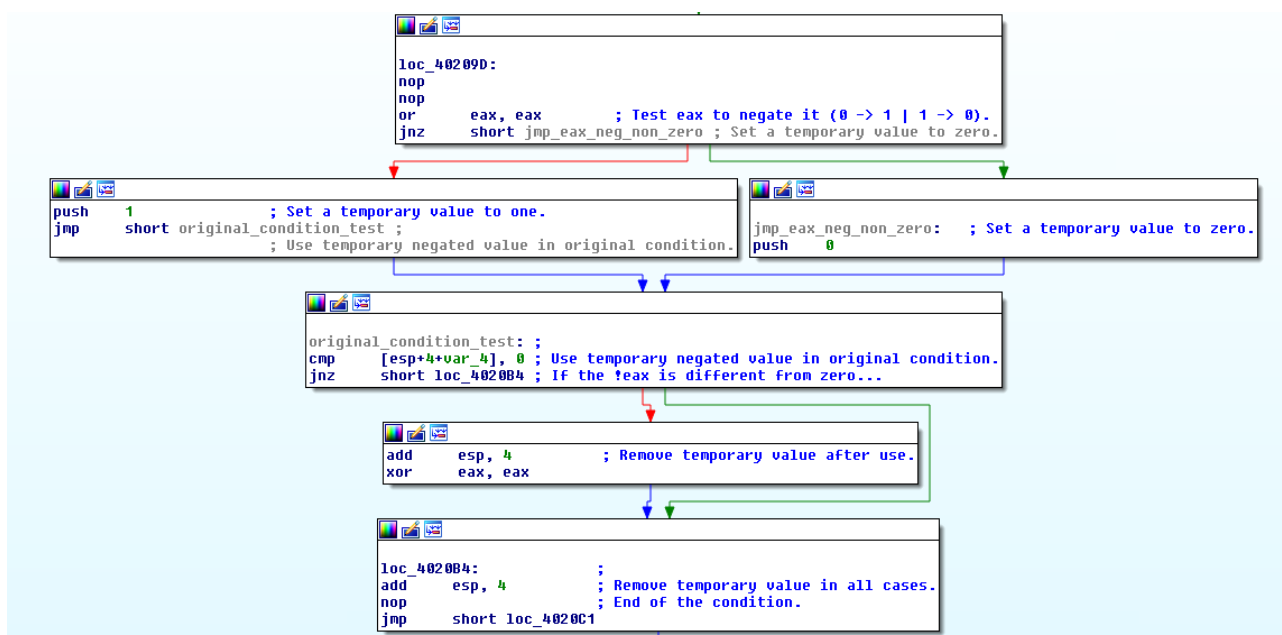


Figure 2.15: Decompiled correction for the proposed solution.

```

Microsoft (R) Macro Assembler Version 14.15.26730.0
Copyright (C) Microsoft Corporation. All rights reserved.

Assembling: pres.asm

*****
ASCII build
*****

pres.asm(23) : error A2154:syntax error in control-flow directive
pres.asm(25) : error A2154:syntax error in control-flow directive
pres.asm(27) : error A2154:syntax error in control-flow directive

```

Figure 2.16: MASM updated version is now using error A2154 to prevent such bug to be exploited.

be updatable (lack of internet connection, code written in ROM for embedded devices, small portion of code used in a math library which is part of a bigger project...). Including the fact that MASM is an old compiler, it represents a large number of software which have been made with it by a lot of people. Facing the impossibility to give an exact estimation, it must also be taken into account that the skills needed to write assembly code are rare. It is also possible to see that projects requiring assembler are often critical projects that need to execute code with high privileges (less critical or more common tasks can be written in high level languages).

Even if a CVE number has been assigned and a security update pushed, it could be hard to fix everything, especially if the source code of one software has been developed at former time by a company which has collapsed since that time. In such a case, a reverse engineering process should be made on compiled code to check that logic flow for each condition is designed to perform the task for those it has been designed.

Without getting into a conspiracy, it might be interesting to think — just for the pleasure of imagination — that a small number of highly qualified consultants or freelancers may have been aware of this bug in the

past. Working on critical projects all over the world for companies that hired them, they could very well have used this mechanism to trap the software they were delivering. The motivation may reside in various national interests, espionage or control of technology when it would not be for blatantly criminal motives. Whatever would be the motivation, if the source code was unclaimed, the crime did not need to be covered up. Otherwise, the trick allowed to create an illusion in the source code by pretending that the program worked securely even if it did not. At worst, even if the trick was discovered, the malicious developer could still plead good faith. After all, the code he had written was logical, consistent with what was expected and *a priori* flawless. How could he be expected to know about a bug that was unknown at the time? The perfect crime in itself, the ultimate backdoor...

## 7 Conclusion

### 7.1 Impacts and achievements

As explained in the state of the art, evolution of compiler's backdoor can be spitted in different periods. Everything started in 1974 with the rapport from Karger and Schell [102] who worked in the US Air Force. The idea in the rapport is publicized by Ken Thompson [115] in 1984 but this one does not publish or explain how to do it in practice. Fuzzing and evaluation of compilers are present since 1970, where Hadford [158] used a dynamic grammar to generate test data for a PL/I compiler. A complete survey can be found in [131] about history of the first fuzzing compiler methods. But we can consider as modern fuzzing the method used by Csmith [126] tool using approach presented in 1998 by McKeeman [129]. The difference is the use of several compilers to compile one single code in order to compare execution output at runtime. This approach allowed to find bugs in a larger proportion and potentially exploitable ones for compilation trapping purposes. Finally, in 2015, Bauer [137] presented an approach reusing an old bug in LLVM to write a source code able to include a backdoor at runtime. But his paper relies on shaky assumptions to justify the backdoor's operability (far-fetched source code, already fixed vulnerability, backdoor that breaks compatibility with trapped features). It will finally require our work to present an unknown bug (0-day) exploitable in a real compiler (MASM) and used to present an effective and operational trapping mechanism. This evolution is summarized in the Figure 2.17 as a timeline.

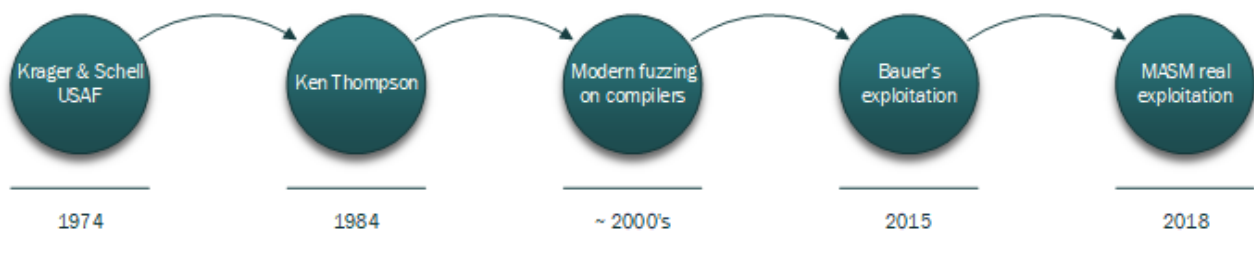


Figure 2.17: Timeline about compiler's backdoor evolution in history.

Assembler programming may not be the most popular one (even if we still find it in different projects<sup>16</sup>), but as already explained many times, it is present in some critical projects, such as firmware. And it should be recognized that transparency is not one of the cardinal virtues in this programming community. This makes assembly language a potential ideal candidate as the vector of backdoor in different projects of software development.

More generally, using a bug in a compiler to introduce a backdoor in a software is a nice and smooth manner to get access to a targeted system in the stealthiest way. This fits perfectly with the specifications of a backdoor as defined in section 2.2. Our technique of backdoor follows the four main criteria defined. First, this backdoor is perfectly stealth since it is not possible to find it from checking the logic flaw from the source code. This type of backdoor is able to bypass source code auditing. Then, this backdoor is perfectly deniable since we cannot blame people to not know about an unknown flaw in Microsoft's compiler. Moreover the logic in the code

<sup>16</sup><https://www.tiobe.com/tiobe-index/>

written is perfectly respected. In addition, as we illustrated it, the written source code is perfectly consistent with the logic expected in an officially correct condition. The persistence of the backdoor is easy to prove since it made many decades the bug was present in MASM compiler. This backdoor is operational: changing logic of a program fits with previous requirements written by Thompson [115] or by Thomas and Francillon [85]. In addition, assembly is usually used for critical code. And that backdoor is secret, at least until we publish our researches.

From all of these points, our research thus continues the evolution of what has been done previously by responding to a problem left open and identified in our state of the art in section 2. Of course, we have not exploited the potential of this backdoor in the wild either. In a way, it has remained in the laboratory, which is ethically better. In the same philosophy than Bauer [137], our work remains different since it does not require to use obfuscation to hide the trap, which makes ours perfectly justifiable. There is nothing new in the concepts used since Ken Thompson [115] in 1984. But our work made it possible for real by exploiting an existing and operational flaw in the compiler and then compiling a trapped code. Here, we exploited a zero-day vulnerability in the compiler to do the job.

## 7.2 Postmortem documentation

From a specific compiled version dedicated to a specific target to a wildly used open-source software, possibilities are endless. Undetectable for humans, perfectly justifiable for malicious developers, thin and efficient, this type of backdoor is designed for long term and high efficiency. We can wonder about how to fix the software impacted by this compiler and how to prevent ourselves from this type of problem.

One may wonder if it is possible to detect problems in source codes used with MASM compiler. Writing detection automatic test procedure is more complex than it seems. Indeed, this procedure must be calibrated to record and take care of suspicious signal or undefined states in a condition. Of course, example provided in our case is for illustration purpose but it could be possible to build many different ones for different purposes. The **NOT** operator may be used for malicious purposes or legitimate ones... Indeed, even when detecting suspicious constructions, it would be necessary to distinguish between those that exploit the bug for malicious purposes and those that suffer from the bug without knowing it — and thus have an unexpected behavior.

Detect it with a suit of dedicated tests is not easy. But, with no prior knowledge that a backdoor has been inserted, detection by a specific test procedure can be almost impossible to perform. While the source code describes what needs to be done by the processor, it does not always reflect exactly what the developer had in mind at the time. Note that, even if tests would find something tendentious, a real source code audit would be required to fix the problem. Finally, it will remain that it will be difficult to say if a dangerous line in a source code was a *bug* or a malicious backdoor inserted on purpose...

What is explained above presupposes that one still has access to the project's source code. What can we do when the source code is not available? Reverse engineering is the only solution to understand what has been compiled, but it is not a mass sport and it requires specific skills far from being earned by each developer. Note that such operation supposes that developers suspect potential bugs could be present in the software. Where such bugs come from compiler misbehavior and not from their own source code. A state of mind which is far from being common if we are not aware it could be possible. Finally, to help the researches about potentially impacted software, one should be able to know that the software has been written in assembly or that, in our case, MASM compiler has been used to craft it. In fact, if one does not have the source code or knowledge of the compiler used to compile the software, a complete reverse-engineering audit must be carried out on each of the program's condition-instructions in order to check that the logic of the program is always respected. Impossible in practice since it amounts to looking for all potential bugs in all software. It is perhaps for this reason that Microsoft did not seek to specifically alert about the problem — it might be lost in advance...

From an operation point of view, such a backdoor is almost perfect. Especially with the bug provided here, attacker has the ability to insert new possibilities in any critical condition targeted without modifying original behavior of the condition. Bonus, attacker can stop diffusion of the attack by correcting the exploited bug in the

---

compiler. It would remove the trap for new compiled version without changing targeted software's source code. Another bonus lies in the difficulty to update critical pieces of code. Most of the time, it concerns firmware and other close to hardware or kernel components. Last attacks such as Spectre [159] and Meltdown [160] demonstrate how hard it could be to update firmware in critical pieces of software. Not using such extreme examples, thinking about pieces of software in firmware, written in assembly, sometime years ago, by companies which could not exist any more or were original authors are now out of business not to say dead, could make impact of such backdoor very critical.

### 7.3 Future of this work

Of course, the goal of this work was to report this vulnerabilities so that it could be patched as soon as possible. Also, it aims to explain how to find such a bug, to make it visible to any developer and to alert about how important the damage could be if it has been exploited by malicious developers. More important, it underlines the necessity to not believe blindly that open-source projects are secure since the source code can be checked. Even if they are correctly audited by experts, compilers must be taken into account. Compiler is a milestone in the building procedure of an application and it must be not underestimated as a potential source of vulnerability. Using open-source compilers such as LLVM and others would be a better solution in the future but it would not be sufficient.

Finally, the main lesson learned is to do the job all by yourself and not by any third parties. The real problem is the disloyal developer. Whether he or she has access to secret high technology to carry out his nefarious action or not. Mastering the technology is the key point of security. In software development as much as anywhere else.

And when we cannot do it on our own, we should never forget that trust does not exclude control. Control the behavior of the program via automatic or manual tests is a true minimal requirement. It is necessary and belonging to the most basic public health. Critical part of programs should be stressed to check everything is correct, not only in perfectly and trivially defined situations but also in less defined ones. Ideally developed by people of great confidence or where the tests could be carried out by independent parties.

Perhaps one of the greatest source of pride that this work has given us, it is the feedback we received from it. Aside from the sense of vanity associated with any recognition of our work by others, the fact that we have presented our work at Zero Night [161] and at C0c0n [162] conferences may have been inspired by others. Menkhus Mark from Hewlett Packard Enterprise PSRT informed us at the beginning of 2019 that CVE-2019-6291<sup>17</sup> referenced a similar vulnerability on NASM compiler (but consequences resulted in the crash of the compiler). It is finally encouraging to see that the research is focused on checking for bugs similar to the one we helped to discover.

---

<sup>17</sup><https://nvd.nist.gov/vuln/detail/CVE-2019-6291>

## 7.4 Research contributions

### Contribution 1: Protection at development level: backdoor in compilers

- ☞ State-of-the-art about the different backdoors.
  - ✍ We have made a survey of the different definitions of backdoor to propose a generic one.
  - ✍ We have made a technical and exhaustive state of the art of the different backdoor methods, especially for compilers.
  - ✍ The state-of-the-art showed that if there had already been works on the subject, none was able to release a credible version of the attack.
  - ✍ The initial hypotheses are often strong (modified compiler, exploitation of an existing and already corrected bug, writing strange source code to justify even stranger constructions).
  - ✍ The fact that there was no credible attack contributed to consider this type of attack as negligible (although the consequences are unanimously recognized as potentially dramatic).
- ☞ We found a bug in the MASM compiler.
  - ✍ It turns out that we found a way to improve it into an exploitable vulnerability.
  - ✍ This one has been recorded as CVE-2018-8232 with a responsible disclosure by us.
  - ✍ The vulnerability is easy to trigger, discreet, stealth and already present for decades.
- ☞ We proposed a backdoor insertion method with a well-known compiler, using an unknown vulnerability, to insert backdoors in a stealthy way.
  - ✍ Our method is very difficult to detect when reading the source code (unless one knows what the backdoor looks like).
  - ✍ We have created a backdoor in a source code by fulfilling the requirements set by previous work [115].
- ☞ Our research has resulted in many other achievements.
  - ✍ Microsoft has fixed the bug in the compiler so that it is no longer possible to create such backdoor.
  - ✍ This may have helped to raise awareness of the risk carried by this type of vulnerability.
  - ✍ All software compiled with MASM can carry this type of vulnerability and it is very complicated to correct it today.
  - ✍ It may have given ideas to other researchers who have found similar results with different compilers since our publication (CVE-2019-6291).

THIS PAGE INTENTIONALLY LEFT BLANK

---



## Chapter 3

# Protection of analyzed executable files: malware

### 1 Introduction

In the defense of systems against offensive threats, there is of course the analysis of malware. That way, the analysis of malware is part of the actions specific to the defense of computer systems. This analysis can be carried out manually by a human operator (an analyst in an antivirus company, for instance) or automatically via a set of dedicated software. In both cases, it appears that the objective is to document the actions of a given program in order to be able to classify it as malicious or benign code [163]. Such a classification is hard task [164, 165].

Designing countermeasure or technologies that can withstand a high level of sophistication would be possible merely by understanding the precise inner workings of such malware. Malware analysis is the way to achieve this understanding [166]. Initially, with the help of disassemblers, decompilers, etc. analysts inspected the malware's binary and code to observe its functionality. This approach, which is also referred to as static analysis, became far more arduous and complex with the rise, the evolution of code obfuscation tactics [167, 168, 169, 170] and other evasion techniques targeting static analysis e.g. opaque constants [171], packers [172], etc. As a resolution, a promising approach that was adopted was dynamic analysis in which the basis of analysis and detection is what the analyzed file does (behavior) rather than what the file is (binary and signature) [173]. Put differently, in dynamic analysis, an instance of the suspected program is run and its behavior is inspected in run-time. This approach would prevent the obstacles posed by the static analysis evasion tactics. To thwart these efforts, however, malware authors turned to a new category of evasion tactics that targeted dynamic analysis.

For obvious reasons, malware developers do not want to see their code to be identified as malicious. Not only this prevents it from being executed, but it also reduces its spread and the benefits it brings to its creator (money, stolen data, undesired and harmful effects on the infected systems, and so on). This is to avoid detection and thus extend the *benefits* that malware authors seek to hide the real purpose of their software. While the latter ought to privilege design approaches that avoid detection patterns, it may be more cost-effective to directly attack the tools used to detect malware.

It goes without saying that the protection of an information system also lies in its ability to identify a potential threat as malicious. Being able to neutralize the sensors of defense systems represents a blank check for malicious programs. Thus, knowledge of the various techniques for bypassing and neutralizing analysis sensors is an approach to better understand the malicious threat. In addition to the fact that seeking to neutralize or evade detection constitutes in some cases a malware signature, it also allows us to design analysis tools to be effective and resilient against such threats.

Another point is relevant from a research point of view. This is the ability to imagine new techniques capable of hijacking the security of analysis tools. Why such an approach? Apart from the fact that achieving a state

of the art makes it possible to do so (or at least, it makes easier), it makes possible to progress on two axes. On the one hand, it is possible to anticipate a future trend that does not yet exist and which could be used by malware authors. It is by anticipating threats and adopting the approach of malware writers that we can force the reduction of their capacities. The result is beneficial on two levels: the first deprives them of potential means of action while the second corrects and improves the analysis tools so that they no longer fall into the identified traps.

On the other hand, our research may discover a mechanism that is new to us but not necessarily new to the attackers. More directly, a mechanism used by malicious programs can be discovered independently of any malware analysis. This feature offers an even wider range of possibilities. On the one hand, because programs using this new type of mechanism can potentially be classified as malicious without any prior suspicion that they would have been known as malicious. There is of course the possibility of false-positives with the detection of mechanisms used fortuitously and without harmful purposes. But considering the specificity of the analysis tools, this is a relatively low probability. And even if, such detection procedure sought to lead to an in-depth analysis. On the other hand, security software that would have been abused by this mechanism can be corrected. Thus, they are no longer victims and they regain the ability to detect, once again, the potentially malicious behavior of the objects they analyze.

This is this *offensive* approach that we are illustrating in this chapter. We are first illustrating a state-of-the-art about malware dynamic analysis evasion techniques in section 2. After a few preliminaries to set the approach (section 2.1), it is divided into two parts. One about manual dynamic analysis evasion (section 2.2) and the second about automated dynamic analysis evasion (section 2.3). Put differently, section 2.2 will deal with the different techniques to detect or escape a debugger when section 2.3 will present the different tactics to do the same in a virtualized environment. Once the state-of-the-art has been established in these two areas, a new contribution should be made in each of them. On the one hand, we will present in section 3 a new operational detection method for Microsoft's Windbg debugger by exploiting an interpretation bug in its decompilation engine. On the other hand, we will present in section 4 our latest work on a method able to detect in a generic way several types of analysis environment (*Dynamic Binary Instrumentation* (DBI), debugger and VM).

We hasten to note that all of this work was done in collaboration with other researchers on different publications. Significant portions of the material written in this chapter are taken from our published articles (with some simplifications, adaptations and improvements whenever necessary). Research about this subject — when it is not confidential — is neither a solitary exercise nor an exercise limited by country's borders. In this sense, the state of the art is taken from an article we published in the magazine ACM Computing Surveys [174] with Afanian Amir, Niksefat Salman and Sadeghiyan Babak from APA Research Center, Amirkabir University of Technology in Iran. The two other sections bearing our contribution to the researches were realized with Plumerault François [175, 176], master degrees student in our research laboratory in France. On a personal level, it is an unspeakable pride to have had the opportunity to evolve in an international context and with people of great quality, both in scientific and human.

## 2 State of the art

### Resume 9:

- ☞ Definition and classification for a taxonomy about evasion techniques from analysis environments.
  - ☞ We have written a state-of-the-art about manual evasion techniques in the context of debuggers.
  - ☞ We have written a state-of-the-art about dynamic evasion techniques in the context of sandboxes (virtual machine).
  - ☞ Each time, we make a distinction between detection dependent and independent evasion techniques.
  - ☞ We propose a brief survey on countering malware evasion.

### 2.1 Preliminaries

#### Resume 10:

- ☞ In this subsection, we expose many definitions of vocabulary words used in current chapter.

In this section, we define several keywords that we extensively use throughout the paper. Some terms specifically need explanation since the passage of time has overloaded or altered their pervasive meaning. Moreover, it is not uncommon to find in literature various terms that cover the same reality. The passage of time, not to say the different trends, are sometimes responsible for this achievement.

#### Sandbox

Traditionally, *sandbox* was developed to contain unintended effects of an unknown software [177]. Hence, the term sandbox meant an isolated or highly controlled environment used to test unverified programs. Due to similarities in nature, virtualized machines and emulated environments are often seen to be called sandbox. But, in this chapter, we use the term sandbox to refer to the contained and isolated environments that analyze a given program *automatically*, without the involvement of a human. The dividing line for us, in this paper, given the trend of the industry, is the autonomous nature of the system. For instance, for a modified virtualized machine such as *Ether* [178], we consider the term debugger rather than sandbox.

Generally speaking, the sandbox should be seen as a generic set of technical tools to analyze a running process. These technical tools are usually composed of emulators and virtual machines, but not only. One might think about real machines — connected on a distinct network partition (e.g. "*air-gap*") — which are resetting after each analysis. Whatever is the sandbox's shape, the main idea being automation and the non-intervention of a human in the loop. Some debuggers can be automated (via a scripting system) to allow an automatic analysis (for example with the Mathieu Tarral's tools [179]). Nevertheless, debuggers are still tools where human action is privileged, that is why they partially fit into the sandbox framework. Figure 3.1 illustrates these remarks.

#### Evasion and Transparency

In literature, *evasion* constitutes a series of techniques employed by malware in order to remain stealth, avoid detection, or hinder efforts for analysis. For instance, a major evasion tactic as we will discuss is fingerprinting [180]. With fingerprinting, the malware tries to detect its environment and to verify if it is residing in a production system or an analysis system.

In the same level, one major strategy to counter the evasions is to hide the clues that might expose the analysis system. A system is more *transparent* if it exposes fewer clues to malware [181].

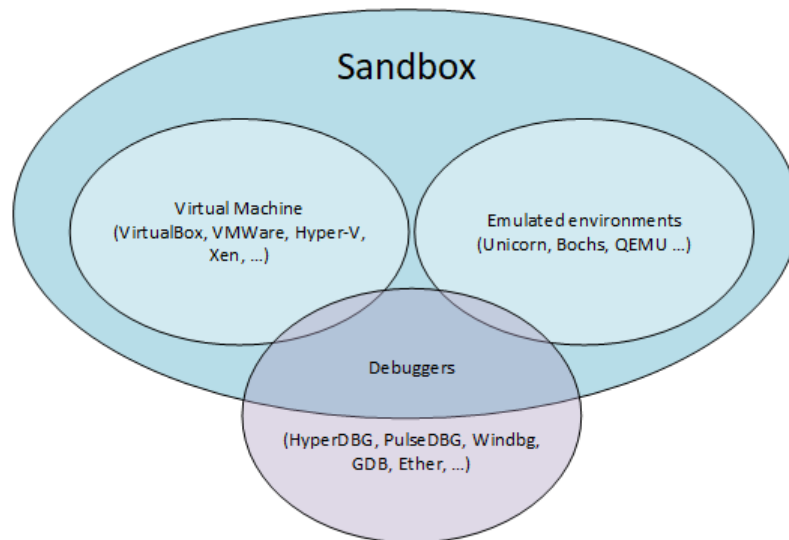


Figure 3.1: Representation of the sandbox concept with different technical tools.

### Manual vs Automated

*Manual* and *automated* analysis are two major terms which form the basis of our classifications. The *manual analysis* is when the analysis procedure is performed by a human expert with the help of dedicated tools (for instance with a debugger). The *automated analysis*, on the other hand, is the procedure that is performed automatically by a machine or software, also known as sandbox.

### Detection vs Analysis

Previously, there was no need for defining these two terms. *Detection* would simply refer to the process of discerning if a given file is malicious or not while *analysis* would refer to the process of understanding how the given malware works. Today, however, this dividing line is blurry. The reason is that the role of automated analysis tools such as sandboxes is now extended. In addition to reporting on malware behavior, sandboxes are now playing their role as the core of automated detection mechanisms [181]. In this chapter, we follow similar concepts when using the word of *analysis*. For *manual analysis* it would mean understanding the malware behavior [182] and for the automated one, additionally, it can mean detection.

### Static vs Dynamic

There are two major types of analysis: static and dynamic. *Static analysis* is the process of analyzing the code or binary without executing it. *Dynamic analysis* is the process of studying the behavior of the malware (API, system calls, files touched, network activity, etc.) at the run-time. Both types of analysis can be performed either manually or automatically.

In this chapter, our focus is on dynamic analysis and how malware tries to prevent or evade such analysis. Why such a choice? Quite simply because static analysis only allows to process a small number of samples (but with a certain efficiency). It is difficult to give an accurate (if not credible) estimate of the number of malware (new or variants of old ones) that is created every day. It is difficult because of the lack of real capacity to collect such information. Nevertheless, some antivirus vendors agree to publish some statistics on the subject, based on their own detection rate of submitted samples per day [183, 184]. Statistics show an increase in submissions, on the one hand due to the production of malware authors, but also due to the always increasing capacities of samples collection (and the multiplication of devices protected by antivirus tools) from antivirus vendors.

As a result, antivirus publishers obviously do not have the human resources to handle every program submitted to their services. That is why they shift the analysis to automated tools. And this is usually the first line for analysis and detection from antivirus vendors. If a malicious program can quickly and easily escape

this first detection, the benefit is immediate because it avoids further analysis (which could actually expose it). Of course, it would be possible to automate the static analysis procedures. But obfuscation techniques are sophisticated enough to compromise many of these analyses. To bypass obfuscation, we could increase the sophistication of analysis tools and this would represent a certain calculation cost per program to be analyzed... Moreover, considering the large number of malware to be analyzed per day, it is not possible to spend "too much" computing time on each program received. Therefore, the simplest (and equally effective) solution is to run the program in a controlled analysis environment. This is where dynamic analysis comes from. Of course, dynamic analysis cannot see everything and there are different levels of analysis. From the most generic and automated tools (sandbox) to tools used by humans (debuggers).

This is the main reason why we choose to focus on dynamic analysis. Since it is usually the first analysis produced by antivirus vendors, this is therefore the analysis that should be bypassed as quickly as possible for malware, at all costs.

### Category, Tactic, and Technique

Throughout this paper, we use the terms *category*, *tactic*, and *technique* that are the basis of our classification. *Category* of evasion is our high-level classification. Each category has the goal of evasion with a specific attitude for achieving it. This attitude is highly correlated with the efficacy of the evasion and is pursued by the *tactics* under each category. Tactics, in other words, are the specific maneuvers or approach for evasion with the specified attitude of its parent category. Finally, *techniques* are the various practical ways of implementing those tactics.

## 2.2 Manual Dynamic Analysis Evasion

### 2.2.1 Evaluation to manual dynamic analysis evasion techniques

#### Key Point 3.1:

☞ We call "Manual Dynamic Analysis" the set of analysis tools that require manual control.

☞ More directly, it corresponds to debuggers.

As cited in the introduction, due to the employment of code obfuscation, packers, etc. static analysis of malware has become a daunting task. To prevent issues and limitations of this approach, analysts opt for dynamic analysis in which malware's behavior is inspected at the run-time and often with the help of debuggers. We view this approach which is aided by debuggers, under the term "manual dynamic analysis". This way of examination has two major benefits:

- It relieves us from the impediments inflicted by packers, polymorphism, etc.
- It explores the activities that manifest themselves only in run-time [185] such as the interaction of the program with the OS [173].

The corresponding evasion tactics to this approach involves the set of employed techniques within malware code with the goal of inhibiting, distracting or evading the analysis process. These measures include approaches such as detecting the presence of analysis tools on the system (e.g. Wireshark, TCPDump, etc.) or detecting virtual-machines as a sign of analysis environment. But, the majority of manual analysis evasion techniques are targeted toward debuggers which are the primary tools of manual dynamic analysis.

Chen et al. ran a study on 6,222 malware samples to assess changes in malware behavior in the presence of virtualization or an environment with debuggers attached. They discovered in presence of a debugger (not in a virtualized environment) around 40% of malware samples exhibited less malicious behavior [186]. The same study revealed that when malware are executed in a virtualized environment, merely 2% of the samples exhibit malicious behavior. This shows that even though there are similar tactics to evade both the automated and manual dynamic analysis (fingerprinting), the techniques are different. And sheer detection of virtualization or emulated environment does not suffice to evade manual analysis since more and more production systems

are running on virtualized machines. Our coverage in this sections includes the traditional (and still relevant) anti-debugging techniques to the more advanced, recent AI-powered techniques.

Technically, anti-debugging references one or more techniques able to prevent manual dynamic analysis (debugging) or reverse-engineering. It must be noted that the presence of such techniques in a program does not necessarily mean we are facing a malicious behavior. Indeed, historically, anti-debugging techniques — and more generally any obfuscation ones — were legitimate practices conducted by developers to protect intellectual property their own software. Malware authors just use these techniques for other purposes...

For the sake of brevity, we will focus our survey on the most cited manual dynamic analysis evasion (anti-debugging) tactics and corresponding techniques [173, 185, 187, 188, 186, 189, 190, 182, 191, 192, 193] that are more relevant to the context of malware. For the sake of consistency, we propose to evaluate the different methods presented according to five detailed criteria<sup>1</sup>:

- **Complexity:** The difficulty of incorporating and implementing the technique within malware code.
- **Resistance:** Resistance pertains to the difficulty level of counteracting the evasion technique.
- **Pervasiveness:** Even if it is hard to evaluate the popularity of each tactic, we try to provide a fair view of this metric based on other works [187, 194] in addition to our own observations and experience in the field.
- **Efficacy-Level:** Due to the diversity of debuggers' nature, we appoint 1 to refer to user-level debuggers, 2 to refer to kernel-level debuggers, 3 to refer to virtualization-based/emulation-based debuggers, and 4 to refer to bare-metal debuggers.
- **Countermeasure:** We briefly note how the anti-debugging technique could be circumvented.

### 2.2.2 A Briefing on Debuggers

Since we are dealing with anti-debugging techniques, it makes sense to have a brief introduction on debuggers. Sikorski states, "A debugger is a piece of software or hardware used to test or examine the execution of another program" [182]. Technically speaking, the debugger has not the ability to execute itself instruction per instruction a given program. This is the aim of an emulator (such as Bochs [195] or Unicorn [196]). Instead of, among the core functionality of debuggers [197], there are several features such as stepping through the code one instruction at a time, pausing or halting it on desired points, examining the variables, etc.

To provide each of the mentioned functionality, debuggers rely on different tactics and each tactic is often aided with specific hardware or software provisions that inevitably result in subtle changes on the system. For instance, to provide the pausing capability, one of the debuggers' tactics is to set breakpoints in the debugged process. Using breakpoints is further assisted with either special hardware [198] (e.g. DR Registers of CPU and specific opcodes such as breakpoint instruction [199]) or software with specific API to access/alter them [200, 201]. Single stepping, as another instance, is made possible by triggering exceptions in the code which is aided by the trap flag [202]. The trap flag is inserted in the context of one of more debugged thread.

Based on the needed functionality, debuggers are implemented following different approaches [203]. Generally speaking, we can make the distinction between debuggers which run in the context of the debugged target (user-mode or kernel-mode context) or in a Virtualization context. Among the most popular debuggers, we can cite:

- **User-mode and Kernel-mode debuggers:** *OllyICE*, *OllyDbg* [204], *LLVM* [205], *Radar2* [206], *GDB* [207], *Visual studio debugger* [208], *WinDbg* [209] ;
- **Virtualization-based debuggers:** *Ether* [210], *BOCHS*, [211], *HyperDBG* [212], *Winbagility* [213], *VirtICE* [214] and more recently, the bare-metal debuggers such as *MALT* [193].

---

<sup>1</sup>The description given is intended to be brief. More details are given in the published article [174].

Each of the designs, yield different levels of transparency. Kernel-level debuggers are more transparent than User-level debugger and provide more detailed information as they operate in ring 0 (same level of privilege as the operating system). Virtualization/emulation-based debuggers have an even higher level of privilege than kernel debuggers since the operating system in this setting is running atop the simulated (virtualized/emulated) hardware [210, 211] and consequently are more transparent to malware. Finally, the bare-metal design such as MALT, which often rely on System Management Mode (SMM) offers the highest level of transparency against which many of the traditional anti-debugging techniques lose efficacy.

### 2.2.3 Proposed Anti-Debugging Classification

#### Key Point 3.2:

- ☞ There are two anti-debugging strategies for malware to perform automated analysis evasion: *detection-dependent* and *detection-independent*.
  - ☞ *Detection-dependent*: malware tries to detect a specific environment by exploiting a specificity of this one.
  - ☞ *Detection-independent*: malware tries to detect any environment analysis without targeting any specificity.

We propose two major categories, as malware's initial anti-debugging strategies which are similar to the categories of automated analysis evasion: *detection-dependent* and *detection-independent*. In the context of *detection-dependent*, a malware probes for detecting its environment in order to escape analysis. Put differently, a successful evasion is subjected to detecting or finding signs of an analysis environment. This is possible since debuggers were originally designed to debug legitimate software [185], who therefore had no reason to seek to detect them. This is why there has been no stealthy countermeasure provisioned by them. In addition, we often have to instrument the system with necessary tools which obviously results in a wide spectrum of traces in different levels of the system [186, 194]. Looking for the presence of a debugger by analyzing the system for such traces is one major strategy that malware utilizes to detect an analysis environment.

Unlike the previous category which tried to detect or infer debugger's presence or analysis environment, tactics of this category do not rely on the detection of the analysis system. They do not try to detect whether the system on which they are has a debugger attached to it or not. The malware operates the same way regardless of their target system. We will have a survey on the tactics of this category in the following.

For the sake of brevity, in the context of a detection-dependent strategy for evasion, malware incorporates techniques to explicitly detect its execution environment. A different strategy is detection-independent in which the malware behaves the same, regardless of its execution environment, but consequences are to escape from the analysis environment. In other words, to evade analysis, malware does not have to detect the execution environment.

We elaborate a classification of each anti-debugging tactic based on fact that the tactic used is detection dependent or independent. It is certainly an arbitrary classification, but it allows us to define the goals behind the techniques used. If we divide our analysis between manual and automatic dynamic analysis tools, it is necessary to subdivide within these tools two approaches (dependent or independent detection) as given below in the different tactics.

### 2.2.4 Detection-Dependent Evasion

#### 2.2.4.1 Tactic 1. Fingerprinting

Fingerprinting is the malware's endeavor to spot, find, or detect signs that attest the presence of an analysis environment or debuggers. Fingerprinting is the most common evasion tactic regarding both manual and automated analysis. However, the techniques are different. In contrast with the automated analysis evasion, the majority of fingerprinting techniques aimed at evading manual analysis involves fingerprinting the environment



to detect the presence of debuggers [186]. Major fingerprinting techniques used by malware include the following:

- *Analyzing Process Environment Block (PEB)*. (*PEB*) is a data structure that exists per process in the system and contains data about that process [215, 216]. Different sections of *PEB* contain information that can be probed by malware to detect whether a debugger is present. The most obvious one is a field inside *PEB* named *BeingDebugged* which can be read directly or as Microsoft recommends — and malware like *Kronos* [217] or *Satan RaaS* [218] implement — through the specific APIs that read this field, i.e. `IsDebuggerPresent` [219] or `CheckRemoteDebuggerPresent` [220] to check if the debugger resides in a separate and parallel process. Implementation of this technique is of trivial complexity which can be countered through alteration of *BeingDebugged* bit [221] or API hook [188]. Collectively, anti-debugging tactics relying on *PEB*, constitute the majority of anti-debugging techniques observed in malware [187].
- *Search for Breakpoints*. To halt the execution, debuggers set breakpoints. This can be accomplished through hardware or software techniques. In hardware breakpoints, the breakpoint address, for instance, can be saved in CPU DR registers. In software breakpoints, the debugger writes the special opcode *0xCC* (*INT 3* instruction) into the process which is specifically designated for setting breakpoints. Consequently, the malware, if spots signs of these breakpoints, presumes the presence of a debugger. This can be accomplished through a self-scan or integrity check, looking for *0xCC* value or using `GetThreadContext` [222] to check CPU register [190]. The latter technique has been employed by malware such as *CIH* [223] or *MyDoom* [224]. This technique is the second most observed anti-debugging technique in malware’s arsenal [187] and it is simple to implement (less than 10 lines of assembly code often suffices). Countering this category of tactics, however, is not trivial. In the case of software breakpoints, for instance, the debugger has to keep and feed a copy of the original byte that was replaced by *0xCC* opcode [221]. Another solution is to update the output of `GetThreadContext` function whenever this one is called. In the output structure returned, it is possible to edit the value of the register directly via the debugger, that way avoiding any detection.
- *Probing For System Artifacts*. From installation to configuration and execution, debuggers leave traces behind in different levels of the OS, e.g. in the file system, registry, process name, etc. Hence, malware can simply look for these traces. `FindWindow` [225] function or any technique enumerating all processes running in the current session [226] are a couple of APIs that *shcnhdss* [227] exploited to detect debuggers. The malware, for instance, can give the name of debuggers as the parameter to `FindWindow` to verify if this process is present in the system or not [191, 228].

Most often, simple anti-debugging techniques are defeated with trivial complexity. The countermeasure to this category of tactic is randomizing the names or altering the results of the aforementioned query through simple API hooks. Even though attributed to one of the anti-debugging techniques in literature, we have rarely observed it in the wild.

A similar technique to note here is called *parent check*. Ordinarily, applications are executed either through double clicking of an icon, or execution from command line, the parent process ID of which is retrievable accordingly. It would be a pronounced sign of debugger if the examined parent process name belongs to a debugger or is not equivalent to the process name of *explorer.exe* [190] (or command line process). One straightforward technique is using `CreateToolhelp32Snapshot` [229] and checking if the parent process name matches the name of a known debugger [188]. A malware such as [227] uses such a technique [187]. Countering this technique would be skipping the relevant APIs [194]. Our observations demonstrate few utilization of this technique [187]. Another technique for fingerprinting for system artifacts is to use the `NtQuerySystemInformation` [230] function. Stored in *ntdll.dll*, `NtQuerySystemInformation` is a function that accepts a parameter which is the class of information to query [190]. Defeating this technique without using hook *ntdll* (which could lead to stability issues) requires to manage it at kernel level. For instance, *vti-rescan* [231], *Wdf01000.sys* [232], and *Inkasso trojaner* [233] are some examples that leverage this technique.



- *Timing-Based Detection.* Timing-based detection is among the most efficacious fingerprinting techniques for inferring debugger’s presence. Adroit employment of this technique reliably exposes the presence of a debugger and circumventing them is an arduous task.

The logic behind this timing-based detection follows malware authors’ presumption that a particular function or instruction set, requires merely a minuscule amount of time. Thus, if a predefined threshold is surpassed, malware would infer the presence of a debugger or analysis environment. Timing-based detection can be carried out either locally with the aid of local APIs (`GetTickCount` [234], `QueryPerformanceCounter` [235], etc. [236]) or CPU `rdtsc` (read time stamp counter) instruction. Note that it can be performed by inquiring an external resource through the network [237] to evaluate the timing. Local timing is simple to employ and difficult to circumvent. Countering timing-based detection conducted with the aid of external resource (using NTP or tunneled NTP), however, is still an open problem [193]. *W32/HIV* [238], *W32/MyDoom* [224], *W32/Ratos* [239] are infamous malware that are known to have exploited the timing discrepancies.

#### 2.2.4.2 Tactic 2. Traps

Not to be mistaken with the trap flags, we define this category of tactics as “traps”. Following this tactic, the malware provisions codes that when traversed or stepped through by a debugger, production of specific information or lack thereof would help the malware to infer the presence of a debugger. These inferences mostly rely on exploiting the logic of the system (e.g. *SEH* Exception handling [240]). Many techniques fit this category and we elaborate a couple of them here.

One technique used by malware to fool a debugger in order to disclose cues of its presence is using specific instructions and exploiting the logic of how these instructions are handled. The handling is performed through a Structured Exception Handling (*SEH*<sup>2</sup>). For instance, Max++ malware [242] embeds “`int 2Dh`” instruction within its code. According to exception handling documentation [240, 241], when this instruction is executed, in a normal situation i.e. absence of debugger, an exception is raised and malware can handle it via a try-catch structure (which defines a structured exception handler). However, if a debugger is attached, this exception will be transferred to the debugger first rather than the malware; the absence of expected exception is the logic that the malware entertains to deduce the presence of a debugger. Malware may employ other techniques to lay their traps such as embedding specific instruction prefixes [191], or other instructions such as `interrupt 0x41` [190]. These techniques are of low complexity and can be implemented with less than 20 lines of code (in assembly). One way of countering these traps requires debuggers to skip these instructions ([243] in the case of Windbg). According to our survey, utilizing traps is a fairly common approach [187].

#### 2.2.4.3 Tactic 3. Debugger Specific

Debugger specific evasion exploits vulnerabilities that are exclusive to a specific debugger. These vulnerabilities are difficult to discover, but simple to put into action. A famous instance pertains to *OllyDBG* [192]. Regular versions of this debugger have a format string bug which can be exploited to cause it to crash by passing an improper parameter to `OutPutDebugString` function [244]. Another pervasive-at-the-time technique was related to *SoftICE* debugger which was susceptible to multiple DoS attacks because of two vulnerable functions [245]. Malware like [246] that exploited these vulnerabilities would cause the *bluescreen of death*. *SoftICE* is no longer supported and the *dll* file that caused the vulnerability in *OllyDBG* is now fixed. But the idea still remains, if a vulnerability within a specific debugger is discovered, exploiting it would be a potent anti-debugging technique.

#### 2.2.4.4 Tactic 4. Targeted

The last tactic of the detection-dependent category is targeted evasion. Through this tactic, the malware ciphers its malicious payload with a specific cipher key. This cipher key, however, is chosen to be an attribute or variable that could be found only on its target system. This key could be the serial number of a component

---

<sup>2</sup>One can refer to [240, 241] for more information about exception handling.

in the target system, specific environment setting, etc. The targeted tactic is considered as one of the most advanced analysis evasion tactics and it is effective against all kinds of debuggers from user mode to bare-metal. Countering the targeted tactic inherently is an arduous task and we will discuss why as we proceed. Following are two techniques for using the targeted tactic.

- *Environment Keying.* With this technique, the malware author encrypts the payload with a key that is possible to derive merely from the target environment. This could be a specific string in the registry or the serial number of a specific device. Gauss [247] and Ebowla (a framework) [248] are instances that use this technique to prevent their payloads from being analyzed. An obvious way to countering this technique is brute-forcing the payload to come up with the cipher key. And depending on the cryptographic algorithm and key complexity used, it might not be feasible. Another glimpse of hope for combating this technique is brought by path exploration techniques such as [249, 250, 251, 252]. But these techniques have their own limits and may not be effective all the time. In fact, they lose their efficacy when facing the next generation of AI-powered targeted technique.
- *AI-Powered Keying.* Attacks on Deep Neural Networks (DNN) are on the rise, so are cases of abusing them [253]. A recent instance relevant to this section involves IBM researchers who came up with a proof of concept about how malware authors can benefit from AI to craft extremely evasive malware [254]. They developed DeepLocker [254] which ciphers its payload with a cipher key. The crucial difference, though, is that DeepLocker uses AI for the "trigger condition". Using the neural network, DeepLocker produces the key needed to decipher the payload. This technique leverages the black-box nature of DNN to transform a simple *if this, then that* condition into a convolutional network. And due to the enormous complexity of neural networks, it becomes virtually impossible to exhaustively enumerate all possible pathways and conditions [255].

## 2.2.5 Detection-Independent Evasion

### 2.2.5.1 Tactic 1. Control Flow Manipulation

Through this tactic, malware exploits the implicit flow control mechanism conducted by Window operating system. To implement these techniques, malware authors often rely on callbacks, enumeration functions, thread local storage (*TLS*), etc. [190, 194]. There are several noteworthy techniques here and each deserves a brief introduction.

- *Thread-hiding:* A simple and effective technique is thread-hiding which if used, prevents debugging events from reaching debugger. Microsoft has provisioned special APIs to this end [256, 257]. This technique uses `NtSetInformationThread` function to set the field `HideThreadFromDebugger` of `ETHREAD` kernel structure [192]. This is a powerful technique, simple to implement and which can be countered by hooking the involving functions. *LockScreen* [257] is an instance known to have utilized this technique.
- *Suspending Threads:* A more aggressive way malware might step into is striving to halt the process of the debugger to continue its own execution with little trouble. This technique can be effective against only user-mode debuggers and it can be carried out by leveraging `SuspendThread` [258] (internally calling `NtSuspendThread` from `ntdll`) [187]. Suspending threads is one of the anti-debugging techniques that *Kronos* banking malware used in its arsenal [217].
- *Multi-threading:* Another technique to bypass debuggers and to continue the execution is multi-threading. One way to implement this tactic is utilizing `CreateThread` function [259]. A malware that is packed often spawns a separate thread within their process to perform the decryption routines [192]. However, there are instances where malware executes a part of its malicious code through a different thread outside the debugger. *McRat* [260] and *Verteernet* [261] have incorporated such a technique. Countering this technique is tricky. One way is to set breakpoints at every entry point [262, 245] of executed function by a thread.

- *Self-debugging*: Self-debugging is an interesting technique which prevents the debugger from successfully attaching to the malware [263]. By default, each process can be attached to merely one debugger. Malware such as *ZeroAccess* [264] exploits this by running a copy of itself and this one attaches to it as a debugger. Hence preventing another debugger to own it. There are several ways to implement this tactic, for instance by leveraging `DbgUiDebugActiveProcess` or `NtDebugActiveProcess` undocumented functions.

Collectively, control-flow manipulation techniques are not much common among the samples we have observed.

### 2.2.5.2 Tactic 2. Lockout evasion

In Lockout tactic, malware continues its execution by impeding the debugger operation without having to look for its presence. One way is to opt for `BlockInput` function [265] as in the case of *Satan RaaS* [218], through which malware prevents mouse and keyboard inputs until its conditions are satisfied. Other techniques involve exploiting a feature in Windows NT-based platforms that allow the existence of multiple desktops. Malware such as *LockScreen* [257] with the help of `CreateDesktop` [266] followed by `SwitchDesktop` [267] can select a different active desktop and continue its working unbeknownst to the debugger [245]. This tactic is mostly effective against traditional debuggers.

### 2.2.5.3 Tactic 3. Fileless malware

Fileless malware, non-malware, and occasionally called Advanced Volatile Threats (*AVT*), are among the latest trend in the evolution of malware [268, 269]. In contrast to all prior existence of malware, fileless malware requires no file to operate and they purely reside in memory and take advantage of existing system tools e.g. *PowerShell* [270].

The purpose of such attacks is to make the forensics much harder. Generally speaking, analyzing malware is about to analyze its executable file. And in fileless malware, there is no executable to begin with. In some cases such as *SamSam* [271], the only way to just retrieve a sample for analysis would be to catch the attack taking place live. These attacks inherently are not easy to conduct; but, with the help of exploit-kits, those ones are more readily available. A 2018 report by McAfee shows a 432 % increase of fileless malware in 2017 [272] and projected to constitute 35 % of attacks in 2018 [273].

Fileless tactic complicates the analysis by a debugger because there is no executable file to launch. Nevertheless, the analysis with a debugger remains possible, under certain conditions. On the one hand, it must be understood that to see malicious code to be run, a thread must execute it, usually within a process. If it is possible to get hold of this process contaminated by the attack running the fileless malware, it may be possible to attach a debugger to it. On the other hand, if it is possible to replay the attack allowing to execute the fileless code, it becomes potentially possible to debug the attack itself and, *in fine*, the malicious code.

## 2.2.6 Resume about manual dynamic analysis evasion techniques

With the aid of debuggers, the analyst can overcome many hurdles and limitations of static analysis. However, new trends and real-world scenarios in which the vendors face thousands of new malware samples daily demands a more agile approach — beyond the capabilities of manual dynamic analysis. In the next section, we discuss the emergence of automated dynamic analysis approach and sandboxes as a response to these challenges and will further elaborate on malware's tactics to thwart them. Table 3.1 summarizes our survey of manual dynamic evasion techniques.

Cat	Tactic	Criteria	Technique	Complexity	Resistance	Countermeasure Tactic	Pervasiveness	Malware Sample	Efficacy	
Detection-Dependent	Fingerprinting	Reading PEB	IsDebuggerPresent	Low	Low	Set the <i>BeingDebugged</i> flag to zero	Very high	[217, 218]	1	
			CheckRemoteDebuggerPresent	Medium	Low	Set <i>heap_growable</i> flag for flags field and <i>forceflags</i> to 0			1	
			NtGlobalFlags	Low	Medium	Attach debugger after process creation			1	
		Detecting Breakpoints	Self-scan to spot INT 3 instruction	Low	Medium	Set breakpoint in the first byte of thread	High	[223, 224]	1, 2	
			Self-integrity-check	Low	Medium	Reset the <i>context_debug_registers</i> flag in the <i>contextflags</i> before/after Original <i>NtGetContextThread</i> function call			1, 2	
		System Artifacts	FindWindow	Low-High	Low-High	Randomizing variables, achieve more transparency	Medium	[227]	1, 2, 3	
		Mining NTQuery Object	FindProcessFindFirstFile	Medium	High	Modify process states after calling/skipping these API	Medium	[232, 233, 231]	1, 2	
		Parent Check	ProcessDebugObjectHandle ProcessDebugFlags ProcessBasicInformation	Medium	Medium	API hook	Low	[227]	1, 2	
		Timing-Based Detection	GetCurrentProcessId CreateToolhelp32Snapshot Process32First Process32Next	Low	High	Kernel patch to prevent access to <i>rdtsc</i> outside privilege mode, Maintain high-fidelity time source, Skip time-checking APIs	Medium	[238, 224, 239]	1, 2, 3, 4	
			Local Resource: RDTSC time GetTime GetTickCount QueryPerformanceCounter GetLocalTime GetSystemTime	Medium	N/A	None, open problem				
	Traps	Instruction Prefix (Rep)	High	Medium	Set breakpoint on exception handler, Allow single-step/breakpoint exceptions to be automatically passed to the exception handler	High	[242]	1, 2, 3		
		Interrupt 0x03, 0x2D	Low	High						
		Interrupt 0x41	Low	High						
	Debugger Specific	OillyDBG: InputDebugString	Low	High	Patch entry of <i>OutputDebugString</i>	Low	[246]	1, 2, 3		
		SoftICE Interrupt 1	Low	High	Set breakpoint inside <i>CreateFilefileA/W</i>					
	Targeted	APT Environment Keying	High	Very High	Exhaustive Enumeration, path exploration techniques	Low	[247, 248]	1, 2, 3, 4		
		AI Locksmithing	Very High	Very High	N/A	Rare	[254]	1, 2, 3, 4		
Detection-Independent	Control Flow Manipulation	Self Debugging	DebugActiveProcess DbgUiDebugActiveProcess NtDebugActiveProcess	Medium	Low	Set debug port to 0	Low	[264]	1, 2, 3	
		Suspend Thread	SuspendThread NtSuspendThread	Low	Low	N/A			[217]	1, 2
		Thread Hiding	NtSetInformationThread ZwSetInformationThread	Low	Low	Skip the APIs			[257]	1, 2
		Multi-threading	CreateThread	Medium	Low	Set breakpoint at every entry			[274, 257]	1, 2
	Lockout Evasion	BlockInput SwitchDesktop	Low	Low	Skip APIs	Low	[218, 257]	1, 2, 3, 4		
Fileless (AVT)	Web-based exploits System-level exploits	High	Very High	N/A	Low	[189, 271]	1, 2, 3, 4			

Table 3.1: Classification and comparison of malware anti-debugging techniques.

## 2.3 Automated Dynamic Analysis Evasion

### Key Point 3.3:

- ☞ We call "Automated Dynamic Analysis Evasion" all evasion techniques able to escape from *automatic analysis tool*.
- ☞ We oppose *automatic dynamic analysis environment* to *manual dynamic analysis environment* used by human in from of a computer.
- ☞ Automatic analysis environment does not require any human interaction since they are fully automatized.
- ☞ Automated dynamic analysis environments regroup different types of sandboxes: virtualized or emulated *sandbox*.

Although effective, the manual dynamic analysis suffers a critical limitation, that is: time. 2018 statistics provided by *McAfee* reports on receiving more than 600K new samples each day [275]. Analyzing this massive number of malware samples calls for a far more agile approach. This demand led to a new paradigm of analysis which we referred to as automated dynamic analysis. Sandbox is the representative technology for this paradigm. In this subsection, we will have a brief introduction to sandboxes and further propose a classification and comparison of malware evasion tactics.

### 2.3.1 An overview of malware sandboxes

The concept behind a malware automated dynamic analysis system is to capture the suspicious program in a controlled and contained testing environment called *sandbox*, where its behavior in runtime can be closely studied and analyzed. Initially, sandboxes were employed as a part of the manual malware analysis. But today, they are playing their roles as the core of the automated detection process [181]. Sandboxes are built in different ways. To better grasp the evasion tactics, and depict how they stand against different sandbox technologies, first, we must have a sense of how they are made.

#### 2.3.1.1 Virtualization-based sandboxes

A virtual machine (VM) according to Goldberg [276], is "an efficient, isolated duplicate of the real machine". The hypervisor or virtual machine manager (*VMM*) is in charge of managing and mediating programs' access requests to the underlying hardware. In other words, every virtual machine atop the *VMM*, in order to access the hardware, must first pass through the hypervisor. There are a couple of ways to implement sandboxes based on virtualization. One way is to craft the analysis tools directly into the hypervisor as in the case of Ether [210]. The other approach would be to embed the analysis tools (e.g. installing hooks) within the virtual machine that runs the malware sample. Instances of this design are: *Norman* sandbox [277], *CWsandbox* [278] and more recently *Cuckoo* sandbox [279]. Both of the methods inherently leak subtle cues which malware could pick on to detect the presence of a sandbox. In the latter case, for instance, the *VMM* has to provide the required information to the analysis VM which means whenever a sensitive system call is being made by the malware, the *VMM* has to pass the control to the analysis tools inside the VM. Challenges of performing these procedures without leakage are profound which we will elaborate accordingly.

#### 2.3.1.2 Emulation-based sandboxes

An emulator is a software that simulates a functionality or a piece of hardware [280]. An emulation-based sandbox can be achieved through different designs. One would be to simulate the necessary OS functions and APIs. Another approach is the simulation of CPU and memory and is the case for many anti-virus products [280]. Simulation of I/O in addition to memory and CPU is what in literature is referred to as the full system simulation. *QEMU* [281] is a widely famous full system simulation based on which other famous sandboxes such as *Anubis* [282] are built. Among the eminent features of emulation-based sandboxes are the great flexibility and detailed visibility of malware inner workings (introspection) that they offer. Especially, with the full system emulation, the behavior of the *program under inspection* (PUI) could be studied with minute details.

### 2.3.1.3 Bare-metal sandboxes

Recently evolved and perplexing evasion tactics, employed by sophisticated malware, demands a new paradigm of analysis. The emerging idea is to execute the malware in several different analysis environments simultaneously with the assumption that any deviation in behavior is a potential indication of malicious intents [283]. The feasibility of this idea requires a reference system in which the malware is analyzed without the utilization of any detectable component and the ideal choice would be a bare-metal environment equal to a real production system in terms of transparency. There have been several products in this vein e.g. *Barebox*, bare cloud, etc. [283, 284, 285].

## 2.3.2 Proposed Classification of Automated Dynamic Analysis Evasion Techniques

### Resume 11:

☞ We are reusing the classification of detection-dependent or independent as explained in Key-Point 3.2.

Along with the merits that each design offers, subtle flaws or specific working principles that a malware exploits to forge their evasion tactic. Indeed, if the malware achieves one specific goal, it has the ability to triumphantly evade the sandbox. This goal is to behave nicely or refrain from executing its malicious payload as long as it resides within the sandbox. This strategy capitalizes on two facts. The first is that due to a massive number of malware samples and limitation of resources, sandboxes assign a specific limited time to the analysis of a sample. The second lies in an inherent limitation of dynamic analysis. In dynamic analysis since the "runtime behavior" and "execution" is being inspected, only the execution path is visible to the inspector (sandbox). Thus, if a malware provides no malicious behavior while under examination, the sandbox flags it as benign.

Similarly to *manual dynamic analysis evasion*, we propose to classify *automated dynamic analysis evasion* tactics under two categories, that is to say detection-dependent evasion and detection-independent evasion. We will elaborate on these tactics along with several techniques under each tactic. In addition, we briefly note the ways through which sandboxes try to defeat these tactics.

### 2.3.3 Detection-Dependent Evasion

In the same way as with the debuggers, the main goal of the malware is to detect its environment to check whether the host is a sandbox or not. A successful evasion in this category is subjected to correctly detecting the environment. If the environment is detected to be a sandbox — or by contraposition, it is not the intended environment — the malware will not reveal its malicious payload, hence, evades the detection. Following are the tactics a malware might use to achieve this.

#### 2.3.3.1 Tactic 1. Fingerprinting

Fingerprinting is a tactic pursued by malware to detect the presence of sandboxes by looking for environmental artifacts or signs that could reveal the indications of a virtual/emulated machine. These signs can range from device drivers, files on disk and registry keys which only belong to emulated/virtualized environments. Indications of *VM* or emulation are scattered at different levels [286, 287, 288, 289]. It is noteworthy to mention that initially, many sandboxes such as Norman [277] were developed upon *VMs*. Thus, in literature, we may still observe the terms sandbox and *VMs* being used interchangeably to imply a contained analysis environment. The very same fact is also the reason for the rise of techniques referred to as anti-*VM*, suggesting that detection of a virtual machine would potentially mean an analysis environment.

Different studies have been conducted to uncover the levels at which sandboxes leave their marks [186, 290]. These levels are:

- **Hardware:** Devices and drivers are artifacts that malware might look for to identify its environment. In the case of devices, *VMs* often emulate devices that can be readily detected as in the case of Reptile malware [291]. This ranges from obvious footprints such as the *VMWare* Ethernet device with its identifiable



manufacturer prefix, to more subtle marks. Moreover, specific drivers are employed by VMs to interact properly with the host OS. These drivers are other indications of an analysis environment for malware. For instance in the path: "C:\Windows\System32\Drivers" there are such signs that could expose *VMWare*, *VirtualBox*, etc. (e.g. *Vmmouse.sys*, *vm3dgl.dll*, *VMToolsHook.dll*, etc.) [292].

- **Execution Environment:** A malware inside sandbox experiences subtle differences in the environment within which they are executed. Some kernel space memory values, for instance, are different between a sandbox and native system that can be detected by malware as in the cases of *Agobot* and *Storm* Trojans [293, 294]. Artifacts of this level manifest themselves either in memory or execution. As in the case of memory artifacts, for instance, to allow inspection and control between host and guest OS, *VMWare* creates a channel between them ("*ComCHannel*"). Virtual PC hooks work in the same way [295]. *Anticuckoo* [296], for instance, detects and crashes the cuckoo sandbox through detecting and manipulating the system hooks.
- **Application:** When an analysis application is used in an environment, its presence is usually disclosed due to the artifacts of installation and execution which can be picked up by inquisitive malware such as *Rbot* [297], *Phatbot* [298], [299], or [300]. Even if not executed, evidence of the analysis tools might be residing on the disk, registry keys, etc. which could be readily found by malware especially if they contain well-known file name or locations e.g. "HKLM\SYSTEM\CurrentControlSet\Control\VirtualDeviceDrivers". In a similar instance, if the names are not altered or the corresponding processes concealed, they can be enumerated by malware with little trouble. For instance, *VMtools.exe*, *Vmwareuser.exe* or *vboxservice.exe* if queried by malware, are vivid indicators of a virtual machine and analysis environment.
- **Behavior:** Perhaps, the most troublesome of all, for anti-malware vendors to conceal, are the leakages caused by imperfect virtualization/emulation, or characteristics that are innate to such environments. Of particular interest are discrepancies of behavior between an emulated CPU and a physical one [286]. Performance of an application running under emulation is inferior to a real system. This performance penalty stems from translations or interceptions that have to be carried out in an emulation/virtualization. Such discrepancies can be disclosed through a diverse set of timing attacks [301, 237, 302]. Computing absolute performance for fingerprinting is an arduous task due to the diversity of hardware configuration.

An alternative for malware is to calculate the relative performance of code execution [286]. Pursuing this method, malware compares the performance ratio of two or more operations on the same system. If the measurement varies significantly among production and emulation systems, there's a high chance of sandbox presence. Another interesting way of exploiting the limitations of emulation is observing and comparing the effects of caching on emulated and real environments. In this technique, a function is executed a number of times. As expected, the first run must be the slowest (due to cache performance effects). The same test is performed again, this time, in the absence of caching. Timing analysis depicts the effectiveness of the employed caching. Simulation of processor cache is a complex task and emulators may not support it, the result of which is about to expose the emulation nature of the running environment in the aforementioned test [286].

Another noteworthy set of techniques are the red pill tests [303]. An inevitable behavioral artifact sandboxes share, are the imperfect simulation of CPU and residing instruction bugs. If a malware specifically finds such bugs and if during the execution it notices a mishandled instruction, it will suspect the presence of the sandbox.

- **Network** In addition to the previous levels that were suggested by Chen et al. [186], malware can also probe the network in pursuit of sandbox's marks. These marks are manifested in many forms such as known fixed IP addresses, [304], limitations pertaining to the sandboxes that prevent or emulate the internet access [305, 306] or extremely fast internet connection [307]. One technique, for instance, proposed in [304] is detecting sandboxes based on their known IP addresses which is acquired in an earlier attack
-

through a decoy malware.

Fingerprinting tactic was the initial objective of malware authors to detect and evade sandboxes. That way, we observed the techniques have evolved significantly. In countering fingerprinting, most solutions are reactive, namely, the fingerprinting technique must first be disclosed, then the corresponding counter evasion will be provisioned. An offensive way of fighting against fingerprinting are projects such as Jing et al.'s *Morpheus*. *Morpheus* is a tool that *automatically* finds fingerprints on QEMU and VirtualBox-based emulators [36]. Collectively, the fingerprinting tactic is still the dominant approach to detect and evade sandboxes [308].

### 2.3.3.2 Tactic 2. Reverse Turing Test

The second tactic that aims at detection, is checking for human interaction with the system. This tactic capitalizes on the fact that sandboxes are automated machines with no human or operator directly interacting with them. Thus, if malware does not observe any human interaction, it presumes to be in a sandbox. Such tactic is referred to as Reverse Turing Test since a machine is trying to distinguish between human or AI. This tactic can be carried out through various techniques [289, 309, 310, 307, 311]. For instance, the *UpClicker* [311] or a more advanced one, *BaneChant* [312] await the mouse left-click to detect human interaction [313].

In a large portion of reverse Turing test techniques, the malware looks for last user's inputs. To do so, malware often leverages a combination of `GetTickCount` [234] and `GetLastInputInfo` [314] functions to compute the idle time of the user. To test whether it is running on a real system, the malware waits indefinitely for any form of user input. On a real system eventually, a key would be pressed or mouse would be moved by the user. If that occurs for a specific number of times, malware executes its malicious payload.

To counter this tactic of evasion, simulating human behavior might seem intuitive but it might be counterproductive. One reason is that digitally-generated human behavior can also be detected [315]. The second reason is that limitation of designing such reversal Turing test seems to be merely a function of imagination. To demonstrate our point, consider another technique in which the malware waits for the user to scroll to the second page of a Rich Text Format (RTF) before it executes the malicious payload; or in another approach, using Windows function `GetCursorPos` [316], which holds the position of system's cursor, the malware checks the cursor movement speed between instructions; if it exceeds a specific threshold, it would imply that the movement is too fast to be human-generated and malware ceases to operate [313]; or another more recent technique which relies on seeking wear and tear signs of a production system [317]. The seemingly endless possibilities for designing reversal Turing tests make it a puzzling challenge for sandboxes to counter. This tactic is becoming more prevalent, but still not as pervasive as fingerprinting [308].

### 2.3.3.3 Tactic 3. Targeted

The third tactic of the detection-dependent category is targeted detection. This tactic is slightly different from the previous ones in that instead of striving for detecting or evading a sandbox directly, the malware fingerprints the environment to verify if the host is precisely the intended (targeted) machine. In other words, the malware looks for its target, not sandbox. This tactic can be employed following different routes:

- **Environmentally-targeted:** *Stuxnet* is known to be the first Cyber weapon that incorporated targeting tactic as a portion of its evasion tactics [310]. *Stuxnet* explicitly looked for the presence of a specific industrial control system and would remain dormant otherwise. Depending on the infection approach, APT attacks follow different routes.
- **Individually Targeted:** In the case of *Stuxnet*, the strategy was to keep probing victims (a wormy behavior) until the target was reached. Other classes of APT include the attacks such as *darkhotel* [318], in which the infection is conducted through spear phishing (i.e. directly aimed at the target). In other words, the attackers make sure that their malicious code is directly delivered to their target.



- **Environment-dependent Encryption:** Such targeted malware, have an encrypted payload the decryption of which, is subjected to a key that is derived from its victim's environment. The key might be a hardware serial number, specific environmental settings, etc. You can refer to [319] for a thorough discussion of the topic.

### 2.3.4 Detection-Independent Evasion

The major difference in this category of tactics is that they do not rely on detecting the target environment, and their evasion tactic is independent of target system which relieves them from having to employ sophisticated detection techniques. Consequently, efforts directed at achieving more transparency has no effects on these tactics. In the following, we elaborate on the tactics of this category and several techniques to deploy them.

#### 2.3.4.1 Tactic 1. Stalling

This tactic of analysis evasion capitalizes on the fact that sandboxes assign a limited amount of time to the analysis of each sample. Malware has to simply postpone its malicious activity to the post-analysis stage [320]. To this end, malware authors came up with the idea of stalling [295] which can be achieved through a diverse set of techniques that range from a simple call to *Sleep* function [321] to more sophisticated ones. Here we examine some of the known major stalling techniques.

- **Simple Sleep:** Sleeping is the simplest form of stalling and just as simple to defeat. The idea was to remain inactive for  $n$  minutes in order for the sandbox inspection to timeout before observing any malicious activity. After being released to the network, the malware would execute the malicious payload. The infamous *DUQU* [247] exhibits such technique as one of its prerequisites for ignition. It requires that the system remain idle for at least 10 minutes [168]. Another example would be the *Khelios* botnet [322]. A new variant of *Khelios* sample found in 2013 (Called Nap) calls the *SleepEx* function [323] with a timeout of 10 minutes. This delay in execution outruns the sandbox analysis timeout which is followed by achieving the harmless flag. To counter such techniques, sandboxes came up with the simple idea of accelerating the time (called sleep patching). Although seemingly logical, sleep patching has unpredicted effects. An interesting side effect of sleep patching was observed in specific malware samples where the acceleration actually leads to inactivity of malware instances that wait for human interaction within a predefined period of time [315]. Despite its side effects, sleep patching turned out to be effective in some cases and malware authors came up with more advanced techniques as a response.
- **Advanced Sleep:** New malware instances such as *Pafish* [324] were found to opt for more advanced sleeping tricks. They detect sleep-patching using the *rdtsc* instruction in combination with *Sleep* function call to check the acceleration of execution. More directly, this means timing the *Sleep* function between the expected wait and the actual wait.
- **Code stalling:** While *Sleep*, delay the execution, in code stalling the malware, opts for executing irrelevant and time-consuming benign instructions to avoid raising any suspicions from the sandbox [315]. *Rombertik* [325] is a spyware aimed at stealing confidential data. To confuse sandboxes, it writes approximately 960 million bytes of random data to memory [326]. The hurdles this technique impose on sandboxes are twofold. The first one is the inability of the sandbox to suspect stalling as the sample is running actively. The second one is that this excessive writing would overwhelm the tracking tools [325]. In another striking code stalling technique, the malware encrypts the payload using a weak encryption key and brute-forces it during the execution [283].

#### 2.3.4.2 Tactic 2. Trigger-based

The second tactic that does not rely on detection is a trigger-based tactic or more traditionally logic bombs. As we have already noted, the basis of evading sandboxes is to simply refrain from exhibiting the malicious behavior so long as they are in the sandbox. Another tactic for remaining dormant would be to wait for a trigger. There are many environmental variables that can serve as malware triggers ranging from system date

to a specially crafted network instruction. For instance, *MyDoom* [224] is triggered on specific dates and it performs *DDoS* attacks, or some key-loggers only log keystrokes for particular websites; and finally, *DDoS* zombies which are only activated when given the proper command [327]. In the literature, this behavior is referred to as trigger-based behavior [249]. In the following, there are a number of triggers embedded within malware as a protection layer against sandboxes.

- **Keystroke-based:** The malware gets triggered if it notices a specific keyword, for instance, the name of an application or the title of a window [328]. What occurs when the trigger happens, depends on the purpose and context. For instance, malware might initiate logging keystrokes.
- **System time:** In this case, the system date or time would serve as the trigger [328, 329, 330]. A newer malware that recently used this technique was the *industroyer* [274].
- **Network Inputs:** A portion of malware are triggered when they receive certain inputs from the network as in the case of Tribal flood network [187]. Note that some malware are able to go on internet by themselves, checking the content of a given website to know if they are in an analysis environment or not. The case of *Wannacry* malware [331] is an interesting example. Even if it was not to evade analysis, this ransomware checked a hard-coded url in order to cipher files only if it cannot contact this domain. The goal was probably to avoid infection in the environment in which it has been developed, the same logic could be practiced to detect a url that would exist (or would not exist — in the case where some environments would respond to all requests, even those impossible) and act accordingly.
- **Covert trigger-based:** Previous techniques often presuppose lack of code inspection to remain undetected (as it is often the case for compiled malware). However, there have been advances regarding the automatic detection of such program routes e.g. State-of-the-art covert trigger-based techniques are being devised as a response to automated approaches that aim at detecting such triggers instruction. These techniques use instruction-level stenography to hide the malicious code from the disassemblers. In addition, they implement trigger-based bugs to provision stealthy control transfer which makes it difficult for dynamic analysis to discover proper triggers [310] return values from system calls are other instances of malware triggers.

Trigger-based tactic initially was not used to evade sandboxes and they are still relatively seen in the wild. Finding these triggers is often pursued through path exploration approaches e.g. symbolic execution with the goal of finding and traversing all the conditional branches in an automated manner. These approaches require significant levels of resources and they are still below a fair level of efficiency.

### 2.3.4.3 Tactic 3. Fileless Malware

In section 2.2.5.3, we discussed fileless malware. In addition to the rigorous resistance against manual analysis, fileless malware is profoundly adept at evading security mechanisms. A major difference of this tactic is the most often the malware is not subjected to the analysis environment in the first place. This is an inherent outcome of this tactic. The techniques that fileless malware utilizes are similar to drive-by attacks [332]. Generally, fileless malware exploits a vulnerability of the target system (OS, Browser, Browser plugins, etc.) and it injects its malicious code directly into the memory. The new trend of fileless malware uses windows *PowerShell* to carry out its task. As mentioned earlier, this attack is on the rise, and defending against it is of great complexity.

### 2.3.5 Resume about automatic dynamic analysis evasion techniques

In table 3.2, we summarized our survey on malware automated dynamic analysis evasion. In this table, we compared the techniques based on four criteria i.e. complexity of implementation, pervasiveness, efficacy level, and detection complexity. In addition, in the example column, we provided malware samples that incorporate the corresponding techniques. Moreover, under Sandbox countermeasure tactics, we hint on how the defensive side strives to counter the evasion tactics and how complex/effective these countermeasures are.

Criteria		Complexity	Pervasiveness	Efficacy Level	Sandbox Countermeasure Tactics		Detection Complexity	Examples
Cat.	Tactic				Complexity	Effectiveness		
Detection Independent	Stalling	Low-Medium	Medium	All Architectures	Sleep Patching		Very High	[325, 247, 324, 322]
	Trigger-Based	Low	Medium	Emulation-Based, Baremetal	Low	Low		
					High	Moderate		
Fileless (AVT)	High	Low	All Architectures	N/A		Very High	[271]	
Detection Dependant	Fingerprinting	High	High	VM-Based, Hypervisor based, Emulation-Based	Using heterogeneous analysis, Artifact Randomization		Moderate	[291, 300, 298, 297, 293, 294, 299, 304]
	Reverse Turing Test	Medium	Medium	VM-Based, Hypervisor based, Bare-metal	Moderate	Moderate		
					Digital Simulation, path exploration			
					Low	Low		
Targeted	Very High	Low	VM-based, Hypervisor based	Emulation based	Path exploration		Very High	[310, 318, 277]
High	Low							

Table 3.2: Classification and comparison of malware sandbox evasion techniques.

## 2.4 A Brief Survey on Countering Malware Evasion

### Resume 12:

☞ In this subsection, we are presenting briefly the different techniques used to counter malware evasion.

👉 Even if it is hard to design a perfect analysis test able to be totally transparent, it is possible to improve different points.

Countering evasive malware can follow different tactics. In this section, we briefly survey such defensive tactics against evasive malware, for the sake of sobriety. Further explanation is given in the original article and the reader is invited to refer to it for more details.

There are different countermeasure tactics against evasive malware. It depends on the analysis environment, what is analyzed and how it is analyzed. It also depends on how a malware operates, what it looks at and how it reacts to its environment. Generally speaking, we can define four modes of action to counter the evasive techniques of malware.

- **Reactive Detection:** In this case, defenders tap into their knowledge of specific evasive behaviors to craft their countermeasure accordingly. Namely, the detection is subjected to knowing the evasion technique in advance [287]. Reactive approach could circumvent only the known tactics and is vulnerable to novel evasion techniques. They are relatively easy to implement.
- **Multi-System Execution:** In this approach, malware is executed on several different analysis platforms [36]. Their execution then is studied to determine if their behavior has been diverged based on the execution environment. As a drawback, multi-system execution tactic is only effective against detection-dependent evasion strategy.
- **Path-Exploration:** Another solution is to trigger as many conditional branches as possible in a program in order to provide the greatest code coverage. The goal is to trigger any condition in a process such as, in the case of a malware, the malicious payload will be executed which should expose the malware. It is possible to do the same with path-exploration tactic to identify the detection-dependant malware, as in [251]. But this solution is far from being perfect. For instance, malware authors can incorporate anti-symbolic execution obfuscation [333] into their code and they can impede with the path-exploration tactic. In some cases, it is possible to force the path-exploration to go on a corrupted one, leading to crashes.
- **Towards Perfect Transparency:** Research efforts for finding solutions to counter evasive malware, by a drastically wide margin, are focused on achieving more transparent systems. Transparent systems are the the most used response to the proliferation of fingerprinting tactics. We can point several key approaches to achieve transparency and hiding analysis from malware's eyes:

- **Hiding Environmental Artifacts** For instance, CWSandbox [278] prevents malware from detecting the analysis environment by instrumenting the system with *rootkit-like* functionality to cover the environmental artifacts (e.g. files, registry entries, etc.).
- **Hypervisor-Based Analysis** Given the higher privilege in hypervisors, researchers have proposed to take their instrumentation of the system into the hypervisor itself. For instance, in [210], activity of malware is monitored by catching context switches and system calls using Xen hypervisor. Even though the experimental results of such design might seem promising, they are shown to be detectable as well. Regardless, such systems are shown to be detectable as well [237, 334, 335, 336].
- **Bare-Metal Analysis** The bare-metal analysis approach is the most idealistic endeavor which targets perfect transparency. In *BareCloud* [283] for instance, authors refrain from incorporating in-system monitoring tools. Rather, they rely on analyzing disc and network activity at the hardware level. They achieve a high level of transparency. However, at the cost of introspection. If a malware opts for a stalling tactic, for instance, *BareCloud* would fail to detect the suspicious activity. Recently, the new trend of making a bare-metal system seems to be concerning the incorporation of system management mode (SMM) [337]. In [337] authors introduce *Spectre* for transparent malware analysis. Zhang et al., further employed this technique to introduce *MALT*—a transparent debugger.

In the end, Table 3.3 summarizes our survey on countermeasure tactics against evasive malware. This one regroups all the cases previously described.

Criterion	Countermeasure	Effective Against	Complexity	Weakness	Examples
<b>Reactive</b>		Only Known Evasion techniques	Low	Vulnerable to zero-day techniques.	[334, 287, 338]
<b>Multi-System Execution</b>		Detection-independent-dependent category	Medium	Ineffective against detection-independent tactic.	[339, 340, 341, 283, 326]
<b>Path-Exploration</b>		All tactics except Fileless	High	Vulnerable to anti-symbolic execution obfuscation. Not scalable, resource intensive.	[249, 251, 342]
<b>Towards Perfect</b>		Transparency Fingerprinting Tactic	Very High	Vulnerable to Targeted, Reverse Turing, Stalling, Fileless, and trigger-based tactics. Unless the sandbox is equipped with other countermeasure tactics as well.	[210, 283, 343, 344, 278, 337]

Table 3.3: Classification and comparison of countermeasure tactics against evasive malwares.

## 2.5 Conclusion about the state-of-the-art

With regard to the different techniques proposed here, it should be noted that the competition between evasion measures and countermeasures is always ongoing. Fingerprinting by far is the most pervasive tactic and has proven to be an effective technique if executed with precision. It should be noted that these techniques are generally identified and specific to a given analysis environment, whether the escape method is automatic or manual.

Generally speaking, as with the uncertainty of the measurement, any direct measurement action of a system induces, in one way or another, a certain disturbance of this system via the measuring device. It is thus in the research of this type of disturbance (cause) or in the exploitation of the latter (consequence) that it is possible to work to escape.

Very directly, from our observations the most effective methods over time have some common characteristics: be based on the exploitation of a technical characteristic that is global to several tools, exploitation of a consequence whose cause is inherent to the analysis tool (and whose correction could be challenging for the tool) and be based on legitimate and documented mechanisms. The second point is illustrated by the exploitation of the communication mechanisms of the analysis environment (manual method) or the installation of a hypervisor in a *hypervised* environment to see if there is already one (automatic method), etc... Allowing this type of action such as a malware does not realize it is under analysis is equivalent to neutralizing or strongly modifying the analysis tool.

Therefore, our research was conducted to find new methods capable of performing an escape while maximizing these characteristics. Our researches have been focused, like this state-of-the-art, on two main axes. On the one hand, we first present a new form of evasion against a manual dynamic analysis evasion in the context of a debugger and, on the other hand, against an automatic dynamic analysis evasion evading DBI, VM and even debuggers. This last method is intended to be generic and synthetic in order to act on all analysis environments.

### 3 New manual dynamic analysis evasion technique on debuggers

#### Resume 13:

- ☞ This section propose one new evasion technique from manual dynamic analysis environment.
  - 👉 We have rediscovered a bug in Windbg debugger but never corrected despite publication by a former Microsoft's employee.
  - 👉 After introducing this bug, we show how to escape from Windbg in a stealthy way.
- ☞ This section proposes three different techniques to fool the disassembly engine in debuggers.
  - 👉 The first uses an undocumented behavior by using an operand-size override prefix on a relative jump.
  - 👉 The second uses REX prefix in a way which is not correctly handled by Windbg.
  - 👉 The third uses unsupported instruction by the debugger or exotic `nop` instructions which are not correctly interpreted.
- ☞ The goal is to make the disassembled code unreadable in the debugger but perfectly executable for the CPU.
  - 👉 All these methods are based on interpretation differences of the assembly language by Intel and AMD CPU vendors (they use the same op-codes for similar instructions but they do not have exactly the same way to implement it).

This section explains why it is relevant to present and fix new techniques or bugs exploited from debuggers to detect, escape or subvert analysis. Our study is based on Windows operating system, no matter the version of the operating system used. This choice is explained by the fact that most threats are on this platform. And the most famous — not to say the most used — debuggers on Windows is the one developed by Microsoft: Windbg [345]. This one is part of the *Windows Driver Kit* [209] and a new version, Windbg Preview [346], is actually developed with nice extensions such as *Time Travel Debugger* [347]. We are going to show four ways to disrupt the functioning of Windbg disassembler (version 10.0.17763.1 for Windbg and 1.0.1904.18001 for Windbg Preview). Whenever possible, we will also try to see if our escape techniques could have success with other debuggers.

### 3.1 INT 3 mishandling exploitation

#### Key Point 3.4:

- ☞ There are several ways to implement *breakpoint instruction* from assembly ("int 3h") to opcodes.
  - ☞ The *short version* usually used in Microsoft world (with MASM) uses 0xCC (one byte).
  - ☞ The *long version* used by NASM uses 0xCD 0x03 (two bytes).
  - ☞ Both represent the same instruction but with a different number of instructions.

A trick used to evade debuggers is to abuse from bugs in the debugger itself. Indeed, debuggers are software like any others, which means they can be improved. As explained in the debugger specific tactic of detection (section 2.2.4.3), it is possible to exploit existing bugs in a debugger to detect it.

One of the bugs used in this section to exploit Windbg is already partially known [190]. It is based on "int 3h" instruction. This one corresponds to an interruption — the third one — referencing a debug break. When this instruction is encountered, the CPU gives back the control to the debugger process. From the CPU, this instruction can be encoded in two ways, for the same result. Indeed, it could be encoded, from assembly to opcodes executable by CPU, by using either 0xCC or 0xCD 0x03 writing. The reason behind this two encoding stands in the approach used to encode the instruction by the compiler. Indeed, the 0xCC encoding always refers to the third interruption, (debug break). This is the short version of the instruction. However, there is a longest one: 0xCD 0x03. This encoding can refer to any interruption where the id of the interruption is encoded using the second byte of the encoding — 0x03 in our case. Most of the time, only the shortest instruction is used by compilers. Thus, the 0xCC encoding is used a lot more than 0xCD 0x03, except with NASM compiler [348].

More than the waste of a byte to encode the same instruction, the longest version can potentially lead to a bug if it is not taken into account correctly, as suggested in [349, 350]. The same way, this behavioral error has already been observed without being explicitly linked to a particular debugger by Peter Ferrie [351]. In this paper, the author has tested a lot of debuggers and emulators without linking this bug to one or more products. More generally, this bug has been used as a "debugger killer". Example is provided in [352] to illustrate how Windbg can be trapped by using these opcodes to make it crash. The same technique is used by malware's packer to avoid analysis under a debugger [353]. But this technique is expected to make crash the application since this one is debugged.

This is the misinterpretation of the "int 3h" instruction in its long form which can be exploited. From a general point of view, any misinterpretation resulting in a wrong execution by a debugger is a vulnerable flaw which could result to hidden code execution or badly interpreted one. Such a behaviour could lead to debugger detection or the execution of obfuscated code that is difficult to analyze. This is what we are going to show in the two next subsections. The first is about technical details about the flaw in Windbg and the second about the exploitation of that flaw.

### 3.1.1 Technical details

#### Key Point 3.5:

- ☞ Windbg debugger from Microsoft does not handle correctly long version of "int 3h" instruction.
  - ☞ The representation is misinterpreted by the decompiler (which does not seem to know this shape of the instruction).
  - ☞ In addition, the debugger's step-by-step operation dealing with the long version executes only one byte (when two would be required).
  - ☞ This misinterpretation wrongly affects the following step-by-step execution of the debugger.
- ☞ Microsoft was aware of this trick years ago but did not fix it...

As explained previously, Windbg does not correctly handle the "int 3h" instruction when this one has been encoded by the compiler in its long form. Before execution, Windbg correctly handles the debug break as Figure 3.2 illustrates it below.

```
00561026 cd03      int    3
00561028 90          nop
00561029 90          nop
```

Figure 3.2: Interpretation of the debug break instruction before execution.

When the previous code is executed, the next instruction about to be executed by Windbg is the one provided in Figure 3.3.

```
00561027 0390909090  add   edx, dword ptr [eax-6F6F6F70h]
0056102d 90          nop
0056102e 90          nop
```

Figure 3.3: Interpretation of the debug break instruction after execution.

Actually, when we are processing step by step, Windbg is incrementing by one the execution pointer (eip in 32 bits — rip in 64 bits) of the debugged thread, as if it would have been the short form version. This is incorrect since it is expected that the debugger increments the current instruction pointer by two as the long form of the debug break instruction would have expected it. The result of this misbehavior is a jump in the middle of the opcodes of the next valid instruction (the one following the debug break) to execute opcode 03 which is an addition (`add`) instruction. Of course, such behavior is obviously not about to increase the stability of the program. Most of the time, this behavior will eventually rise an access violation exception, resulting in the crash of the debugged program.

Interestingly, Ferrie worked at Microsoft, in 2008, at a time he published his paper [351]. Unfortunately, this one seemed not to notice that this bug has concerned Windbg. Maybe it is because this technique is not used efficiently by malware. Indeed, by default, the bug is usually about to crash an application — not the most discreet thing to do for a malware. If it would happen, a study the reason for the crash may be performed, resulting in the probable discovering the trick used by the malware. The idea is then to propose a method that allows detecting a debugger without crashing by exploiting this old bug, known for more than ten years.

### 3.1.2 Technical exploitation

#### Resume 14:

- ☞ In this subsection, we show how to exploit this bug in Windows to design analysis escape.



**Key Point 3.6:**

- ☞ Assembly instructions used in x86 or x64 architectures are not formatted with a fixed number of bytes.
  - ☞ This means that assembly instructions can be represented as opcodes with a variable number of bytes.
  - ☞ Since the debugger does not skip the right number of bytes on a long version of breakpoint, we can execute "an instruction within an instruction".
  - ☞ The trick of the evasion is to make sure that this instruction shifted remains consistent and achieves a different result than the original instruction (without crashing).
- ☞ To automatically resolves a breakpoint without a debugger, we use a *Structure Exception Handling* (SEH) in x86 — a *Vectored Exception Handler* (VEH) in x64.

The main issue with the use of "int 3h" Windbg's bug is the resulting crash of the application under the control of the debugger. The problem stands in the fact that 0x03 opcode usually results in an invalid execution flow or in an invalid access memory. The operation expected in such a case is an addition performed with registers or a direct access read in memory, which is usually not stable when it is not correctly driven. But it is possible to combine this misinterpreted opcode with other opcodes so that the result is still executable and valid.

Firstly, before trying to implement debugger detection, the code we are writing must be able to survive when there is no debugger attached. Executing "int 3h" instruction results in an application crash when there is nothing registered to handle it. To avoid that, we must ensure that our code which will be executed in an exception handled context. This one is triggered in case of absence of the debugger. Otherwise, this is the debugger itself which is notified first, by default.

In C language, this operation is implemented through a `_try/_except` block statement [354]. This one can be directly implemented in x86 assembly architecture since exception handling is stored at offset zero in the Thread Information Block (TIB) [355, 356] of the current thread. The field responsible for exception handler in the TIB is named *Structure Exception Handling* (SEH) [240] and it is directly accessible through "fs:[0]" in assembly language [357]. Update this field is enough in order to insert an exception handler function, called when an exception occurs. Technically speaking, it is just about storing a pointer referencing a handler function [358] in this structure. This is usually performed via the following instructions, where "handler" corresponds to a handler function. Such function is the "except" statement bloc in C language.

```

1 | assume fs:nothing ; Avoid segment registers to be considered as an error.
2 | push offset handler ; Store address of the handler pointer function.
3 | push fs:[0] ; Store the previous SEH on top of the stack.
4 | mov fs:[0], esp ; Update the SEH stack.

```

Code 3.1: Implementation of the beginning of exception procedure.

The procedure in x64 would be a bit different [241] since the exception handler is no more stored in the TIB directly, as it was in the x86 architecture, but it is mapped from the MZ-PE executable file. This can be done by the use of *Vectored Exception Handler* (VEH) [359] with the same logic. A dedicated list of functions is registered in order to be used if an exception occurs. This one is able to handle correctly the "int 3h" instruction when there is no debugger attached, the same way it does under 32-bit architecture.

Once the handler exception is registered, we have to deal with the case of the debugger's misinterpretation. The main problem is that the opcodes used must produce a valid result both in the normal case (no debugger attached) and in the case where they are misinterpreted (with the debugger attached). The easiest way to achieve this result is to try to reach one of two unconditional jumps in both cases. That is to say, the first in the case when the process is running under a debugger and the second when there is no debugger. All the code between the "int 3h" instruction and the two jumps is designed to avoid or cancel bad consequences of a



misinterpretation while keeping normal execution flow.

One way to encode unconditional jump operation is through 0xEB opcode, followed by a single byte indicated the relative offset to jump. This offset value is interpreted as a signed number which results in a jump of -128 to +127 bytes in memory. This is far enough for our code which is just about to jump to a dedicated location returning zero or one, depending on the presence or absence of a debugger.

The most efficient way to build the code is to stick the unconditional jump after the debug break instruction. In this way, when there is no debugging, the code is handled by the exception handler function which gives back the hand to the jump instruction. Figure 3.4 illustrates this procedure.

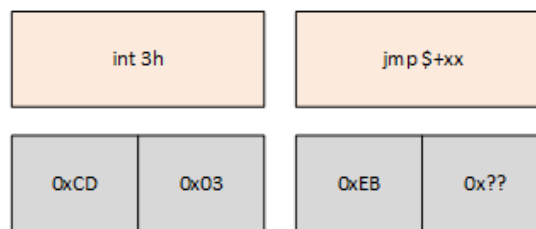


Figure 3.4: Illustration of the correct disassembling of "int 3h".

When Windbg is present, the misinterpretation of the debug break occurs, resulting in a shift inside the instructions flows. These ones are now interpreted as shown in Figure 3.5.



Figure 3.5: Illustration of the incorrect disassembling of "int 3h" by Windbg.

To cancel the "add ebp, ebx" instruction (resulting from the misinterpretation), it is enough to compute instruction "sub ebp, ebx". This last one is encoded with 0x2B 0xEB opcodes. Then, it comes the final unconditional jump to return zero or one, depending if this function must be able to detect the presence or the absence of a debugger. This construction allows the function to cancel consequences of misinterpretation (corruption of ebp register) while keeping the final jump to a location where we can handle if a debugger is present. The final result looks like in Figure 3.6.

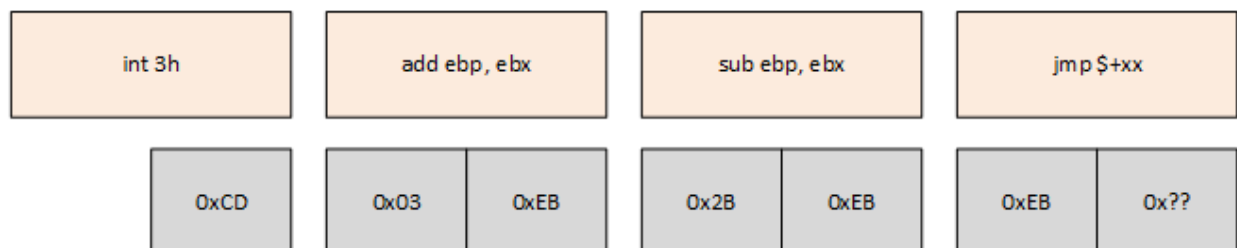


Figure 3.6: Correction to cancel side effect of the misinterpretation from Windbg.

The case of unconditional jump offsets remains to be resolved. In the case where a debugger is present, the last jump can point to any relevant location for our function. Whatever is the offset, it will not change anything

is can of correct or misinterpretation. In the case where there is no debugger and it is our exception handler function which resolves correctly the debug break, offset of the jump is fixed at +0x2B (offset of 43 bytes) as represented in the figure 3.7.

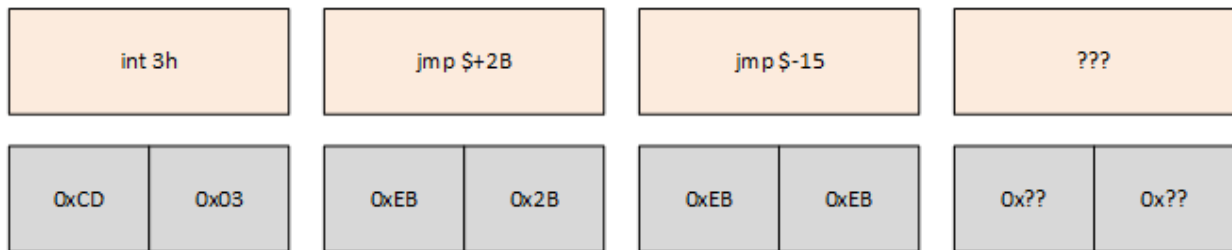


Figure 3.7: View of the assembly code executed where there is no Windbg.

The fixed offset of the unconditional jump in the case where there is no debugger forces us to write relevant code at that point. Instead of using nop instructions as a junk code, we propose to optimize the code so that this one is as optimized as possible. Actually, we are going to write the exception handler function right after the debug break part. It means the jump at a fixed offset will be inside the opcodes of the exception handler function.

Without loss of generality, we decided to build the present code to detect if the program is under the influence of Windbg debugger. More directly, it means the code returns one if a debugger is attached and zero otherwise. Since the exception handler function returns zero (under the defined value EXCEPTION\_EXECUTE\_HANDLER) to continue execution [358], it makes sense to reuse the last opcodes of the exception handler procedure to return zero when there is no debugger attached.

### 3.1.3 Illustration with operational code

#### Resume 15:

☞ In this subsection, we provide an operational code (in x86) and a graphical view (Figure 3.8) of the procedure to perform the evasion safely and efficiently.

In the case where we are executed under Windbg control, the unconditional jump is linked to a dedicated part supposing to return one. Then, it reroutes the execution flow to the original epilogue of the code. Note that there is no need to realign the stack after the exception handler is set up. Indeed, the function's epilogue reset the stack alignment thanks to "mov esp, ebp" instruction. Finally, taking into account everything, the code of debugger detection is the one presented in Figure 3.2.

```

1 | main proc
2 |
3 |     push ebp
4 |     mov  ebp, esp
5 |     assume fs:nothing
6 |
7 |     ; Register the SEH.
8 |     push offset__handler
9 |     push fs:[0]
10 |    mov  fs:[0], esp
11 |
12 |    ; In case of misinterpretation.
13 |    ;     [add ebp, ebx] [sub ebp, ebx] [jmp $+0E]
14 |    db 0CDh, 003d, 0EBh,      02Bh, 0EBh,      0EBh, 00Eh
15 |    ; [int 3h] [jmp $+2B]
16 |    ; In case of correct interpretation.
17 |
18 |    ; Restore original SEH.
19 |    pop  fs:[0]

```

```

20     add esp, 4
22     mov esp, ebp
23     pop ebp
24     ret
26     ; Return one if there is a debugger.
27     or eax, 1
28     jmp $-7 ; Go upper to the epilogue of the main function.
30 __handler:
31     push ebp
32     mov  ebp, esp
33     mov  eax, [ebp+08h] ; Retrieve the first parameter.
34     mov  edx, dword ptr [eax+0B8h] ; Get access to the value of instruction pointer.
35     add  edx, 2 ; Add two to jump correctly over "int 3h".
36     mov  dword ptr [eax+0B8h], edx ; Store the new value of rip.
37     xor  eax, eax ; Return zero.
38     mov  esp, ebp
39     pop  ebp
40     ret
42 main endp

```

Code 3.2: Final version of the Windbg's detection shellcode.

It results that the execution flow depends on the presence or the absence of Windbg debugger. Since assembly programming can be complex to understand, we propose to illustrate the execution flow of the program with Figure 3.8. This one represents the different jumps the code uses to achieve the Windbg detection goal.

The first case is where there is no debugger. In such a case, the "int 3h" instruction is handled by the `__handler` function registered just before and displayed in blue boxes. The role of the handler [360, 361] is to retrieve, from its first parameter (stored at `[ebp+08h]`) a pointer to an `EXCEPTION_RECORD` structure [362] where it will be able to change the instruction pointer where the exception handler must give back the control to the main code. In the case where there is no debugger, the execution still continues to restore the original version of the SEH and to finish the function, with the return value (stored in `eax`) equals to zero.

The main right arrow symbolizes when there is Windbg debugger attached to the process. In such a case, misinterpretation forces to go a little further on a bloc of code responsible to return one (or `eax, 1`) and it reuses the epilogue<sup>3</sup> of the main's function by a jump with a negative offset (`jmp $-7`, the small left array) to get access to `mov esp, ebp` instruction. Such a way, the function returns one when there is a debugger.

Finally, if there is a debugger able to handle correctly the long form of "int 3h" instruction, this one will execute the next instruction which is an unconditional jump with a positive offset (`jmp $+2b`). Compiled with MASM compiler, this last jump in our codes goes on the epilogue of the handler function, the same than the main function. Such a way, it does not change the logic of instructions present in the main functions and it returns zero. There is no detection (since another debugger is not vulnerable to this issue) but there is no crash, guaranteeing the operational use by a malware of the code provided.

<sup>3</sup>This optimization is perfectly optional even if it shows how it is possible to optimize the size of the code. Smaller the code is, easier it is to insert in a given process.

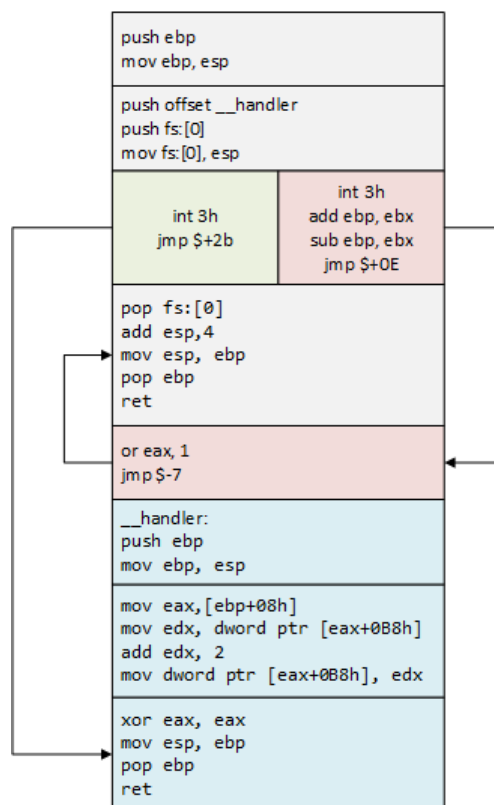


Figure 3.8: Graphical view of the Windbg's detection procedure.

### 3.2 Wrong jump interpretation

#### Key Point 3.7:

- ☞ We show in this subsection how identify the type (Intel or AMD) of processor used by a malware without using `cpuid` instruction.
- ☞ We use a difference in interpretation, for Intel and AMD processor families, of the same `jmp` instruction.

To work properly, debuggers need a disassembler. Generally, the compilation procedure aims to move from a code, written for instance in C, to a single compiled code. The compiled version of the code then depends on the style chosen by the compiler used. The disassembling procedure, that is to say starting from the compiled code to retrieve a human-readable code, is more complex. Indeed, it is necessary to take into account various backwards compatibility issues of the assembler language or the architecture of the CPU manufacturers. There are also the different optimization procedures, compilers style procedures and various freedoms taken or imposed by CPU manufacturers from the compiled source code. In short, it is notoriously complex to write a truly perfect disassembler.

But there is more, since assembly language is a complex mix of different norms. Historically, Intel was leader in the 32-bits architecture. In the old days, when a new architecture was released by Intel, AMD had no choice but to follow it in order to keep its market share. However, when a 64-bit architecture rose on the market, Intel and AMD both proposed their own norms. The first was the IA-64 architecture, with Itanium CPU, developed by Intel, a totally new architecture which broke with the x86 one. The second was the AMD64 architecture developed by AMD, which mainly consist of adding 64-bit operations in existing x86 architecture. At the end, the success of the IA-64 architecture was mitigated, which resulted, for the first time, that Intel followed the architecture developed by AMD (AMD64), under the name Intel 64. A more neutral name for the AMD64 and

Intel 64 architecture has been finally used in the industry to identify this architecture: x64. This architecture is very well documented by both Intel and AMD. However, there are still few differences between AMD and Intel to know in order to disassemble code correctly. Therein lies the rub.

### 3.2.1 Technical description

#### Key Point 3.8:

- ☞ Some of the 64-bit instructions have a behavior described as unpredictable in the Intel documentation.
  - 👉 It means such instructions does not have a precise behavior in official assembly documentation.
  - 👉 CPU manufacturers (Intel or AMD in our case) are free to implement it as they want.
- ☞ Operand-size override prefix (encoded with 0x66 byte) with a relative jump is one of these instructions.
  - 👉 And there is a difference of interpretation between the two manufacturers we can exploit.

Some of the 64-bit instructions have a behavior described as unpredictable in the Intel documentation [363]. However, it does not mean that these behaviors are really unpredictable. In fact, it just means that the x64 instructions set reference does not provide a precise behavior for these instructions.

One of this undocumented behavior is the use of an operand-size override prefix (encoded with 0x66 value at compilation time) on a relative jump. This use of the operand-size override prefix is not a common practice because there is no real operational use for it. Usually, to obtain the destination of the relative jump, we use the following formula:

$$\text{instruction\_pointer} = \text{address\_of\_jump} + \text{size\_of\_jump} + \text{sign\_extended\_displacement}$$

To illustrate with a real example, let us suppose we have a jump at address 0x100000. This jump is an unconditional jump supposed to jump 5 bytes after. Technically, it is encoded on 3 bytes (0x66 for the prefix, 0xEB for the jump and one byte to encode the offset, in our case, +5 bytes). The final result jumps at address 0x100008, since it is the sum of the size of the instruction (3 bytes), plus the offset inducted (5 bytes).

$$0x0100008 = 0x100000 + 3 + 0x05$$

The logic which is described above is always true on Intel processor. Nevertheless, on AMD processor the use of operand-size override prefix creates a totally new logic. Indeed, when a relative jump has an operand-size override prefix on AMD processor, the formula is different. It includes now a final operation to only keep the last 2 bytes. The following equation illustrates the procedure to compute the new address where to jump. Note that 0xFFFF represents a cast to only keeps the last two bytes of the resulting operation, as an AMD processor does.

$$\text{instruction\_pointer} = (\text{address\_of\_jump} + \text{size\_of\_jump} + \text{sign\_extended\_displacement}) \& 0xFFFF$$

Now, in the same conditions using a jump (base address 0x100000 and an inducted offset of 5 bytes) as the one we use before, it will result in a jump at address 0x0008 on AMD processor. Details of the computation are provided as follows.

$$\begin{aligned} 0x0000008 &= (0x100000 + 3 + 0x05) \& 0xFFFF \\ &= (0x100008) \& 0xFFFF \\ &= 0x0008 \end{aligned}$$

Of course, the difference in interpretation, for the two processor families, of the same instruction can open interesting possibilities. In addition to identifying the CPU family used in a precise way (other than with `cpuid` instruction which can be handled in the case of an hypervisor), it can be used to try to fool a debugger.

### 3.2.2 Technical exploitation

Windbg is able to partially handle this kind of behavior. For instance, we will use a relative jump with an operand-size override prefix and an 8-bits displacement (sign extended to 64-bits value) with the same characteristic as the precedent example. Such a jump will be encoded "0x66 0xEB 0x05" and Windbg will give the following interpretation (Figure 3.9).

```
00007FF634111730 66 EB 05          jmp          0000000000001738
```

Figure 3.9: Correct interpretation under AMD CPU but misinterpretation on Intel CPU due to the prefix used.

We can see that the result is exact if we assume we are on an AMD processor. Moreover, whenever we execute it on an AMD processor, we are going to jump at address 0x000000000001738. The issue stands in the fact that Windbg interprets this jump the same way it is interpreted on Intel processor. However, on Intel processor, it is the first logic presented which is used for jump calculation. Indeed, when we execute the "0x66 0xEB 0x05" jump on an Intel processor, we will jump at address 00007FF634111738 instead of address 0x000000000001738. Thus, the correct interpretation that is expected from Windows, when running on Intel processor, is given in Figure 3.10.

```
00007FF634111730 66 EB 05          jmp          00007FF634111738
```

Figure 3.10: What will be executed on an Intel CPU with the provided opcodes.

This behavior also concerns any conditional jumps (Jcc instructions) also and it can be easily checked by executing them. It is not a big deal since it does not change the expected execution of the process. But it could lead to misunderstand the assembly code analyzed by the debugger. Even if this error will not impact the functioning of the debugged processes, it could confuse the human person using Windbg to negatively affect the analysis of the process.

This trick is not new by itself. This is one of the few about CPU differences we can find online, even on Wikipedia [364]. Note that if the goal is simply to identify precisely the type of CPU on which malware is likely to run, it is possible to refer to performance difference studies [365, 366, 367, 368] on a whole bunch of technical characteristics.

## 3.3 Partial instruction prefix handling

### Resume 16:

☞ In this subsection, we show how to fool the disassembler of Windbg debugger by abusing of *prefixes* for instructions.

#### 3.3.1 Technical detail

All AMD64 and Intel64 instructions have a structured form which is described in the documentation of the manufacturer: the Intel documentation [3]. An instruction is described as represented in Figure 3.11 below. First, before the opcode itself, we find *prefixes*. These ones are used as an extended information provided to an instruction. Usually, the main purpose of a prefix is to custom or repeat a specific instruction. In the area of prefixes, we can split them in two different types.

The firsts are legacy prefixes used in x86 architecture to compute specific actions on instructions. One instance is lock prefix which is used on certain read-modify-write instructions in order to prevent simultaneous access to the memory. A second one is the repeat prefix that causes string handling instructions to be repeated. This last one is usually driven by the content of the rcx (ecx when using 32-bits architecture) register to evaluate the number of times an instruction must be repeated. The same way, we can talk about branch

taken/not taken prefixes which give clues to the CPU to lessen the impact of branch misprediction. Another one is the operand-size override prefix, which is normally used to switch between an instruction from 32-bits to 16-bits operand. Finally, there is the address-size override prefix, which can be used in 64-bit mode to use 32-bit addressing memory.

The second type of prefixes is REX prefix. This last one has a particular encoding. In fact, unlike the legacy prefix, this one is encoded using values from 0x40 to 0x4F range. Its lower nibble allows encoding several properties that have two main purposes. The first is to allow 64-bits operand size on some instructions which usually use 32-bits operand size. This is a smart way to extend existing opcodes from x86 to be usable easily for x64 architecture. The second use is to access newly added registers in 64-bit mode (r8 to r15, xmm8 to xmm15, ymm8 to ymm15, cr8 to cr15 and dr8 to dr15). In fact, the REX prefix could modify the initial behavior of the ModR/M byte and SIB byte [369] to make them access new registers. But the REX prefix, despite being part of the x64 instruction semantic and defined in the documentation, is still perfectly optional.

Then, it comes the remaining of the instruction encoding with the opcode itself which provides the real meaning of the operation. Finally, the last bytes are about memory or register involved in the operation and how they are involved.

Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes	Immediate data of 1, 2, or 4 bytes or none

Figure 3.11: Illustration of the assembly semantic by Intel [3].

Even though the position of the REX prefix is defined, in Figure 3.11 extracted from Intel documentation [363], between the legacy prefix and the opcode, it is not always encoded in such a way. Indeed, the REX prefix property (64-bit instruction and access to new registers) will only be taken into account when it is right before the opcode. However, there could have several REX prefixes mixed with legacy prefixes, as long as the total instruction size does not exceed the instruction maximum size of 15 bytes [370]. One example of undocumented use of the REX prefix can be seen in Figure 3.12.

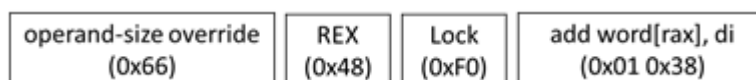


Figure 3.12: Illustration of undocumented use of REX prefix in assembly semantic.

In this example, we can see that there is a legacy prefix (lock) between the REX prefix and the opcode (direct addition of a value stored in memory where its address is referenced via rax register). This lock prefix is used for two main purposes. Firstly, it ensures that access to the memory is locked during the operation. Secondly, the impact of the REX prefix is now disable and we can consider the 0x48 byte as irrelevant (sort of transparent "nop" prefix, in a way). However, the operand-size override prefix is still valid and it impacts the size of the operand. Thanks to the operand-size override prefix, the operation is using WORD (16-bit) instead of DWORD (32-bit) operand size.

### 3.3.2 Technical exploitation

Nevertheless, Windbg does not handle this kind of use of the REX prefix correctly. Thus, for the preceding example encoded "0x66 0x48 0xF0 0x01 0x38", Windbg is going to interpret it baldy, as given in Figure 3.13.

```

00007FF714821743 66          ?? ??
00007FF714821744 48          ?? ??
00007FF714821745 F0 01 38    lock add    dword ptr [rax],edi

```

Figure 3.13: Misinterpretation of code by Windbg due to REX prefix.

The correct interpretation (that will be executed at run-time by the CPU) would be about to accept the REX prefix inside the instruction but to ignore its property and therefore the operand-size override prefix. Thus, the correct interpretation of the instruction would be as given in the assembly code in Figure 3.14.

```

00007FF714821743 66 48 F0 01 38    lock add    word ptr [rax], di

```

Figure 3.14: The code provided in figure 3.13 should be interpreted like this one.

This, behavior — which also concerns GDB [207] debugger — allows shellcode to be potentially unreadable by Windbg when they are analyzed. In fact, it would be very easy to abuse the use of useless prefix in order to disturb the use of Windbg and GDB or any vulnerable debugger. Indeed, we can easily force Windbg to wrongly guess more than half of the opcodes bytes used in a single instruction. Thus, it would be very troublesome to analyze assembly code for human readers while preserving the correct behavior of the code executed by the CPU.

### 3.4 Unsupported instruction

Despite the fact that most of the AMD64 instructions are well documented, some instructions are not. These instructions are not documented for two reasons. The first reason is that they could be used internally by Intel or AMD for their own purposes. The second reason stands in the fact that they could create new instructions in the future. In order to keep these instructions work on former processors, they assign them at `nop` (no operation).

In consequence, it could be complex to handle every single instruction. In that respect, there is not a lot of tools able to handle all possible instructions. Indeed, radar2, GDB, x64gdb as well as Windbg [206, 207, 371, 209] do not manage all instructions. For instance, there is a NOP encoded `0F 19 /r4` that Windbg does not handle correctly. This one, encoded with any register selected, should be observed in a debugger as a regular `nop`, as given in Figure 3.15.

```

00007FF6C21A1743 0F 19 37    NOP

```

Figure 3.15: Unsupported instruction should be interpreted as a `nop`.

In fact, Windbg is totally lost when it meets this instruction and it describes it as something entirely different, as we can see in Figure 3.16.

```

00007FF6C21A1743 0F          ?? ??
00007FF6C21A1744 19 37      sbb        dword ptr [rdi],esi

```

Figure 3.16: Unsupported instruction is not correctly interpreted by Windbg which tries to provide an irrelevant meaning to it.

Moreover, Winbdg, like IDA, radar2, GDB and x64dbg, only partially checks the `cpuid` instruction of the processor before disassembling and debugging. It means, they do not take into account the exact features sup-

<sup>4</sup><http://ref.x86asm.net/coder32.html#{}x0F19>



ported by the CPU of the machine they are executed from. Thus, when Windbg faces Intel MPX instructions [372], it always describes them as if they were supported on the current CPU, even if the CPU does not support them. It means that the debugger tries to interpret them even if such instructions behave as NOP or invalid instruction when they are not supported. This is a disadvantage since the debugger is describing a reality that is quite different from the one perceived by the program currently debugged on the machine. The debugger is interpreting its own reality, not the one of the current CPU about the execute instruction from the process the debugger is attached to. The Figure 3.17 shows an instance of such Intel MPX instruction not supported on our CPU (AMD Ryzen 7 1800X).

```
|.text:0000000140011812 F3 0F 1A 00 bndcl bnd0, qword ptr [rax]
```

Figure 3.17: Instance of unsupported CPU instruction interpreted as nop.

This reality is also true for the IDA software [373], which decompiles everything it can. It should be noted, however, that IDA is not just only a debugger and that its role is first and foremost to disassemble. Thus, having the ability to interpret instructions that the CPU of the host machine cannot execute is not a bad thing in itself, even if IDA might highlight in a better way the fact that some instructions are not supported on the host machine.

In a more general way, a good remediation would be to take into account the information from the current CPU to determine on which architecture the debugged program is running. This is true for debuggers who are required to execute code on the machine on which the targeted program is running (the case of the network debug procedure [374] aims to identify the remote hardware in the same way). The case of static analysis tools such as IDA is more complex. Indeed, they may have to evaluate binaries that are not supported by the CPU architecture of the current machine.

### 3.5 Conclusion about exploiting of Windbg flaws

This section aims to present bugs, mainly in Windbg debugger and to explain how to exploit existing generic vulnerabilities in debuggers to evade Windbg. More than the result about vulnerability exploitation, the method presented and used here is sufficient to allow some kind of bugs or misinterpretations in a debugger to be exploited by a malware. The goal for malware is to avoid detection when it knows it is currently executed in an analyzed environment. The method provides such an opportunity for malware. Otherwise, it is possible to complicate the analysis task by human using debugger tools by the use of misunderstand specificities between different architectures of different processors. Debuggers, since they allow to find bugs in software or to analyze malware, should be reliable, efficient and accurate. Otherwise, the trust between the analyst and the tool could be broken, complicating much more the work of analysis.

Our study has been mainly focused on Windbg since it is one of the most famous debuggers and one of the most largely used by malware analysts. All the tests reported here have been driven on third party debuggers with the results provided in the table below. The main conclusion is that there is no debugger definitely perfect and that all of them could improve their software in order to provide a much more accurate result. Note that, we have contacted Microsoft the 13/06/2019 and the 23/07/2019 to inform them about troubles in Windbg. After acknowledging receipt of emails and forwarding them to the appropriate person, we had no further news. Of course, bugs are always present and they are still exploitable. The disclosure time elapsed from a long time, this is why we publish them.

In addition to Microsoft with Windbg, all the possible bugs exploitation reported in this paper have been submitted organizations responsible to develop debuggers. The main recommendation is to fix disassembling issues and to use `cpuid` instruction check so that the debugger knows exactly on which architecture it is running. Such a way, it could be able to calibrate efficiently the methods it uses to perform the disassembly operations. In addition, a better implementation of debug break management for Windbg is strongly recommended so that it is not exploitable by malware.

---

	Windbg	IDA	Radare2	GDB	x64dbg
Manage Rex	No	Yes	Yes	No	Yes
Manage int 3	No	Yes	Yes	Yes	Yes
Manage AMD specific instruction	Yes	Yes	No	No	Yes
Manage CPUID	No	No	No	No	No
Manage undocumented instruction	Partially	Yes	Yes	Partially	Yes

---

Finally, this study could be continued by the test of new instructions provided by last generations of CPU on the market in addition to other old ones, kept for backward compatibility purposes. Checking the differences between what the debugger expects and the reality of process execution is always a good way to find tricks to detect or evade debuggers.

## 4 New universal dynamic analysis evasion technique

### Resume 17:

- ☞ This section presents an original and universal method to detection automated dynamic analysis environment (both manual and dynamic).

### 4.1 Introduction

Regardless of the analysis method used to try to understand a given piece of software (which could be malware or not), there is no generic method to know whether or not if a code is under analysis. There are disparate techniques (as presented in section 2) that often need to be combined to achieve appreciable results (section 2.4). But for each of them, there is more or less an effective way of countering it. The idea of this paper is therefore to present a new technique that allows both to detect all analysis environments, but also to offer a certain robustness against counter methods.

### 4.2 Preliminaries

#### Key Point 3.9:

- ☞ Technically speaking, there is no unique method able to generically evade from all analysis environment.
  - ☞ Most of the methods are specific to a given environment.
  - ☞ Malware need to chain different methods to detect different analysis environments.
  - ☞ Providing a universal method able to evade from many analysis environment would be a hit.
- ☞ In addition to debuggers (manual dynamic analysis environment) and virtual machine (automatic dynamic analysis environment), we add a new type of tool called *Dynamic Binary Instrumentation* (DBI).
  - ☞ Such a tool can be seen as an automatic debugger, programmed to debugger automatically a given sample.
  - ☞ It is generally used for vulnerability investigation but also as *automatic dynamic analysis environment* with malware.

Escaping automated dynamic analysis is a notion that concerns several kind of tools. As explained previously, there are several kind of tools, including debuggers and sandboxes. In the context of sandboxes, as given in section 2.3.1, there are hypervisor also known under the name of Virtual Machine (VM) used to execute a given executable file in a controlled and closed environment. More directly, VM tries to look very much like a real environment to allow normal execution, there are subtle differences. But we can include another type of analysis tool called *Dynamic Binary Instrumentation* (DBI) framework.

Detecting a debugger for an analyzed process is a complex operation detailed in section 2.2. From simple (and highly used) techniques involving Windows' API to detect a debugger [375, 376, 377], Also, there are advanced methods to perform detection of debuggers without using a given API from the system [378, 379]. In addition, there are more complex methods exploiting flaws or unexpected behaviors inducted by such tools [175] as given in section 3. But these techniques are inherently limited since debuggers are powerful control tools which are often deployed on systems in which analyst has full control. More precisely, this means that once the detection trick is known, it is thus possible to control it and ultimately to turn it off. We can think about variables from API updated on-the-fly via hooked API methods [380, 381]. It is therefore very hard to counter a debugger on a technical detail.

Regarding Virtual Machine, there are a lot of tricks to detect it, as given in section 2.3. In addition, public projects like Pafish [324] or Al-Khaser [382] provide operational example of codes. They both propose a great

number of tricks to detect they are running in a virtualized environment. From the use of obvious tricks to detect hypervisors which are badly disguised [383], the methods used are inherited from early research when Joanna Rutkowska published her work at Black Hat conference in 2006 about a project called Blue Pill [384]. During her talk, Joanna Rutkowska claimed that her method would be "100% undetectable". But researches showed it could be detected by different means [385], including one from Joanna Rutkowska herself called Red Pill [386]. Further researches have been conducted to develop different means of detection [387, 388, 389] (based on time elapsed by VMExit, for several of them), including original ones [390]. But the methods of detection can be prevented by using specific techniques from hypervisors [391] or by optimizing VM or with stealth hypervisor [392, 343].

The case of the DBI is a bit different. This one is a sort of automatic debugger, driven by a program, able to analyze efficiently and automatically a target executable file. In a way, it can be seen as the intermediate tool between the debugger and the VM (although in practice it shares more with a debugger than a VM). Note that in addition to execute the targeted code in a controlled environment, a DBI has the possibility to perform additional operations clandestinely alongside regular program execution. It means that between the execution of two code segments (code segments split with an arbitrary size: per instruction, per basic-block, per function...), it has the possibility to execute a specific code. In this additional code, it is possible to perform any analysis task, including malware analysis for DBI tool. It is also on this last type of analysis tool that we are going to test our research.

Since DBIs were not mentioned in our state of the art given in section 2, it may be appropriate to explain a bit about the background of DBI tools in an initial statement. The latter will be reinforced later with the specific part about the case of DBI.

The evasion techniques presented are usually used to detect a specific type of analysis environment. Their detection methods are far from being generic. In fact, in an operational context, it is required to use different tricks for different purposes. Pafish [324] or Al-Khaser [382] projects are a good illustration of different techniques grouped in a single project [391]. This strategy of detection guesses that one system can handle some detection tricks and not all of them. But this hypothesis is far from being accurate in practice. Technically speaking, it is a kind of race between detection and countermeasure. This is one of the main drawbacks of existing techniques of analysis environment detection. They are unreliable over the time. Thus, having an effective and generic method — capable of being effective over time — would be an important contribution.

The new detection method presented in this section is not subject to the previously stated problem. This one aims to be generic and hard to prevent despite the knowledge of mechanisms used to perform the detection. It is based on uncovered behavior of cache of CPU from Intel and AMD vendors.

### 4.3 Method of detection

#### Resume 18:

- 📖 This section presents our detection method.
- 🔗 Our method is based on an undocumented extend from a method to cross-modify code between two threads [4].
- 🔗 We explain how to exploit this cross-modifying code to perform a detection, based on what we suppose to be the cache actualization time-lapse.

### 4.3.1 Where the idea came from

#### Key Point 3.10:

- ☞ In the Intel documentation, it is written how to correctly implement certain procedures so that everything works as expected.
  - ☞ The good question is to know what would happen if these procedures are incorrectly implemented...
  - ☞ We are exploiting here the side effects and undefined behaviors in the Intel's documentation.
- ☞ One relevant mechanism to exploit concerns cross-modifying code between two threads.

Formally speaking, our new method is based on two threads performing cross-modifying code between each other. The idea came after reading Intel's documentation [4] which describes how to perform such operations (figure 3.18). Extract from the documentation below explains how it is possible to update code from one thread to another.

To write cross-modifying code and ensure that it is compliant with current and future versions of the IA-32 architecture, the following processor synchronization algorithm must be implemented:

```
(* Action of Modifying Processor *)
Memory_Flag ← 0; (* Set Memory_Flag to value other than 1 *)
Store modified code (as data) into code segment;
Memory_Flag ← 1;

(* Action of Executing Processor *)
WHILE (Memory_Flag ≠ 1)
  Wait for code to update;
ELIHW;
Execute serializing instruction; (* For example, CPUID instruction *)
Begin executing modified code;
```

(The use of this option is not required for programs intended to run on the Intel486 processor, but is recommended to ensure compatibility with the Pentium 4, Intel Xeon, P6 family, and Pentium processors.)

Like self-modifying code, cross-modifying code will execute at a lower level of performance than non-cross-modifying (normal) code, depending upon the frequency of modification and specific characteristics of the code.

The restrictions on self-modifying code and cross-modifying code also apply to the Intel 64 architecture.

Figure 3.18: Extract from Intel documentation [4] explaining how to perform cross-modifying code between two threads.

In Intel's documentation, one *execution unit* is called "*processor*", but we propose to keep the word "*thread*"<sup>5</sup> in the rest of the section since it is the most commonly used word in operating system world for different architectures. However, we should keep in mind that the two "*threads*" must be executed on different logical cores from the CPU. For the sake of simplicity, we call *modifying thread* the one responsible to modify, on-the-fly, the code (composed of *opcodes*) which is about to be executed by the *executing thread*. Note that the method could be used in the context of more than two threads with different graphs of representation with modifying threads and executing threads (with hybrid approaches where modified thread can be a modifying thread too). Such constructions are direct consequences of the approach we have with two threads.

For short, the method presented by Intel consists in locking the executing thread to subsequently modify its opcodes by the modifying thread. The lock mechanism is described through a *while loop* spinning on a memory flag value. Technically speaking, this loop can be seen as a *spinlock* [393]. Once modifications are done, the modifying thread unlocks the executing thread thanks to the spinlock mechanism. After that, the executing

<sup>5</sup>Note that the word "*task*" although used in Intel's documentation might have suited the situation perfectly. But this one is a bit less used in operating system world and it would not have suited the implementation of code example provided in the following to interface with API from Windows and Linux.

thread executes the modified code. More than trying to reproduce this algorithm, we wondered what would happen if the proposed steps were not followed. Especially if we do not respect the synchronization mechanism proposed here.

### 4.3.2 Detection mechanism

In our case, we decided to synchronize the threads before the code modification (see Figure 3.19). More directly, we are going to remove the spinlock (before modifying opcodes). By swapping these two steps from Intel’s procedure, the code modifications are not taken into account by the executing thread. Indeed, the spinlock removed, the original following opcodes are executed long before the modifications made by the modifying thread are visible to the executing thread. It is always the case on a real hardware system.

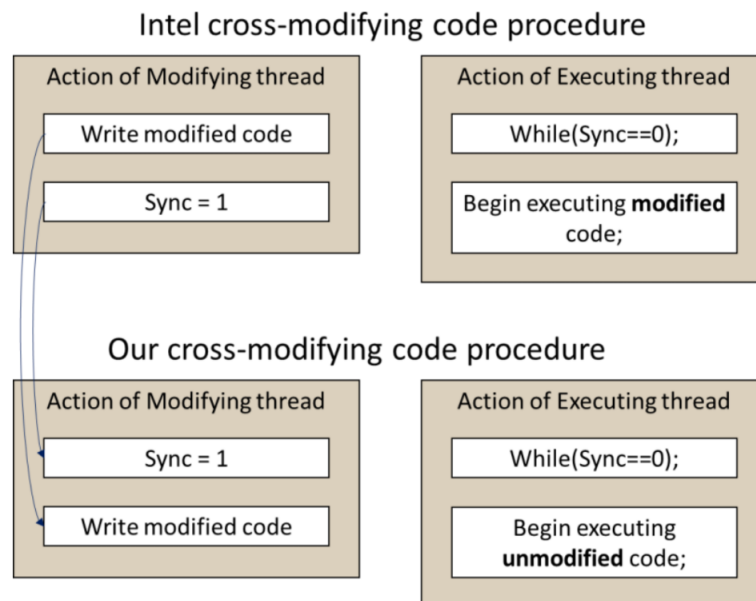


Figure 3.19: Intel’s cross-modifying code procedure.

The explanation of this observation lies in the way that the CPU’s cache works. Indeed, mechanism of synchronization before opcodes modification is presented in the Intel’s documentation [4] for at least two reasons. First, it ensures that the operations performed by the modifying thread are finished before letting the other thread running. Secondly, it allows the cache to be synchronized correctly between the two threads which are working on different CPU’s cores. Thus, the illustration of our method presented in Figure 3.19 is not actualized because the cache may not have sufficient time to synchronize.

Therefore, it becomes interesting to understand how the executing thread can take into account modified opcodes from the modifying thread. If we increase artificially the execution time between the synchronization spinlock and the spot where modified opcodes are supposed to be executed, it is possible to observe the executing thread executing modified opcodes, instead of original ones. Indeed, the CPU’s cache has a large enough time-lapse to perform an update (Figure 3.20).

The question is about to know how analysis environments manage such case of opcodes modification, taking into account that our method cheerfully transgresses official Intel’s documentation procedures. Indeed, there is no official way to manage this behavior, since nothing has been officially documented by Intel about these special circumstances. In addition, by design, analysis environments increase generally execution time of any code. And this the main problem we are facing. Since the execution time is different and longer than without analysis environment, the cache is updated. It means that modified opcodes are executed even if they should not be executed in a real environment.

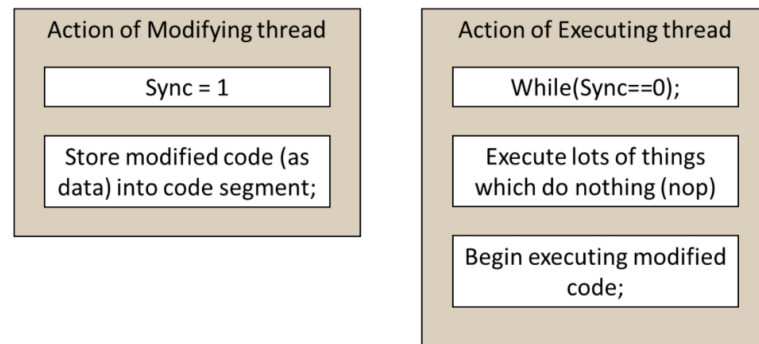


Figure 3.20: Modified cross-modifying code procedure.

By consequence, our detection method is based on watching if there is any delay in the execution of modified opcodes by the executing thread. The watch is performed via the check on the result of the operation modified (or not) after synchronization spinlock has been released. If the result corresponds to original opcodes of the modified thread, it means that nothing delayed it or updated the cache. If the result is different from original opcodes, it means we are dealing with an analyzed environment (see Figure 3.21).

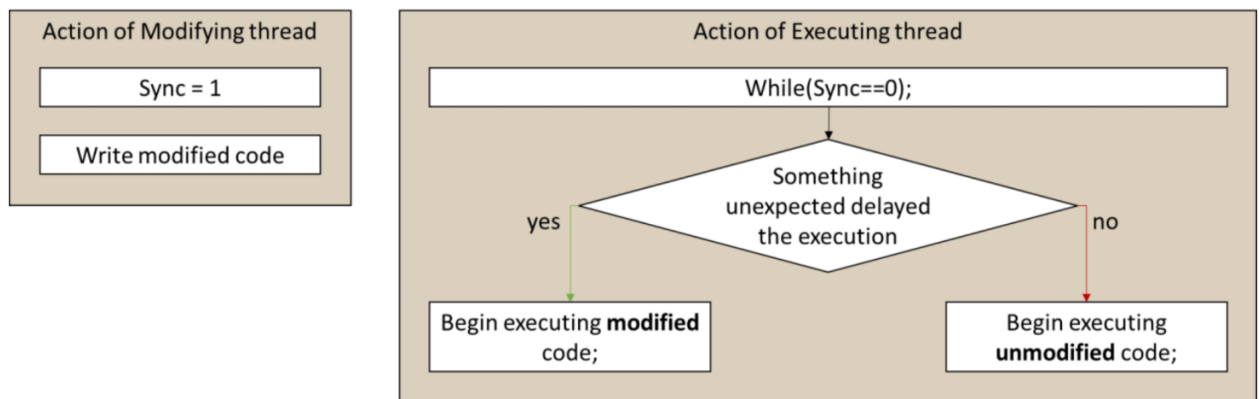


Figure 3.21: Basic detection mechanism



### 4.3.3 General resume

#### Resume 19:

- ☞ We explain in this subsection how to incorrectly manage cross-modifying code between two threads to design an analysis evasion mechanism.
  - ☞ The main idea is to not respect Intel's requirements about synchronizing cross-modifying code.
  - ☞ We expect that the cache system used to execute instruction quickly by the CPU will not be correctly updated on a real machine (due to a short timing attack).
  - ☞ But in an analysis environment, things are going slower than a real one. And cache update can occur automatically — thanks to the design of such environment — where it should not happen on a real environment.
  - ☞ This is how we build our evasion technique.

Our technique is based on the difference of cache refreshment between execution on regular hardware without any analysis environment (ie: software in between the CPU and our running code). In a general environment, the behavior can be seen as given in the time-line provided in Figure 3.22. This illustration aims to describe step-by-steps the different changes and interactions between the modifying thread and the executing one. In green, we have represented the current step currently executed by a given thread at a given moment.

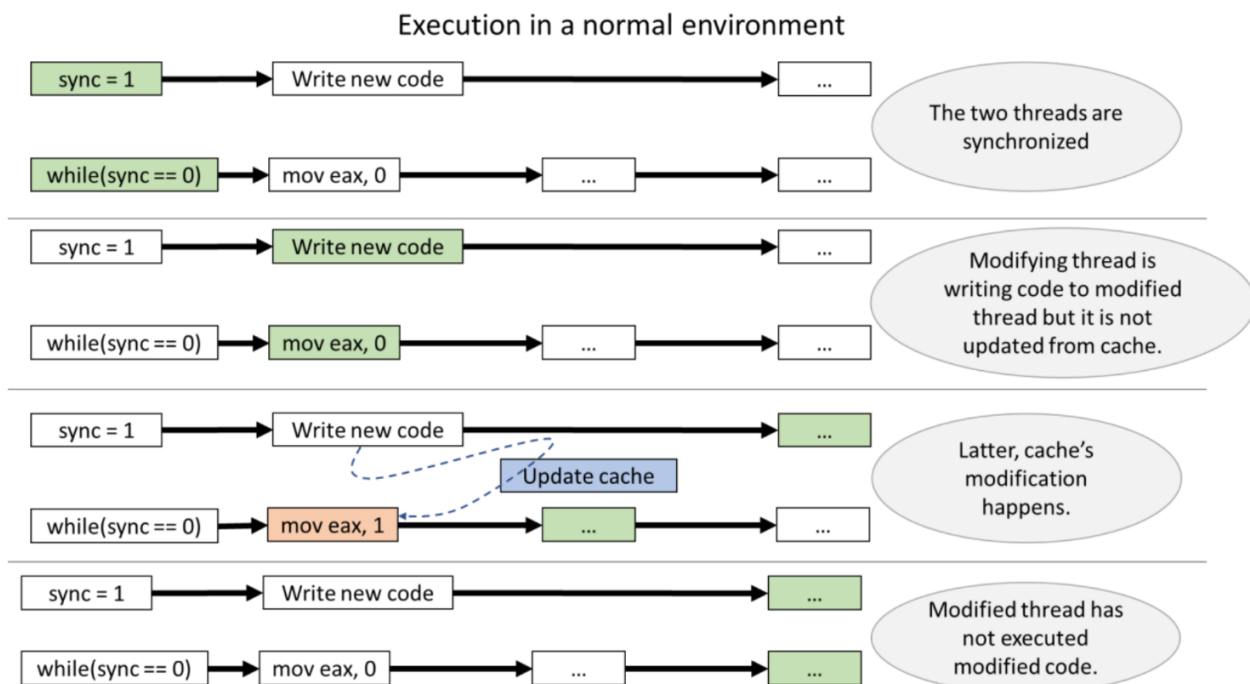


Figure 3.22: Regular execution of our method without analysis environment.

In the case of Figure 3.22, we can see that once the spinlock has been released by the modifying thread, the executing one is about to execute the following opcodes long before the modification of opcodes is performed and long before cache refreshment updates opcodes of the executing thread. Speed and timing are key concepts here.

When we are dealing with an analysis environment, execution is quite different. There is time between the release of spinlock and the actual execution of following opcodes. This time-lapse allows the modifying thread to change the following opcodes of the executing thread. More than just changing them, there is enough time so that the CPU's cache is updated, allowing modified opcodes to be executed. This is illustrated by the time-line given in Figure 3.23.



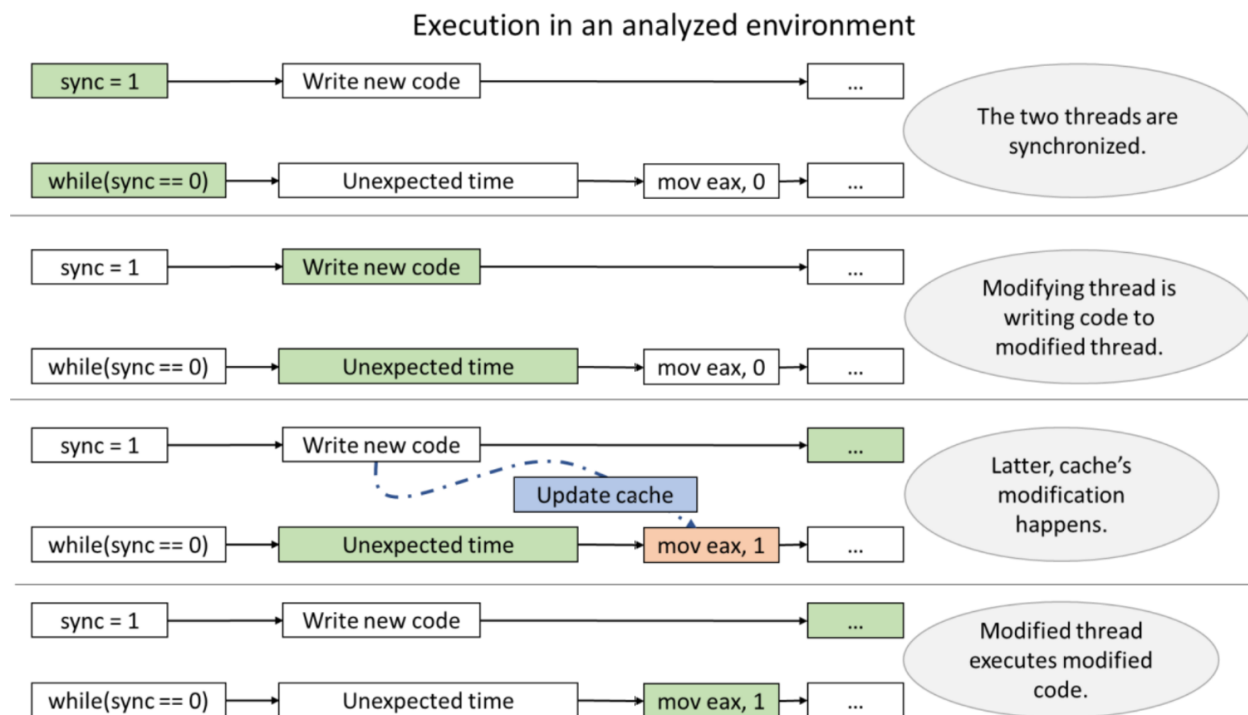


Figure 3.23: Execution of our method on an analysis environment.

Contrary to a regular environment, opcodes modified are executed by the executing thread. It results that the behavior produced on the executing thread is different to the one observed on a real environment. This difference is enough to build a detection method. More than just returning true or false about detection, it allows a full conditional modification of the following code, depending if we have detected that our executed code is under analysis or not.

Note that this operation must be performed by at least two different threads running in a real parallel environment. Pseudo-multitasking is not enough to execute correctly our method. In fact, due to the different cache's rules of the pseudo-multitasking mechanism, applying our method on a single CPU would trigger false positives. Our method needs to run on two different cores to be efficient.

#### 4.3.4 Implementation

Implementation of the code is quite simple and it does not require specific skills. A version written in C and assembly code for Windows is provided to easily build shellcodes (but it can be easily adapted to be running on Unix operating system). This one is given in codes 3.3 and 3.4. The code 3.3 is written in C to launch a dedicated thread (with `CreateThread` function [259]) holding our detection method after changing the rights of a memory page (with `VirtualProtect` function [394]) to "read-write-execute" the modified page where the original code belongs.

```

1 HANDLE hThread;
2 DWORD oldProtect;
3
4 // Change the page's rights to allow both execution and writing
5 if (VirtualProtect(ModifiedThread, 100, PAGE_EXECUTE_READWRITE, &oldProtect) == 0) {
6     return -1;
7 }
8
9 // Create a new thread able to modify the code to execute
10 hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadWhichModify, NULL, 0, NULL);

```

```

12 if (hThread == NULL) {
13     return -1;
14 }
15
16 // Execute the function that is modified and display its result
17 printf("%d", ModifiedThread());
18
19 //Restore the right of the page
20 VirtualProtect(ModifiedThread, 100, oldProtect, &oldProtect);
21
22 return 0;

```

Code 3.3: Launch of the detection procedure in a dedicated thread (Windows).

The assembly code 3.4 holds the `ModifiedThread` function we use for detection. This one is written such as it both works on a 32-bit (x86) and a 64-bit (x64) CPU. For the sake of completeness, it can be assembled using NASM assembler or MASM assembler with few modifications. That one first sets both "Synch" and "codeToModify+0x01" values to one. The first is used to wait until the modification of the current code is actualized (wait operation is performed with the couple of `cmp` and `jz` instructions). The update procedure of the opcode at offset +1 from `codeToModify` is used to change a "mov eax, 0" to a "mov eax, 1" instruction. This way, the function returns 0 or 1, depending if the update has been clearly taken into account. Note that "Synch" value is held in `.data` section while `codeToModify` value is directly in the body of the function (as a regular opcode). We expect a difference of synchronization between the `.data` section and the self-modifying memory code page to allow the detection of an analysis environment.

```

1 GLOBAL _ModifiedThread
2 GLOBAL _ThreadWhichModify
3
4 section .text ; makes this executable
5
6 _ThreadWhichModify:
7     mov byte[rel Sync], 1 ; Synchronize the thread, but to early
8                             ; in order to exploit the FIFO
9                             ; property of the processor cache.
10    mov byte[rel codeToModify + 1], 1 ; Transform 'mov eax, 0' into 'mov eax, 1'.
11    ret
12
13 _ModifiedThread:
14    cmp byte[rel Sync], 0 ; Wait for synchronization.
15    jz _ModifiedThread ;
16
17    ; If a DBI is here, or a debug breakpoint, the detection is about to success.
18    ; In order to detect VM, we should add several CPUID here (depending on the CPU).
19
20    codeToModify: ; 0B8h is the opcode for mov eax, imm32,
21                    ; where imm32 is 0 in our case.
22    db 0B8h, 00h, 00h, 00h, 00h ; mov eax, 0 -> about to be modified in mov eax, 1
23
24    ret
25
26 section .data ; Makes this writable and ensure
27                ; it is not in the same memory page
28                ; than the code above.
29    Sync : db 0

```

Code 3.4: Assembly code holding the detection procedure.

For the sake of reproduction, a complete version of the project can be found in the following github link: <https://github.com/fplu/Detector>. This project gives full sources for the tool able to run on Windows and Unix operating systems. There is no real difference from the algorithmic point of view between the two operating systems. The only difference lies in the API they are providing to change memory rights. This project allows everyone to verify the veracity of our assertions since there is nothing complex about recompiling these

codes and executing them in an analysis environment, as described in this section. There is no need for a dataset, just to install the analysis environments.

## 4.4 Detection of analysis environment

### Resume 20:

- ✎ In this subsection, we propose to test our method on three different analysis environment (DBI, Debugger and virtual machine).
  - ✎ We present what the *Dynamic Binary Instrumentation* (DBI) is, how it works and a state-of-the-art (similarly to the one given in section 2) about DBI evasion methods.
  - ✎ We show that results of evasion by our method on debuggers and DBI is at 100 %.
  - ✎ We show that results of evasion by our method on virtual machine environment is probabilistic but close to perfection with correct calibration for a given CPU.

We tested different DBI frameworks with our method compared with an environment where there was no analysis tool. Indeed, since our detection method is based on the delay of cache actualization to execute opcodes modified by another thread, it is possible to detect different analysis environment since all of them potentially induce time between execution of different opcodes. We are able to detect, several kinds of tools: Dynamic binary instrumentation (DBI) framework, debuggers and virtual machines (VM).

For our tests, we created a small software program that uses our new detection method. This one is common for both DBI and debugger environments. The case of VM detection requires to add several CPUID instruction to force a cache synchronization but the logic remains the same. This evaluation software, when executed in a clean environment displays 0 on the screen if it does not detect any analysis environment. Otherwise, it displays 1.

### 4.4.1 Dynamic binary instrumentation framework

#### Resume 21:

- ✎ In this subsection, we present what a *Dynamic Binary Instrumentation* is, how it works and why it is vulnerable to our evasion method by design.

#### 4.4.1.1 Introduction about DBI technology

One of the categories of tools we detect is the *Dynamic Binary Instrumentation* (DBI) framework. The goal of this kind of tool is to be able to make an automatic and detailed analysis of different thread in a target process. In order to achieve this goal, they can put hook everywhere in the target process, including between instructions. Such design implies to strongly modify the execution of a program by editing the entire code running, on-the-fly. For the sake of simplicity, depending on the hook granularity provided by the DBI framework, it is possible to monitor specific instructions, block of instructions or functions. In a way, we can talk about *basic blocks* and the *granularity* of the partitioning of these basic blocks. This is done by a permanent alternation between code executed by the DBI and code executed by the original process. All this context switching (notwithstanding the inducted amount of code executed) increases the execution time compared to a regular execution in a clean environment.

Why focusing on DBI? *De facto*, DBIs are only automated debuggers. More directly, they have the same capabilities as debuggers (step-by-step execution, directly on the CPU of the host machine) but they do not require human interaction, in the sense that they are programmed to react automatically where a human was required with a traditional debugger. This allows to gain speed and therefore increasing analysis capacities. Originally, DBIs were — among other things — mostly used to search for vulnerabilities [395, 396, 397]. Of course, this type of tool is especially useful when it comes to analyzing potential malware [398]. It combines the

qualities of a debugger while minimizing human work. Literature references many projects about it [399, 400]. Note that DBI can be used in conjunction of hypervisor, to benefit from both world (DBI for analyzing smoothly what is happening on a given process and the hypervisor (VM) to contains potential damage produced by the malware). For instance, in [401], the goal is to bridge the automatic and manual analysis processes. That is to say, to offer an environment where dissection can happen without incurring the anti-analysis hassle and benefit from capabilities missing in previous approaches, e.g., stealthy instruction patching, cloaking of tools, and surgical use of program analyses.

If analyses can be performed in the context of a DBI, it makes sense for malware vendors to develop strategies to evade them [402, 403, 404, 405]. In [403], authors have performed an impressive state of the art on the various evasion strategies in the context of a DBI analysis. Their study is based for a good part on the work of [405] who had realized a similar state-of-the-art about the different techniques used to escape to DynamoRIO [69], while proposing tracks of remediations as well as the possible consequences to see implemented these solutions. The contribution of [403] consists in generalizing the state-of-the-art about DBI evasion and escape problems. This one has been extended to other DBI (in particular Pin) where attack surfaces, transparency concerns, and possible mitigations have been evaluated. Finally, a *library of detection patterns and stopgap measures* has been presented in this paper for DBI users.

We find DBI tools on different CPU architectures (Intel, AMD or ARM [406], for instance). There are different methods to design DBI [407] which implies different performances [408] and different methods to detect them. Kirch et al. [402] give many description of different DBI detection techniques. These techniques have been grouped and sometimes improved by Zhechev [404, 409]. An efficient one is about *self modifying code* based mainly on work from Mario Polino [410]. Zhechko [404] presents a tool called PwIN to detect with different methods Intel Pin DBI [68, 411]. But there are some methods which are specific to few DBI tools [412].

#### 4.4.1.2 Technical elements to perform DBI detection

Without pretending to be as exhaustive as given in [403, 404, 405] but based on their work, we propose to evoke here a synthesis of various existing techniques allowing to detect DBI and how to possibly avoid such a detection. In practice, there are several possibilities to carry out such an action. The attack surface is composed of different elements:

- *Time overhead*: Because a DBI needs to decompile, translate, and instrument the original instructions it traces introduces an inevitable slowdown in the execution comparing to an execution without the DBI. The overhead inducted by a DBI is not easy to hide and it can easily measured<sup>6</sup> not to mention any side effects in consequence to time changes about time-sensitive code. A solution can be about faking the results of time queries from different sources (API or instructions) [410]. But at the difference with virtual machine sandbox where it is possible to master time sources, in the context of the DBI, this operation is much more tricky. In the end, a malware could always query other processes not running under DBI or external attacker-controlled time sources, as a web-server.

The best solution is perhaps to have a DBI as efficient as possible, in order to minimize the impact produced (and to postpone malware analysis once the information has been captured by the tool and not at runtime).

- *Leaked code pointers*: Since execution is instrumented in the context of the DBI, there is a *decoupling* between the real instruction pointer and the virtual one exposed to the analyzed code. Under some assumption, there are two famous ways to get access to the real instruction pointer for an analyzed code. One is about special x87 instructions that are used to save the FPU state: `fstenv` and `fsave` [413]. These instructions keeps a structure in memory where one field holds the instruction pointer value for the last performed FPU instruction. The other way is in 32-bit Windows version with `int 2Eh` instruction. That one is normally used to enter kernel mode on such systems. But, by clearing `eax` and `edx` registers before

---

<sup>6</sup>With the same logic and possibility the same API as presented in section 2.2.4 with timing-based fingerprinting techniques.

invoking it, the real instruction pointer address (eip register) is leaked to edx [410].

The solution to both cases is to act when the DBI is about given the hand to a section of code containing such instructions. By decompiling and patching both instructions, it is possible to handle it correctly. Of course, it increases the overhead DBI framework.

- *Memory contents and permissions:* In a lot of architectures of DBI, both DBI engine and targeted code share the same address space without a perfect isolation. By looking in the the address space for additional sections or unexpected exported functions from non analysis run-time, it is possible to detect the DBI [413]. In addition, memory consumption is higher than without the DBI because of the use or extra-code to drive the DBI analysis. In the end, some memory rights (page execute, read-only, ...) are not perfectly handled by DBI [414, 402]. This last point is usually to allow programs like *Just-In-Time* compilers (JIT) to be executed correctly. It is a kind of compromise between an efficient (and therefore relatively fast) execution and a certain freedom offered to the instrumented process.

This is why there is no simple solution dealing with such issue. On the first hand, checking all memory accesses would solve the evasion problem but the cost in performance would be dramatically high. On the other hand, solutions based on underlying hardware such as non-readable executable memory (XnR) [415] or thin-hypervisor solutions based on *Extended Page Tables* (EPT) technology. But such solutions are nowadays quite experimental.

- *DBI engine internals:* Since all thread are sharing the same address space, it may be relevant for each thread to get access to a unique data spot, accessible using a global index. Such a place is called *Thread Local Storage* (TLS) [416, 417]. The problem is that very frequently, in order to have a tight management for each thread, DBIs monopolize some entries in the TLS [418]. By trying to check if a slot at an unexpected index is already reserved or by trying to allocate all, a malware could detect the presence of the DBI. With comparable logic, a DBI may legitimately need to know when a thread is about to be created by a process, just to be able to follow it. To proceed, the DBI often hooks<sup>7</sup> [419] certain functions of the OS [410].

In practice, there is no real perfect solution here. On the one hand, apart from trying to do without TLS (by using an *ad-hoc* system that mimics TLS' properties), there is no solution. On the other hand, we can try to hide the hook procedures by controlling the memory accesses (at the cost of an important performance loss) or to control the actions from a driver (via `PsSetCreateThreadNotifyRoutine` routine [420]).

- *Interactions with the OS:* DBI acts at the same level as analyzed process, in an identical environment. In fact, they have the same rights since they are usually part of the same process. This means that a malware can try to detect them through the operating system. And all shots are allowed. Looking for the parent process [413, 418], listing all active processes to reveal Pin or DynamoRIO, checking *handles* [421] list available for objects (files, registry keys, and so on) manipulation (more or less handles could be present, not to mention that some of them can have a name in full-text, formally identifying the process). More generally, any tricks provided in sections 2.2.5 2.3.4 seem to be good candidates to attempt an escape.

To avoid such issues, for instance, DynamoRIO intercepts memory query operations about to access its own code to make believe that areas are free [405]. Concerning the interface with the rest of the system, it is a subtle mix of very targeted hooks on some sensitive APIs or the use of a driver whose goal is to use rootkit techniques to hide the critical parts of the DBI. This is a reminiscent of a game of cat and mouse.

- *Exception handling:* As explained in section 2.2.4, *SEH* Exception handling [240] can be very useful for evasion tactics. The same way, DBI needs to handle such exception handler carefully. Indeed, if a caught

---

<sup>7</sup>For short, a hook gives access to a targeted function before (or after) that one is about to be called by a another function. This can be done with several means, more or less discreetly, more or less efficiently.

exception occurs, the DBI must be able to follow the execution of the handler function (the *except* section of the *try/except*). This operation is far from being obvious since exception handling mechanism is passing through kernel management thanks to software interruptions. The DBI lost the hand for a little time-slice and the goal is to get it back before the analyzed process. Internally, it supposes to hook undocumented function such as `KiUserExceptionDispatch` from `ntdll.dll` in Windows. Few DBI are able to do it correctly since it is not a classical function and managing it correctly might induce a higher runtime overhead.

- *Translation defects*: To work properly, a DBI generally needs to be able to disassemble on-the-fly the instructions it is going to execute from the analyzed code. For instance, the `enter` instruction that is not implemented in Valgrind [402]. This may be justifiable because some instructions are rarely used or they are too intrinsically challenging for DBI to be perfectly handled. For instance, `far-jump`<sup>8</sup> instructions or any transition from 64-bit mode to 32-bit mode by altering the CS selector (sometime called "*Heaven's gate code*" [403]) [422] can be impossible or highly limited with some DBI (for instance Pin).

In the same way as in section 3 where we presented the exploitation of flaws in the reliability of the disassembler, the fact of badly disassembling or badly handling instructions obviously leads to potential evasions. Beyond this assertion, it may be possible to justify these issues. On the first hand, implementation gaps can be explained because of architectural issues (such as with transition from 64-bit to 32-bit mode) and in such a case, it is a known limitation. On the other hand, this issue can be fixed with extended features. On the other hand

- *Self-modifying code*: Self-modifying code (SMC) is always a problem for DBI. Why? Because often, for optimization purposes, instructions cached by DBI means there is code that has already been executed and which will not be re-evaluated by the disassembly procedure by the DBI. This is a very significant performance gain. But if a code modifies an already executed section of code, it becomes possible to escape. Worse, if the code self-modifies a small enough portion<sup>9</sup>, then it becomes possible to completely escape the DBI without even having to attack the cache system. In 2010, Martignoni et al. reported: "*the presence of aggressive SMC prevents the attacker from using efficient code emulations techniques, such as dynamic binary translation and software-based virtualization*" [423].

The case of dealing with SMC is highly complicated for DBI. DynamoRIO handles it correctly, while Pin provides a strict SMC policy option to deal with such cases from its 3.0 release [411, 403]. Such an optional switch is possible at the cost of an increased overhead about performances.

#### 4.4.1.3 Particular case of self-modifying code for DBI

As explained before about SMC, DBI tools need to face the case of execution of *Just-In-Time* (JIT) compiled code. Used by .NET technologies [424] and others, JIT technology represents a great challenge for DBI which have to deal with code coherency while keeping strong execution performances. Indeed, a modification of an executable buffer at running time is translated by the DBI to be correctly handled by itself, but there is no correct way to simulate the cache refresh of such procedure by the DBI. Either the DBI lets the cache of executable buffers being refreshed by themselves, but the charge of code inducted by the DBI offers quite a big time-lapse to update the cache compared to a real environment. Or they reproduce modification when asked and they have to manually deal with the flush of the cache to allow consistency of execution. It means that the modifications of opcodes are propagated instantaneously from the eyes of the executing thread which is targeted by the modifications.

In all cases, modified code does not become effective the same as way in real environment. It is very hard to mimic the real behavior of cache actualization since it is not documented by CPU vendors. It could be possible with calibration tests. However, this is far from being obvious in the context of multiple threads. Another solution would be to not take into account JIT code, as some DBI do and an easy way to escape them. But it

---

<sup>8</sup>A far-jump is a jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an inter-segment jump.

<sup>9</sup>The size of a basic-block if the execution granularity is this one.



would result in a large break of support for a lot of applications which use that technology. Finally, it means that DBI tools are condemned to take modifications of run-time code into account even if it would not be taken immediately into account by the processors in a regular environment. This is why they are perfectly detectable with our method.

#### 4.4.1.4 Detection of DBI with our method

We tested different DBI frameworks with our method compared with an environment where there was no analysis tool. The list has been crafted from previously existing studies [403]. We use a set of different analysis tools such as Pin [68], Flow<sup>10</sup>, QBDI [74], Frida [72], Valgrind [70], Anaconda [425], Qemu [281], and DynamoRIO [69]. A last one is referenced in our observation table 3.4 as *None*. This one correspond to an environment of execution where there is no analysis tool.

In the case where there is no analysis tool, the detection fails since, as explained in section 4.3.2, cache actualization has not enough time to happen. It is noted ✗ as observation in Table 3.4. In the case where there is a DBI framework analysis and the detection occurs, our observations are reported in table 3.4 and are noted ✓ in table 3.4. The second is where the DBI has crashed due to our detection method, it is noted \* in table 3.4. This can be due to a lack or a bad management of the JIT code by some framework. This is sometime a result of design choice to not support JIT code. Despite possible execution stop, such environments are still detectable. Indeed, it is possible to execute code in an exception handled section [240], where crash is correctly handled by our code. Since JIT is correctly supported by regular CPUs, any failure to execute such a code is a way to detect we are running in a modified environment.

	Pin	Flow	QBDI	Frida	Valgrind	DynamoRio	Qemu	None
Result	✓	✓	*	✓	✓	*	✓	✗

Table 3.4: Results of DBI framework detection test.

During our experiments, in the case of detection, the detection is always present on the DBI, as far as we have carried out our tests. There has been no false positive or detection error observed. As implemented and given on the internet, our method is calibrated to succeed on all the materials tested. After all, this is what is expected of it. But why there is no false-positive detection? After all the question is legitimate, it could happen that the cache does not update as slowly as one might observe. To answer this, it is necessary to understand that the additional activity induced by the DBI is very important, in particular to correctly process the self-modifying code or simply to take and to give back the hand between two basic-blocks (with the one executed by the modifying thread and afterwards, with the one of the modified thread). The cache has plenty of time to update itself during the execution time of the instructions that are specific to the DBI (and that are not executed when the DBI is not there, which changes the execution time considerably). This difference allows a systematic detection. The CPU cache update time is negligible compared to the DBI run-time.

Nevertheless, it would be interesting to test on more hardware and on different CPUs. We limited ourselves to several models of Intel and AMD brands, those we had, without observing any break in the detection method. Further studies, on different CPU architectures (ARM, in particular), could also make sense. Nevertheless, within the scope of our research, we did it with the equipment that was ours and with as many as possible different machines (notably with the help of students coming from ENSIBS school who were volunteers to take part in the tests).

#### 4.4.1.5 Difference with existing self modifying code methods

As explained in section 4.4.1.2, Kirsch et al. [402] propose a method to detect DBI framework which might look close to our. Indeed, in their case, they propose to change the execution rights of a memory page to read-write-execute. In case of success, they are going to modifying on-the-fly coming instructions to be executed. In

<sup>10</sup>Until now, Flow is a private DBI tool which has been developed in our laboratory. It is thanks to the development of this tool that we have been able to understand in detail the functioning of DBI and thus having the possibility to carry out these researches.

a general CPU, modification is automatically taken into account. But running in the context of some DBI, this assumption is no longer true. For instance, the following code change the value of RAX register from 1 to 0.

```

1 | ; Evaluation procedure with a self modifying code.
2 | call $+5                ; e800000000    - Push the current rip register.
3 | pop rax                 ; 58          - Store in rax the value of rip.
4 | mov BYTE PTR [rax+4+4], 0 ; c6400800    - Modify the third opcodes of the following
   |     instruction.
5 | mov rax, 1              ; 48c7c001000000 - Modification should change instruction to "mov rax,
   |     0"

```

Code:

Executed with Pin DBI, the result is zero. But it is possible to manage this side effect with Pin by the use of `"-smc_strict 1"` command line argument [411] as explained by Zhechev [404]. When this command line is provided, the instrumentation platform starts monitoring self modifying code. When such situation happens, Pin raises a code cache invalidation notification followed by recompilation of already present code in the code cache [404]. Such a way, modified code is taken into account.

One might think that our method could be countered by this approach. That is not the case. We detect Pin DBI whatever the command line `"-smc_strict 1"` (or `"-smc_support"`) is set or not. This is due to the fact that our method does not rely on *self modifying code* but on *cross modifying code* [4]. Indeed, the approach of Kirsch et al. [402] is based on modifying the next opcodes to be executed on the current thread, which corresponds to *self modifying code*. But the procedure they are using is not exactly in the expected shape of Intel's documentation because they are not using jump or a serializing instruction (CPUID for instance). The idea behind this procedure is to force the CPU to take the modified code into account again to force the cache to be updated.

Not doing so leaves the authors in a situation described by Intel as a *blurred* state. Even if it is not documented to work, in reality, when a thread modifies an instruction in an area close to the current opcode executed, the cache actualizes instantly. It means this *close update* of opcodes is likely to be instantly taken into account by the modifying thread. More directly, if a thread modifies the next instruction to be executed, then this thread is going to execute the modified version of the coming instructions.

And it is precisely on this principle that [402] bases its detection method. Without a DBI, the opcode change is always executed because the CPU takes it into account (albeit this is not documented). But in the context of a DBI, changing the following opcodes for a thread does not make much sense. Technically speaking, the opcodes following those executed in the context of the DBI are those of the DBI itself (to take the control, to analyze what has been executed, to prepare the next execution and so on). We can understand why the DBI then reflects the modification not on its own opcodes but on those which will be executed afterwards by the DBI. And DBIs follow the Intel documentation, because they have no choice but to ensure consistent execution. And also because Intel's documentation, in such a case, is the first — not to say the only — source of information. In code of self-modification-code, DBI tools are waiting for a serializing instruction to update the modified code. And it is by respecting the Intel documentation that they end up performing a behavior that is different from reality (there is no need for a serializing instruction to update the cache on a code segment very close to the one currently executed).

Our approach based on *cross modifying code* is different because we are acting on opcodes from another thread running on a different CPU's core, in a way which is not planned by Intel's documentation. Moreover, the approach shown in [402] consists of detecting Pin by not viewing the cache actualization of the code while our method consists of detecting DBIs framework by making them to actualize the code when they should not. It means we are changing opcodes on different cores so on a different CPU's cache. In such a situation, it takes time for the thread whose code has been modified by the other thread to *see* the changes. DBIs such as Pin are naturally trapped since they are forced to handle the case of cross-modifying code while the operations they conduct behind the back of the program cause a premature update of the cache and thus a detection. Moreover, Pin is following the documentation from Intel to update opcodes in cache, but our method does not follow the documentation so that we are exploiting a side effect. A side effect which is not taken into account by Pin and



others DBI. In consequence, this difference of strategy allows our method to provide a more efficient method than those existing about self modifying code with DBI tools.

#### 4.4.2 Debugger

##### Key Point 3.11:

- ☞ Debuggers are systematically detected as long as they execute our step-by-step procedure.
  - ☞ In automatic mode (different from step-by-step), they have so little impact on the system that they cannot be detected with our method.
  - ☞ At the opposite, in automatic mode, it is possible to evade them with one of our method (see section 3.1 — Key-Point 3.4).
  - ☞ In this case, our method can be used to protect a very specific region of code that should not be debugged.

Another example of analysis tool is debugger. This type is not systematically detected at the difference of DBI framework. More directly, it is only detectable when the debugger is trying to analysis a section of code, otherwise, execution is transparent. This is due to the fact that debuggers, when they are running code, are non invasive into the process. It allows an efficient execution as close as possible to a real environment. Operationally, we detect the debugger when this one is trying to interact with a section of code which is supposed to be protected by our method. Indeed, a breakpoint or a step-by-step execution is required between the synchronization mechanism and the modified code to detect it. This can be done when the debugger is targeting our code and a step-by-step execution is currently happening.

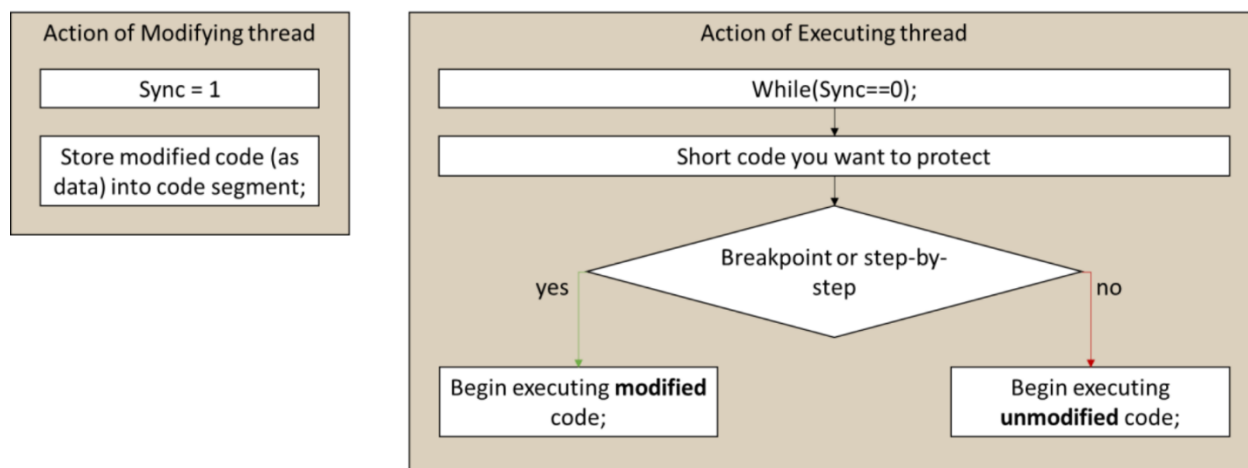


Figure 3.24: Debugger detection mechanism

Such a way, execution step-by-step increases the time between the release of the spinlock and the potential execution of modified opcodes by the modifying thread. This is where the difference with reality lies and how we can perform the detection. Of course, it supposes that only the executing thread is stopped by the debugger and other threads (including the modifying one) is still running. This is definitively a strong assumption. By contrast, it may be hard to detect the pattern of our method since we modify on-the-fly instructions, resulting in a great challenge to handle modifications propagated with a delay in the code. Indeed, there is no assumption about the shape of the modified code in the executing thread (it can be a value changed or a full update of instructions). Such design allows to protect one or several specific portions of code which can be analyzed only to prevent the analysis of the rest of the program thereafter.

We tested our method on different debugger software, privileging the newest and the most popular ones [426]. In the list of tested debuggers, we have GDB [207, 427], Microsoft Visual Studio Debugger (MVSD) [428], x64dbg [371], Ollydbg [429] and Radar2 [430]. There is no real difference under Windows or under Unix operating system since the logic and the hardware used are the same. Results are given in the table 3.5 that follows. As in section 4.4.1, we included an environment with no analysis tool to check false-positive detection.

	GDB	Windbg	x64dbg	Ollydbg	Radar2	MSVS	None
Result	✓	✓	✓	✓	✓	✓	✗

Table 3.5: Results of debuggers detection test.

Here again, the detection is always absolute and does not suffer from error. The reason is that the operation is precise and that the reaction time of a man who drives the debugger is much longer than the cache update. More directly, the analysis must focus on a particular thread while leaving the modifying thread active. This configuration remains rather "theoretical" although possible (especially if the modifying thread appears as a thread responsible for the GUI display). Nevertheless, it shows the generic side of the method which also applies to debugging tools.

#### 4.4.3 Virtual machine

##### Key Point 3.12:

- ☞ Detection of Virtual Machine is relevant if and only if a specific calibration is performed before running our method.
  - ☞ Calibration procedure concerns the of the number of CPUID instructions to use.
  - ☞ CPUID instructions are used to trigger a *vmexit* procedure from the guest virtual machine to the host (hypervisor). Such operation consumes time and allow the CPU cache to update cross-modified code, unlike a real machine where CPUID instruction is much faster.
  - ☞ In practice, with the machines we have, a number of twenty CPUID instructions is enough to have a reliable method.
- ☞ Although effective with, the problem of calibration is twofold.
  - ☞ It is complex to calibrate the operation without knowing in advance the exact CPU used. We depend on the CPU model of the hosting machine.
  - ☞ The method is no longer deterministic (as with DBI or debugger) but becomes probabilistic (with a possible false positive rate).

Our method allows detecting virtual machines or more generally any code running under the control of hypervisor technology. This one has been defined by Intel as VT-X [431] under other names for other vendors [432]. However, the VM detection method requires a little extra to our detection method presented until now. Indeed, the time elapsed between synchronization spinlock and modified opcodes is almost as short in the virtual environment as in reality. This way, the detection method fails if there is not enough time. To increase this time artificially in order to deal with hypervisor environment, we propose to provoke a *VMexit* operation which is a time-consuming operation.

Technically speaking, hypervisor technology allows execution of a code in a *virtualized* context. This code (called *guest* in Intel's documentation) is under the control of the hypervisor (called *host*). For different reasons (hardware access, memory management, some specific instructions, etc), some events coming from the guest code force the CPU to give the control back to the host. *VMexit* is the name of this operation. More directly, the guest stops running to give the control to the host, which is the hypervisor code itself. Once the hypervisor code has finished, the control can be returned back to the guest. This context switching and execution time of the hypervisor is not negligible and consumes times.

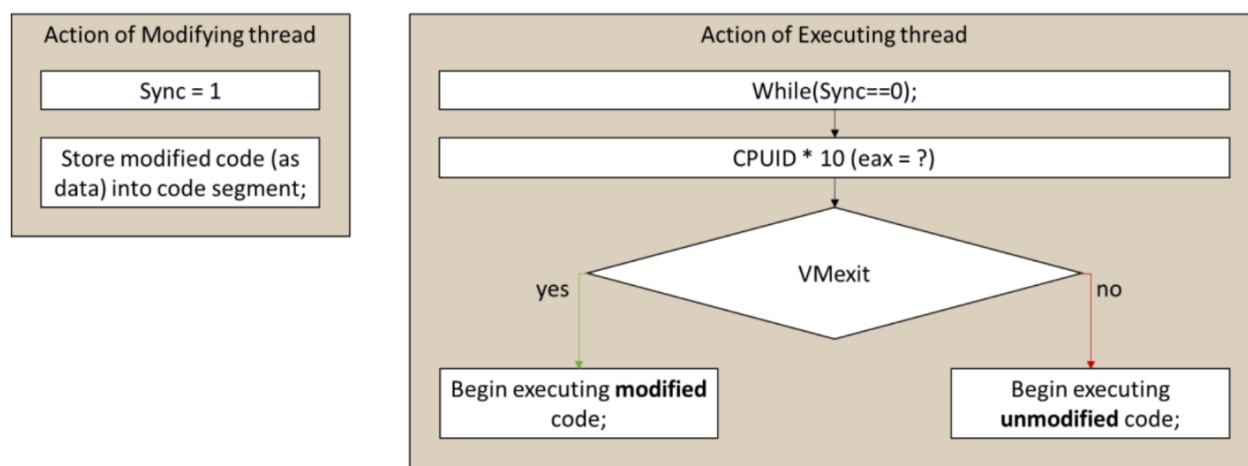


Figure 3.25: Virtual machine detection mechanism

To add *VMexit*, we propose to use an instruction which always results from a *VMexit* under a virtualized environment [433], such as *CPUID* [434]. If we add several *CPUID* instructions (about a dozen) between the synchronization mechanism and the modified code (see Figure 3.25), the time wasted in the hypervisor *VMexit*'s handler should be enough to let the cache refresh.

Normally, on a normal environment, about twenty *CPUID* instructions cannot give the sufficient time to the CPU's cache to actualize, a hundred of them neither. The exact number of *CPUID* instructions required is not really accurate until now and it is a source of future work to calibrate it efficiently. However, within the VM environment, when we test with only with ten *CPUID* instructions, on most of the machine we have, it is more than enough to cause the cache actualization. Such result allows to detect the VM. And this detection is reliable and efficient since Intel VT-X technology for hypervisor makes instructions like *CPUID* unconditionally exit the guest to go back to the host. Note that it is perfectly legitimate for software in user mode to use the *CPUID* instruction to detect which hardware assets are present and for calibration purposes.

We tested our method on different products of virtualization. Among them, we selected VmWare [435], Virtual Box [436] and Hyper-V [437, 438]. Results of observations are given in Table 3.6 correspond to the methodology provided in section 4.4.1 but adapted in the context of VM. More directly, we evaluate here the detection method on a number of machines at our disposal after having calibrated the number of *CPUID* instruction. Thus, we try to measure if it is possible to realize a detection or not (and not a detection rate, which would require more machines to test).

	VmWare	Virtual Box	Hyper-V	None
Result	✓	✓	✓	✗

Table 3.6: Results of hypervisor detection test.

## 4.5 Improvement of the test campaign and reproducibility of results

### Key Point 3.13:

- ☞ Due to a lack of machines to perform tests with virtual machine environments, we proposed to students in the university to test our method on their own machines to help us to evaluate its efficiency.
  - ✍ The diversity of machines used by students allows a more accurate evaluation of our method.
  - ✍ The diversity in the approaches taken by students to evaluate our method has been relevant to rise relevant remarks.
- ☞ Our detection method does not depend from the operating system from where it is executed (Linux or Windows, the result is the same).
- ☞ Our detection method has some limitations in the context of virtual machines.
  - ✍ The prerequisite calibration prevents malware to use the method in an operational context (since it does not know on which machine it will be executed).
  - ✍ Calibration on a given CPU does not necessarily work on another CPU, even if it comes from the same manufacturer.
  - ✍ The overload of different CPU cores can potentially pollute the results in some cases.
  - ✍ The strong use of CPUID instruction can be an easy detection pattern for an antivirus software.

We have to be direct and recognize that our detection method, if it is exact and deterministic within the framework of DBI and debuggers, remains probabilistic within the framework of virtual machines, in particular if the number of CPUID instructions is not skilfully adjusted. In our experiments on our own machines (with Intel and ADM CPUs), a number of 12 CPUID instructions is sufficient to achieve a detection without almost any false positives.

Nevertheless, if we can calibrate the experiment so that it works on our machines, we had to check it on a larger scale. With the help of the university where we were giving lessons in the context of our PhD, we proposed to the students to take the scientific article [176] as it was initially published (with the concept of the method used) to test it by themselves. The objective was twofold. On the one hand, to take advantage of the diversity of the machines that the students have to evaluate our method, especially concerning the detection of VMs. On the other hand, it aimed at checking the reproducibility of the results presented as well as having a critical feedback on the paper, as written. It was also an opportunity to exchange with students between the authors of a research paper and those who read it.

In order to be transparent about our approach, no instructions were given to the students except to read and check the validity of the paper. They were given *carte blanche* to evaluate the paper. The goal was to challenge our detection method and for us to see it stressed in conditions that we might not have imagined or dared to produce. It is an extraordinary way to keep the research alive and to allow, if necessary, new researches completed by the observations realized here.

From the feedback of our testers, several conclusions could be drawn. We compile here the main results directly extracted from the studies written by the students and provided to us. Here we have only made a synthesis of the student's work and gathered various remarks.

- DBI and debuggers have a successful detection rate of 100 % with the provided source code, as expected, until the experimental protocol given in the research paper is respected.
- Evaluations have been conducted on several different CPUs and disparities in effectiveness have emerged, especially to calibrate the number of CPUID instructions required. It became clear to the students that they needed to empirically find the ideal number of CPUID instruction that fits for their own machines.

More directly, this means finding the ideal number of CPUID to detect a VM while avoiding this same number of CPUID rises a detection when the program is executed on the host machine directly.

For the sake of illustration, we take an experiment performed on 3 different CPUs: Intel i7-7500U, Intel i7-8550U and Ryzen 7 2700. The experiment was done by successively recompiling the project but increasing each time the number of CPUID instructions by 1, 5, 10, 15 ... until 50. From there, each compiled program is run 100 times and the results reported as a ROC curve in Figure 3.26.

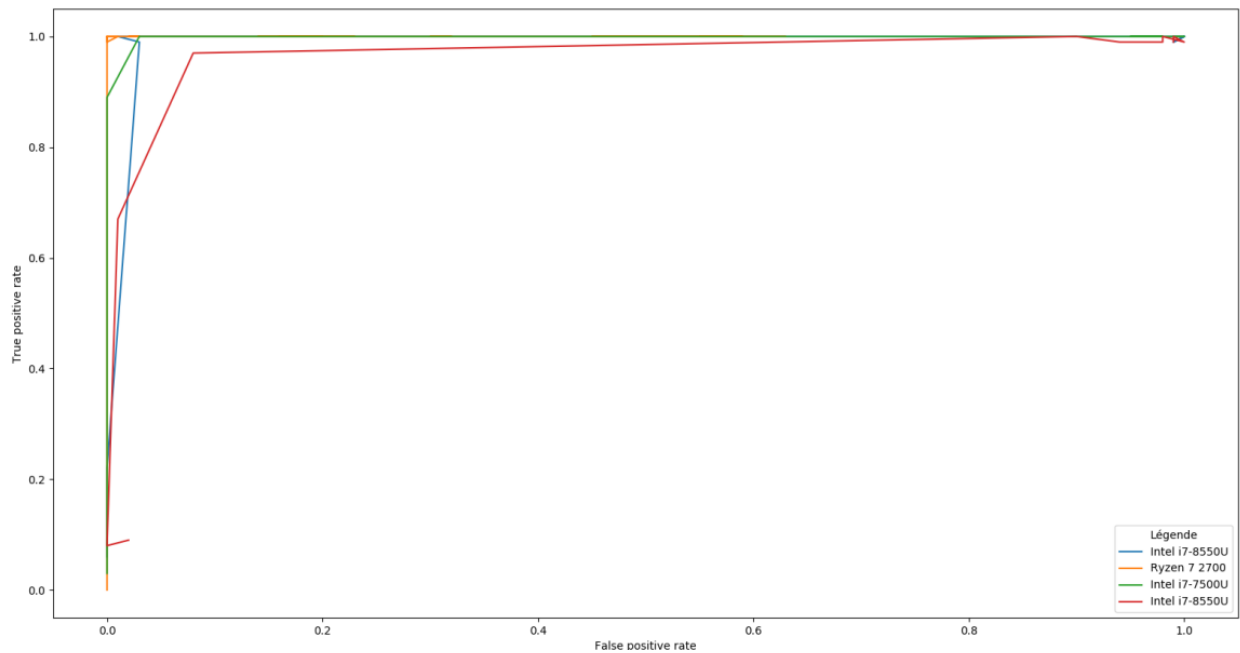


Figure 3.26: Comparison of the ROC curves for the different CPUs.

From these curves, several observations can be made. First of all, in this experiment, for a sufficient number of instructions (and in accordance with the idea of a dozen, twenty instructions necessary), the detection seems to work quite correctly (without being perfect). Secondly, there is some disparity between the two Intel CPUs, especially with the Intel 8550U. However, these two CPUs are not very distant in terms of the technologies they use. With closer inspection, it appears that the CPU cache specifications are different, despite their similar version numbers. This observation leads us to consider the problem of automatic calibration as being more global than simply being based on the characteristics of certain CPU families.

The last observation is that CPU Ryzen 7 2700 has a very low number of false positives (only one case in all), regardless of the number of CPUID instructions. In practice, by increasing this number beyond 50 (starting from 80 instructions in particular), the false positive rate increases. A similar case has been observed with Intel Core i5-7300HQ CPU. In this test, students used Windows 10 Professional 64 bits, build 1809 17763.1577 with VirtualBox 6.1.16 r140961 as host. In virtual machine, Windows 10 Professional 64 bits build 1909 18363.1256 has been used.

Another study was performed with three types of CPU (Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz, Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz and Intel Core i7 vPro), testing each time the host machine and two different virtual machines (one with Windows, other with Ubuntu). Extended tests shows that with a dozen of CPUID instructions, we observe a high detection rate with virtual machine environment while keeping a low detection level on the host machine. Nevertheless, it is noted that if a number of CPUID instructions can correspond to a VM/Host pair, this number is not guaranteed for another VM/Host pair.

- If the CPU architectures found on client machines can be measured, is it the same on server machines, sometimes equipped with different CPUs? After all, many VMs are hosted in the cloud, often on server architectures, quite different from popular client-desktop computers. An experiment has been performed with a bi-xeon 2x(4 cores /8 threads) 32 Gb memory and for each VM, 4 cores and 8 Gb of memory. The experimental protocol took here consist in increasing the number of CPUID instructions (fixed at compile time) from 0 to 30. Each test is performed 100,000 times and the percentages of success (or failure) are then reported in Figure 3.27 as a graph. The tests were performed on the host machine (*barre metal* machine) and in a VM, both using the server hardware.

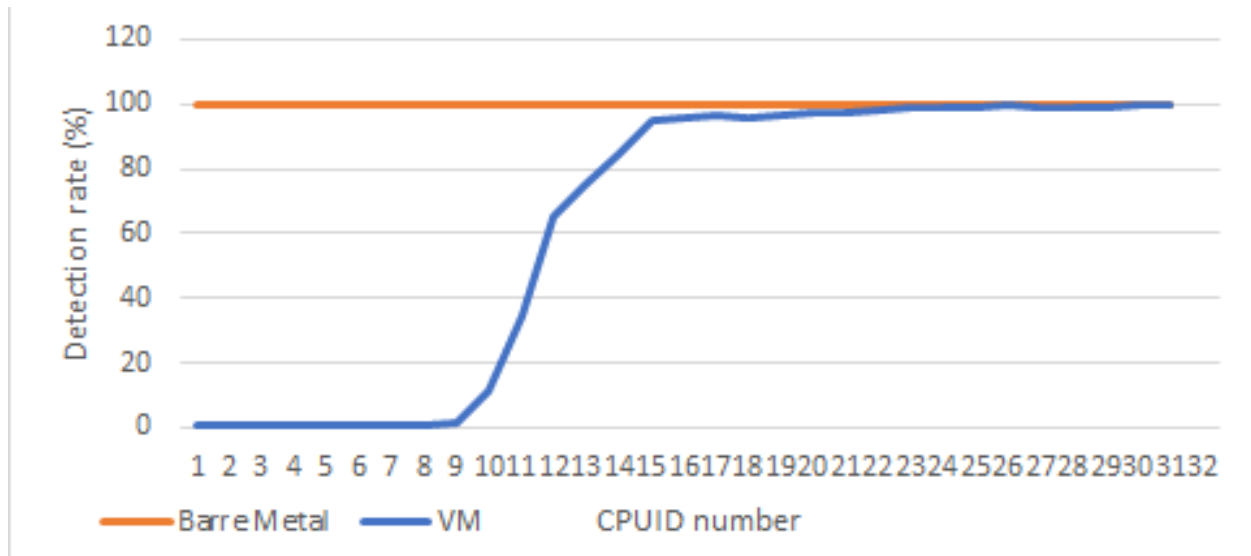


Figure 3.27: Detection rate of the method with server hardware configuration.

Two observations emerge from this graph. First, the false positive rate (erroneous detection of the VM when the method is applied to the host machine) is very low. On the other hand, the detection is done well for a certain number of CPUID instructions, like with a more classical machine.

- The *containerization* [439] approach has been elided from our study. For the sake of simplicity, this technology aims to isolate a process from the various resources of the system environment (CPU cores, RAM memory, I/Os on a hard disk, namespaces, and so on), for ergonomic or security reasons. Many solutions are present nowadays, most of them using the same standards based on *Open Container Initiative*<sup>11</sup> (OCI).

There are two main technologies for this purpose. On the one hand, *virtualized containment*, such as Docker [440, 441, 442, 443], which is based on the use of hypervisors. On this point, our detection works because these environments are virtual machines (though very light, which requires a fine calibration). On the other hand, at the opposite to virtualization, *containerized procedure* gives direct access to the system's resources, making the containment technology lighter (but faster) than the virtualized one. We propose to evaluate the two technology.

- A dedicated group of student was focused on *virtualized containment*, especially with Docker software [444]. This software is available for three operating systems: Linux, MacOS, and Windows. The experiment has been performed with a Intel Core i7 (eighth generation) by ranging the number of instructions from 0 to 20. Each experiment has been executed 100 times. The students assumed that each OS has its own way of implementing Docker. For instance, under Windows, a container must emulate a UNIX system and it must use its architectural notions to generate a partition [445]. This means that even if virtualization is quite special, it should still be possible. On a Linux system,

<sup>11</sup><https://opencontainers.org/>

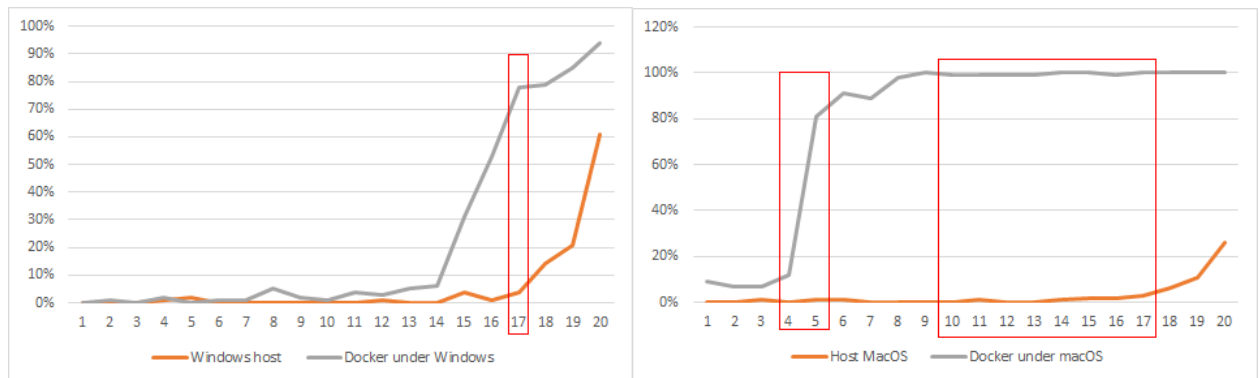


Figure 3.28: Detection of Docker under Windows.

Figure 3.29: Detection of Docker under MacOS.

the docker does not need to simulate an environment because the notions of *namespace* and *cgroups* are specific to the host system. Thus, because there is no virtualization, the detection should not be done. Finally MacOS being a proprietary system, based on UNIX systems, containers must virtualize a *LinuxKit* VM environment within Mac through the *xyhye* hypervisor [446, 447, 448]. This way, it should also be detected.

In Figure 3.28, we have the result under Windows. This one is partially detected when the number of CPUID instructions is equal to 17. But even in this case, the detection rate is about 78 % with 4 % of false-positive detection rate. This is not as good as with conventional virtual machines. Such results can be explained by the particular architecture used by Docker on Windows. Concerning MacOS, the results are excellent as showed in Figure 3.29. From 5 instructions, the detection is already very effective and between 10 and 17 instructions, it is almost perfect.

- The second technology is about *containerized procedure*, where no virtualization technology is involved. This is the case of the use of Docker under Linux. And after experimentation, the case of detection under Linux is consistent with expectations. Indeed, not being really controlled by a hypervisor, there is no *VmExit* due to the use of CPUID instruction and therefore no timing problem at the CPU cache level. Thus, there is no clear trend observed in Figure 3.30 to conclude a potential detection.

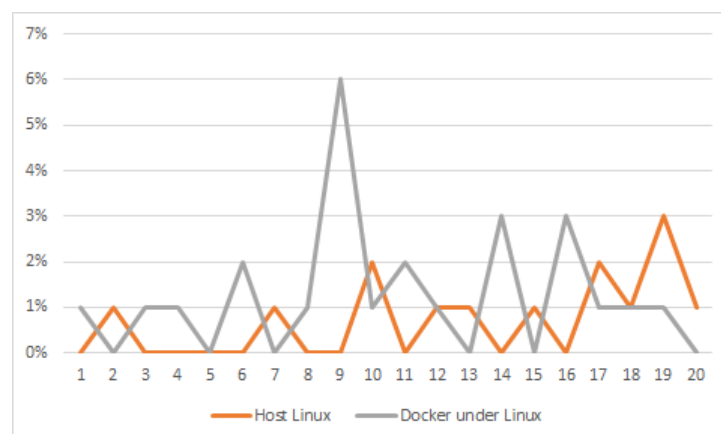


Figure 3.30: Detection of Docker under Linux.

- Concerning *containerized procedure*, two hypotheses were proposed by another group of students. On the one hand, containerization makes the resources of a CPU directly accessible. Therefore, the detector should not identify that it is running in a containerized environment. On the other hand, containerization



can limit CPU resources and this limitation is detectable in particular cases.

If the first hypothesis has been verified as correct after some experiments, as shown previously with Docker, the second hypothesis is a bit more complex to evaluate. Indeed, the load induced to isolate the process requires a little more work than without, but is negligible compared to virtualization and cache management. The second hypothesis is evaluated by setting a CPU quota. The detection rate is measured as 55 %. This rate is not enough to make the difference. In addition, it is not the containerization that is detected but moreover the overloading of our system which force the cache du update sooner as expected.

- One assumption assumes that the diversity of operating systems has an impact on the proposed method. More directly, the problem is to know if there can be any difference in results depending on whether our method would be executed under Linux, MacOS or Windows. After all, each of these OS is different and if they all follow the requirements of the CPU manufacturers (especially Intel), they have implementation differences. One hypothesis is whether a difference is discernible, measurable, and actionable to challenge the proposed detection method.

A first study performs 1000 tests at different levels of CPUID instructions. The results (extracted from the report) are shown in Figure 3.31.

OS / CPUID	0	10	20	30	40	50
Windows 10 (True OS)	0,1%	0,0%	0,1%	0,4%	0,1%	0,0%
Linux (True OS)	0,0%	0,1%	0,0%	0,2%	0,1%	0,0%
VM Windows (on Windows)	0,1%	0,5%	98,1%	99,9%	99,9%	99,9%
VM Linux (on Windows)	3,4%	9,77%	9,95%	100%	100%	100%
LXC (on Linux)	2,3%	2%	2,2%	4%	2,2%	3,3%
VM Windows (on Linux)	0,5%	52,1%	100,0%	100,0%	100,0%	100,0%
VM Linux (on Linux)	0,4%	56,9%	100,0%	100,0%	100,0%	100,0%

Figure 3.31: Detection rates according to the host/Guest operating system used.

The results in Figure 3.31 confirm that between 20 and 30 CPUID instructions on this machine, a high detection rate is achieved while keeping the false positive rate minimal. This confirms that there is no significant difference in the performance of the method between Windows and Linux. We note that students have tested *Linux Containers* (LXC) [449], which is a light container as presented before. In this case too, since this type of containment does not use any hypervisor, it is not detected by our method, no matter we are dealing with Linux or Windows.

To confirm this conclusion, another group undertook a similar study. They undertook this study with both Windows 10 (2004 - 19041.685 for host & 1809 - 17763.1637 in VM) with a CPU Intel(R) Core(TM) i7-9700K CPU at 3.60GHz and Linux (Arch - kernel 5.10.4 for host & Ubuntu - ubuntu 5.4.0-59-generic) with a Intel(R) Core(TM) i5-8300H CPU at 2.30GHz. Results with Windows operating system are given in Figures 3.32 and 3.33.

We observe that in the case of Windows, with the CPU used here, a number between 7 to 9 CPUID instructions is correct to guarantee a detection without too many false positives. In the same way with Linux, the students also undertook to check what it gave in addition with a virtual machine running Windows. Results are given in Figures 3.34, 3.35 and 3.36.



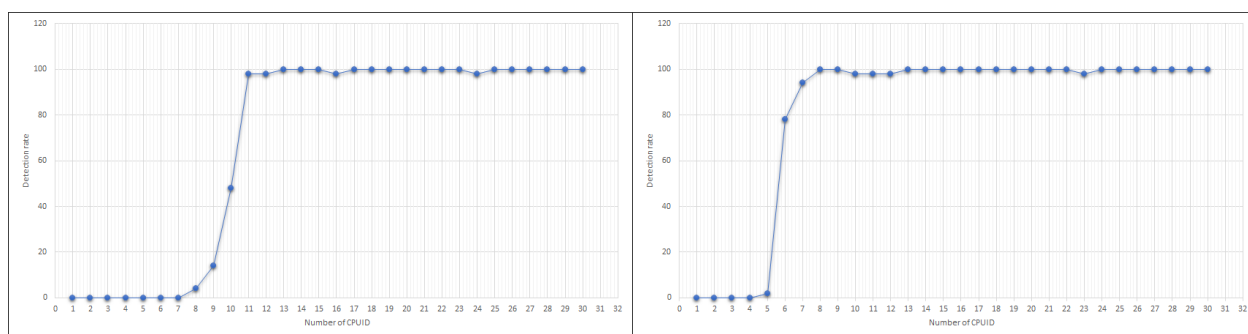


Figure 3.32: Windows host machine.

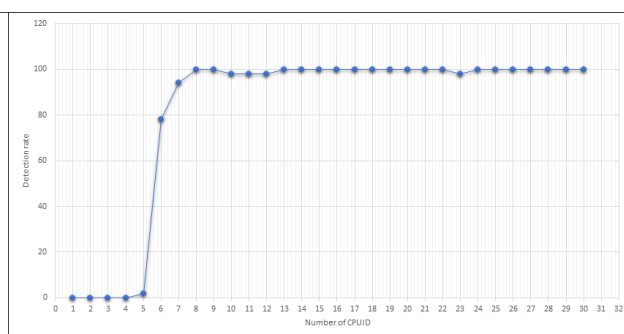


Figure 3.33: Windows guest on Windows host.

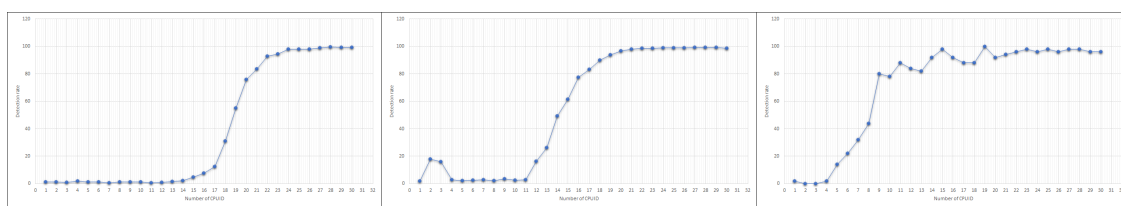


Figure 3.34: Linux Host.

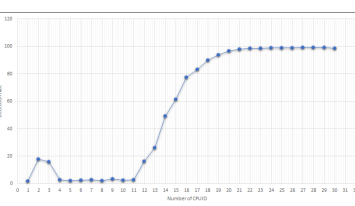


Figure 3.35: Linux on Linux.

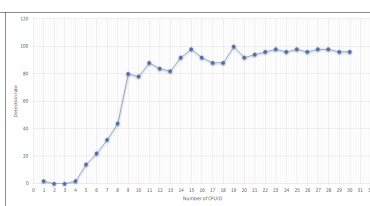


Figure 3.36: Windows on Linux.

Remembering that here the tests were with another CPU<sup>12</sup>, we notice that for a number of instructions close to 8, it becomes quite possible to detect the VM that is executed from Linux. But the results are less stable than under Windows because the host Linux machine can provides more false positives at this level of CPUID instructions.

- The analysis of the cache given in the article has been seen by some students as *speculative*. Indeed, acting in an undocumented area on the Intel CPU, one cannot state with precision what is going on. Our hypothesis about the cache management timing, corroborated by several observations and whose logic has been presented by us to explain the phenomenon, remains a possible explanation but not definitive. This is the difficulty when interacting with CPUs whose documentation remains partial and whose architecture is not open and very difficult to analyze (the hardware means are important to analyze a modern CPU, not to mention the complexity inherent to this type of system). This also makes it possible to question the reliability of research based on these systems when it is not the most elementary confidence that it is necessary to grant to these systems whose side effects are finally unknown nor definitively explicable.
- The question of the impact of the external environment (and in particular the sharing of execution on the CPU cores) was raised by different groups. Is the detection method impacted if the system on which it is tested is overloaded with work? More directly, if many processes are executed on the machine with a high CPU and memory load, this forces an important solicitation of the cache and then may "pollute" the detection, especially in virtual machine. For the sake of simplicity, one might say that a *stressed* environment (where an important work is running on CPUs' cores) might have a lower detection efficiency.

In practice, it is possible to observe a difference when the method is running on systems with few CPU core available and a high activity on different cores. This could have been a plausible explanation for some of the observed results for some students. But one group of students decided to look into this particular issue. They started with two different machines, one called "A machine" (Intel(R) Core (TM) i5-8250U CPU, 4 physical cores with 8 logical ones, Windows 10 Family Edition 18362.1139) and another called "B

<sup>12</sup>For the sake of methodology, it would have been better to compare on the same CPU, just to check that the number of instructions required is close enough between Windows and Linux as host. But this was not possible due to time constraints. Nevertheless, based on our own observations and those from students (Figure 3.31), while the number of instructions may differ slightly, the approach observed here tends to be confirmed.

machine” (Intel(R) Core(TM) i5-4300U CPU at 1.90GHz, 1 physical cores with 4 logical ones, OpenSUSE Tumbleweed (Linux)). They tested each time an increasing number of CPUID instruction, both in host and guest with stressed and unstressed environment. Each time, tests are performed 10,000 times.

First, they conducted a test to measure the performance of the detection on a host machine, without hypervisor, to verify that the detection is consistent with what is expected, ie near zero. The results are reported in Table 3.7 with the correct detection rate given in percentage (here, the number of times our method does not see a virtualized environment).

Machine A - Host environment									
00 CPUID		05 CPUID		10 CPUID		20 CPUID		50 CPUID	
Correct	Wrong	Correct	Wrong	Correct	Wrong	Correct	Wrong	Correct	Wrong
99.87	0.13	99.90	0.10	99.88	0.12	99.87	0.13	92.47	0.763
Machine B - Host environment									
00 CPUID		05 CPUID		10 CPUID		20 CPUID		50 CPUID	
99.87	0.13	99.90	0.10	99.88	0.12	99.87	0.13	92.47	0.763

Table 3.7: Test with two host environments, *unstressed*.

The first observation is that if the method seems quite good with a low number of CPUID instructions, it remains nevertheless probabilistic (the method is not always exact) with nevertheless very important success rates. If the CPUID instruction is not free to use, its impact is marginal compared to the activity of the cache. In this sense, the observation is consistent with the expected behavior of our method. For some unknown reason, the students are restrict their evaluation to machine B only. For the sake of transparency, they prove the activity on the logical cores of the machine with the *htop* command from Linux (Figure 3.37).

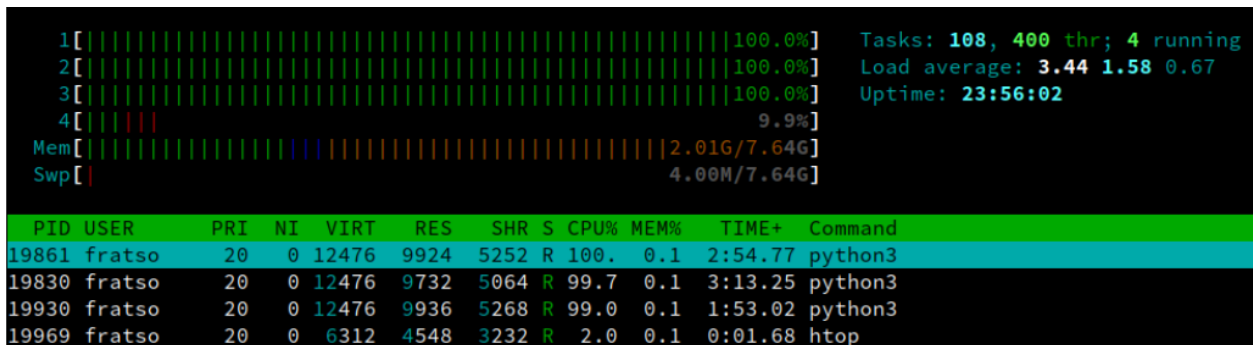


Figure 3.37: 3 logical cores on 4 are overloaded at 100 % of activity.

Their experimental protocol is about to progressively overload each logic cores to observe the impact on the detection rate. For the sake of readability, the data has been compiled in Table 3.8.

We can split the conclusions from Table 3.8 into two parts. On the one hand, observations concerning less than 50 instructions. These one show more than 90% of false positives as soon as there is only one free core left on the machine. On the other hand, beyond 50 instructions, 80 % of false positives appear as soon as half of the machine’s cores are overloaded and the maximum number of false positives also appears when there is only one non-overloaded core left. From these two conclusions, we can confirm that the detection method needs at least 2 logical cores (not overloaded) to run. This is a clearly written condition in our article. In this sense, the students’ evaluation confirms our detection protocol in a non-virtualized environment to avoid false positives.

Concerning the virtual environment, two machines were used here. From the A machine (Intel(R) Core (TM) i5-8250U CPU — W10), Virtual Box software has been used (6.0.4 r128413 (Qt5.6.2)) and Windows

Correct	Wrong	Correct	Wrong	Correct	Wrong	Correct	Wrong
00 CPUID							
1 cores on 4		2 cores on 4		3 cores on 4		4 cores on 4	
99.63	0.37	99.06	0.94	0.52	99.48	15.92	84.08
05 CPUID							
1 cores on 4		2 cores on 4		3 cores on 4		4 cores on 4	
99.71	0.29	99.04	0.96	0.60	99.40	16.27	83.73
10 CPUID							
1 cores on 4		2 cores on 4		3 cores on 4		4 cores on 4	
99.67	0.33	99.01	0.99	0.69	99.31	15.31	84.69
20 CPUID							
1 cores on 4		2 cores on 4		3 cores on 4		4 cores on 4	
99.71	0.29	99.08	0.92	0.64	99.36	17.18	82.82
50 CPUID							
1 cores on 4		2 cores on 4		3 cores on 4		4 cores on 4	
99.49	0.51	16.32	83.68	0.46	99.54	15.44	84.66

Table 3.8: Test with two host environments, in *stressed* conditions.

Server 2016 used as a guest operating system. The same applied with B machine (Intel(R) Core(TM) i5-4300U CPU at 1.90GHz — Linux) using the same visualization software and the same guest operating system. For the sake of brevity, only the results from machine B are reported (but nothing indicates that they were different on machine A). For practical reasons, at least one logical core must be left at the host machine, which means that the tests were performed on only three logical cores.

First, students tested the detection under *unstressed* conditions with these two virtualized environments, as they did host environments. Results are provided in Table 3.9.

Machine A - Host environment									
00 CPUID		05 CPUID		10 CPUID		20 CPUID		50 CPUID	
Correct	Wrong	Correct	Wrong	Correct	Wrong	Correct	Wrong	Correct	Wrong
100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00
Machine B - Host environment									
00 CPUID		05 CPUID		10 CPUID		20 CPUID		50 CPUID	
0.28	99.72	35.18	64.82	97.46	02.54	99.39	0.61	99.70	0.30

Table 3.9: Test with two guest environments, *unstressed*.

Then the experiment was reproduced in a virtual machine. For operational reasons, the virtual machine has difficulty supporting the saturation of its three cores with the detection method (there may even be crashes), so it was not possible to push the results as far as in the previous tests. This is why "N/A" appears in some columns from Table 3.10 which compiles the results.

We can observe in Table 3.10 that despite the overloading of the cores, there are not significantly more false positives. Therefore, it can be said that the use of other CPU-intensive programs within a virtual machine will not impact the reliability of the method. In a way, this is an additional difference with the non-virtualized environment. Nevertheless, it is difficult to support this statement without having been able to carry out a complete test, especially with the overload of the three cores. To solve this problem, the students proposed to perform an additional test by overloading all the VM's cores but by allocating twice more logical cores and by performing 100 tests instead of the previous 10,000 ones (to reduce the risk of crash). Results are given in Table 3.11.

Correct	Wrong	Correct	Wrong	Correct	Wrong
00 CPUID					
1 cores on 3		2 cores on 3		3 cores on 3	
0.39	99.61	10.60	89.40	N/A	N/A
05 CPUID					
1 cores on 3		2 cores on 3		3 cores on 3	
35.84	64.16	35.44	64.56	N/A	N/A
10 CPUID					
1 cores on 3		2 cores on 3		3 cores on 3	
97.22	02.78	90.08	09.92	N/A	N/A
20 CPUID					
1 cores on 3		2 cores on 3		3 cores on 3	
99.30	0.70	98.50	01.50	N/A	N/A
50 CPUID					
1 cores on 3		2 cores on 3		3 cores on 3	
99.67	0.33	99.57	0.43	N/A	N/A

Table 3.10: Test with two guest environments, in *stressed* conditions.

Overloading of all the logical cores in the VM									
00 CPUID		05 CPUID		10 CPUID		20 CPUID		50 CPUID	
Correct	Wrong	Correct	Wrong	Correct	Wrong	Correct	Wrong	Correct	Wrong
30	70	30	70	97	3	100	0	100	0

Table 3.11: Test in the guest environments, fully *stressed* with only 100 tests per instance of CPUID instruction.

We note that the results are roughly similar to what observed previously and the method starts to be relatively reliable between 10 and 20 CPUID instructions, as expected in our researches. The overload of the virtual logical cores does not have any impact on the reliability of this detection method within a VM.

But if the impact cannot be measured in a virtual machine, it has been measured at some level on the host machine where all the logical cores are overloaded. Indeed, if all CPU cores are fully overloaded by other tasks, this detection method is completely flawed, since host would be wrongly detected as virtualized environment. But as explained, the detection method needs at least two logical cores to work. These two cores must not be busy with other processes during the execution of the detection program.

This observation allows us to imagine a corrected solution with a consideration of the current CPU load as a weighting element. In addition, to avoid the "memory" effects from the cache between two successive tests of the method, it is advisable to wait a little for the cache to become empty (otherwise the modification of the thread is automatically inserted since it is already present in cache memory). This is a consideration that was originally taken into account but which is now reinforced, in particular by testing the general activity of the CPU cores on the machine beforehand to take this into account.

- Most of students claimed that it would be easy for an antivirus to detect our method. They claimed that it would be easy to detect the method because the addition of 10 adjacent CPUID instructions represents a very unique pattern that would be easily detected. Since the number of such instructions is driven by the environment to detect, antivirus software could simply detect the abusive use of CPUID instructions via signature-based detection.

This judicious remark can be addressed via two arguments. On the one hand, it is true that many CPUID instructions following each other in memory is not so common. But the code given to illustrate our detection method is only present for the sake of illustration. It is out of question to give a too fully operational code that would make life easier for malware authors. All things being equal, it is not forbidden

to imagine a code that would execute our detection method in a dynamic way. In the same way that a packer seeks to hide the original code that it protects, it could be possible to reconstruct the instructions in memory of our method. That way, they would be executed, afterwards, by a dedicated thread. This would also justify the use of memory with read, write and execute rights to host our method's opcodes... On the other hand (and assuming the previous argument is applied), it is no longer possible to easily detect such a construction of several CPUID instructions with static means. To do this, malware that would dynamically embed instructions would have to be dynamically analyzed. This would ideally be done via a DBI or a virtual machine... And that is perfectly appropriate, because it is precisely this type of tool that our method is designed to detect. In such a case, the method will be detected, but not the malicious code protected by the method. But there are better solutions than looking for a solution in this direction.

## 4.6 Limitations and further improvements

### Resume 22:

- ☞ This subsection proposes to resolve some of the limitations outlined in section 4.5.
- ☞ To no longer depend on the need for calibration, the detection method has evolved by using robust statistics.

### 4.6.1 Limitations of the method

There are few limitations with our method. First, if the goal is to protect from reverse engineering, static analysis is still possible. This is due to the fact that our method is designed for dynamic analysis with tools such as debuggers, DBI or VM. Note that it is possible to limit this point by deciphering following opcodes when detection returns that there is no analysis environment and to keep ciphered opcodes in case of detection.

The second limitation of this method is that it only works on architecture where we have at least two CPU's cores. Multicore CPUs are required since if our method is running on a system where only one core is available, false positive rise. Ideally, execution of two threads on different logical cores guarantees optimal results in terms of performances and reliability. This is due to the fact that both threads who are running on the same logical cores are sharing the same cache, a situation we prefer to avoid. Thus, the cache will no longer need to actualize explicitly. In the case of a single physical core with a single logical core, there are still some rules to correctly consider code modification [4]. However, they are different from multiple cores CPU cache's rules that we are exploiting here.

One last limitation lies in the number of *VMexit* instructions mandatory to allow the cache actualization in case of VM detection. From empirical observations performed, it depends on the type, family or technology or CPU used. Nevertheless, we can consider that about ten instructions (and at most about fifty) are likely to produce, generally, correct results. Nevertheless, this depends on the hardware (CPU) used. More directly, the proposed method requires some calibration. One idea would be to propose a calibration method as an improvement, but it would remain theoretical. Indeed, this presupposes to know the CPU used in advance, which removes the generic property expected in our research. For a better result, the detection method itself should be improved so that it no longer depends on material differences. Indeed, the phenomenon used for detection is measured on all the tested CPUs, but at different degrees.

In addition, the stealth of the proposed method must be improved. It was noted that it was possible to detect the method used, in particular by an excessive call to the CPUID instruction. Researching other tactics would be a good way to make this detection method even more effective.

### 4.6.2 Further improvements

#### 4.6.2.1 General considerations

The content discussed here is intended to report on current developments and research about our method.

As the topic is sensitive (since it can directly serve malware authors), it needs to be published with caution. Moreover, the results may depend on the CPU hardware used, which implies that many tests will have to be done to validate the results. We will therefore propose here some possibilities currently being studied to answer limitations.

The main limitation concerns the detection of virtual machines (the case of DBI or debuggers is deterministic). There are three issues with virtual machines: the method requires several logical cores, it needs calibration and it lacks stealth. The first problem cannot be solved because it is a necessary prerequisite for the method. Also, single-core environments nowadays are quite unusual. The other two issues can be answered in some way.

Concerning the instructions able to cause *VmExit*, it would be relevant to find a set of instructions able to provide a *VmExit* event. From the whole list of instructions that can cause an *VmExit* event [450], it is possible to find several alternative candidates. Nevertheless, not all instructions are equal to be used in the context of the detection method. On the one hand, some instructions require to be in kernel-mode (ring 0) to be executed (which would force to restrict implementation of the method only for drivers). On the other hand, some instructions optionally trigger an *VmExit* event. More directly, during the configuration of the virtual machine control structures [451], it is possible to define which instructions will trigger an *VmExit* event by the host and which instructions will not trigger one. This selection depends on the needs of the virtual machine, what is expected to process or the expected effects. In fact, it is possible to filter which instruction can cause an *VmExit* event. But there is only CPUID that does it from user-mode unconditionally.

What does this mean? It means that it is hard to perform the detection without this instruction because it combines all the right properties (user-mode and non-maskable *VmExit* instruction). There are `rdrand/rdseed`, `rdtsc/rdtscp` and `xsaves/xrstors` which are able to realize this event from the user-mode, but they are optional. Note however that in the context of malware analysis, the `rdtsc/rdtscp` instruction is a key instruction. Indeed, in the context of tools such as Pafish [324] using `rdtsc` instruction in order to time instruction execution (section 2.2.4), mastering it is a key point in the strategy to neutralize this VM escape tactic. The same way, using the Kirsch's tactic [409] based on `xsaves/xrstors` instructions, malware can detect DBI tools such as Pin. This may explain why, in some sandbox, these instructions are configured to be handled by the host when executed thanks to a *VmExit* event.

Other tactics could be considered, based on other instructions. The idea here is no longer to exploit the *VmExit* mechanism specific to virtual machines, but to come back to the fundamentals of our detection method, namely the synchronization of the cache in the context of a cross-modification of code. Working on memory access (which is handled by the virtual machine to guarantee the separation of host and guest memory) or on other mechanisms are currently research tracks. The details are not given to preserve a certain confidentiality of the work.

The last problem is the reliability of the method. More directly, if the reliability can be seen as acceptable but it is not high enough for a fully operational use in comfort. The problem is not so much the reliability as the necessary calibration on a given CPU to become reliable. And this is finally the main issue to solve: removing the required calibration.

#### 4.6.2.2 Possibilities to remove calibration

In order to better understand the cache update phenomenon, we propose to modify our detection method. The idea is no longer to observe the result of an execution after a cross modification of code, but to measure when this modification takes place. In practice, this is equivalent to implement a loop that increments a value in the modified thread. At the end of this loop, the modified thread initializes a register to 0. This loop is conditioned to stop when the register previously set to zero is equal 1. Of course, this operation is only possible when the modifying thread updates the initialization code for the register in the modified thread (to change the initialization from 0 to 1, for instance). From this point, by retrieving the counter value, we can have an estimate (in number of loops), to know when the modification takes place.

Using a machine equipped with a processor Intel(R) Core(TM) i9-9900K CPU at 3.60GHz, 8 Core(s), 16

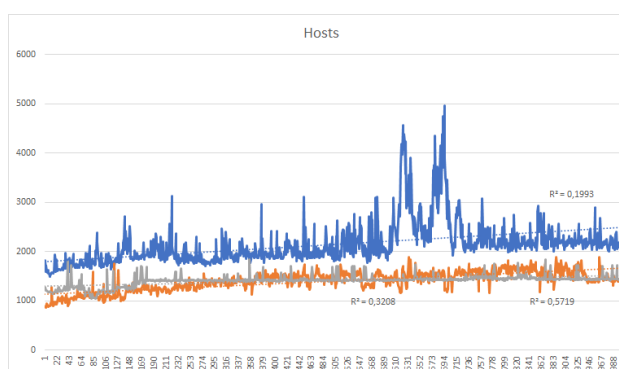


Figure 3.38: Tests from host machine.

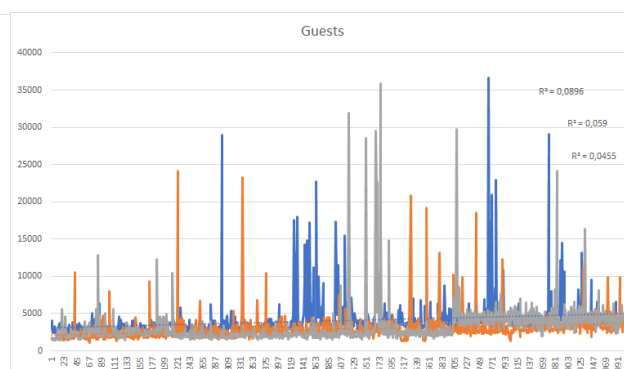


Figure 3.39: Tests from guest machine.

Logical Processor(s), we performed tests both in host and guest (Virtual Box - Version 6.1.16 r140961 (Qt5.6.2)) environment. In both environment, Windows 10 operating system has been used. For the sake of brevity and readability, we have kept only three tests from each environment. The results are given in Figures 3.38 and 3.39. On the x-axis is given the number of tests performed (a thousand times) and on the y-axis the value obtained (in number of times the loop has been executed).

In an ideal world, assuming that the CPU is perfectly deterministic and our method perfect, whatever is the environment analyzed, the number of loops should be constant (because the cache refresh, assuming we are in this ideal world, would then be deterministic). In practice, we should see parallel lines (since we are measuring a constant), more or less close (since it is the same CPU used). Of course, we are not in an ideal world. And that is why we can see variations on the curves. In a way, these variations can be considered as random noise added to a constant. A statistical bias in short. From there, the statistical analysis can begin.

From these results, we observe the greater variation of a guest environment (Figure 3.39) compared to a host one (Figure 3.38). More directly, we observe greater variations on the guest machines than on the host machines. More important variations are based on range and frequency. From that observation, several tactics could be implemented to perform a detection.

First, one could think of modeling a generic behavior for the host and for the guest and then determine, using a statistical test such as chi-square, whether a measured distribution corresponds to the one sought (and statistical deviation possibly due to chance with a confidence score) or whether the observed deviation is not the one looked for. Unfortunately, this approach does not work in practice. The variations are too great to establish a reference model. Moreover, in practice, we would probably have to determine a model per CPU, which would mean going back to our initial calibration issue. The use of artificial intelligence techniques does not solve the problem: initial data are not reliable and accurate enough to allow any credible learning.

Another (and maybe naive) idea, looking at the given graphs, is to observe a stronger linear trend in the host compared to the guest. By using a linear regression technique, it may be possible to model the observed trend line of the distribution. Of course, if we had to rely only on the linear equation obtained from the regression, this would also mean creating an equation per CPU. To avoid this, one can try to guess how close a linear model is close to the observed data. The most obvious answer is to measure the linear correlation between two sets of data thanks to the Pearson correlation coefficient [452, 453, 454]. This one is easy to compute from a least squares regression analysis and written for each curve in Figures 3.38 and 3.39. It seems natural to take a decision criterion by choosing an arbitrary limit on the correlation coefficient at 0.10. Above, it seems to be a host machine, below a guest one.

Unfortunately, this technique does not work. Indeed, when testing on other machines, we realize that some host machines, under some configurations, have a correlation coefficient close to zero. In practice, this means that the correlation is not established in terms of linear modeling (Figure 3.40). The explanation of this particularity is however simple. We are not really measuring a linear trend here. The x-axis gives the number of



the test and not an incremental factor that would have an impact on the value resulting from the test (i.e. the number of loops). Ideally, the number of loops should be a constant, that is to say a line parallel to the x-axis. And therefore a straight line with a slope equal to zero. And this slope is linked by a linear relation with the correlation coefficient. This explains why when we are modeling a straight line, the correlation coefficient is equal to zero. This is why this detection tactic, based on this coefficient, is not reliable. Simply because there is no linear relation between the test number and the number of loops.

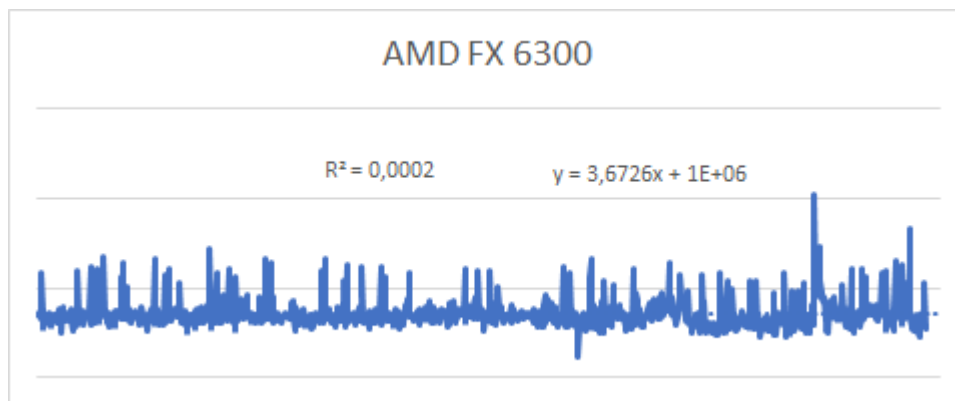


Figure 3.40: Example of test on a host machine which has a correlation coefficient close to zero (and thus seen as a guest).

The real problem of statistical modeling of the observed phenomenon is the pollution of the data by extreme values. And these extreme values can happen both in the host (punctual overload of CPU cores for example) and in the guest. Of course, it would be easy to think that the most extreme variation (in terms of absolute value) is a decision criterion in itself. It is true that execution in a virtual machine most often produces the highest values (in terms of the number of loops executed) but this criterion is neither objective (what would be the threshold to set for detection?) nor true (it happens that some hosts on a given CPU have higher values than guests on another CPU). Once again, we would end up making a database of extreme values per CPU and therefore, finally, a calibration.

To address data pollution issue, the solution is the use of robust statistics [455, 456]. For the sake of simplicity, this type of statistics can be described as less sensitive to extreme values or outliers. The advantages of both robustness and superior efficiency when dealing with contaminated data is balanced by a more limited efficiency on clean data from distributions. For instance, the mathematical mean is more sensitive to extreme values than the mathematical median. Indeed, mathematical mean can be significantly impacted with a single observation. Another instance could be the Theil-Sen estimator [457, 458] to keep the idea of linear regression by robustly fitting a line to sample points in the plane. And this is directly the type of property expected for our detection.

Our goal is not so much to determine a general trend in the measurements as an irregularity between measurements. Indeed, since the repeated experiment is always the same, notwithstanding a statistical bias due to the context of the execution environment (other processes activity, devices activity, background tasks...), the results must be relatively constant on a real environment. Unlike a virtual environment where virtualization, *VmExit* and other constraints of this type of environment contribute to the irregularity of the measures. This is what is observed on Figure 3.38 and 3.39.

For the sake of formalism, we propose to represent the results of our repeated experiment as a random variable. Thus,  $X$  is the random variable whose outcome is the number of loops measured in the experiment described previously. An instance of the experiment noted  $x_i$  corresponds to a measure of number of loops executed before the opcode update occurs and  $i$  is the current index of the experiment processed. Since the experiment is performed  $n$  times, we have  $1 \leq i \leq n$ .



In order to measure the average of the absolute deviations from a central point, we usually use the *average absolute deviation* (AAD) [459]. In its shape, it is comparable to first central moment in statistics. Formally speaking, the average absolute deviation is defined such as:

$$D_{aad} = \frac{1}{n} \sum_{i=1}^n |x_i - m(x)|$$

where  $m(x)$  is any measure of central tendency. In practice, we can use the mean  $\bar{x}$  as a central moment to compute the *mean absolute deviation* as the average (absolute) distance of the deviations from the mean. In other words, it is the average distance to the mean. But in practice, since the mean is not a robust moment, it is not possible to make a detection with polluted data. But it is possible to use the *median absolute deviation* which is the average of the deviations from the median. In this case, the central tendency is the median  $\tilde{x}$ . This results in:

$$D_{med} = \frac{1}{n} \sum_{i=1}^n |x_i - \tilde{x}|$$

In practice, the results are interesting but it is not possible to directly obtain a detection criterion. Why? Simply because the deviations depend on the CPUs on which the tests are performed. The values obtained are arbitrarily large compared to the CPUs used. This observation comes from the fact that  $D_{med}$  is an *absolute measures of dispersion*. More directly, it indicates how much the values of a distribution deviate from a reference value, such as the median. Since absolute measures of dispersion use the original units of data (number of loops in our case), they are useful for understanding the dispersion of measurements in the context of an experiment.

In practice, absolute measures of dispersion are used to compare the deviation from their central value of two distributions whose central values are identical and whose unit of measurement is the same. But in our case, by doing the experiment on two different CPUs, we obtain two distributions  $X$  and  $Y$  that describe the same phenomenon but for two different populations (CPUs) and such that the order of magnitude of the distributions is significantly different. They are not able to compare the dispersion of two distributions that have different units of measurement or orders of magnitude. And that is what we are dealing with. Indeed, CPUs of different brands, different generations and different capacities are based on the same physical phenomena and execution conventions, but offer measurements with different orders of magnitude.

To solve this issue, we can use relative measures of dispersion. Such relative measures of dispersion are perfect to make comparisons between separate data sets or different experiments that might use different units or different orders of magnitude.

A relative measure of dispersion is a measure of the relative deviation of the values of a distribution from a given central value. It is therefore a ratio built from an absolute measure of dispersion by taking the ratio between an absolute dispersion parameter and a central value [460]. In all cases, the dispersion parameter is a dimensionless number (the ratio of two numbers with the same unit of measurement) that expresses how much the values differ from the central value in relative value. For instance, the *coefficient of variation* (also known as *relative standard deviation*) is the ratio of the standard deviation to the mean. In our case<sup>13</sup>, we propose to perform the ratio of the median absolute deviation to the median of the distribution. More directly, our measure is written:

$$D = \frac{1}{n\tilde{x}} \sum_{i=1}^n |x_i - \tilde{x}|$$

It is not very complicated to show that the measure created here is a ratio whose value is always non-negative, since it is composed of factors and terms always non-negative. But it is relevant to explain in which context the measure can be minimal. Taking into account that all  $x_i \in X$  are non-negative values (because it corresponds the number of executed loop), with  $x_{\min}$  the minimum from  $X$  distribution, we have:

<sup>13</sup>Relative measures of dispersion have some limitations in their use. They cannot be used in all cases (because we can only compare comparable objects or phenomena). In our case, we have a comparative approach between two variables with the same unit of measurement (number of loops executed) driven by the same phenomenon considered to be random in nature. This allows us to have a comparison that makes sense.

$$\begin{aligned}
x_{\min} &\leq \tilde{x} \\
\sum_{i=1}^n |x_i - x_{\min}| &\leq \sum_{i=1}^n |x_i - \tilde{x}| \\
\frac{1}{n} \sum_{i=1}^n |x_i - x_{\min}| &\leq \frac{1}{n} \sum_{i=1}^n |x_i - \tilde{x}| \\
\frac{1}{n} \sum_{i=1}^n \left| \frac{x_i}{\tilde{x}} - \frac{x_{\min}}{\tilde{x}} \right| &\leq \frac{1}{n\tilde{x}} \sum_{i=1}^n |x_i - 1|
\end{aligned}$$

The left part of the equation is minimal when all terms in the sum are equal (since the absolute value is always non-negative). More directly, it means that  $\forall i \in \llbracket 1, n \rrbracket$  all  $x_i = x_{\min} = x_{\max} = \tilde{x}$ , that is to say when there is no deviation from the central value observed (the median in our case). In such case, all terms of the sum are zero and then it comes that our measure is always non-negative and its minimum is zero.

But there is no conclusion about the maximum value that our relative measure of dispersion can take. In practice, the observations obtained from our own machines give the results provided in Tables 3.12 and 3.13. All results are given in percentage, for the sake of readability.

Host					
CPU	AMD RYZEN 7 1800X	Intel Core i5-8300H	Intel Core i9-9900K	AMD FX 6300	Intel Core i5-8265U CPU
Measure	13.69	11.32	4.07	3.57	13.21

Table 3.12: Result from test with our measures of dispersion based on the median for host machines.

Guest					
CPU	AMD RYZEN 7 1800X	Intel Core i5-8300H	Intel Core i9-9900K	AMD FX 6300	Intel Core i5-8265U
Measure	45.79	130.39	27.86	31.11	48.56

Table 3.13: Result from test with our measures of dispersion based on the median for guest machines.

From what we observe, it is possible to set a detection threshold around 20 percent (more than 20 % of deviation from the median means that we are likely to deal with a virtual machine). This leaves enough room to avoid false positives. But here again, it would be necessary to be able to test on more machines, with more virtual machine software to be able to write more definitive conclusions. The research presented here is provided as it is. In addition, this one is always subject to evaluation. Nevertheless, the approach taken shows the procedure used to solve the problem we are facing, with correct results as far as we have been able to perform tests.

## 5 Future work and broader approach

### Resume 23:

- ☞ This section proposes original approaches to use evasion techniques for malware neutralization.
  - ☞ It provides a broader view on how to use the protection technologies developed here.
- ☞ This section explains new trend in malware protection based on packer driven by artificial-intelligence, as a new step after evasion techniques.
- ☞ This section highlights former research work we did years ago to present what — at that time — was new software protection techniques (to prevent reverse engineering).
  - ☞ It shows that our former work was in line with what was being done at that time.
  - ☞ It shows that our current work presented in this document is still relevant to current research.

As an overture, we propose three axes to discuss the future of the work presented here. If on the one hand there are the obvious improvements from our own work, it is interesting to see other contexts of exploitation of these researches to finally discuss more original approaches in malware protection.

### 5.0.1 Improving our own researches

First, it should be noted that the methods proposed here can always be improved. Both by a more thorough test campaign on more different machines and by a search for techniques to optimize results (in performance, efficiency and reliability). It should also be noted that there is always more than one way to do things. To increase the stealth of the detection methods, it could be possible to try to produce effects similar to those presented here, but using different APIs, different assembly instructions, different tactics... Research in this area seems to be quite substantial.

### 5.0.2 Using evasion techniques to design a secure end-user system

But perhaps the most interesting part of this work would be to use it directly for more defensive purposes. An article published by Zhang & al. [461] proposes to use malware's evasion technology to caught out at its own game. It is an extended version of the idea proposed by Chen & al. [186]. Authors from [461] observe that malware authors introduce more and more efficient techniques to probe analysis environments before exposing malicious behaviors. Usually, when malware sample detects an analysis environment, they stop all their malicious activity. According to authors, such a strategy is a double-edged sword. Because, if we can turn a working environment into an analysis environment, then most of the malware that detects these environments would become non-functional. For short, the idea is to execute malware (or unknown software) on end-user systems and prevent nefarious activity by means of the malware's very own evasive logic.

In [461], they propose an environment called SCARECROW to deactivate evasive malware before they are executing malicious code (either they are stopping their activities or they acts are benign software). Technically speaking, their solution is about to transform real end-user systems into sandbox-like platforms. It is a giant camouflage of the user's system so that it has all the characteristics of a malware analysis sandbox. To proceed, they are reproducing and copying elements from real analysis environment, such as specific files and directories, processes, libraries, GUI Windows running, registry keys and values, function hooks set for each executed process (with similar techniques such as thos proposed in [419]), exception processing, fake limited hardware resources<sup>14</sup>, and network resources (with fake DNS and always responsive network to simulate fake network connections).

The results obtained are promising. In particular, malware often tend to use a small set of well-known evasion techniques and they do not look beyond that point when they detect an analysis environment. That

---

<sup>14</sup>Due to limitations and maybe for economical purposes, public sandbox usually provide small hard disk size and low memory or CPU cores resources.

way, applying the *Pareto principle*, authors prefer to only focus on satisfying a small subset of conditions to deactivate most of malware samples. This strategy offers a neutralization rate around 90 % according to authors. From the benign software used to detect false-positive or unexpected effects, even if authors admit to not have exhaustively explored all possible program paths, they noted little or no impact for benign programs. At the opposite, most troubles came from malware executed in such context. Indeed, some malware use self-spawning strategy when detecting the presence of a debugger (most of the time just with `IsDebuggerPresent` [219] function). More directly, they are creating another instance of the process, hoping that one will not be followed by the debugger. Since a debugger is always detected, such a procedure executed by some malware samples becomes endless. Despite this undesirable effect (and the possibility of fork-bombs), it would be possible to consider control mechanisms to neutralize this type of behavior.

If the proposed solution can appear as very interesting, it is not without some drawbacks. Notwithstanding some ugly reactions from malware detecting an analysis environment, authors claim that their solution is able to be detected as other analysis environment. Using Pafish [324], they are triggering almost all the same evasion techniques tested, becoming *indistinguishable* for evasive malware. This statement can be balanced by the fact that the detection of Pafish also relies on the timing mechanism based on `rdtsc` instruction. As explained in section 2.3.4, this tactic is used by some malware samples and even if it is a probabilistic, it remains efficient to detect a virtualized analysis environment, what SCARECROW is not. Thus, since SCARECROW is not virtualized, our proposed method in section 4 would not detect it. Thus, a malware based on our method would not be sensitive to the ruse set up by SCARECROW.

In addition, it becomes possible to develop various strategies to detect this particular analysis environment. Based on the fact that if a real user is actually present on the machine for everyday use, there will be possible to probe user's activity (movement from mouse, screen refresh, keyboard activity, and so on). This is typically the kind of events that do not occur on a analysis environment. Of course, some antivirus vendors try to emulate this type of behavior, but advanced malware can see the real from the fake here (especially by the lack of user reaction to specific events). In an even more advanced way, carrying out a forensic study of the machine's activity (in particular by analyzing SuperFetch's databases [462, 463] — which has been documented by us<sup>15</sup>) would make it possible to measure real activity on the machine, which would be in total contradiction with the analysis environments that are generally reset after each analysis (and thus SuperFetch's database is empty or totally outdated). In the same way, the date of the data in the Windows DNS cache or the last modified files could be decisive clues to show that we are not on a "restored" machine state but on a real environment — while we are also able to detect, via other techniques, that we are in an analysis environment. Faced with such a contradiction, it is possible to identify the SCARECROW environment.

In the end, the approach used by Zhang & al. [461] has the merit of using the evasion logic of malware against their own interests. That is a bit rich. But it is difficult to deploy their idea in a real operational context, for three reasons. The first is that it will always remain possible to detect such an environment, as we briefly explained before, notably by the contradiction coming from different measures. Secondly, this security is only valid for malware using evasion methods. Some malware does not use evasion methods and therefore would be free. To this argument, it would be possible to answer that the proposed solution is only one element in a chain of defense (in addition to antivirus software, etc). This brings us to the third reason. Which user would use such an environment? It is necessary to see that the user's machine would be polluted by many analysis software, some of them displayed directly on the screen (like debuggers). Not to mention the fact that some software are unstable (voluntarily or not) when they are executed under the control of a debugger, the security would be at the price of the user's comfort, not to say about usual habits of using a computer.

### 5.0.3 Improvement of malware protection

On closer look, the evasion logic of a lot of malware can be analyzed as a two-step action. First, the collection of information via various evasion methods and then, at the end, a conditional test to decide whether to execute the malicious payload or not. Except for detection-independent evasions (section 2.2.5), such a procedure is usual.

---

<sup>15</sup>The project is freely available at <https://github.com/MathildeVenault/SysMainView>.

While collecting various features to detect the presence of an analysis environment may seem suspicious, it is not necessarily deterrent. Many software programs use the same elements to avoid reverse engineering (commercial software, video games, etc.) without being malicious. However, the strategy remains the same: allow the execution of the code if and only if there is no detection of the analysis environment. In the end, it is in the management of the final condition where the difference is made. From the point of view of a human analyst, analyzing this final condition often allows to understand the real protection set up by the malware in its evasion strategies, but it also allows to trigger the protected code (the analysis is then possible to know whether it is malicious or not).

Finally, the real secret of an evasion strategy, nowadays, lies more in the final triggering condition than in the techniques implemented to achieve the detection. Why? Simply because the protected code cannot be executed without satisfying the condition. Of course, naively, it might be possible to think that modifying the condition so that it is true every time is enough to solve the case. However, this is not the case because in some cases, in advanced malware, the final condition verifies the validity of a cipher key that can decipher the protected code (Figure 3.41). In a way, it is quite similar to packer strategies... More advanced techniques about efficient crystallographic strategies against reverse-engineering could be retrieved in [464, 465, 466].

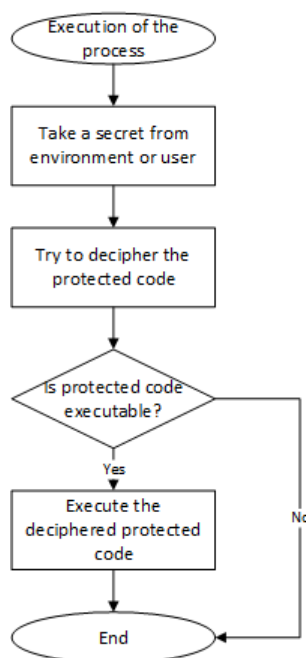


Figure 3.41: Illustration of simple procedure about how to protect code in an executable file.

Of course, with this type of approach, the security of the protection is linked to the security of the cipher key. No way to store it somewhere in clear text in memory since it would only postpone the problem. The solution is to create the cipher key with elements coming from the environment where the malware is executed. In a conference we made years ago [467, 468], we presented techniques to manage the cipher key efficiently in this context. Our objective was to reduce the possibilities for an analyst to get access to our protected code while keeping execution. The solution is to develop highly targeted malware. To proceed, the idea is no longer to detect an analysis environment but the environment of the target we are trying to reach. The execution of the malware then becomes probable. More directly, the execution is driven by the environment on which the malware is running. More directly, this means capturing the cipher key directly from the environment where the malware is executed (better if a precise targeting allows to execute only on a limited number of machines). The cipher key is never stored in the malware, but there is a *key-maker*, able to craft the cipher-key on demand if the malware is running on the expected system.

In practice, this means evaluating the runtime environment for a cipher key. For trivial reasons, we should

avoid using keys that are too obvious (registry values, IP in the DNS cache, hard disk files, time and date, process list ...). On the contrary, it is required to build up the encryption keys so that they are as long as possible, coming from different sources and difficult to guess (no default or predefined values). For example, in a targeted attack, one might try to attack the director of a company or the director's secretary. These two people both have a computer but the execution environment is different (not the same software, habits, activity schedules, web browsing history...) as is the use of the machine (it is likely that the secretary types faster to the keyboard than the director). For instance, listening the keyboard activity to record the keystrokes or the frequency with which those keystrokes are pressed. Depending on the language of the target, the keys pressed will not be the same. In the same way, the frequency of typing may indicate the target's profession (secretary or director). And this is only an example of channel to collect information able to build a cipher key.

It is possible to design many channel from environment. For instance, one might think about keyboard management, mouse position management, network history and website historic (listing, frequency of connections, and so on), voice recording, face recognition through web-cam... Generally speaking, it is an application from the notion of *environmental key generation towards clueless agents* [469]. And of course, many channels can be combined together to design an environmental key. Each channel can be seen as source of noise, constantly listening in order to collect input. The goal is to correctly manage this source of noise, since this one is quite probabilistic. To proceed, we need to transform this probabilistic input noise so that this one can produce quite relevant output. As signal processing, we quantify the input noise so that output is calibrated according to a precision predefined. That way, it is possible to manage probabilistic models with correctly defined input.

Each noise source used gives a certain amount of information. This information can be checked (to know if it matches the desired target — but this should be avoided in order to not give too many clues to an analyst) or directly exploited to build a cipher key (by any transformation function, in practice a cryptographic hash function). Note that it is possible to chain different sources of noise, each protecting a given section of code, as illustrated on Figure 3.42. Once a channel has provided access to a protected code, that code can also provides a new source of noise to manage the protection of another code, and so on. This technology deployed stage by stage — with an encapsulation as Matryoshka dolls — allows to hide efficiently which are the eavesdropping channels and it allows a more and more precise targeting of the target, as all the stages are executing. More channels are used, more difficult it is to guess the real cipher key used, and harder it is to analyze the malware.

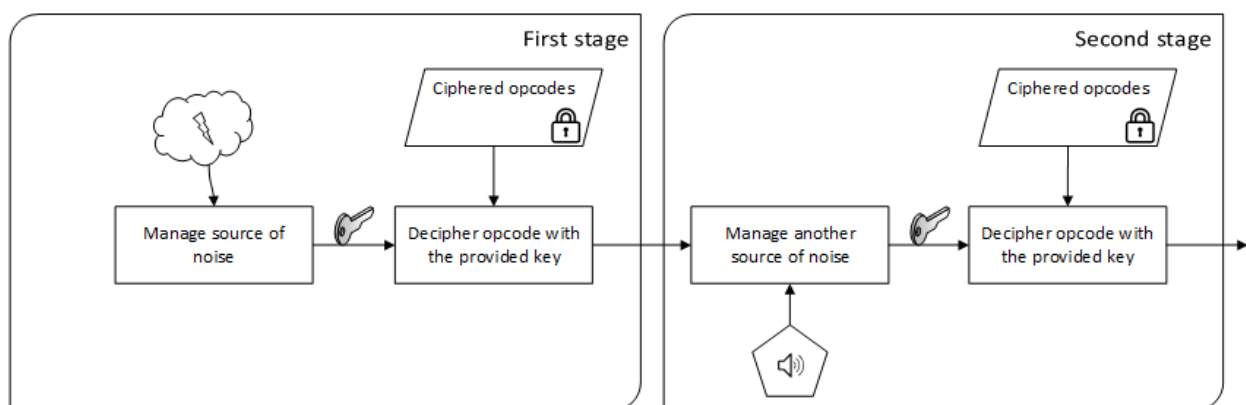


Figure 3.42: Illustration of a chain of different source of noise, each opening the access to the next when the environmental key is correct.

However, this security remains probabilistic. The user must perform certain actions or have a machine configured more or less as expected. And even if it is possible to have a certain tolerance (to allow the generation of a valid key when enough criteria are present — without them all being necessarily there), it remains that the protection can be analyzed. This is usually for extremely targeted malware because greater the diversity of targets, the easier the key can be generated and the more likely an analyst will find it. In any case, once the target has been impacted by the malware and if this one is aware of it, it will be possible to try to discover what finally triggered the malware since the final environment is known.

In the end, we always come back to the protection of the triggering mechanism. In a conference given at Black Hat USA in 2018, Dhillung & al. [5] provides another approach to protect the malware. From their point of view, malware concealment can be seen as locksmithing, where the history started with obfuscation and mutating payloads in the 80s to mature with packers in the 90s, seeing the 2000s using evasive malware trying to avoid being analyzed and the 2010s targeted attack disclosed only at a target. The Figure 3.43 is extracted from their conference [5] to illustrate the historical evolution of malware protection, from authors' point-of-view.

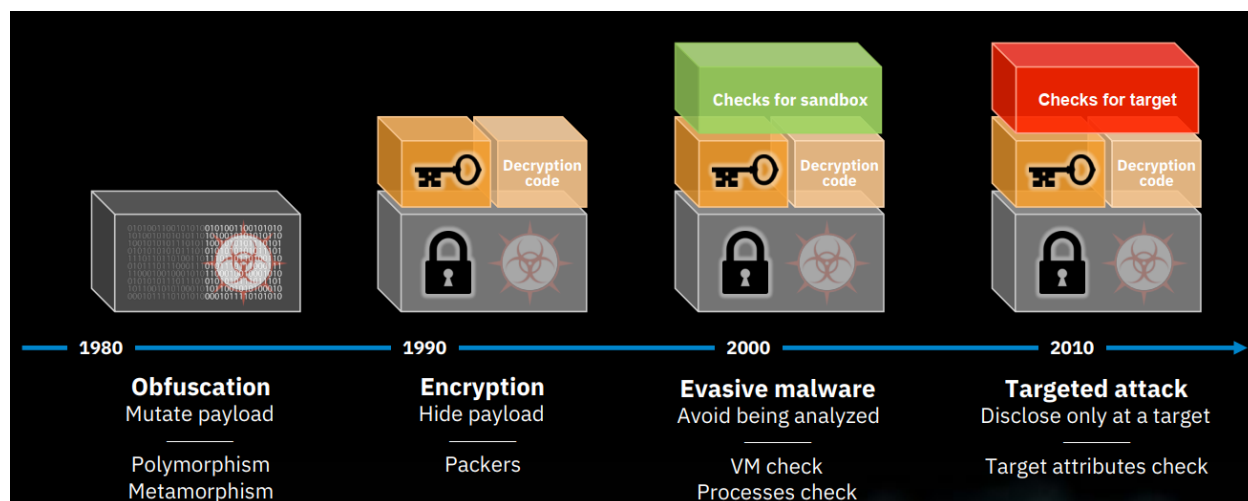


Figure 3.43: Timeline of the evolution of malware protections, taken from [5].

In addition to confirming that our work published in 2013 [467, 468] was finally relevant, the authors propose a new approach with the hope that this may provide a new milestone a day. The idea is to entrust the management of the key manufacturing to an artificial intelligence — a *Deep Neural Network* (DNN) in author's case. Thus, by collecting various target attributes (Audio, visual, geolocation, software environment, user activity, sensors or physical environment), it becomes possible for a trained DNN to recognize its target and to generate a cipher key according to the recognition. Note that it is possible to use another DNN with a key generation model to generate the cipher key in order to get reliable results.

This technique increases the difficulty of reverse engineering malware potentially equipped with such technology. Since artificial intelligence and DNN in particular has become popular those last years, researchers have tried to look for reverse engineering techniques on neural networks [470, 471]. And of course, there are some possibilities to mitigate techniques of reverse engineering with DNN [472].

Forcing to reverse engineering an artificial intelligence is a pretty efficient way to make malware analysis complex. It is in a way an improvement of what we did with our probabilistic models for key-generation. The difference is that in our case the code that dealt with the management of the environment to generate the key were specific and sometimes tedious to write (because we have to program while anticipating a probabilistic input). The advantage of using an artificial intelligence is that the models are generic and easily accessible online. But if there is a plus in the ease of implementation (and finally also in complexity of analysis because these generic models are combinatorially complex), there is a lost in the control of the trigger conditions. In our case, although probabilistic, we can drive exactly the characteristics that allow us to target closely the victim. In the case of artificial intelligence, the logic is more fuzzy and the control is built empirically. But here we go from a handmade and complex to implement production to a mass production facilitated by generic tools and finally quite reliable. It is perhaps the trend and the possible democratization of the threat that we should fear more than the search for an optimal solution.



## 6 Conclusion

### 6.1 Reminder of the achievements

Throughout the achievements reported in Chapter 3, we can summarize the achievements along two main lines. On the one hand, there is the realization of a state of the art (section 2) to define which technologies are currently present so that a malware can escape from an analysis environment. By analysis environment, we mean that there are two main types of environment. The first type is about *manual dynamic analysis* and concerns, to put it simply, all debugger software that are driven by human. In contrast, the second type corresponds to analysis tools named as *automatic dynamic analysis*. Such tools do not require a human being to be used. Technically speaking, it corresponds to malware sandbox composed by virtual machines and DBI tools mainly.

After exposing the goals and tools used in the context of malware analysis, we turned our attention to the various known methods that malware could use to escape from the analysis tools. There are two well-defined strategies, whether the analysis tool is manual or automatic. On the one hand, a target strategy that seeks to identify or exploit a property induced by the analysis tool. This often involves exploiting the fact that an analysis tool is never “free” compared to a normal environment. More directly, such a tool always leaves a trace or induces a difference in the behavior of the analyzed program, simply by its ability to collect and report information. On the other hand, the other strategy is often to act generically, without worrying about the presence of a particular tool. The idea is often to exploit a specific API from the operating system or a CPU feature in order to evade from the control of the analysis environment.

From the state-of-the-art, it was possible to define two original and operational solutions to complete the panel of existing evasion techniques. Since the state-of-the-art is divided between manual and automatic dynamic analysis evasion, one solution has been proposed to address each type of evasion. The idea is to humbly try to complete and contribute to a better knowledge in the field by proposing improvements or new approaches. Of course, the development of these techniques aims to better understand the different techniques of evasion. To do this, we can either guess a future technique (and therefore study it better) or try to improve a past one (to see how far it is possible to go).

We have in section 3 the exploitation of negligence in the Windbg debugger (more or less already known but we rediscovered it by ourselves) to allow different methods of evasion. Then, in section 4, the objective was to create a generic escape method for all automated dynamic analysis environments by exploiting some physical phenomena specific to the CPU architecture from the most popular vendors. This is unique both by the possibilities offered (generic detection, no other approach offers such a result) and by the complexity to correct this problem. Indeed, the approach we propose exploits a phenomenon neglected by CPU designers, namely in our case the use of undocumented techniques. Correcting such issues would lead to update CPU’s micro-code or to review the cache management architecture of the CPU. All other things being equal, the consequences are somewhat the same as for Spectre attacks [159]. Correcting this type of problem often means having to make a trade-off in terms of performance (the cache is a central performance mechanism of modern CPUs). The difference is that our detection method does not have the same impact as the Spectre attack. But it is possible to note that approaches exploiting conceptual or physical limits from CPUs could be — in the future — a promising source of research in computer security.

It is possible to see here an educational approach to fight against evasion techniques. By putting ourselves in the role of finding a new evasion method, we have to face all the constraints inherent to the conception of this type of technique. That way, it allows us to better understand the compromises that malware must do to have operational methods. And therefore, potentially, to better understand the problems and biases they have to face. In consequence, we can better understand the weaknesses inherent in the design of this type of method. And of course, to be able to exploit these weaknesses to better counter or detect malware.

More than the technical aspect which has been detailed already in this chapter, it is perhaps on this aspect of understanding the offensive mechanics that it is advisable to put forward. The contribution of section 4 on how to improve a generic evasion method seems for us to be particularly relevant. On the one hand, because it proposes to evaluate the reproducibility of an experiment by third parties who only have access to our research paper and no specific instructions for reviewing. Leaving the academic path and confronting ourselves to operational

context. On the other hand, by the ability to synthesize the different feedbacks, sometimes original in their approach of evaluation of our method, to propose an even more robust system. This way of doing things is to response to the needs of the people who were able to test our method, but also in order to better understand the constraints specific to this type of evasion (diversity of environments, hardware specificity, technical complexity, false-positives, etc.).

## 6.2 Research contributions

### Contribution 2: Protection of analyzed executable files

- ☞ State-of-the-art about different techniques used by malware for evasion.
  - ✍ We present a comprehensive survey of malware dynamic analysis evasion techniques for both modes of manual and automated (debuggers, virtual machines, DBI).
  - ✍ For both manual and automated modes, we present a detailed classification of malware evasion tactics and techniques.
  - ✍ To the best of our knowledge, this would be the first comprehensive survey of dynamic analysis evasion tactics that offers a thorough classification.
  - ✍ We provide a brief survey on countermeasures against evasive malware that the industry and academia is pursuing.
- ☞ We propose a new method of evasion for manual dynamic analysis evasion technique.
  - ✍ We propose one evasion method based on an existing bug in Windbg which does not interpret correctly one specific implementation of interruption breakpoint.
  - ✍ We propose three different ways to fool disassembly engines from debuggers based on badly interpreted assembly instructions. Result produces unreadable code displayed to user.
- ☞ We propose a universal method of dynamic analysis evasion for malware.
  - ✍ This method is based on an original way to perform cross-modifying code between two threads and not covered by Intel documentation.
  - ✍ Our method allows to process both manual and automatic dynamic analysis evasion technique.
  - ✍ If the method works every-time with debuggers and DBI, it remains probabilistic with virtual machines if there is no prior calibration based on the host CPU.
  - ✍ Based on the feedback from a test campaign, we have improved our detection method to avoid requiring a calibration step.

THIS PAGE INTENTIONALLY LEFT BLANK

---

## Chapter 4

# State of the art about Windows keyboard management

## 1 General introduction

### 1.1 About keylogger threat and the organization of the following chapters

The idea of improving the security of a computer system, the fight against malicious threats is a key point. Among these threats, keyloggers have a very special role. Many definitions have been given to a keylogger in literature. Among them, Trend Micro antirvirus defines keyloggers [473] as *"keyloggers are programs that log keyboard activity"*. This is a general definition, comforted by [474] which references a keylogger as:

*"A keylogger is a software designed to capture all of a user's keyboard strokes, and then make use of them to impersonate a user in financial transactions."*

Sophos offers a more inclusive definition [475] where:

*"Keyloggers are activity-monitoring software programs that give hackers access to your personal data. The passwords and credit card numbers you type, the webpages you visit - all by logging your keyboard strokes"*.

It can be observed that from all its definitions, the role of keyloggers, if it is focused on the keyboard, can be extended to other areas, and more generally for any data manipulated by the keyboard. But the two definitions do not necessarily detail the form a keylogger can take. Therefore, the antivirus company Kaspersky proposes a dual definition [476]. From Kaspersky's point of view:

*The concept of a keylogger breaks down into two definitions:*

1. **Keystroke logging:** *Record-keeping for every key pressed on your keyboard.*
2. **Keylogger tools:** *Devices or programs used to log your keystrokes.*

In [476], there is a difference between the pupurpose of a keylogger (i.e. a tool for collecting information, mainly focused on all the data coming from the keyboard) and the technical means implemented to proceed. That way, there is an important distinctions in terms of the methods used to collect information. Indeed, there are several types of keyloggers, adapted to different architectures and responding to different needs. A keylogger can be hardware or software. On the first hand, hardware keylogger are designed to handle the keyboard device

physically while, on the other hand, software keylogger is designed to be run on the operating system receiving information from the device. Detailed explanations about keylogger shapes are provided in sections 2 and 3.

Form our point of view, a keylogger is more a concept, a tool, a feature, a way for doing something, than a definitively fixed definition. But generally speaking, we can say that a *keylogger* is a *malicious feature* (hardware or software) that aims to *retrieve* what a *user sends* as data into a machine.

Data retrieved in the system can include document contents, passwords, user-names, and other potentially sensitive piece of information. More generally, it concerns all critical information manipulated by the user. The goals of keyloggers authors are diverse. It includes everything from stealing bank credentials for money theft to espionage (industrial or state espionage). With credentials from systems, the attacker has a great advantage to success any attack, stealthy and efficiently. The consequences can be dramatic for the victims of these tools while they can be very lucrative for their perpetrators. And this explains why this type of tool is so popular.

According to Check Point Software Technologies Ltd’s cyber security report [6], in the top ten malware sample families that have been found worldwide in 2020 (provided in Figure 4.1), seven [477, 478, 479, 480, 481, 482, 483, 484] are using a keylogger technology (the other three malware Jsecoin, Cryptoloot, and Coinhive are *crypto-miner* designed to perform online mining on the infected machine).

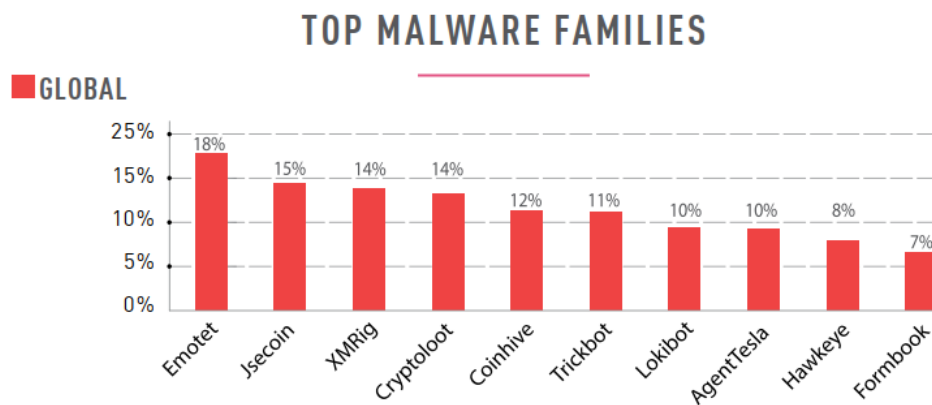


Figure 4.1: Top malware families - extracted from [6].

Nowadays, keyloggers are generally embedded as a feature in more general purpose malware (such as info-stealer or botnets malware). This is also one of the reasons that makes this type of malware complex to detect [485, 486]. And it may even be necessary to question the need to detect it. Indeed, it may be potentially more interesting to neutralize this type of behavior and the effects it produces, whatever the software, rather than trying to characterize it.

It is with this approach that our study was conducted. By understanding how keyloggers work and the underlying technology they rely on can enable us to deploy effective defense systems. The purpose of this and the next two chapters is to better understand how the keyboard works in Windows in order to propose solution designed to counter keylogger malware. The articulation of the chapters, the links between the parts and the key points explained in each section is given in Figure 4.2.

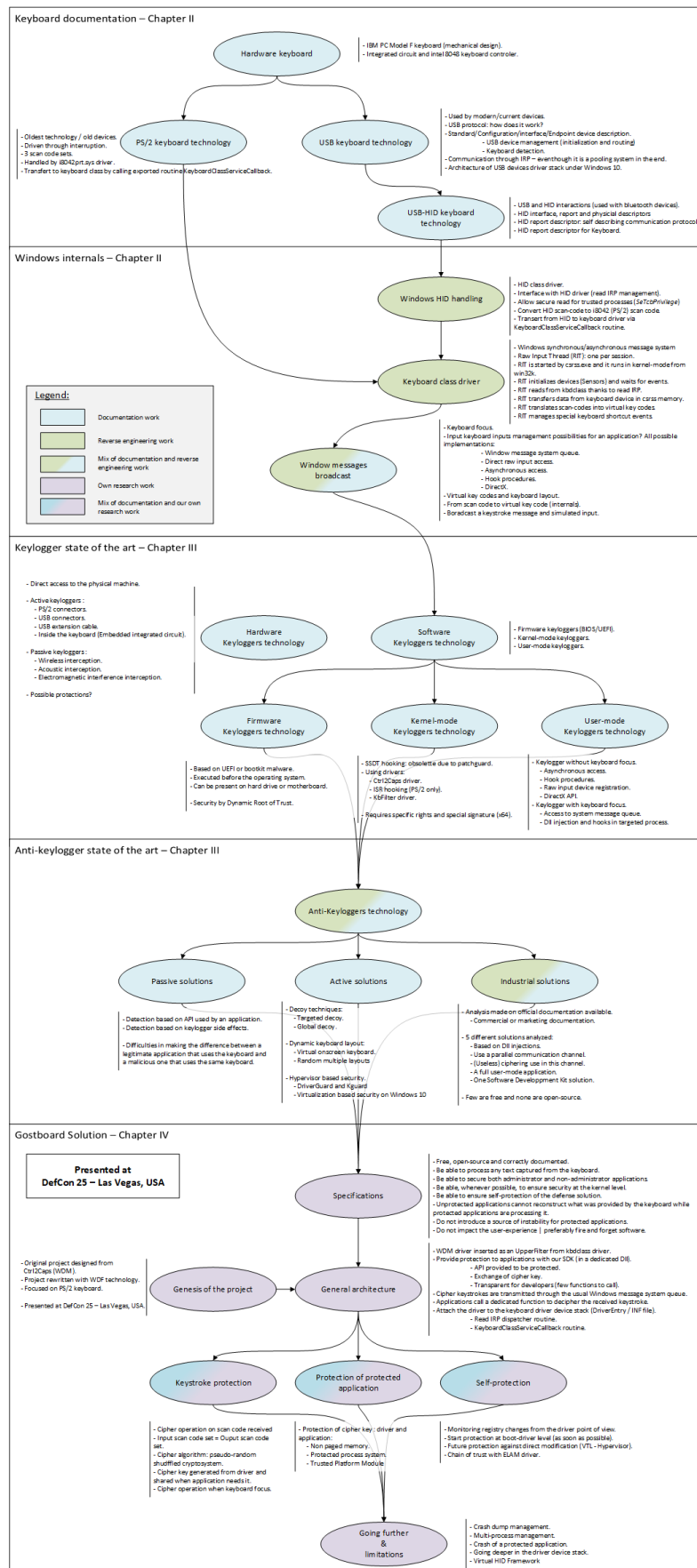


Figure 4.2: Plan of the next chapters dealing with keylogger threat management.

## 1.2 Introduction about keyboard technology

To understand how a keylogger malware works or any solution responsible to neutralize one, it is necessary to understand first how keyboard is managed under Windows operating system. From a general point of view, the way the keyboard is managed under Windows has not really evolved since Windows NT. The original spirit remains the same since retro-compatibility fundamental forces Microsoft to keep existing technology close to the shape it has before. Various third-party drivers or application software depends on API functions or structures provided by Windows. It means that API functions which have been used before must continue to provide the same service in the future.

However, while backwards compatibility requires maintaining old structures, it is important to note that technology keeps on evolving. From the first keyboards connected to PS/2 ports to Bluetooth keyboards, both the hardware and software have been subject to updates. One of the objectives of this section is to present the way the keyboard works and the associated technological developments. From both historical and technological point of views, we will see here how to drive and manage the keyboard at different levels and with different hardware devices. Finally, it will be interesting to note that the further we move away from the hardware and its specificities, the easier is the interface for the programmer. This makes sense, because, even-though the hardware has evolved, the software interface to interact with it has not changed to keep compatibility with older versions.

The technology used for both USB and PS/2 keyboard is not complex from an electronic point of view. In the end, both are serialized communication devices where interface has been standardized. When they are plugged for the first time in the computer (at boot-time or at running time), electronic management is taken into account by the motherboard. In both cases, connectors are composed with many pinouts including at least a ground, a clock, a data and an input Vcc (+5V). Power specifications is only relevant if we are willing to design the silicon for a PS/2 or USB device/transceiver. To go further, we would like to mention the following reference documents for PS/2 [487, 488, 489, 490, 491, 492] and for USB [7, 493].

Automatic detection and setup configuration of both devices are different. For PS/2 keyboard, this one does not works as it works with USB. In the first case, device discovering is guaranteed by electronic connections with the motherboard followed by PS/2 commands [492, 490]. With USB devices, a most complete procedure is fully described in section 4 (Key-Point 4.6). Note that, in the case of PS/2 connectors, improvements have been made to allow a friendly-user interface (especially by removing the difference between keyboard and mouse connectors) but it inadvertently created more issues [494].

We propose in this chapter to drive the explanation by following the path taken by information when a key is pressed. From the electric signal generated from the keyboard device to the graphical application reacting to that signal in the computer. We propose first to explain the starting point, when the keyboard device itself transfers information to the computer (section 2). Then, the kernel handles it depending on the technology the keyboard used (PS/2 in section 3 or USB/HID in section 4). Finally, the Windows kernel is in charge of handling signals received from keystroke in order to broadcast them to applications running in the system (section 5 — Key-Point 25).



## 2 Keystroke from hardware keyboard

### Key Point 4.1: Keyboard hardware devices

- 👉 This section explains the electronic architecture of the keyboards used on our computers.
  - 👉 The current design configuration of our keyboards comes from Model F keyboard designed by IBM.
  - 👉 The internal electronics of the device are quite simple. The goal is to identify the keys pressed and transmit their code to the host machine through dedicated micro-controllers (historically with the Intel 8048 CPU).
  - 👉 The transmission of information depends on the connectivity used: PS/2 or USB.

Technically speaking, the first keyboards were directly embedded in computers and managed specifically by operating system (like the DataPoint 3300 in 1969). The first general public keyboards were connected with serial wire (such as the Model F keyboard [495] from IBM which has used, among others, a 5-pin/180 DIN connector). Since they are no more really used, it does not appear relevant to talk about, even if the philosophy to manage them is similar to others.

Among the different hardware keyboards manufactured all over the world, the design of the IBM PC Model F keyboard [495, 496] is one of those that have had a long-term influence on the design of keyboards. This is why we chose to focus on its architecture since the principles of this one has been reused a lot in modern keyboards. In this part, we propose to detail the different layers of the keyboard from the point of view of the hardware equipment (and not of the computer to which it is connected). Technically, a keyboard is a set of keys which, whenever they are pressed, deliver an electrical signal that is transmitted to the computer to which the keyboard is connected. Formally, it is a device as given in Figure 4.3 [497].



Figure 4.3: IBM PC Model F Type 1 keyboard device.

If we remove the keys from the keyboard and the external framework of the device, we have the result given in Figure 4.4.



Figure 4.4: Keyboard IBM-PC model F type 1 mechanism from top without key.

If we remove the plastic part, we discover the bottom barrel plate (Figure 4.5) composed of hammers that come to strike the metal plate (Figure 4.6) underneath. This plate is the metallic physical contact between the keys and the *Printed Circuit Board* (PCB) representing the electronic matrix of keys.



Figure 4.5: Bottom barrel plate with hammers.



Figure 4.6: Bottom cover removed to get access to the metal bar that makes contact between the hammers and the PCB underneath.

The electronic matrix of keys is given in Figure 4.7. When a key is pressed, the hammer behind the key strikes the metal bar which makes a physical contact on the matrix. This matrix is a capacity circuit able to give a different input depending on the key pressed, allowing to differentiate them. This behavior is similar to a piano.

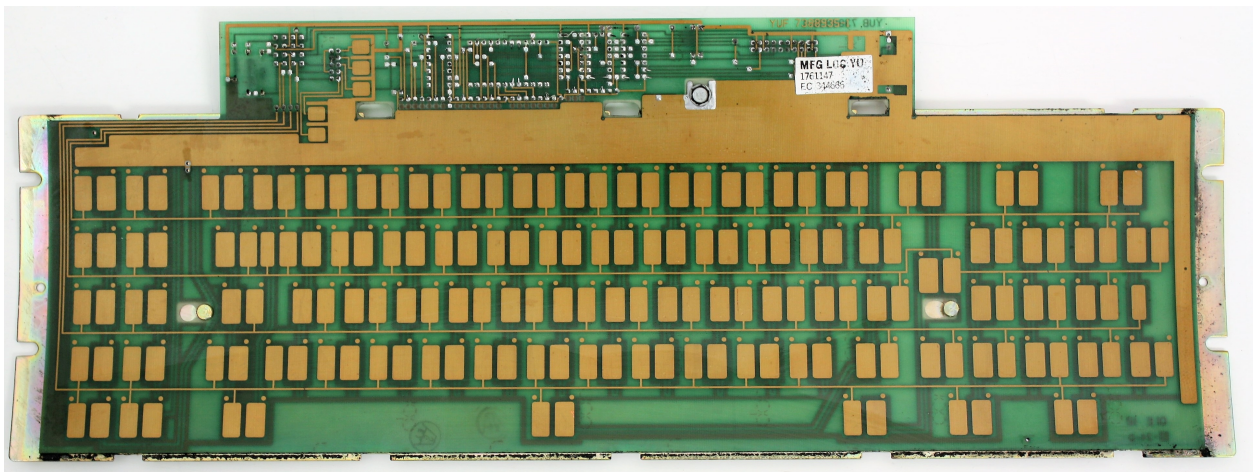


Figure 4.7: Capacitive PCB representing the electronic matrix of keys.



If we focus on the top of Figure 4.6, we can see two electronic circuits which are supposed to manage the keystrokes. The first top right PCB is a chip responsible to manage keystrokes. Technically, a keyboard is made up of switches constituting a key matrix [498] continually read by the internal keyboard's micro-controller [499] (Figure 4.6). Each of these switches is associated with a hardware key which each produces a dedicated value. This one is directly managed by the internal micro-controller of the device, usually an original Intel 8048 [500, 501] from the MCS-48 micro-controller series [502] (Figure 4.9).

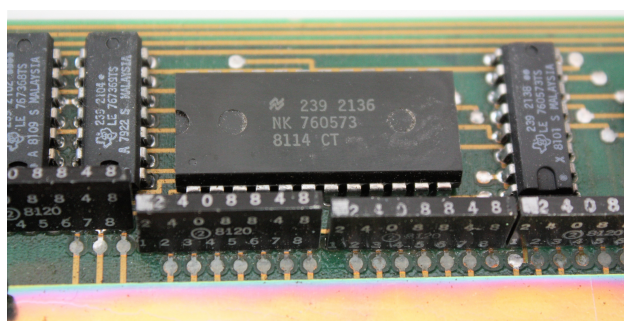


Figure 4.8: Keyboard matrix integrated circuit manager.

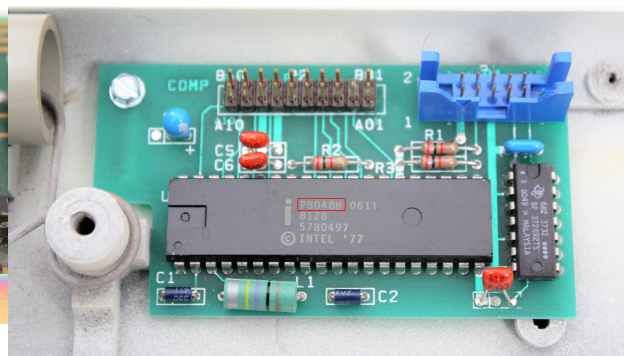


Figure 4.9: Intel 8048 keyboard controller.

Of course, the exact micro-controller used in a keyboard is very dependent from one manufacturer to another. But if the chip is different, the required task to perform is always the same: keyboard encoder. Since a key is pressed (or released), a mechanical movement forces a switch to be connected on the key grid composed by the electronic circuit of the device. Because the rows and columns within the key grid are connected to 8 bit I/O ports on the keyboard encoder, a keystroke generates a signal on the dedicated port connected by the physical key. Logical diagram extracted from [496] is given in Figure 4.10 to represent this action.

With this architecture, the goal of the keyboard encoder is to scan in real time the ports enabled to see whether a key is down or not. Once the key is identified, the second PCB (top left in Figure 4.6) is supposed to translate it to an understandable code for the host computer where the keyboard is connected. This process is performed with the help of an Intel 8048 micro-controller (Figure 4.9). This one reads in a read only memory (ROM) which value is associated to the signal received. The electronic diagram of this process is given in Figure 4.11 [496]. Note that key matrix handler (Figure 4.10) is more or less embedded in a box "keyboard capacitive matrix" on that scheme.

Finally, the micro-controller of the keyboard device generates a signal, through the wire of the keyboard, to carry the value from the device to the computer. It is therefore up to the receiving machine to process this information so that it can be understood by its software.

The signal of a keystroke is transferred from the keyboard device to the host machine. At that point, it is the responsibility of the host operating system to handle it. The kernel of the operating system, since it handles hardware management, is the first involved. Then comes applications (user-mode applications) which are usually final consumers from keyboard input. From the kernel glasses, the main responsibility is to handle the signal, wherever it comes from, to commute it on the keyboard device stack in such a way it will be correctly dispatched to the rest of the system. Of course, keyboard signal can come from different places using different types of technology to interface with the host they are connected. From PS/2 connection [503] to wireless device, including USB/HID connections, it must be correctly handled and recognized. Because of keyboard manufacturers use different types of interface, it remains that the software still need to interact the same way with the device. For retro-compatibility purposes first but also because the operating systems are not motivated to handle different drivers software for each keyboard interface type. Explaining how different types of connectivity work and how they are implemented through Windows operating system is the aim of the next sections.

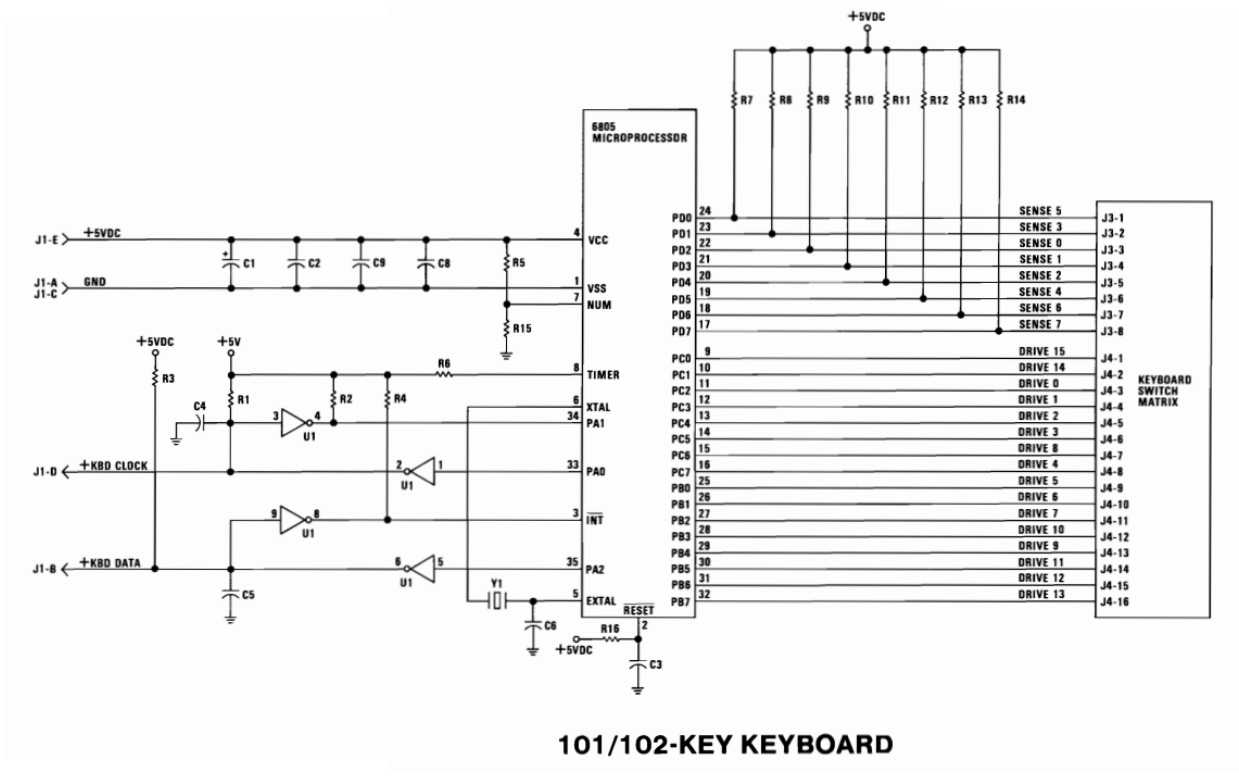


Figure 4.10: Logical diagram representing the circuit given in Figure 4.8.

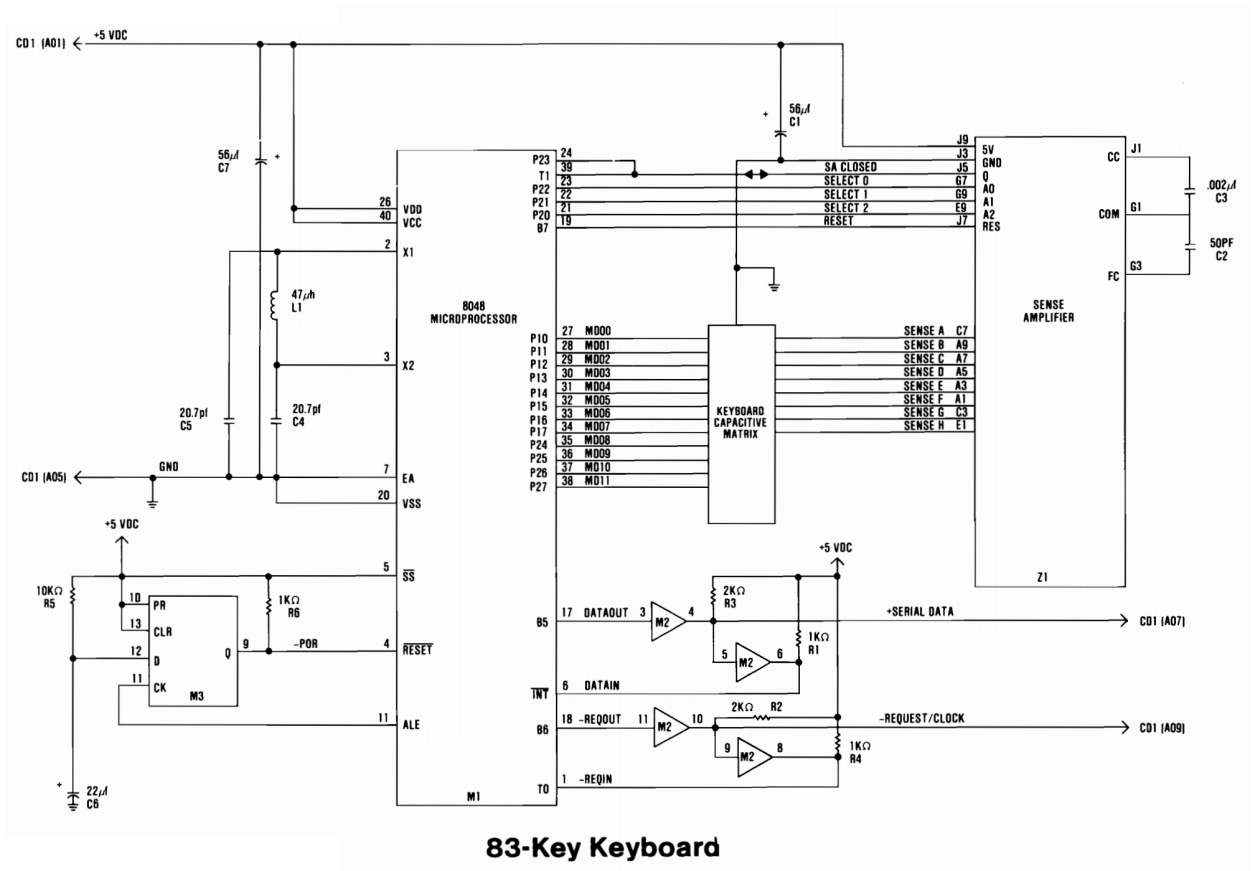


Figure 4.11: Logical diagram representing the circuit given in Figure 4.9.

### 3 PS/2 technology

#### Key Point 4.2: PS/2 technology

- ☞ PS/2 standard is a communication protocol used to communicate with PS/2 connected devices. Today, this one is mostly obsolete.

#### 3.1 Presentation of PS/2 technology

#### Key Point 4.3: PS/2 technology presentation

- ☞ PS/2 standard is an old technology (released in 1987) coming from IBM AT keyboard port.
  - ☞ PS/2 keyboard management is generally handled by micro-controllers present on the side of the motherboard on the computer.
  - ☞ Historically, Intel 8042 chip has handled keyboards. Whatever is the chipset used today, it is still commonly referred as "8042".

One of the technology to manage keyboard which is still in use (even if it decreases nowadays) is PS/2 [504]. Technically speaking, the PS/2 port is a 6-pin mini-DIN connector (Figure 4.12 [496]) which can be used by keyboard or mouse to interact with any type of PC compatible computers (in the sense of original IBM PC [505], XT, and AT, able to use the same software). This one is an extension of the "AT" standard crafted to be a serial interface with a bi-directional interface. Except the plug-connector, the PS/2 standard is electrically identical to the "AT" one. The name of PS/2 comes from the IBM Personal System/2 computer [506] (PS/2, for short) which is the third generation of personal computer, released on 1987. One of the main innovation it carried was the new interface for keyboard and mouse. The latter became *de facto* a standard because it was later adopted on a large number of computers. The success of the Model M keyboard from IBM is part of the story. Indeed, this one looks like most modern keyboards.

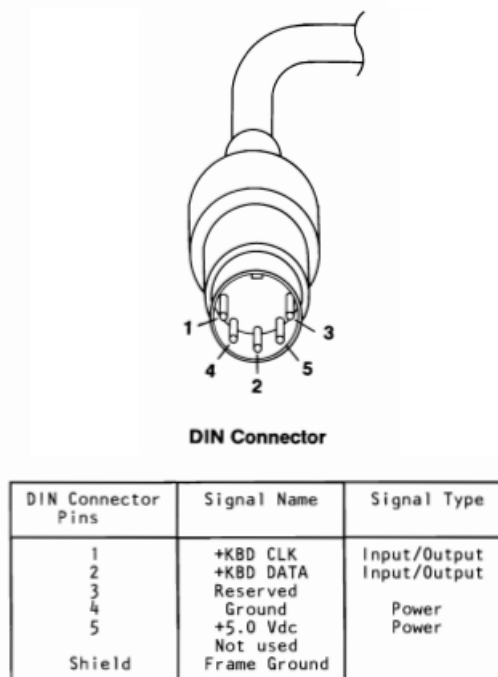


Figure 4.12: Representation of DIN connector.

The internal functioning [507] of the PS/2 protocol is quite obsolete nowadays. Even if this connector is still used in some (old) industrial systems, it is no more present in consumer computers. This is why we will not see all the details about this technology because it concerns only a few systems, in the end (nonetheless, more information about PS/2 keyboard technology can be retrieved in [492]). Nevertheless, this one can still be relevant to build a hardware PS/2 device. Indeed, PS/2 keyboard management is generally carried out by micro-controllers present on the side of the motherboard on the computer. More directly, when we are interfacing with the keyboard, we are not communicating directly with the keyboard device through the bus. Instead, a keyboard controller provides an interface between the keyboard and the peripheral bus.

The keyboard controller interfaces with the keyboard encoder chip through the keyboard communication protocol and it provides a way to interface it. This micro-controller is present on motherboards from the early days of keyboard history since it is responsible, among other things, to handle commands coming to and from computers, translation from one scan code set to another scan code set and any third-party modern functionality such as specific functionality keys and commands [508]. In the early days, the controller was constituted by a single chip (Intel 8042) used to control access to the A20 address line [509] responsible to correct a compatibility issue with the Intel 80286 CPU. Even in modern computers, this chip is still one of many components present on a motherboard. Whatever is the model of the chip used today [487], there is still a part of motherboard chipset responsible to handle the keyboard and it is still commonly referred under the "8042" name. Note that, if it is the Intel 8042 which is used on computer side, it is the Intel 8048 which is used on keyboard side (Figure 4.13).

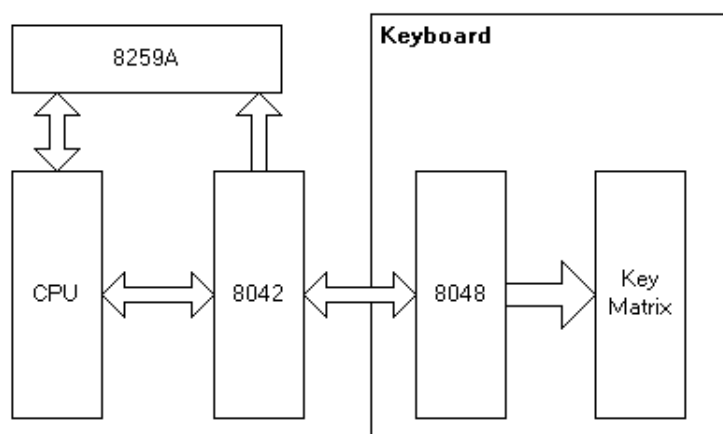


Figure 4.13: Presentation of the architecture between the keyboard and the host's motherboard.

### 3.2 Kernel interface with a device and scan code sets

#### Key Point 4.4: PS/2 scan code sets

- ☞ PS/2 standards uses three different scan code sets (for historical reasons).
  - ☞ A scan code is an *alphabet* that associates a specific value (one or several bytes) to each key on the keyboard.
  - ☞ The scan code set is selected when the device is initialized.
  - ☞ For the same key, the scan code value sent is different depending on the key is pressed, released or held down.

From the computer points of view, there are two ways to receive bytes from the device: polling or interruption request. The first is about to wait on an IO port until a signal is receive. This solutions has at least two issues. The first is that, like all polling solutions, it wastes a lot of CPU time for nothing. The second lies in the



case where the device embeds mouse and keyboard together, connected on the same controller (as it is possible with PS/2). In such a case, there is no reliable way to determine from which device data received has been sent, unless one of them has been disabled [491].

The second way to transfer data relies on interruption requests (aka IRQ). For the sake of brevity, this method maps a dedicated interruption port from the CPU (linked via the motherboard) to the device. This way, the CPU is interrupted each time a specific event occurs on that port. More directly, the current CPU task is suspended to handle the notification which is provided by the device each time it is required. This implies a notion of priority of task to allow one to preempt a second one. Nowadays, the interruption system is the most widely used.

The signal transmitted through the wire is composed of the value of the key. This value is defined by the type of keyboard used and it depends on the layout (with geographic specificity) of this one. For instance, German, English, Russian, Chinese or French keyboards have different layouts. More directly, the location where characters are mapped on the keyboard are different. The different values produced by a keyboard are grouped together in a scan code set which determines when a key is pressed, repeated or released. Even if any manufacturer is free to build its own one, there are three different sets [510] of scan codes. The oldest is originally named "scan code set 1" [511] defined by IBM PC XT [512]. The default one used by a lot of keyboards is "scan code set 2" [513] (from IBM PC AT [505]), and there is a newer (and more complex) one named "scan code set 3" (from IBM PC 3270 [514]).

The way the computer understands all different scan codes depends on the architecture of the keyboard and on the driver installed on the computer. It happens on specific devices that the keyboard itself uses scan code set 2 and the keyboard controller translates this one into scan code set 1 for compatibility purposes. Depending on the device, it can be possible to ask it to determine which scan code set this one uses (and eventually changing it, when it could be possible). A synthetic list of scan codes content from the different scan code sets is given in table 4.1.

IBM Key No.	Set 1 Make/Break	Set 2 Make/Break	Set 3 Make/Break	Base Case	Upper Case
1	29/A9	0E/F0 0E	0E/F0 0E	,	~
2	02/82	16/F0 16	16/F0 16	1	!
3	03/83	1E/F0 1E	1E/F0 1E	2	@
4	04/84	26/F0 26	26/F0 26	3	#
5	05/85	25/F0 25	25/F0 25	4	\$
6	06/86	2E/F0 2E	2E/F0 2E	5	%
7	07/87	36/F0 36	36/F0 36	6	^
8	08/88	3D/F0 3D	3D/F0 3D	7	&
9	09/89	3E/F0 3E	3E/F0 3E	8	*
10	0A/8A	46/F0 46	46/F0 46	9	(
11	0B/8B	45/F0 45	45/F0 45	0	)
12	0C/8C	4E/F0 4E	4E/F0 4E	-	-
13	0D/8D	55/F0 55	55/F0 55	=	+
15	0E/8E	66/F0 66	66/F0 66	Backspace	
16	0F/8F	0D/F0 0D	0D/F0 0D	Tab	
17	10/90	15/F0 15	15/F0 15	q	Q
18	11/91	1D/F0 1D	1D/F0 1D	w	W
19	12/92	24/F0 24	24/F0 24	e	E
20	13/93	2D/F0 2D	2D/F0 2D	r	R
21	14/94	2C/F0 2C	2C/F0 2C	t	T
22	15/95	35/F0 35	35/F0 35	y	Y
23	16/96	3C/F0 3C	3C/F0 3C	u	U
24	17/97	43/F0 43	43/F0 43	i	I
25	18/98	44/F0 44	44/F0 44	o	O
26	19/99	4D/F0 4D	4D/F0 4D	p	P
27	1A/9A	54/F0 54	54/F0 54	[	{
28	1B/9B	5B/F0 5B	5B/F0 5B	]	}
30	3A/BA	58/F0 58	58/F0 58	Caps Lock	
31	1E/9E	1C/F0 1C	1C/F0 1C	a	A
32	1F/9F	1B/F0 1B	1B/F0 1B	s	S
33	20/A0	23/F0 23	23/F0 23	d	D
34	21/A1	2B/F0 2B	2B/F0 2B	f	F
35	22/A2	34/F0 34	34/F0 34	g	G
36	23/A3	33/F0 33	33/F0 33	h	H
37	24/A4	3B/F0 3B	3B/F0 3B	j	J
38	25/A5	42/F0 42	42/F0 42	k	K
39	26/A6	4B/F0 4B	4B/F0 4B	l	L
40	27/A7	4C/F0 4C	4C/F0 4C	:	:
41	28/A8	52/F0 52	52/F0 52	'	"
43	1C/9C	5A/F0 5A	5A/F0 5A	Enter	Enter
44	2A/AA	12/F0 12	12/F0 12	Left Shift	
46	2C/AC	1A/F0 1A	1A/F0 1A	z	Z
47	2D/AD	22/F0 22	22/F0 22	x	X
48	2E/AE	21/F0 21	21/F0 21	c	C
49	2F/AF	2A/F0 2A	2A/F0 2A	v	V
50	30/B0	32/F0 32	32/F0 32	b	B
51	31/B1	31/F0 31	31/F0 31	n	N

52	32/B2	3A/F0 3A	3A/F0 3A	m	M
53	33/B3	41/F0 41	41/F0 41	,	<
54	34/B4	49/F0 49	49/F0 49	.	>
55	35/B5	4A/F0 4A	4A/F0 4A	/	?
57	36/B6	59/F0 59	59/F0 59		Right Shift
58	1D/9D	14/F0 14	11/F0 11		Left Ctrl
60	38/B8	11/F0 11	19/F0 19		Left Alt
61	39/B9	29/F0 29	29/F0 29		Spacebar
62	E0 38/E0 B8	E0 11/E0 F0 11	39/F0 39		Right Alt
64	E0 1D/E0 9D	E0 14/E0 F0 14	58/F0 58		Right Ctrl
75	E0 52/E0 D2 (base)	E0 70/E0 F0 70 (base)	67/F0 67		Insert
76	E0 4B/E0 CB (base)	E0 71/E0 F0 71 (base)	64/F0 64		Delete
79	E0 4B/E0 CB (base)	E0 6B/E0 F0 6B (base)	61/F0 61		Left Arrow
80	E0 47/E0 C7 (base)	E0 6C/E0 F0 6C (base)	6E/F0 6E		Home
81	E0 4F/E0 CF (base)	E0 69/E0 F0 69 (base)	65/F0 65		End
83	E0 48/E0 C8 (base)	E0 75/E0 F0 75 (base)	63/F0 63		Up Arrow
84	E0 50/E0 D0 (base)	E0 72/E0 F0 72 (base)	60/F0 60		Down Arrow
85	E0 49/E0 C9 (base)	E0 7D/E0 F0 7D (base)	6F/F0 6F		Page Up
86	E0 51/E0 D1 (base)	E0 7A/E0 F0 7A (base)	6D/F0 6D		Page Down
89	E0 4D/E0 CD (base)	E0 74/E0 F0 74 (base)	6A/F0 6A		Right Arrow
90	45/C5	77/F0 77	76/F0 76		Num Lock
91	47/C7	6C/F0 6C	6C/F0 6C		Keypad 7
92	4B/CB	6B/F0 6B	6B/F0 6B		Keypad 4
93	4F/CF	69/F0 69	69/F0 69		Keypad 1
95	E0 35/E0 B5 (base)	E0 4A/E0 F0 4A (base)	77/F0 77		Keypad /
96	48/C8	75/F0 75	75/F0 75		Keypad 8
97	4C/CC	73/F0 73	73/F0 73		Keypad 5
98	50/D0	72/F0 72	72/F0 72		Keypad 2
99	52/D2	70/F0 70	70/F0 70		Keypad 0
100	37/B7	7C/F0 7C	7E/F0 7E		Keypad *
101	49/C9	7D/F0 7D	7D/F0 7D		Keypad 9
102	4D/CD	74/F0 74	74/F0 74		Keypad 6
103	51/D1	7A/F0 7A	7A/F0 7A		Keypad 3
104	53/D3	71/F0 71	71/F0 71		Keypad .
105	4A/CA	7B/F0 7B	84/F0 84		Keypad -
106	4E/CE	79/F0 79	7C/F0 7C		Keypad +
108	E0 1C/E0 9C	E0 5A/E0 F0 5A	79/F0 79		Keypad Enter
110	01/81	76/F0 76	08/F0 08		Esc
112	3B/BB	05/F0 05	07/F0 07		F1
113	3C/BC	06/F0 06	0F/F0 0F		F2
114	3D/BD	04/F0 04	17/F0 17		F3
115	3E/BE	0C/F0 0C	1F/F0 1F		F4
116	3F/BF	03/F0 03	27/F0 27		F5
117	40/C0	0B/F0 0B	2F/F0 2F		F6
118	41/C1	83/F0 83	37/F0 37		F7
119	42/C2	0A/F0 0A	3F/F0 3F		F8
120	43/C3	01/F0 01	47/F0 47		F9
121	44/C4	09/F0 09	4F/F0 4F		F10
122	57/D7	78/F0 78	56/F0 56		F11
123	58/D8	07/F0 07	5E/F0 5E		F12
124	E0 2A E0 37/E0 B7 E0 AA	E0 12 E0 7C/E0 F0 7C E0 F0 12	57/F0 57		Print Screen
125	46/C6	7E/F0 7E	5F/F0 5F		Scroll Lock
126	E1 1D 45/E1 9D C5	E1 14 77 E1/F0 14 F0 77	62/F0 62		Pause Break
29 or 42*	2B/AB	5D/F0 5D	5C/F0 5C or 53/F0 53	\	-

Table 4.1: List of different scan codes from all different scan code sets — IBM PS/2 Model 50 and 60 Technical Reference.

A scan code set is composed of scan codes values. Scan codes themselves are sequences of one or more bytes. Technically speaking, a scan code is a data packet that represents the state of a key. When the state of a key is changing (pressed, released or held down), a scan code is sent to the computer through keyboard controller. For the sake of precision, we talk about two types of scan codes: make codes for pressed keys and break codes for released keys. There is a unique make code and break code for each key referenced on keyboard. But these scan codes are not always composed by a single byte value. In some cases, it could evolve more than one bytes for specific sequences (on scan code set 1, operation "print screen pressed" is composed of 0xE0, 0x2A, 0xE0, 0x37 bytes). Technically, there are two types of extended scan codes where the prefix byte is 0xE000 or 0xE100 [515].

Of course, this situation is far from being ideal for operating systems which would prefer to handle only one integer value instead of a sequence of unknown number of bytes (with parsing operations or using large lookup tables to know which sequence corresponds to what). From operating system point of view, each keystroke must be represented as a single value called "key code". The main problem lies in the fact there is no real standard for key codes. Each operating system has its own one, including Windows [516] and Linux<sup>1</sup>. Such architecture means, when the keyboard's driver knows it has received a complete scan code, it has to convert the sequence of bytes representing the scan code into a key code, understandable by the operating system.

<sup>1</sup><https://github.com/torvalds/linux/blob/master/include/uapi/linux/input-event-codes.h>

### 3.3 Handling PS/2 by Windows

#### Key Point 4.5: Windows and PS/2 protocol

- ☞ Windows interfaces PS/2 protocol thanks to `i8042prt.sys` driver.
  - ☞ Communication with PS/2 keyboard and host machine CPU is performed with interruptions at 0x60 and 0x64 ports.
  - ☞ Windows kernel uses an *Interrupt Service Routine* (ISR) called `I8042KeyboardInterruptService` to handle keyboard.
  - ☞ Part of the keystroke handling procedure is performed in a *Deferred Procedure Call* (DPC) via `I8042KeyboardIsrDpc` routine.
- ☞ Notification of keyboard driver (`kbdclass.sys`) is performed by `i8042prt.sys` driver via `I8042KeyboardIsrDpc` routine.
  - ☞ To proceed, `I8042KeyboardIsrDpc` routine calls `KeyboardClassServiceCallback` routine directly.
  - ☞ `KeyboardClassServiceCallback` routine is exported by `kbdclass.sys`.
- ☞ There is a system of device names to represent each driver in the keyboard device call stack.

From the Windows operating system point of view, it is the driver `i8042prt.sys` responsibility to handle communications between the kernel and the keyboard controller [517]. The name of this driver owes nothing to chance. It consists of the name of the microcontroller commonly used and an abbreviation of the word "port". The notion of port is central because it allows the computer's CPU to communicate with the keyboard. More precisely, if the CPU needs to interact with the keyboard, it has two dedicated ports to receive or to send information to the keyboard encoder linked to the keyboard controller. The PS/2 interaction is driven via IO ports 0x60 and 0x64 [506, 504, 499, 511, 488]. Like any other ports, read and write operations allow access to different internal registers and data.

The data port (IO Port 0x60) is used for reading or writing data that was received or sent from a connected PS/2 device via the keyboard controller [489]. Technically, read operation concerns the retrieval of keys pressed and stored in an internal buffer of the device. It consist of retrieving data content (representing keystrokes) as status associated to the keyboard controller (output or input buffers full or empty, system flag, command state, time-out, parity error). The second one, the command port (IO Port 0x64), is used to send commands to the keyboard controller (not to device itself, i.e. keyboard encoder). This one can be used to disable or enable the keyboard (commands 0xAD and 0xAE) and more generally to drive and configure state of the keyboard controller (not the device). Other interfaces directly connected to the device (set LED, select the scan code to use, Set autorepeat delay, etc.) are driven by the 0x60 port. An illustration to summarize this is provided in Figure 4.14.

The `i8042prt.sys` driver is in charge of managing notifications from keyboard device. For efficiency reasons, notifications are made by interruptions. The keyboard controller is configured to rise an interruption request whenever a key has been pressed or released. This interruption is handled by an *Interrupt Service Routine* (ISR) [518] that has been registered by the driver in charge of the physical device. The system calls the ISR each time it receives that interrupt. For historical reasons, Windows supports two types of interrupts. Before Windows Vista, devices that used ports and buses prior to PCI 2.2 [519, 520] could only support generated line-based interrupts. It means that a device generates an interrupt by sending an electrical signal on a dedicated pin known as an interrupt line. Starting with PCI 2.2, devices can generate message-signaled interrupts by writing a data value to a particular address. In our case, we are dealing with hardware interruptions. They are nothing more than regular interrupts except that instead of being called explicitly from an assembly program (with `int` instruction), they are invoked automatically by hardware.

Interruptions in Windows are managed through the Interrupt Dispatch Table (IDT). This one is a collection of function pointers responsible to hold notification coming from the hardware to the system. All the interruption routines have been initialized by the operating system (even if it is possible to register some manually [521]

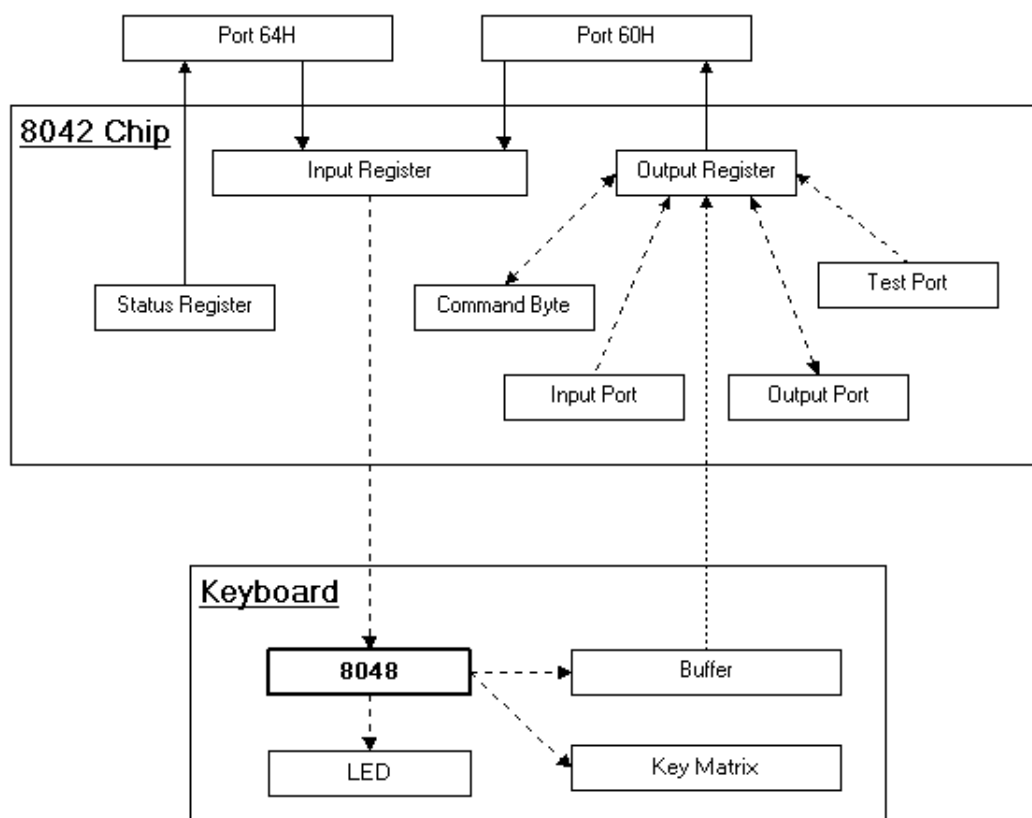


Figure 4.14: Interaction between keyboard and host with interruption ports.

thanks to `IoConnectInterruptEx` routine [522]). This is where the ISR are directly stored. In the case of handling keyboard, `i8042prt.sys` driver exports the `I8042KeyboardInterruptService` function [523] to read data stored in the keyboard controller in order to know which key has been pressed (or released).

For some unknown reason<sup>2</sup>, Windows 10 registers the PS/2 keyboard interrupt on the ISR to vector number `0xA0`. It is possible to see it thanks to Windbg debugger [209] by mapping the IDT. In Code 4.1, the view of the IDT provides us access to address of `I8042KeyboardInterruptService` routine referenced in the opaque `KINTERRUPT` structure [525]. In practice, with the debugger, it is possible to compute address of the `KINTERRUPT` structure from the IDT base address [523]. In such case, we get access to the `0xA0` interruption entry by adding this value multiplied by `0x10` (to get the size of a pointer in 64-bit architecture) to the base address of the IDT. This gives us a `KIDTENTRY64` undocumented structure which references an address in `KiLsrThunkShadow` routine jumping to `KxLsrLinkageShadow` routine after having pushed the `0xA0` vector number. This last routine will finally call `I8042KeyboardInterruptService` routine.

```

1 kd> !idt
3 Dumping IDT: ffff9d8137db4000
5 00: fffff8042ac11100 nt!KiDivideErrorFaultShadow
6 01: fffff8042ac11180 nt!KiDebugTrapOrFaultShadow Stack = 0xFFFF9D8137DB89D0
7 02: fffff8042ac11240 nt!KiNmiInterruptShadow Stack = 0xFFFF9D8137DB87D0

```

<sup>2</sup>To the best of our knowledge, there is no official explanation about such a choice. Official ports `0x60` and `0x64` are mostly used by BIOS and it is convenient for operating system to keep them. But it is possible to manage it differently with modern system (which are not using BIOS anymore). Note that an explanation could be given from [524]. In this case, `0xA0` would be a separate interrupt controllers used to send an *end of interrupt* command at the end. In that case, Windows would wait for the whole operation to finish before processing it.

```

03: fffff8042ac112c0 nt!KiBreakpointTrapShadow
9 04: fffff8042ac11340 nt!KiOverflowTrapShadow
05: fffff8042ac113c0 nt!KiBoundFaultShadow
11 (...)
60: fffff8042ac125c0 USBPORT!USBPORT_InterruptService (KINTERRUPT ffff9d81377fc780)
13 70: fffff8042ac12640 VBoxGuest+0x22d0 (KINTERRUPT ffff9d81377fcb40)
(...)
15 90: fffff8042ac12740 i8042prt!I8042MouseInterruptService (KINTERRUPT ffff9d81377fca00)
a0: fffff8042ac127c0 i8042prt!I8042KeyboardInterruptService (KINTERRUPT ffff9d81377fc8c0)
17 (...)
fe: fffff8042ac12ab0 nt!HalPerfInterrupt (KINTERRUPT ffffd826aaa7240)

```

Code 4.1: "Partial view of the IDT from Windows 10 (running on virtual box virtual machine)"

Note that `KlsrThunkShadow` and `KxlsrLinkageShadow` does not appear in the call stack when a key is pressed. Indeed, with the debugger, when we break in `I8042KeyboardInterruptService`, the call stack is the one given in Code 4.2. Because `KiInterruptDispatch` and `KiCallInterruptServiceRoutine` routines are in charge of handling interruptions. This last routine uses information from `KINTERRUPT` structure to know which ISR to call for handling a specific interruption.

```

Breakpoint 0 hit
i8042prt!:
fffff804 '2f096790 488bc4          mov     rax , rsp
4 1: kd> kn
# Child-SP      RetAddr          Call Site
6 00 ffff9d81 '37 dfff38      fffff804 '2a527ef5      i8042prt!I8042KeyboardInterruptService
01 ffff9d81 '37 dfff40      fffff804 '2a5f72af      nt!KiCallInterruptServiceRoutine+0xa5
8 02 ffff9d81 '37 dfff90      fffff804 '2a5f7577      nt!KiInterruptSubDispatch+0x11f
03 fffffee04 'fe829520      fffff804 '2a5f187f      nt!KiInterruptDispatch+0x37
10 04 fffffee04 'fe8296b8      fffff804 '2a5b8c04      nt!HalProcessorIdle+0xf
05 fffffee04 'fe8296c0      fffff804 '2a471396      nt!PpmIdleDefaultExecute+0x14
12 06 fffffee04 'fe8296f0      fffff804 '2a470154      nt!PpmIdleExecuteTransition+0x10c6
07 fffffee04 'fe829af0      fffff804 '2a5f95a4      nt!PoIdle+0x374
14 08 fffffee04 'fe829c60      00000000'00000000      nt!KiIdleLoop+0x54

```

Code 4.2: "Callstack from `I8042KeyboardInterruptService` routine when notified."

An ISR must run at priority Device Interruption Request Level (DIRQL) for the shortest possible interval of time. Because ISR are designed to run at such priority level where strong restrictions happen about what can be performed, they are not supposed to do a lot except managing the current interruption. More directly, ISR dedicates all the heavy job to a *Deferred Procedure Call* (DPC) [526] routine which should hold it as soon as IRQL falls below `DISPATCH_LEVEL` on a processor.

Once `I8042KeyboardInterruptService` has completed its task, it queues a DPC with `KiInsertQueueDpc` routine [527] such as `I8042KeyboardIsrDpc` routine is called inside the `i8042prt.sys` driver. Its goal is to transfer the notification to the next driver. By design, instructions can only be executed in the context of a thread. It means that the interruption must be handled by a thread to be executed. This is done by one of the idle thread that the kernel (hosted by the virtual process "system") makes available for this type of interruption. This thread is waiting for a DPC in `KiExecuteAllDpcs` routine. A typical call stack of routines called when a key is pressed with a PS/2 keyboard is given in Code 4.3.

```

kbdclass!KeyboardClassServiceCallback
2 i8042prt!I8042KeyboardIsrDpc+0x2f9
nt!KiExecuteAllDpcs+0x30a
4 nt!KiRetireDpcList+0x1ef
nt!KiIdleLoop+0x7e

```

Code 4.3: "Windows' routines call stack when a key is pressed on a PS/2 keyboard"

The operating system represents physical devices by device objects used to interact directly with them. One or more device objects are associated with each device and all these objects can be driven by software called

drivers. The set of all drivers for a given device is called the *driver stack* and objects to hierarchically interact with is called *device stack*. The keyboard is a device like any other one. It means it is represented in the system by a *physical device object* (PDO) created by the root bus driver (responsible to administrate the root bus by enumerating devices on it or responding to Plug and Play notifications). Technically, the PDO is created and owned by the bus controller driver that detects and enumerates the device for the PnP manager. This PDO contains the information (for example, bus address) that the bus controller needs to access the device over the bus [528]. The name of this device is `"\Device\000000NN"` where the last "N" stand for a device number allocated by the system. Above the PDO, the main driver for the device create a *function driver object* (FDO). This one represents the internal state of the device. In our case, it is the responsibility of `i8042prt.sys` to create it, attached to the PDO, since it is loaded by the PnP manager for this type of device. The FDO of the keyboard has no specific name under Windows except its driver reference `"\Driver\i8042prt"`. A function driver provides the operational interface for its device, mainly to handle read and write operations to the device and to manage device power policy. From this point, the functional driver still has a link with the device itself in the sense that it is dependent on the type of connection the device uses (PS/2 here).

But Windows is designed to provide an abstraction layer which is a device-independent keyboard support for any application. In other words, it is designed to interact with the hardware regardless the physical hardware. Such a way, `kbdclass.sys` centralizes interactions with any type of keyboard. This one is called a *class driver* since it supports system requirements independently of the hardware requirements of a specific class of device. This generic view of keyboard for the system is represented by the device name `"\Device\KeyboardLegacyClassN"` where "N" stands for the keyboard device number (0 for the first keyboard). In our case, information is directly supplied by `i8042prt.sys` to `kbdclass.sys`. To be accurate, if no driver is registered between `i8042prt.sys` and `kbdclass.sys`, notification is transferred from `I8042KeyboardIsrDpc` to `KeyboardClassServiceCallback` callback routine [529], exported from the `kbdclass.sys` driver to be notified. Otherwise, other lower filter drivers are notified to handle the *I/O Request Packets* (IRP) [530] used to transfer information from one driver to another on the device stack.

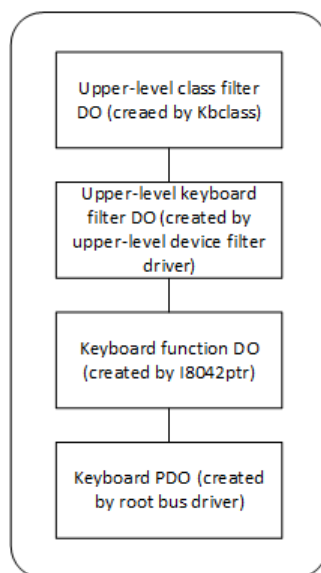


Figure 4.15: PS/2 keyboard device stack.

Optionally, if we need to interact among one of these drivers, we can create an *filter driver* (FD). The goal of a filter driver is to add value to modify the behavior of a given device. We can distinguish three main types of filters: *Bus Filter*, *Lower-Level* and *Upper-Level* [531, 532]. For the sake of simplicity, we can keep in mind that bus-filter are used by Microsoft or OEM to implement proprietary enhancements to standard bus hardware. In the case of lower-level filter, they are dedicated to modify the behavior of device hardware when upper-level filter are typically provided to add value features for a device.

## 4 USB and HID technology

### Key Point 4.6: USB & HID technology

- ☞ Nowadays, keyboard devices use USB protocol to communicate with the host computer.
  - ☞ The USB protocol allows each device to self-describe itself so that the system can identify them.
  - ☞ The Bluetooth technology used by wireless keyboards is similar in its shape to USB.
- ☞ Since keyboard is a *human interface device* (HID), this one can use an extension of USB protocol called HID.
  - ☞ HID is a self-describing protocol that allows devices to be generically handled by software applications.
  - ☞ Every device describes how it will communicate with the host device.
  - ☞ Extensible and robust, an application can talk to any HID device, without necessarily knowing its nature, just by analyzing its *Report Descriptor*.
  - ☞ Communication is performed via *reports* defined at initialization by *report descriptor*.

Since PS/2 keyboard technology remains for retro-compatibility for some types of computers (internal laptops keyboard and Virtual Machine emulation or specific industrial systems), modern hardware are more likely to use keyboard connected via USB cable or wireless technology such as Bluetooth [533]. They are always keyboards managed internally by Windows as regular keyboards, but the way information from device is managed in the system has evolved. It means there are different possibilities to handle keystrokes from a given device from the operating system's point of view.

In order to understand how it works, we propose to explain how USB protocol works [7] for keyboard and especially for modern ones with the use of Human Resource Interface (HID) [534]. It matters to understand the protocol used to understand how the keyboard subsystem works under Windows. Indeed, how could it be possible to protect a system if we do not know how does this one work? Part of information presented here will be used latter to explain how Windows works and how it is possible to interface our solution with it. In addition, it helps to understand how malware could intercept keyboard data and how it would be possible to design a protection system based on this technology. More directly, this is this technology which is used in product such as a *rubber-ducky*<sup>3</sup> (Figure 4.16) to simulate a keyboard. Formally speaking, such device is just a USB stick driven by a micro-controller repeating order through USB protocol. And it is recognized by the operating system thanks to the HID protocol it abuses...



Figure 4.16: Rubber ducky dongle used to emulate a keyboard with pre-recorded sequences of keystrokes. The technologies behind this type of device are direct applications of USB and HID protocols.

This section is more complete than the one about PS/2 keyboards since the technology used here is more

<sup>3</sup><https://shop.hak5.org/products/usb-rubber-ducky-deluxe>



modern and more widely spread. In order to have a detailed knowledge of the subject, we propose to introduce first how USB protocol works and then how HID handles keyboard management.

## 4.1 USB protocol

### 4.1.1 USB introduction

#### Key Point 4.7:

☞ USB protocol is defined through different standards. Today, the version 4.0 is the last version.

USB is often associated with USB sticks or USB hard drives used to store data. But in reality USB — which stands for *Universal Serial Bus* — is a cable bus that supports data exchange between a host computer and a wide range of simultaneously accessible peripherals. There are several generations (which correspond to different norms) of USB that have evolved through the ages. From USB 1.0 [535] started by Compaq, Intel, Microsoft and NEC companies to set up the USB Organization, USB 2.0 [7], USB 3.0 [536], USB 3.1 [537], USB 3.2 [538] and more recently (even if it is not present on the market at the time this text is written) USB 4.0 [539] emerged. While there are always some backwards compatibility between the different versions of the USB standard despite addition of new features and technologies, the main difference known to people is the different speeds allowed with devices using a given standard.

Connectors	USB 1.0 1996	USB 1.1 1998	USB 2.0 2001	USB 2.0 Revised	USB 3.0 2011	USB 3.1 2014	USB 3.2 2017	USB 4.0 2019
Data rate	1.5 Mbit/s	1.5 Mbit/s (Low Speed) 12 Mbit/s (Full Speed)	1.5 Mbit/s (Low Speed) 12 Mbit/s (Full Speed) 480 Mbit/s (High Speed)		5 Gbit/s (SuperSpeed)	10 Gbit/s (SuperSpeed+)	20 Gbit/s (SuperSpeed+)	40 Gbit/s (SuperSpeed+ Thunderbolt 3)

Table 4.2: Speeds of USB devices among different norms.

More than different norms, USB is also known to get different formats of connectors. Indeed, there are the standard format (type-A) principally used by desktop or portable equipment, the mini intended for mobile or embedded equipment (type-B), and the most recent one which is the thinner micro size (type-C) [540] for low-profile mobile equipment such as mobile phones and tablets. Note that with the introduction of USB 4.0 [539], all formats will be deprecated except the type-C which is about to fusion with the Thunderbolt 3 format from Apple devices. To illustrate this historical evolution, we propose to resume the main formats for USB in Figure 4.17.

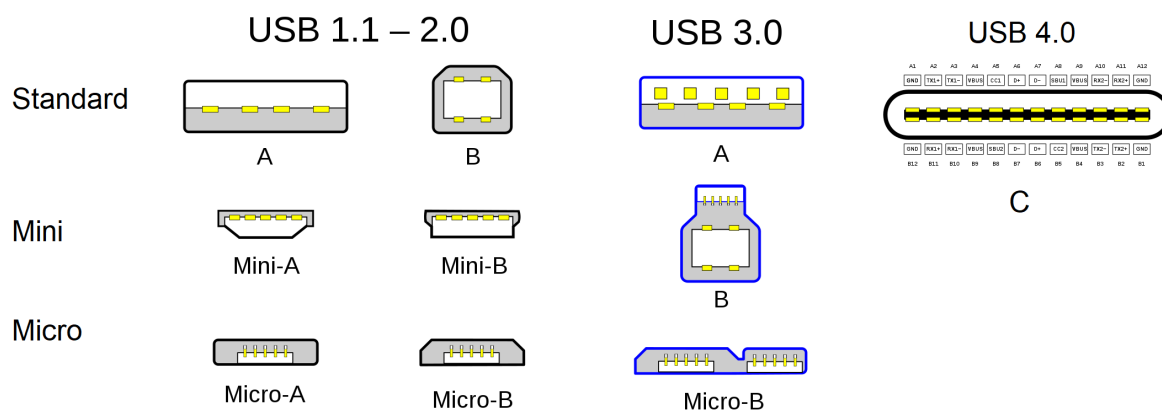


Figure 4.17: Different types of formats used by USB devices.

### 4.1.2 USB standard

#### Key Point 4.8:

- ☞ With USB, communication is performed between several components.
  - ☞ The *host* represents the machine where the USB *device* is connected.
  - ☞ The USB *devices* is called *function* in USB documentation.
  - ☞ Exchanges of information between host and devices are called *transaction*.
  - ☞ Most bus *transactions* involve the transmission of up to three *packets*.
- ☞ With USB, communication is performed between different components.
  - ☞ USB protocol is described with "host-glasses" — input and output directions are referenced from the host point of view.
  - ☞ Host initiates all transactions and different devices communicate together on one single data bus.
  - ☞ The USB is a polled bus. It means that the host controller, on a scheduled basis, sends a USB packet.

Since we are particularly interested in keyboard type peripherals, we have decided to focus on the USB 2.0 standard. This choice is due to the fact that most keyboard devices support this standard or they are at least compatible with it. Moreover, considering the speeds and technologies offered, this standard is more than enough to explain how USB keyboards are designed.

Technically, USB is described by three definition areas: USB interconnect, USB devices and USB host [7]. The USB host is unique to the machine and its USB interface is called Host Controller. The host masters exchanges on the bus since it drives and receives from the USB device. For the sake of clarity, USB is a host centric bus. USB device — also called *function* in USB documentation [7] — corresponds to any device which provides capabilities to the system. To comply with USB standards, devices must have a comprehension of the USB protocol, an ability to respond to standard USB operations, such as configuration and reset with an ability to interact with standard descriptive information. It can be any USB peripherals such as keyboard, mouse, USB stick and others. The USB interconnect is a layer by which USB devices are connected to and communicate with the host. It can be a hub or more generally the USB connector responsible for the interconnection between the host and the device. Communication can be represented schematically as in Figure 4.18.

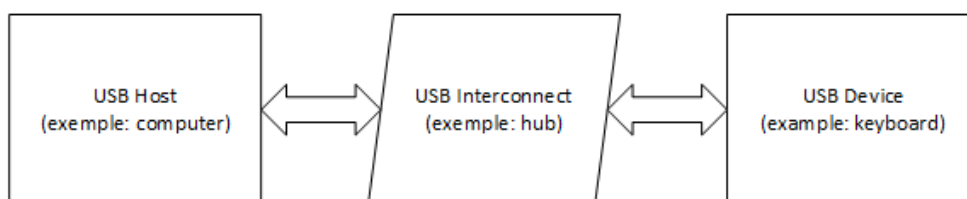


Figure 4.18: USB three definitional areas.

All operations and more directly all data transfers in USB model are initiated by the host. Since the host is central, the USB protocol is described with "host-glasses" which means that input and output directions are referenced from the host point of view. All devices share USB bandwidth so that they can be attached, configured, used, and detached while the host and other peripherals are in operation. The USB physical interconnect is a tiered star topology which means that it is possible to connect to a host several hubs linked together themselves connecting devices. A hub is at the center of each star where each wire equipped with a USB connector is a point-to-point connection between the host and a hub or a device, or a hub connected to another hub or another device. The flexibility of topology allowed by USB explains undeniably part of the successes of its use. A popular representation from [7] of such bus topology is given in Figure 4.19.

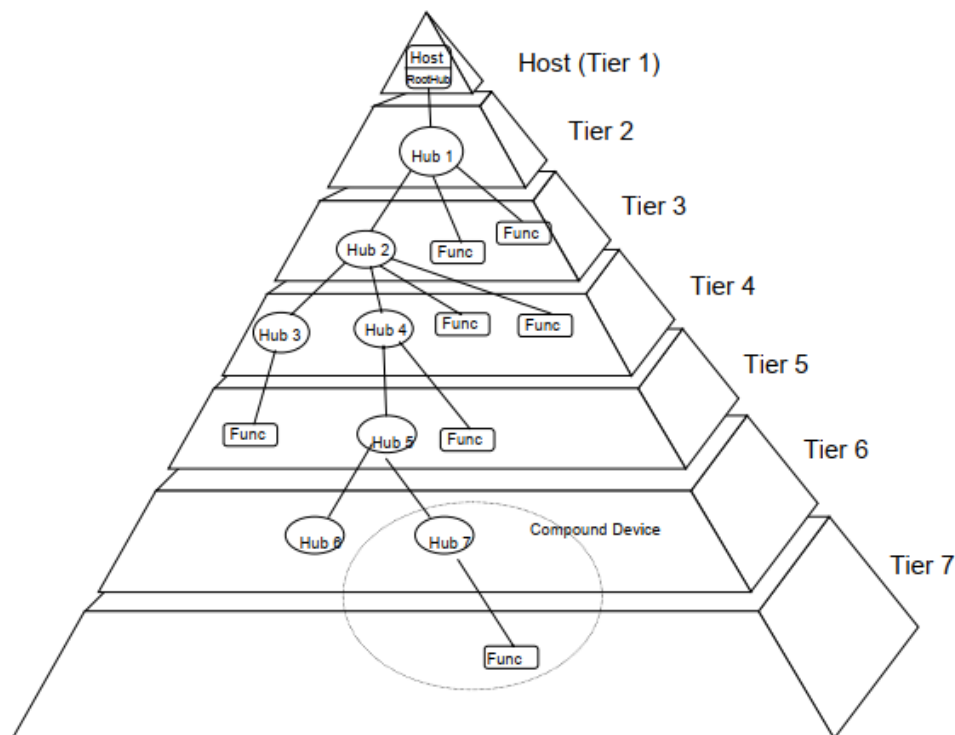


Figure 4.19: USB bus topology.

According to [7], due to timing constraints allowed for hub and cable propagation times, the maximum number of tiers allowed is seven (including the root tier). It means we cannot have more than six hubs linked together. Technically speaking, it could allow a large number of devices connected to all different hubs, which is far enough for everyday use. A good question is to know how all these devices communicate together on one single data bus. The answer comes from the way transactions exchanged on this bus are forged at the host's initiative.

As defined by the USB documentation [7], the host initiates all transactions since USB is a host centric bus. Exchanges must be organized such as each device knows which transaction is addressed to whom and how to respond to control requests. In such a way, it means there is a general shape describing how transactions must be forged to exchange information between host and device. This is done by defining different fields in USB documentation [7] such that information exchanged is properly formatted. On the first hand, data is moved over the bus in little-endian order, which corresponds to least-significant bit (LSB) first. On a second hand, a transaction can be composed by one or more packets holding information to exchange. A packet can be seen as the smallest element of data transmission and a single transaction composed of different packets. USB packets consist of different fields predefined and understandable by all USB devices. Note that if a packet is not big enough to carry all information, this one can be split in many packets.

### 4.1.3 USB packets

#### Key Point 4.9:

- ☞ All transactions in USB are performed through packets.
  - ☞ A *Packet Identifier* (PID) is a value in the packet defining the type of action to be executed with a packet.
  - ☞ Token packets indicate the type of USB transaction to follow.
  - ☞ Data packets contain the payload following a Token packet.
- ☞ There are three main types of token packets with IN, OUT and SETUP:
  - ☞ IN packets are used when host requires to read information from device.
  - ☞ OUT packets are used when host wishes to write to the device.
  - ☞ SETUP packets are used to begin control transfers.
- ☞ To know which device to address on a unique bus, USB devices use *address field* (ADDR) and *endpoint field* (ENDP).
  - ☞ Up to 127 USB devices can be supported per bus with the address field. A given USB device may have many pipes.
  - ☞ There is 16 possible endpoints (called "*pipes*" sometimes) per address. Each endpoint has a specific purpose (IN, OUT, ...).
  - ☞ Like IP technology, we can see USB field's address as the IP address and the endpoint as the port associated.

All packets begin with a synchronization (SYNC) field which corresponds to bits (from 8 to 32 bits, depending the version of the USB device) in order to generate a maximum edge transition density. Technically, a zero bit is inserted after 6 successive 1-bit (this is known as *bit stuffing*) since data is NRZI encoded. It is used by the input circuitry to align incoming data with the local clock. This field is used as a synchronization mechanism and it is not supposed to be shown in packet diagrams since it is purely hardware mechanism. For the sake of sobriety, we can say that SYNC field defined the Start-of-Packet (SOP) which precludes every transaction in USB (the same way, there is an End-of-Packet (EOP) delimiter). The last two bits of the SYNC field indicates where the following PID field starts. But SYNC and EOP do not matter for us since they are managed at hardware level by the USB connector and presented here for the sake of completeness.

The *Packet Identifier* (PID) corresponds to a value defining the action to be executed with this packet. This field is encoded in a 8-bit value but only the first 4-bit are used to encode the code order called in the USB documentation *packet type field* (PID Type). As shown in Figure 4.20, the 4 last bits are a one's complement of the first 4-bit used to check decoding procedure in order to avoid to manage corrupted packets (which are supposed to be ignored if it happens). According to [7], the PID indicates the type of packet and, by inference, the format of the packet and the type of error detection applied to the packet.

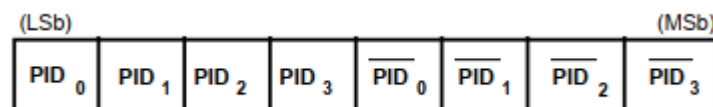


Figure 4.20: PID Format as defined in USB [7]

There are four types of *PID Type* used in different contexts called by *PID Names*. Each combination of PID Type and PID name has a single code composed of 4-bit. Table 4.3 lists all these codes and the description linked to them.

PID Type	PID Name	PID	Description
Token	OUT	0001B	Address + endpoint number in host-to-function transaction
	IN	1001B	Address + endpoint number in function-to-host transaction
	SOF	0101B	Start-of-Frame marker and frame number
	SETUP	1101B	Address + endpoint number in host-to-function transaction for SETUP to a control pipe
Data	DATA0	0011B	Data packet PID even
	DATA1	1011B	Data packet PID odd
	DATA2	0111B	Data packet PID high-speed, high bandwidth isochronous transaction in a microframe
	MDATA	1111B	Data packet PID high-speed for split and high bandwidth isochronous transactions
Handshake	ACK	0010B	Receiver accepts error-free data packet
	NAK	1010B	Receiving device cannot accept data or transmitting device cannot send data
	STALL	1110B	Endpoint is halted or a control pipe request is not supported
	NYET	0110B	No response yet from receiver
Special	PRE	1100B	(Token) Host-issued preamble. Enables downstream bus traffic to low-speed devices.
	ERR	1100B	(Handshake) Split Transaction Error Handshake (reuses PRE value)
	SPLIT	1000B	(Token) High-speed Split Transaction Token
	PING	0100B	(Token) High-speed flow control probe for a bulk/control endpoint
	Reserved	0000B	Reserved PID

Table 4.3: PID Types extracted from USB documentation.

Each USB type has a specific role. Token packets indicate the type of transaction to follow while data packets contain the payload. Handshake packets are used for acknowledging data or reporting errors and start of frame packets to indicate the start of a new frame, but they are beyond the scope of this chapter. Starting by token packets, there are for us three main types of token packets with IN, OUT and SETUP. IN packets are used when the host wishes to read information from device. OUT is the opposite, it means that the host wishes to write to the device. SETUP is used to begin control transfers.

The data field may range from 0 to 1,024 bytes and must be an integral number of bytes. Technically, there are two types of data packets (DATA0 and DATA1), each capable of transmitting up to the maximum data payload size. This one depends on USB version (8 bytes for low-speed devices, 1023 for full-speed devices and 1024 for high-speed devices). With the rise of the high speed mode norm, it is possible to use two another data PIDs (DATA2 and MDATA) which are used for specific types of transaction modes.

To know which device to address on a unique bus, USB uses address field. Device's address (ADDR) field specifies if a given device, via its address, is either the source or destination of a data packet, depending on the value of the token PID (IN, OUT or SETUP for the most relevant ones). The address field is encoded on 7-bit which represent 127 possibilities of devices to be supported. This maximum number of devices is large enough to support the needs of any host. Note that address zero is not a valid address to be assigned to a device. Indeed, it corresponds to the default address, as any device which is not yet assigned to an address must be able to respond to packets sent to address zero.

Technically, devices are addressed using two fields: the address field and the endpoint field. A device needs to fully decode both address and endpoint fields to respond correctly. The endpoint (ENDP) field is made up of 4-bit values, allowing 16 possible endpoints. It allows more flexible addressing of devices since it provides for one address different endpoints with whom to communicate. To use an analogy with IP technology, we can see USB field's address as the IP address of a network and the endpoint as the port linked to this address. When reading USB documentation, endpoints may be called "*pipes*" sometime. Actually, USB communication is based on pipes where a pipe is a connection between the host software and a USB device endpoint. For the sake of simplicity, the two terms are sometimes used interchangeably even if formally speaking, pipes correspond to logical channels while endpoints correspond to real buffer in the device's memory where information is stored before transfer.

Except for endpoint zero, endpoint numbers are device-specific and therefore undefined. They can be used for IN, OUT or SETUP token operations (among others) and all devices must support a control pipe at endpoint number zero (the Default Control Pipe). The maximum number of endpoints depends on the USB version used by the device (low-speed devices support a maximum of three pipes per function while full-speed and high-speed

devices may support up to a maximum of 16 IN and OUT endpoints).

There are other fields in USB definition such as Cyclic redundancy checks (CRCs) used to protect all non-PID fields in token and data packets or frame number field used to manage micro-frames. But may be with the exception of End of packet (EOP) field which defines the end of a packet, those ones are irrelevant in the context of understand how USB keyboard technology works.

The Figure 4.21 represents the general shape of a token packet followed by a data packet. For instance, to transfer data from the host to the device, the host uses a token packet indicating OUT operation. In this packet, the address of the targeted device and the endpoint where to store the content of the buffer in the device is referenced. Then, the host broadcasts data packets holding the content of data to transfer from the host to the device. Such a way, it is possible to use a single endpoint as a bi-directional pipe.

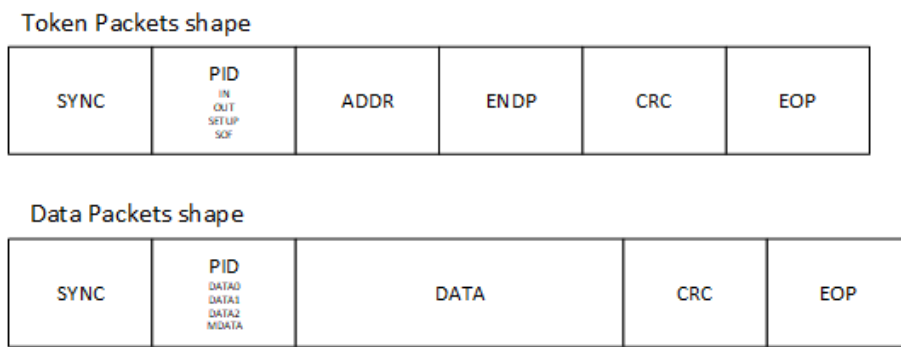


Figure 4.21: Shape of USB packets.

Technically, a successful transaction is a sequence of three packets which performs a simple but secure data transfer(IN/OUT, DATA, EOP). From what has been exposed, it is possible to represent the USB architecture with different peripherals via the Figure 4.22.

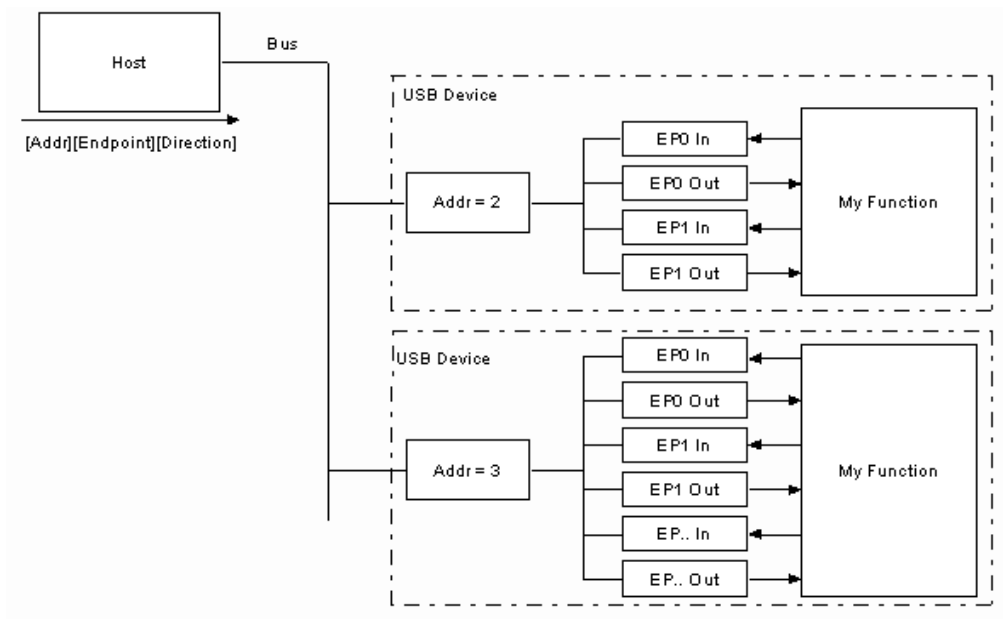


Figure 4.22: Scheme of USB host and devices interactions.

#### 4.1.4 USB pipes and communication

##### Key Point 4.10:

- ☞ In USB, *pipes* are used to move data the host and the endpoint on a device.
  - ☞ A pipe has a transfer type, bandwidth usage and associated endpoint's characteristics (maximum data payload sizes, for instance).
- ☞ Keyboards are likely to use *interrupt data transfers* type.
  - ☞ Despite its name, device does not interrupt and notify the host directly.
  - ☞ USB is host polled and driven by an *I/O Request Packets* (IRPs) which is pending for the interrupt endpoint.
  - ☞ Once an IRP is pending, the host polls the endpoint to know if there is something to transfer.
  - ☞ The polling is performed at a period from 1 ms to 255 ms in case of full-speed endpoint (common for keyboard devices).
  - ☞ An interrupt pipe is always uni-directional (only IN or OUT but not both).

The USB supports functional data and control exchange between the USB host and a USB device as a set of either uni-directional or bi-directional pipes. A given USB device may have many pipes such as one endpoint for input and another endpoint for output. However, it is still necessary to define how to transfer data. There are four different ways to transfer data on a USB bus.

- Control Transfers: used to configure a device at attach time and can be used for other device-specific purposes, including the control of other pipes on the device. This one uses endpoint 0 OUT and endpoint 0 IN whatever the USB device is.
- Bulk Data Transfers: generated or consumed in relatively large and bursty quantities and have wide dynamic latitude in transmission constraints. It is generally used with large amounts of data with error-free delivery, but with no guarantee of bandwidth.
- Interrupt Data Transfers: used for timely but reliable delivery of data, for example, characters or coordinates with human-perceptible echo or feedback response characteristics. Despite its name, interrupt transfers have nothing to do with interrupts since USB is a polling system. The name of *interrupt* is not accurate but it represents a situation where an interrupt would have been used in earlier connection types. Interrupt data typically consists of event notification, such as characters from a keyboard. It is always an uni-directional type.
- Isochronous Data Transfers: occupy a prenegotiated amount of USB bandwidth with a prenegotiated delivery latency (also called streaming real time transfers). If Isochronous Transfers have a guaranteed bandwidth they are prone to deal with erroneous data (missing or corrupted). This is perfect for stream data, such as video or audio stream, where it matters to maintain the data flow but not a big deal if there are few lacks or corruptions.

The data payloads offered by each of these transfer methods is below or equal to the maximum packet size. But it can vary from USB versions and transfer methods (interrupt data transfer, which is mostly used by keyboards, has a data payload coming from 8 bytes to 1024 bytes)... In addition, there are two mutually exclusive pipe communication modes: stream and message. Stream mode means data which is moving has no USB-defined structure. At the opposite, message mode means data which is moving has some USB-defined structure. For instance, the default the Default Control Pipe is always a message pipe.

A software client normally requests data transfers via I/O Request Packets (IRPs) [541] to a pipe and then either waits or is notified when they are completed. IRP refers in USB documentation to any identifiable request by a software client to move data between itself and an endpoint of a device in any direction. IRP offers the



possibility to notify the software client when a bus transaction has been completed, with success or not. The only time bus activity is present for a pipe is when IRPs are pending for that pipe, otherwise the pipe is idle and the host controller will take no action for that pipe. Furthermore, an IRP may require multiple data payloads to move the client data over the bus. In the case we should deal with multiple data payload IRP, the data payloads of each request is expected to be of the maximum packet size until the last data payload that contains the remainder of the overall IRP is sent. IRP allows the operating system to *poll* the USB bus by arming a request each time it makes sense to interact with the USB device.

Since keyboards are most likely to use interrupt data transfers, it must be clearly understood that this is the host controller which engages the current IRP. This is not the device which directly notifies the host as a regular interrupt mechanism would have done it. The endpoint is only polled when the host has an IRP for an interrupt transfer pending. This is once the IRP is pending that the host polls the device in order to know if there is something to transfer. The polling is performed at a scheduled period. An endpoint for an interrupt pipe specifies its desired bus access period (at configuration time). And the host uses this information during configuration to determine a period that can be sustained. A full-speed endpoint can specify a desired period from 1 ms to 255 ms (255 ms is the maximum limit from the host point of view), which is far enough for keyboard purposes.

According to USB documentation [7], without an IRP pending, if the bus time for performing an interrupt transfer arrives, the endpoint will not transfer data at that time. Indeed, if there is no IRP, the pipe is idle and the host controller should take no action with regard to the pipe. This is why the host must continuously engage an IRP to be notified by an interrupt. In addition, we must understand that the host has no way to determine whether an endpoint has something to transfer except without accessing this endpoint and requesting an interrupt transfer. If the device has nothing to transfer when requested, it responds with NAK (*Not-Acknowledge*) and it does not provide interrupt data to avoid the host to erroneously consider the notified IRP as complete.

Since we have defined how communication is managed with USB, let us see what is happening when a device is plugged to the host. From USB standard, USB devices may be attached and removed at any time. It means when a device is attached to a USB bus (via a hub or any USB interconnect means), this one reports the attachment, removal or any change in the state of the port to the host. This report is performed at electrical level. The host enables the hub port where the device is attached upon detection of an attachment. Then, the host initiates a RESET command to start the device in a known state on address zero. Since the device has no address assigned yet, this one responds to the default address which is zero. Note that until the reset has been correctly performed by the device, the host will prevent any transaction to the USB port so that it only resets one device at a time. It prevents two devices to answer simultaneously to a request at address zero.

In USB documentation [7], address assignment could be directly performed once the reset operation has been performed. But this is not the way Windows operates. Indeed, to interact efficiently with the device, the operating system needs to know the maximum packet size used by the device. This operation is driven by a GetDescriptor command sent by the host to the device. Which leads us to discuss how an operating system is interacting with a device.

---

### 4.1.5 USB and host configuration

#### Key Point 4.11:

- ☞ To configure a USB device, the host retrieves information from the device called *descriptors*.
  - ☞ There are several types of descriptors linked to each other by a hierarchical system of dependencies.
  - ☞ Each piece of information in a descriptor is called an *attribute*.
- ☞ The root of all descriptors is called *device descriptor* (only one per device).
  - ☞ It provides USB version number, the type of device (class, subclass), vendor id and product id.
- ☞ There are many *configuration descriptors* for one device descriptor.
  - ☞ Usually only one configuration, but could be more in the case of power management (self or bus powered).
- ☞ Each configuration descriptor has many *interface descriptors* (one per function in the device).
  - ☞ For instance, a USB phone would include a vocal stream and a digital keyboard.
  - ☞ Each function is referenced by an interface class or subclass value.
  - ☞ Each function can use more than one endpoints.
- ☞ *Endpoint descriptors* define how to communicate with a function.
  - ☞ It provides bandwidth requirements, the direction (IN/OUT), transfer type and maximum packet size.
  - ☞ Optional, if no endpoint is provided, the host uses the default endpoint zero.

This is the host's responsibility to configure a USB device. This is done by the host which typically requests configuration information from the device. USB devices report their configuration information (called *attributes*) using descriptors. A descriptor is a data structure with a defined format, usually starting by a byte-wide field that contains the total number of bytes in the descriptor and followed by a byte-wide field that identifies the descriptor type. The design provided by the descriptors resembles individual data records in a relational database organized in a hierarchy of descriptors. This one is represented in Figure 4.23 where each of these descriptors and their use will be described in the following.

The root of all descriptors is the *device descriptor*. There is only one device descriptor per device and it describes general information that applies globally to the USB device. It specifies some basics but highly relevant information about the device such as the supported USB version, maximum packet size, vendor and product IDs and the number of possible configurations the device can have. The structure of the descriptor device is given in table 4.4 and referenced as a C structure in Windows operating system [542].

Note that the device descriptor references indexes in string descriptor. Even if it is optional, it is possible, when it is appropriate, to define a descriptors that contains references to string descriptors that provide displayable information describing a descriptor in human-readable form. If the index referencing a string is equal to zero, it means that no string has been provided for this attribute.

Among relevant information in the device descriptor, we have `bcdUSB` field which provides the USB version used by the device. Version number is encoded on two bytes as Binary-Coded Decimal format (i.e., 2.10 is 0x0210 and 3.0 is 0x0300). Then, we have `bDeviceClass`, `bDeviceSubClass` and `bDeviceProtocol` used to identify the type of the device. But these fields are not used a lot since a lot of device manufacturers prefer to identify the device at the interface level. Codes used in device descriptor and interface descriptor are the same. Indeed,

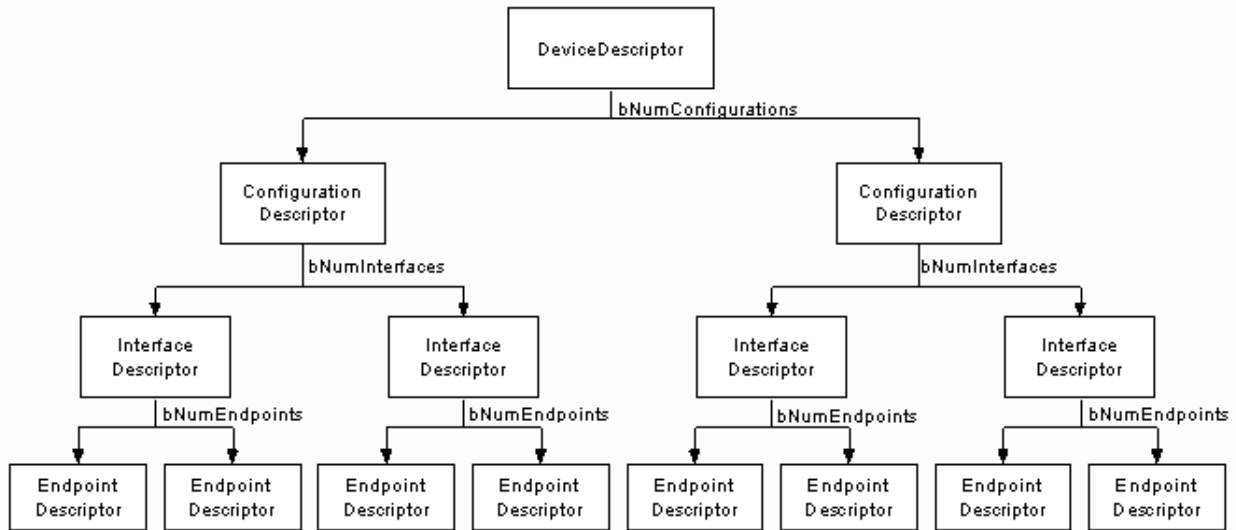


Figure 4.23: USB descriptors hierarchy.

it allows to support multiple classes of devices in a single device, called *composite devices*. `bMaxPacketSize` reports the maximum packet size for endpoint zero (supported by devices). Fields such as `idVendor`, `idProduct` and `bcdDevice` allow to identify a vendor of a product and the serial number of the device if one is provided. Finally, field `bNumConfigurations` provides the number of configuration descriptors supported by the device.

If the device descriptor is unique, there may be more than one configuration descriptor. Indeed, the number of configurations supported by a device is given by `bNumConfigurations` field from the device descriptor. The full content of the configuration descriptor structure [543] is given in table 4.5.

Even if most devices usually have only a single configuration, a device may support different configurations for different reasons. The first reason deals with the power management of the device. There are two possibilities of power management for a device: self or bus powered. For instance, a device may have two energy sources. One directly via the host where the USB device is connected (*bus-powered*) or via an external power supply linked to a power socket (*self-powered*). The device driver may choose to enable the bus powered configuration when the device is not connected to the power socket or if the use does not need to use it. It can be done with the use of *set configuration request* with the value of the configuration given by `bConfigurationValue` field. In the case where the device would be disconnected from its external power source, it tries to continue to operate as best as it can. If it cannot, the device is supposed to fail operations it is no longer able to support. Software using the device may determine the cause of the failure by checking the status of the device's power source.

The configuration descriptor goes beyond power differences. It describes the number of *interfaces* provided by the configuration. According to USB documentation [7], an interface is a related set of endpoints that present a single feature or function of the device to the host. The protocol used to communicate with this related set of endpoints and the purpose of each endpoint within the interface may be specified as part of a device class or vendor-specific definition. Typically, there is a one-to-one correlation between a function provided by a device and an interface. Topically, an interface can have a number of endpoints where each represent a functional unit belonging to a particular class. For instance, we can have a VOIP phone where communications use to endpoints for transferring audio in each direction. In addition, such a phone can have a digital keyboard requiring another interface with a single IN interrupt endpoint. All these interfaces using different endpoints are represented under a single configuration. More than one interface can be active at a time.

The structure [544] used to represent a USB descriptor interface is given in table 4.6. An interface descriptor is returned as part of a configuration descriptor and it cannot be required directly via a *get descriptor request*. Note that the interface descriptor is optional and a device without any would return a Request Error. But such

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	DEVICE Descriptor Type
2	bcdUSB	2	BCD	USB Specification Release Number in Binary-Coded Decimal (i.e., 2.10 is 210H). This field identifies the release of the USB Specification with which the device and its descriptors are compliant.
4	bDeviceClass	1	Class	Class code (assigned by the USB-IF). - 00h means each interface defines its own class information and the various interfaces operate independently. - From 01h to FEh means the device supports different class specifications on different interfaces where they may not operate independently. - FFh means vendor-defined class. - Any other value must be a class code.
5	bDeviceSubClass	1	SubClass	Subclass code assigned by the USB-IF. These codes are qualified by the value of the bDeviceClass field.
6	bDeviceProtocol	1	Protocol	Protocol code assigned by the USB-IF. These codes are qualified by the value of the bDeviceClass and the bDeviceSubClass fields.
7	bMaxPacketSize0	1	Number	Maximum packet size for endpoint zero (only 8, 16, 32, or 64 are valid)
8	idVendor	2	ID	Vendor ID (assigned by the USB-IF)
10	idProduct	2	ID	Product ID (assigned by the manufacturer)
12	bcdDevice	2	BCD	Device release number in binary-coded decimal
14	iManufacturer	1	Index	Index of string descriptor describing manufacturer
15	iProduct	1	Index	Index of string descriptor describing product
16	iSerialNumber	1	Index	Index of string descriptor describing the device's serial number
17	bNumConfigurations	1	Number	Number of possible configurations

Table 4.4: Standard Device Descriptor.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	CONFIGURATION Descriptor Type
2	wTotalLength	2	Number	Total length of data returned for this configuration. It includes the combined length of all descriptors (configuration, interface, endpoint, and class- or vendor-specific) returned for this configuration.
4	bNumInterfaces	1	Number	Number of interfaces supported by this configuration
5	bConfigurationValue	1	Number	Value to use as an argument to the <code>SetConfiguration</code> request to select this configuration
6	iConfiguration	1	Index	Index of string descriptor describing this configuration
7	bmAttributes	1	Bitmap	Configuration characteristics: - D7: Reserved (set to one) - D6: Self-powered - D5: Remote Wakeup - D4...0: Reserved (reset to zero)
8	bMaxPower	1	mA	Maximum power consumption of the USB device from the bus in this specific configuration when the device is fully operational. Expressed in 2 mA units (i.e., 50 = 100mA).

Table 4.5: USB Configuration Descriptor.

a situation is far from being common.

In order to get adaptive device drivers, USB provides information about the type of functionality supported by a device through its interface descriptors. Via `bInterfaceClass`, `bInterfaceSubClass` and `bInterfaceProtocol` fields, it is possible to identify the function(s) provided by a USB device and the protocols used to communicate with. The values [545] used as class code have been assigned to a group of related devices that has been characterized as a part of a USB Class Specification [546]. A class of devices may be further subdivided into subclasses. In addition, a protocol code may define how the host software communicates with the device.

If the selected configuration is not supposed to be changed during the use of an USB device (otherwise, it would require a reset of the device [7]), one interface may include alternate settings. It means that for a given interface, we have the possibility to vary characteristics of endpoints after the device has been configured. It allows a portion of the device configuration to be updated while other interfaces remain in operation. If a configuration has alternate settings for one or more of its interfaces, for each setting, a separate interface descriptor keeping the value of `bInterfaceNumber` field but using a different `bAlternateSetting` is providing. Let us illustrate the case where a device configuration supports a single interface with two alternate settings. In such

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	INTERFACE Descriptor Type
2	bInterfaceNumber	1	Number	Number of this interface.
3	bAlternateSetting	1	Number	Value used to select this alternate setting for the interface identified in the prior field
4	bNumEndpoints	1	Number	Number of endpoints used by this interface.
5	bInterfaceClass	1	Class	Class code (assigned by the USB-IF).
6	bInterfaceSubClass	1	SubClass	Subclass code (assigned by the USB-IF). These codes are qualified by the value of the bInterfaceClass field.
7	bInterfaceProtocol	1	Protocol	Protocol code (assigned by the USB). These codes are qualified by the value of the bInterfaceClass and the bInterfaceSubClass fields.
8	iInterface	1	Index	Index of string descriptor describing this interface.

Table 4.6: USB Interface Descriptor.

a case, the configuration descriptor would be followed by an interface descriptor with both `bInterfaceNumber` and `bAlternateSetting` equal to zero. Then the endpoint descriptors for that setting. After endpoint descriptors, it comes the second interface descriptor with `bInterfaceNumber` equals to zero but `bAlternateSetting` sets to one. After this alternate interface descriptor comes the endpoint descriptors for that setting. The resume of this configuration is given in Figure 4.24 inspired from [547].

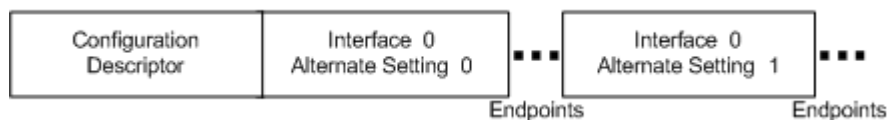


Figure 4.24: Illustration of a configuration descriptor with a single interface but two alternate settings.

To select an alternate setting of an interface, the `SetInterface` request can be used with the alternate setting selected. The `GetInterface` request returns the selected alternate setting. By default, the alternate setting is the one referenced with the number zero.

There is interface descriptor which does not provide endpoint descriptor. In this case, the `bNumEndpoints` field is set to zero. It does not mean that the interface does not communicate but this interface uses the default endpoint zero. Such lack of endpoint descriptor happens since an interface descriptor never includes endpoint zero as the endpoint number to be used.

Each endpoint used for an interface has its own descriptor. This descriptor contains the information required by the host to determine many information such as the bandwidth requirements, the direction (IN/OUT), transfer type and maximum packet size. The structure used is given in table 4.7.

An endpoint is not shared among different interfaces within a single configuration. It means that an endpoint is exclusive to its interface. But a single interface can have several alternate settings which may use the same endpoints. Note that endpoints may be shared among interfaces that are part of different configurations.

To sum up, extracted from the Microsoft documentation [548], we have in Figure 4.25 a representation of a single interface device. This one illustrates how the host can interact with the device. First with the *Default Control Pipe* at endpoint zero and then, after reading the configuration of the device, with the other endpoints described. Note that knowing each endpoint definition is enough for the operating system to deal with the device. For instance, with a keyboard, looking for the endpoint where direction is IN and transfer type is interruption is enough for guessing that we are dealing with the endpoint responsible to provide to the host the key stroke.

For interested readers, there are multiple-interfaces device representation in [548]. It illustrates how all the different descriptor structures are organized in memory when there are multiple interfaces with alternate settings. For short, it is still the configuration descriptor followed by different alternate setting interfaces each

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	ENDPOINT Descriptor Type
2	bEndpointAddress	1	Endpoint	The address of the endpoint on the USB device described by this descriptor. - Bit 3...0: The endpoint number - Bit 6...4: Reserved, reset to zero - Bit 7: Direction, ignored for control endpoints - 0 = OUT endpoint - 1 = IN endpoint
3	bmAttributes	1	Bitmap	Bit 1..0 Transfer Type - 00 = Control - 01 = Isochronous - 10 = Bulk - 11 = Interrupt  If it is isochronous, bits are defined as follows (0 otherwise):  Bit 3..2 Synchronisation Type - 00 = No Synchronisation - 01 = Asynchronous - 10 = Adaptive - 11 = Synchronous  Bit 5..4 Usage Type - 00 = Data endpoint - 01 = Feedback endpoint - 10 = Implicit feedback Data endpoint - 11 = Reserved  Bit 7..6 Reserved and set to 0
4	wMaxPacketSize	2	Number	Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected.
6	bInterval	1	Number	Interval for polling endpoint for data transfers. Expressed in frames or microframes depending on the device operating speed (i.e., either 1ms or 125 $\mu$ s units).

Table 4.7: USB Endpoint Descriptor.

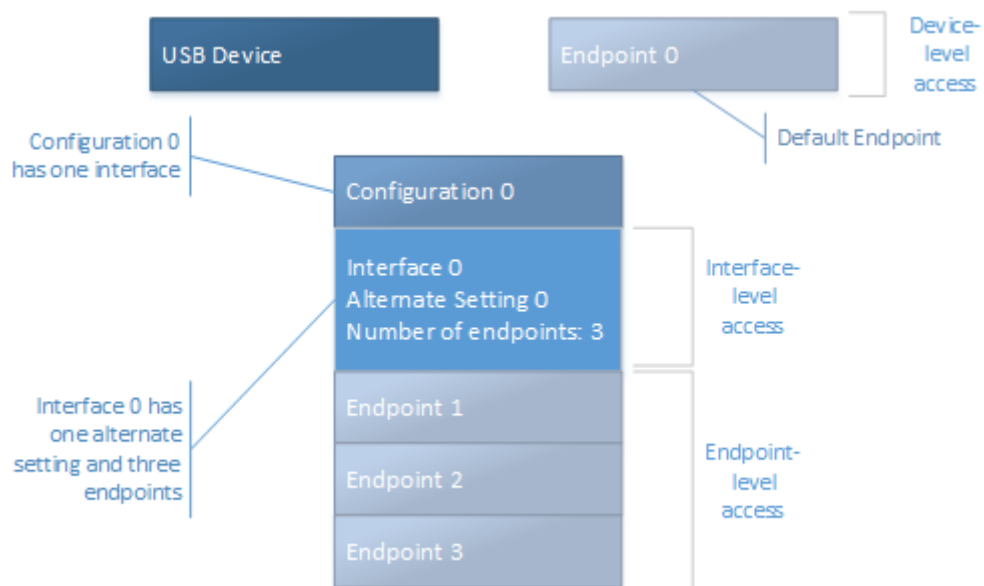


Figure 4.25: Illustration of a configuration descriptor with a single interface and three endpoints.

describing their endpoints if they are not using endpoint zero.

Since we have defined all information from the USB device, it is possible to send standard requests to that device. By default, all requests from the host are managed on the device's Default Control Pipe (endpoint 0). The request and the request's parameters are sent to the device in the Setup packet. Every Setup packet has eight bytes following the structure provided in table 4.8.

Offset	Field	Size	Value	Description
0	bmRequestType	1	Bitmap	Characteristics of request: D7: Data transfer direction - 0 = Host-to-device - 1 = Device-to-host D6...5: Type - 0 = Standard - 1 = Class - 2 = Vendor - 3 = Reserved D4...0: Recipient - 0 = Device - 1 = Interface - 2 = Endpoint - 3 = Other - 4...31 = Reserved
1	bRequest	1	Value	Specific request
2	wValue	2	Value	Word-sized field that varies according to request
4	wIndex	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	wLength	2	Count	Number of bytes to transfer if there is a Data stage

Table 4.8: Format of Setup Data.

This `bmRequestType` field identifies the characteristics of the specific request. This value provides the direction of the `SETUP` request, the type and the recipient targeted by that request (host or device). The recipient can be the device itself, an interface, an endpoint or anything else which is defined by the device's manufacturer. But as explained, only the device is unique and there are many configurations, interfaces and endpoints. To select one of these element, a request must use the `wIndex` field with the id of the target. The `wLength` field specifies the length of the data transferred after the `SETUP` request has been received. On an input request, a device must never return more data than what is indicated in the request. Upon output request, `wLength` always indicates the size of data sent by the host. If the `wLength` field is zero, no data is transferred.

The `bRequest` defines the order associated with the `SETUP` request. Indeed, there are different types or command able to change the meaning of the `bmRequestType` field. Table 4.9 describes the standard device requests defined for all USB devices.

This is the order defined in table 4.9 that defines specific requests. For sake of simplicity, we use in the document requests such as `GetDescriptor`, `SetDescriptor` and so on as a writing facility for a `SETUP` command with the request type (`bmRequestType` field) `GET_DESCRIPTOR` or `SET_DESCRIPTOR`... Note that `GetDescriptor` request is a bit special since it uses the `wValue` field to specify the descriptor type and the descriptor index. Both are encoded with 4-bit values stored respectively in high and low part of the byte. The list of descriptor type codes is given in table 4.10.

Finally, right after the `RESET` code received by the device when this one is electronically recognized by the host, there is usually a `GetDescriptor(Device)` request. This one is used to retrieve maximum packet length in use by the control endpoint stored in the device descriptor. According to [549], when the host is Windows, this first request is performed with the required length `wLength` set to 64. The host is then taking care of one packet in input from the device and ignoring others. This is due to the fact that max packet length value is stored at the 8th byte of the device descriptor and it is enough to know that information to deal with all future control transfers. Then it follows a second `RESET` request. This one is probably present to guarantee that the device does not get confused since the previous transmission did not finished to transfer all the content of the device descriptor.



bmRequestType	bRequest (code)	wValue	wIndex	wLength	Data
0000000B 0000001B 0000010B	CLEAR_FEATURE (1)	Feature Selector	Zero Interface Endpoint	Zero	None
1000000B	GET_CONFIGURATION (8)	Zero	Zero	One	Configuration Value
1000000B	GET_DESCRIPTOR (6)	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
1000001B	GET_INTERFACE (10)	Zero	Interface	One	Alternate Interface
1000000B 1000001B 1000010B	GET_STATUS (0)	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status
0000000B	SET_ADDRESS (5)	Device Address	Zero	Zero	None
0000000B	SET_CONFIGURATION (9)	Configuration Value	Zero	Zero	None
0000000B	SET_DESCRIPTOR (7)	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
0000000B 0000001B 0000010B	SET_FEATURE (3)	Feature Selector	Zero Interface Endpoint	Zero	None
0000001B	SET_INTERFACE (11)	Alternate Setting	Interface	Zero	None
1000010B	SYNCH_FRAME (11)	Zero	Endpoint	Two	Frame Number

Table 4.9: Standard Device Requests codes.

Descriptor Types	Value
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER	8

Table 4.10: Descriptor Types codes.

#### 4.1.6 USB initialization in the system

##### Key Point 4.12:

- ☞ The assignment of an address is done with an `SetAddress` request from the host that assigns an available address to the device.
- ☞ From this moment, the device must respond on this address.
- ☞ The host asks about the available interfaces and chooses one according to its preferences.

Since the packet size of endpoint zero is known, regular transactions with the device can start. The host sends a `SetAddress` request to the device on endpoint zero with one address value available on the host. This is a very simple `SETUP` request where `wValue` field holds the address to be used by the device. This address will be used by the device for all the time it is connected with the host or until it receives a `RESET` request — the address is still valid even if the device is suspended to continue to work after wake-up. When receiving a `SetAddress` request, the device must be able to complete processing of the request within 50 ms and change its address to a new one in less than 2 ms.

Then, the host is going to send `GetConfiguration` requests and in some cases a `SetConfiguration` request to select one which is not the default. Such a choice is made by the host software. Eventually, `GetInterface` requests

followed by `SetInterface` requests could be performed to configure the device as expected by the host's driver software. A nice work with a step-by-step illustration of a USB initialization exchanges under Windows are given at [550, 551].

Once a device has been correctly configured, it is allowed to respond to other transfer types than Control Transfers. Thanks to information in the different descriptors, the host knows what particular transfers on which particular endpoints the device is prepared to support. The host can transfer data from or to the device via IRPs. There may now also be new classes or device's manufacturer requests which may now be supported on the control endpoint in addition to the standard requests. It is all these additional transfers which perform the functionality that the device was designed for.

#### 4.1.7 USB protocol and Windows

Since we are focusing on Windows operating system, it is possible to directly interact with USB thanks to Windows' API at different levels. There is a fully documented API [552] for writing drivers and interacting as close as possible to the device with a USB kernel driver. It is possible to interact with device [553], configuration [554], interface [555] or string [556] descriptors [557, 558]. It is possible to initiate control transfers [551] or USB requests [559] (URBs) directly from kernel to USB devices. From the application API [560], it is possible to interact directly with USB with a large possibilities of action. The most famous way of doing it is via WinUSB API [561]. Architecture of WinUsb is reported in Figure 4.26 extracted from [562]. This one is very interesting since it explains how Windows is interacting with USB devices.

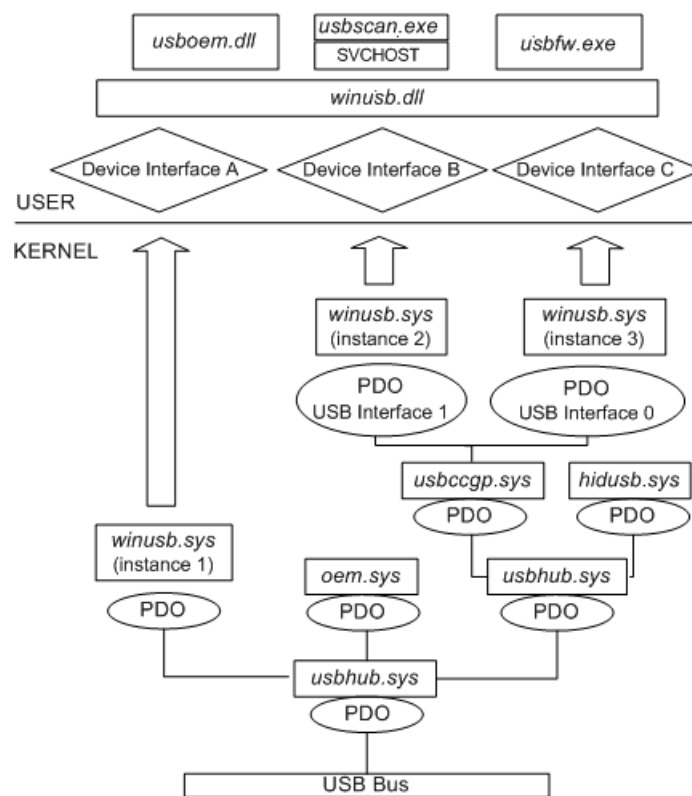


Figure 4.26: WinUsb architecture on Windows Operating System.

The USB bus is driven by `usbhub.sys` (`usbhub3.sys` for USB 3.0 version) driver which is responsible to hold all different USB devices at the lowest level. In case of input, it transfers the IRP from the USB bus to higher instances or third-party drivers used for different purposes. One relevant driver is `usbhid.sys` which is responsible to handle the USB Human Interface Devices norm. Then the content of the IRP is given to `winusb.sys` driver which is responsible for the interface with user-mode applications. The interface with the application API is

performed via `winusb.dll` linked to applications or services to interact with USB. Output interaction follows the same path, from application (or driver) to USB bus. Examples of codes and description about how to access USB devices is given in [563].

For the sake of completeness, it is possible to quickly explain how to interface at any level of the USB driver call stack (given in Figure 4.26). Technically, Microsoft provides in-box *USB device class drivers* for several USB Device classes (defined by the USB-IF<sup>4</sup> with class, subclass, and protocol codes, as explained in device and interface descriptors) [564]. According to Windows documentation [565], if a device that belongs to a supported device class is connected to a system, Windows automatically loads the class driver and the device functions with no additional driver required. It is only in the case where some of the device's capabilities are not implemented by the class driver that device manufacturer should provide supplementary drivers that work in conjunction with the class driver. To perform such an action, Windows categorizes devices by device setup classes, which indicate the functionality of the device. Each of this device setup class has a unique GUID number which can be used in the `.inf` file linked with the driver supplied by the device manufacturer. This `.inf` file is a sort of old script file that contains all the information that device installation components need to install a driver [566, 567]. The complete list of GUID numbers and associated USB Device classes is given in [564] and more generally in [568]. With this information, it is possible to record a driver on a particular USB driver stack [569] (for instance, on every USB audio device or every USB mass storage devices).

In Figure 4.26, there is a driver called `Usbccgp.sys` which has a singular role [570]. This driver is provided by Microsoft to deal with *composite devices*. *Composite devices* correspond to USB devices which expose multiple USB interfaces. From Windows 98 version, Microsoft provides `Usbhub.sys` driver as a generic parent facility in the USB bus driver that exposes each interface of the composite device as a separate device. From Windows XP and Windows Me, this facility is managed by an independent driver called *USB generic parent driver* (`Usbccgp.sys`). Thanks to this driver, device vendors can rely on it to support different USB interfaces automatically. Note that this is this driver which is in charge to manage different USB configurations. Prior to Windows Vista, Microsoft-supplied drivers only select configuration 1 [571]. In Windows Vista and the later versions of Windows, it is possible to select the configuration of a device thanks to a registry key or with specific code [572]. Within a configuration, interfaces (or interface collections) are managed independently to provide different functionalities.

Multiple interfaces from a composite device can behave independently. Microsoft documentation [570] takes example of a composite USB keyboard with regular buttons which might have one interface for the keyboard and another interface for the power buttons on this keyboard. In such a case, the USB generic parent driver enumerates each of these interfaces as a separate device, providing to each interface a single *physical device object* (PDO) [569, 573, 574]. Then, since there are two functionalities, the operating system must handle each of them. This is why it loads the Microsoft-supplied keyboard driver to manage the keyboard interface and the Microsoft-supplied power keys driver to manage the power keys interface. In the case where the composite device has an interface that is not supported by native Windows drivers, it is the device manufacturer indispensability to provide a driver for these unsupported interfaces. This is done thanks to the INF file of this driver. Note that it is written in the documentation [570] that the provided INF file should have an INF `DDInstall` section that matches the device ID of interface (and not the one of the device). This is to prevent the generic parent driver from not being loaded [575].

#### 4.1.8 USB device representation in Windows

Technically, the generic parent driver creates a PDO for each interface and it generates a set of hardware IDs for each PDO [576]. The *device ID* representing the device, for an interface PDO, has the following form:

```
USB\VID_v(4)&PID_p(4)&MLz(2)
```

where each field corresponds to:

---

<sup>4</sup>USB-IF: USB Implementers Forum.

- $v(4)$  is the four-digit vendor code that the USB standard committee assigns to the vendor.
- $p(4)$  is the four-digit product code that the vendor assigns to the device.
- $z(2)$  is the interface number that is extracted from the `bInterfaceNumber` field of the interface descriptor.

It is also possible to use a different name format generated by the generic parent driver. These compatible IDs are crafted using the information from the interface descriptor [542].

```
USB\CLASS_d(2)&SUBCLASS_s(2)&PROT_p(2)
USB\CLASS_d(2)&SUBCLASS_s(2)
USB\CLASS_d(2)
```

where  $d(2)$  is the class code (`bInterfaceClass`),  $s(2)$  is the subclass code (`bInterfaceSubClass`) and  $p(2)$  is the protocol code (`bInterfaceProtocol`). If we are looking for the keyboard we use on our own computer, Figure 4.27 gives the device ID provided to the keyboard by Windows 10 [577].

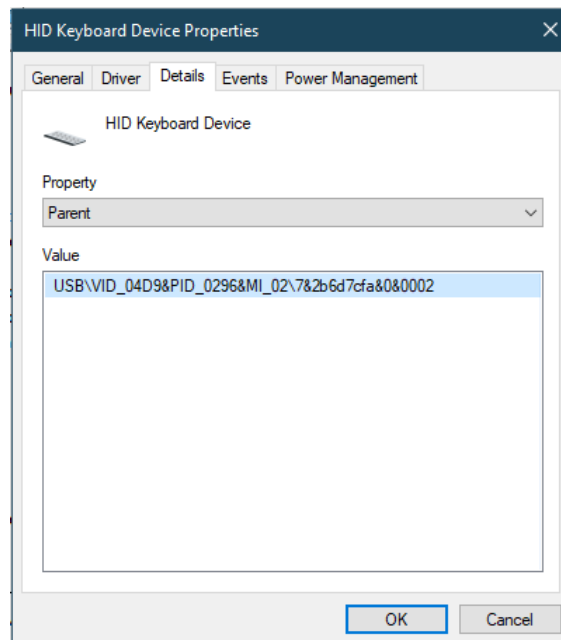


Figure 4.27: Screenshot from the device manager about keyboard’s information.

Note that sometimes it is possible to find *Col* value in the device ID. This one is not always present but it can appear with *interface collections*. Indeed, according to Windows’ documentation [570], some devices group interfaces into interface collections that work together to perform a particular function. In such a case, the generic parent driver treats each collection, rather than each individual interfaces, as a device. More information is given about in Microsoft documentation [578] but this subject is out of our topic since it does not directly impact keyboard devices management.

In the case we are dealing with a composite device, since the system has loaded the different client drivers for the interfaces, it is the generic parent driver’s responsibility to multiplex the data flow from the client drivers. It combines input for the device into a single data stream for the composite device. Note this is its responsibility to handle the power policy, synchronization and PnP requests for the entire composite device and all of its interfaces also. The stated objective of Microsoft is to simplify the task for vendors of composite hardware when Windows does not support some interfaces. That way, Microsoft only requests from vendors to support Windows’ unsupported interfaces.

### 4.1.9 Windows kernel handling USB

#### Key Point 4.13:

📖 The complete architecture of USB management from Windows kernel is given in Figure 4.28.

The technology presented here has evolved from Windows 8 [579]. Indeed, with the rise of USB 3.0, Microsoft decided to separate USB driver stacks for USB 2.0 and USB 3.0 [579]. For short, Windows keeps the existing USB 2.0 stack but only loads the USB 3.0 driver stack when a device is attached to an *eXtensible Host Controller Interface* (xHCI) controller. However, the general philosophy of the stack remains the same. Our representation of the current architecture is given in Figure 4.28 which makes the distinction between the two stacks. In both case, it relies on host controller drivers which are closely connected to the hardware. These drivers (USBOHCI.sys - USB 1.0, USBUHCI.sys - USB 1.1, USBEHCI.sys USB 2.0 and USBXhic.sys - USB 3.0 and above) manage the different physical host controller and their protocols. Among the tasks they manage, they are responsible to map the transfer requests blocks from upper layer drivers and submitting the requests to the hardware above them. The hardware is managed directly by the PCI bus device stack which is inherent in the PnP environment [573]. Once the transfer is complete, these drivers handle transfer completion events from the hardware and propagates the events up the driver stack.

Above the host controller driver, it belong the port driver whose handles USB host with an extra level of abstraction since it is focused on USB protocol itself. Composed by USBPort.sys and Ucx01000.sys, they manage those aspects of the host controller driver's duties that are independent to a specific protocol. Note that Ucx01000.sys is extensible and it is designed to support other types of host controller drivers that are expected to be developed in the future. The USB host controller extension serves as a common abstracted interface to the hub driver. It provides a generic mechanism for queuing requests to the host controller driver and overrides certain selected functions. All input or output requests initiated by upper drivers reach the port driver before the host controller drivers. Finally, it comes hub drivers (USBHUD.sys and USBHUB3.sys) to manage USB hubs and their ports. This one is managed as a *Functional Device Object* (FDO) [580]. The goals of these drivers are to enumerate devices and other hubs attached to their downstream ports and create physical device objects (PDOs) for the enumerated devices and hubs. It is also responsible for dealing with configuration descriptor requests to know which endpoint operations are supposed to be performed with.

The case of USB client driver is quite specific to user-mode driver framework (UMDF) and kernel-mode driver framework (KMDF). Both technology are embedded in *Windows Driver Frameworks* (WDF) [581] context for driver development. For short, these open source frameworks [582] help to write drivers in a modern way, increasing the security while removing complex details prone to error in order to write high-quality device drivers. Thus, it is possible to write specific drivers for USB with this kind of technology, depending on the specifications required [583]. Details about user-mode [584, 585] and kernel-mode implementation [586, 587] are beyond the scope of this part. But for the sake of clarity, we can say that such a technology belongs on WinUSB.sys and WinUsb.dll [561] components.

Finally, to be complete on the Windows USB architecture, we have to mention USBD.sys driver. This one is not clearly represented in Figure 4.28 since it has a more important role than the one presented. This driver is not supposed to do anything by itself (since its driver entry point only returns zero) but to provide a set of routines to support global USB device management by other USB drivers. This one is linked to almost all USB drivers except host controller drivers. It consists of a set of exported routines able to manage USB configuration descriptor from a device and to manage internal list and structures about that device. Last but not least, in USB 3.0 drivers stack, the two top drivers are linked to SleepStudyHelper.sys driver. This one is used as part of the Modern standby SleepStudy [588] technology started with Windows 8.1. It is responsible to provide an overview information about power consumption by the system (active time, idle time, power consumed, processes or devices which consume resources).

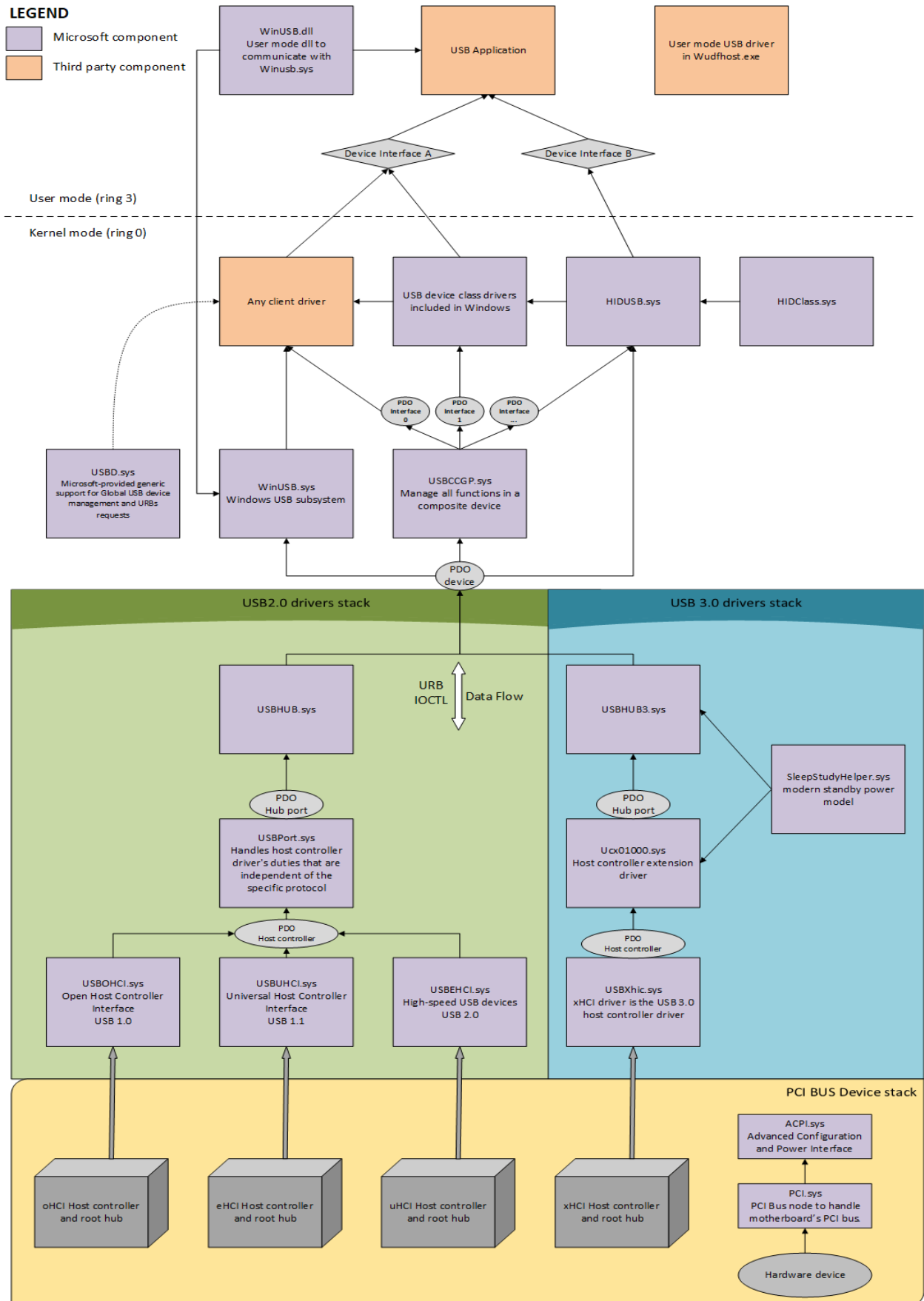


Figure 4.28: Current architecture of USB drivers stack in Windows.

## 4.2 HID protocol

### Key Point 4.14:

- ☞ USB devices are broken into various device classes sharing common behaviors and protocols to serve similar functions.
- ☞ HID is a special class defined through USB interface descriptor which allows a device to interface with humans easily.

### 4.2.1 HID presentation

#### Key Point 4.15:

- ☞ The HID interface corresponds to *Human Interface Device*.
  - ☞ HID is defined in USB interface descriptors to define usage(s) (functions) of a device.
  - ☞ Device self-describing and manufacturer-defined interface to allow generic software applications.
  - ☞ Only one driver software is required on the host to handle HID data whatever the HID device is.
  - ☞ How interactions must be performed is clearly defined through the use of HID format.
- ☞ HID is defined at *interface level* in USB protocol (Key-Point 4.11).
  - ☞ There can be more than one HID interface per device's configuration (several interfaces for several features, for example).
  - ☞ There is usually a default *boot interface* which can be used at start-up time.
- ☞ HID interface is composed by many attributes (providing the class of the HID device).
  - ☞ It informs if the device supports *boot interface* (simplified for UEFI/BIOS setup).
  - ☞ The boot interface is mostly used by mouse and keyboard devices.
  - ☞ In boot interface mode, the keyboard is using a specific table of predefined key-codes (layout from North American keyboard).
- ☞ HID communication:
  - ☞ An interrupt pipe corresponds to an interrupt IN endpoint used for receiving data from the device.
  - ☞ OUT endpoint is optional to export data from the host to the device (otherwise, control endpoint is used).
  - ☞ Communication is performed via data fixed-length structures called *reports*.
  - ☞ *Reports* are regular USB *transactions* (which shows the encapsulation of HID in USB).
  - ☞ *HID report descriptors* explains to the host how information is structured in *reports*.
  - ☞ The format of the HID report descriptor is defined in HID documentation [8].
- ☞ The use of *HID report descriptors* allows a generic communication with the host without device specific defined structures.

USB devices types are defined thanks to various device classes given in device or interface descriptor in `bDeviceClass` field. When we are looking for device class code [545], from USB organization, we can see that each device class defines the common behaviors and protocols for all devices that serve similar functions. For



instance, there are audio, printer, communication, mass storage device classes but no keyboard or mouse classes. This is because all devices interfacing with humans are grouped in a single HID class which stands for *Human Interface Device* [8]. Devices with physical control panels can use a HID interface to send control-panel inputs to the host. Typical examples of HID class devices include keyboards and pointing devices but also any front-panel controls such as telephones, data gloves, rudder pedals and so on. Surprisingly, it includes devices which do not directly interact with users but provide data in a similar format to HID class devices, such as bar-code readers, thermometers, or voltmeters. More generally and contrary to what its name suggests, a HID device may not have a user interface. The device just needs to be able to function within the limits of the HID class specification [589].

According to HID documentation [8], HID class definition follows different underlying goals. The first is to be as compact as possible in order to minimizing data consumption impact on the USB bus. The second is about to be self-describing to allow generic software applications. HID format allows devices to define how to communicate within a specific canvas. Instead of following a generic format for each device (which could be painful for devices manufacturers), it is the responsibility of each device to define how the host must interact with the last. Of course, the definition of how interactions must be performed is clearly defined through the use of HID format. Such a way, it allows HID devices to be extensible, robust and to allow software application to skip unknown information. Finally, it is designed to support nesting and collections.

A USB/HID class device uses a corresponding HID class driver to retrieve and route all data. It means there is only a single driver software on the host to handle HID data whatever the HID device is. The only requirement is that the device follows the HID standard and declares itself to be HID compatible. This is performed at the interface descriptor level.

Indeed, as explained in the USB presentation (Key-Point 4.11), devices manufacturers do not use the device descriptor a lot to define usage of a device. Instead, they prefer to do it at the interface level. This habit has been taken up for adoption in the HID standard and it is recommended in HID standard [8]. HID devices have class/sub-class values of both zeroes in their device descriptors, and instead have the class/sub-class values valid in their interface descriptors [590]. Note that a USB device may be a single class type or it may be composed of multiple classes. For instance, an USB audio loudspeaker might use features of the Audio and HID classes if there are buttons on the device to manage the volume. This is possible because the class is specified in the interface descriptor where multiple interfaces can be defined (and not in the device descriptor which is unique).

Remembering the interface descriptor structure (table 4.6 [544]), the `bInterfaceClass` member of an interface descriptor is always 3 for HID class devices. At early development of the HID specification [8], subclass information (`bInterfaceSubClass` field) was intended to be used to identify the specific protocols of different types of HID class devices, but it quickly became apparent that this approach was too restrictive. On the first hand because it was not possible to define for all possible (and yet to be conceived) devices a specific code. On the other hand because many known devices seemed to straddle multiple classifications (keyboard with embedded pointer or mouse with multiple buttons). Instead, this `bInterfaceSubClass` member is used to declare whether a device supports a boot interface or not. Zero means no boot support, one means a boot support, all other values are reserved for future use. Indeed, the HID protocol requires a lot of data to be parsed from the device to be correctly used. Such parser can be complex to write and it requires a significant amount of code which might not fit in the ROM of UEFI/BIOS setup. The types of *boot devices* able to deal with BIOS only concern the mouse and keyboard devices. This is why a simplified version of HID exchanged data is used by the UEFI/BIOS in many hosts. Full documentation is provided in [8], "Annex B." and an illustration is given in [590].

The `bInterfaceProtocol` member of an interface descriptor has a meaning if and only if the `bInterfaceSubClass` member declares that the device supports a boot interface. In such a case, it is used to identify the type of device the host is dealing with (1 = keyboard, 2 = mouse, all other values are reserved). Note that in such boot mode, the keyboard is using a specific table of key codes. This one is defined in official documentation from USB-IF [591]. Due to the variation of keyboards from language to language, it is not feasible to specify exact key mappings for every language. The list provided in the documentation is not specific for a key function in a language. When there are differences of keys between languages, it is proposed to use the closest equivalent key position but not to replace the keyboard firmware. As an example, Y key on a North American keyboard corresponds to Z key on German keyboard. Instead of replacing Y code by Z code in the list, it is recommended

to use Y code for both keyboards. According to [591], this practice is still being used in the industry, in order to minimize the number of changes to the electronics to accommodate other languages. This may explain why some strokes on non North-American keyboards are not correctly handled by some UEFI/BIOS firmware...

Technically, a HID class device communicates with the HID class driver using either the control (default) pipe or an interrupt pipe. The control pipe is used to receive and respond to requests for USB control and class data. It is also used to transmit data when polled by the HID class driver — via an IRP — or to receive direct data from the host. The interrupt pipe corresponds to an interrupt IN endpoint. This one is used for receiving asynchronous (unrequested) data from the device. For example, there is no way for the host computer to know when a user will press a key on the keyboard. Even if the interrupt pipe is interrupted only by name, the host driver mimics this behavior by using IRP to poll the device periodically to obtain new data from the device. An interrupt OUT endpoint is optional. If a device declares an interrupt OUT endpoint, this one is used as the main channel to export data from the host to the device. Otherwise, this is the control endpoint which is used.

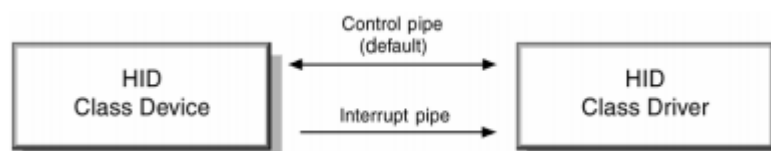


Figure 4.29: From [8], direction of the exchanges on HID pipes.

The rate of data exchange via HID is limited. It corresponds to the rate given for interrupt transfer which could evolve according to different USB versions. But, generally, the endpoints in the default interface should request no more than 64 KB/s [589]. It is not a real limitation for keyboards and users. Under the realistic assumption that a single keystroke is encoded on a single byte, it is impossible for normal human beings to transfer this amount of characters per second from the keyboard to the host. In fact, the keyboard would have stopped managing the keystrokes a while before reaching this limit in reference of the phenomenon called *phantom condition* [590]. Regardless of the economic aspect, this also explains why keyboards are generally devices using 2.0 or at most 3.0 USB version.

A HID can have at most one interrupt IN endpoint and one interrupt OUT endpoint [589]. If for any reason a device needs more interrupt endpoints, this one can be defined as a composite device with multiple HID interfaces. In such a case, an application on the host obtains separate handles for each HID interface in the device. It corresponds to the general USB composite device behavior implemented in Windows. Note that if a HID interface always uses an interrupt endpoint, it does not prevent a device to use another endpoint with another transfer type. Reusing our example with the USB audio loudspeaker, it is possible to use isochronous transfers for audio interface and a HID interface to control different buttons such as volume, balance, treble and bass.

Communication between the host and HID device is performed via data fixed-length structures called *reports*. It corresponds to a regular transaction in USB but with a predefined shape. The report descriptor provides information about the data HID device, especially a description of the data provided by each control in a device. The host sends and receives data by sending and requesting reports in control or interrupt endpoints. The report format is flexible, able to handle any type of data and specific to each type of device. Of course, to make the information transmitted in the reports (which are device-specific) understandable to the host, it is necessary to clearly define how the data is transferred. This is the role of the *HID Descriptors*. There are two types of descriptors: *report* and *physical* descriptors. Both are data structures following HID standard and they indicate the size of data structures exchanged. The first explains how data is formatted between the host and the device while the second provides information about the specific part (or parts) of the human body that are activating a control (or controls). According to HID documentation [8], physical descriptors are entirely optional. They add complexity and offer very little in return for most devices. This is why we are going to focus only on the report descriptor. Figure 4.30 represents the hierarchy of the different structure descriptors used in USB with HID. These structures are used only in configuration steps of the use of the USB device. Note that HID is ultimately an overlay and an extension of what is defined in USB.

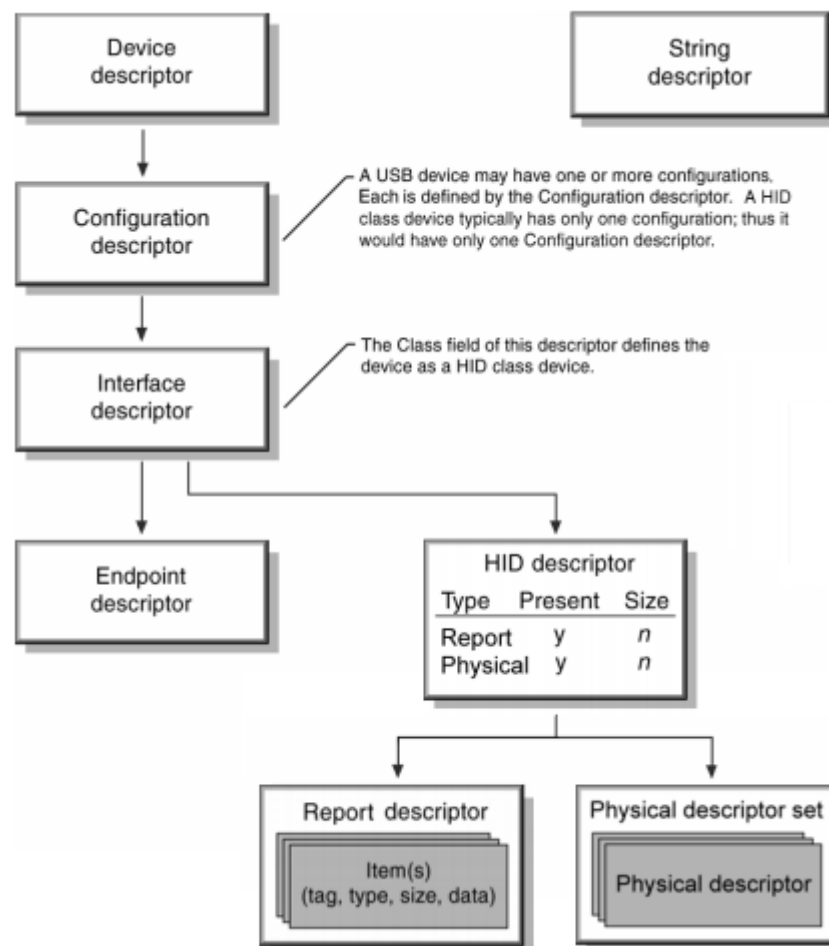


Figure 4.30: From [8], all USB descriptor structures and HID descriptors.

## 4.2.2 HID interface with report descriptors

### Key Point 4.16:

- ☞ Report descriptors are composed of different items, describing how the data is transferred.
  - ☞ Among different items, "Usage" defines the purpose of an item.
  - ☞ Usage tag defines what should be done with data provided by the device.
- ☞ A collection is a meaningful grouping of Input, Output, and Feature items.
  - ☞ To group many small different items to perform a single function in a bigger item.
  - ☞ A mouse device: a collection of four information (x, y, button 1, button 2).
- ☞ Even if in the documentation, a text representation is used for HID descriptor report, this one is compiled in raw data.
- ☞ Fun fact, the management of keyboard device's LED lamps is done by the host and not by the device itself.

The same way that regular USB descriptors use tables of information composed of blocks of data, report descriptors are composed of pieces of information. Each single piece of information about a device in a report descriptor is called an *item*. All items have a one-byte prefix that contains the item tag, item type, and item size followed in memory by optional item data. Figure 4.31 extracted from [8] illustrates how an item is designed (note that it is little-endian, as expected in USB norm).

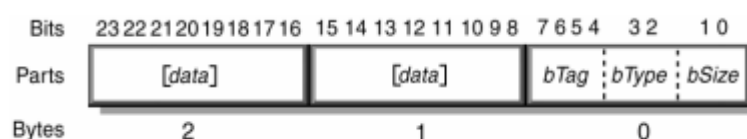


Figure 4.31: Generic Item Format.

According to [8], the size of the data portion of an item is determined by its fundamental type and there are two basic types of items: short and long items. Short items can have a size between 0 to 4 bytes (since the item size is encoded on two bits, it means that item size of  $11_b$  corresponds to 4 data bytes (not 3)). Long items, even if they are mentioned and defined in HID 1.1 documentation, they are not really used [589]. The item tag specifies the item's function and the item type specifies the scope of the item. There are three item types: Main (00), Global (01) and Local (10). Leaving aside Local and Global tags which are specific to the way report descriptors are written, report descriptors may contain several main items. In the case of main item, there are five main item tags currently defined:

- Input item tag: refers to the data from one or more similar controls on a device. For example, one or more push buttons or switches.
- Output item tag: refers to the data to one or more similar controls on a device. For example, it can represent data to one or more LEDs on a keyboard.
- Feature item tag: describes device input and output not intended for consumption by the end user. For example, a control Panel toggle.
- Collection item tag: a meaningful grouping of Input, Output, and Feature items. For example, a mouse could be described as a collection of two to four data (x, y, button 1, button 2). It is a way to group many small different items that together perform a single function in a bigger item.
- End Collection item tag: a terminating item used to specify the end of a collection of items.

A report descriptor is the complete set of all items for a device. A report descriptor alone is complete enough by itself to allow an efficient communication between the device and the host application. It means when an application reads it, it knows how to handle incoming data as well as the usage of data provided. Each main item tag (Input, Output, or Feature) identifies the size of data returned by a particular control, and identifies whether data is absolute or relative, and other pertinent information (data, constant, array, variable, linear or nonlinear, preferred state, null state, nonvolatile, bit field, buffered bytes).

Global items describe rather than define data from a control. It is global to the following items defined in the report descriptor until a new global item is used. From [591], there are many relevant global items which help to understand the meaning of different items. The most important is *Usage* which defines the purpose or the meaning of an item. This usage tag defines what should be done with data provided by the device. This feature allows a vendor to ensure that the user sees consistent function assignments to control across applications.

More generally, usages are also used to define the meaning of groups of related data items. This is accomplished by the hierarchical assignment of usage information to collections. Indeed, usages identify the purpose of a collection and the items it contains. For each Input, Output, Feature, and/or Collection data item within a Collection item can be assigned a purpose with its own usage item. Usages assigned to a collection apply to the items within the collection. In theory, usages could be assigned to any type of HID control (variable, array, etc.) but in practice, usages only make sense when they are attached to particular controls and used in certain ways.

The HID Usage Tables [591] references Usages that have been organized into pages of related controls. Each usage has a unique number called usage ID and it is referenced with a usage name and a detailed description. Such organization should insure that host software would be able to recognize or utilize data item it is looking for (even if there could be sometimes confusions in some software). Usages can also be used to identify functional devices as a whole. This is a good method to facilitate device's identification by an application when the device provides functions of interest. This is why devices manufacturers that write the firmware of their devices use collections. Usages attached to application collections that are wrapped around all the items that describe a particular functional device — or a particular function in a complex device is — is easier to manage. Indeed, host application would query the HID driver for collection usages that concerns it. Example given in [591] is relevant: a gaming device driver might look for Joystick and Game Pad usages, while a system mouse driver might look for Mouse, Digitizer Tablet and Touch Screen usages. Note that usages can be defined by vendors and when an application deals with unknown usages, that one should ignore them.

Usage page values are given in the HID documentation [591]. It consists of a 32-bit unsigned value where the high order 16 bits defines the Usage Page and the low order 16 bits defines a Usage ID. For instance, keyboard devices belong to *Generic Desktop Page* (Usage Page 0x01) with a Usage ID equals to 0x7. This is where one can find keyboard, mouse, joystick, VR controls etc. There are sometime called *application collections* in HID documentation since it corresponds to a group of main items.

As extra information, even if it is optional, a report descriptor could add more global items. For instance, an application which deals with data items as a measurement of time, mass, distance, force, velocity, acceleration, angular acceleration, energy, voltage, and so forth, must look at the *units* (and *unit exponent*) to properly interpret the value defined by a usage. With the unit, it declares the Logical Minimum, Logical Maximum, Physical Minimum, and Physical Maximum items. All of them help to define the resolution of the item measured by the device. Dealing with coordinate system, HID class devices are encouraged, when possible, to use a right-handed coordinate system (Figure 4.32). Padding is also relevant to be aligned on a 8-bit memory length. This is why, in some devices, bits are added in items with no real meaning except to ensure memory alignment. For the sake of completeness, local item tags define characteristics of controls and do not carry over to the next main item. They can be seen as a local version of global items.

If the HID class device uses the usual standard USB descriptors, it also uses its own class-specific descriptors. These descriptors differ from standard USB descriptors. With HID, there are three class-specific descriptors: HID, Report and Physical. The hierarchy between all these different descriptors depends on HID descriptor which references both report and physical ones. This hierarchy is given in Figure 4.33.

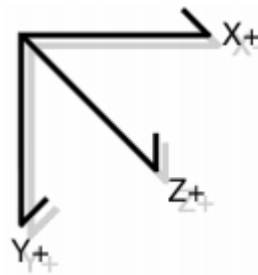


Figure 4.32: Illustration of the right-handed coordinate system recommended in HID (with mouse, for instance).

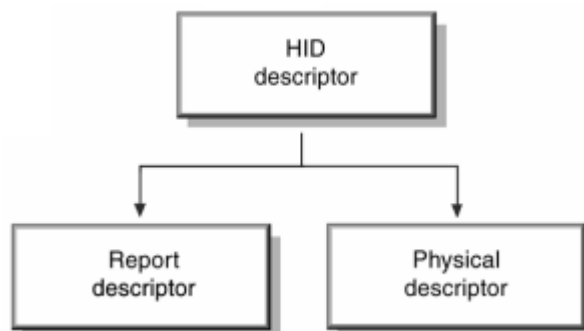


Figure 4.33: Hierarchy between different HID descriptors.

The HID descriptor identifies the length and type of the two other types of HID descriptors for a device. Its structure is given in table 4.11 [592].

Offset	Field	Size (Bytes)	Description
0	bLength	1	Numeric expression that is the total size of the HID descriptor.
1	bDescriptorType	1	Constant name specifying type of HID descriptor.
2	bcdHID	2	Numeric expression identifying the HID Class Specification release.
4	bCountryCode	1	Numeric expression identifying country code of the localized hardware.
5	bNumDescriptors	1	Numeric expression specifying the number of class descriptors (always at least one report descriptor).
6	bDescriptorType	1	Constant name identifying type of class descriptor. - 0x21 HID - 0x22 Report - 0x23 Physical descriptor - 0x24 .. 0x2F Reserved
7	wDescriptorLength	2	Numeric expression that is the total size of the Report descriptor.
9	[bDescriptorType] ...	1	Constant name specifying type of optional descriptor.
10	[wDescriptorLength]...	2	Numeric expression that is the total size of the optional descriptor.

Table 4.11: HID Descriptor.

Among the different fields in HID descriptor, some can be explained. First, bLength only corresponds to the

length of the HID descriptor structure itself, not including the content of the report or the physical descriptor. This value is not fixed and depends on how many descriptors are referenced at the end of the structure. `bDescriptorType` at offset `+0x01` corresponds to the type of current descriptor, that is to say `0x21` since it is a HID descriptor. The same values<sup>5</sup> are used in the context of `bDescriptorType` at offset `+0x06`. In the last case, in practice, only Report (`0x22`) or Physical (`0x23`) are relevant values to describe the coming descriptors. The field `bcdHID` specifies the HID version used with that descriptor, the same way it is used in USB device descriptor with the `bcdUSB` field. In our case, since we are dealing with the last version of HID, that is to say 1.11, it is usually `0x0111` which is used. `bCountryCode` member is used to identify which country the hardware is localized for. Even if it can be relevant for keyboard manufacturers, most hardware devices are not localized and thus this value is zero. Devices located in France are referenced with the ID code `0x08`. It finally comes `bNumDescriptors` member which references the number of descriptors which are provided with this interface. There is at least one descriptor for the obvious reason that without, it would not be possible to communicate through HID with the device. Then comes `bDescriptorType` and `wDescriptorLength` for each of the descriptor referenced in this HID descriptor. Values represented in brackets represent optional descriptors if they are present. Note that, in the case of  $n$  multiple optional descriptors, descriptors and associated lengths may be specified up to offset  $(3 * n) + 6$  and  $(3 * n) + 7$  respectively.

Report descriptor has not a truly defined structure. It is not simply a table of values unlike the other descriptors. The Report descriptor is made up of items that provide information about the device. The number of items varies from device to device which gives it a variable length. But we retrieve main items which define what is the content of the structure about input, output or features. Together these fields identify the kind of information items provide (type, tag, size). The Report descriptor provides a description of the data provided by each control in a device. Note that a report descriptor can define more than one report. In such a case, an item report ID is added in each report in the report descriptor to identify them. That way, the first byte of the report provided by the device will usually starts with the report ID (to identify which type of report the host is dealing with). A general view of all HID class descriptor is given in Figure 4.34 (from [8]).

More than a long description of all possibilities about HID report descriptor, we propose to illustrate with the one from our own keyboard. This is a simple interface used by a device which proposes one of its interface in boot mode (which means a simplified version of HID protocol's possibilities). This one is given in table 4.12. The report is a compiled text. In practice, it is only in compiled form (raw data) that HID parser software and devices use. Humans prefer the text version which is much more user-friendly and there are tools provided by USB-IF to help device manufacturer to write them [593]. Far from being obvious to understand, even in text mode, official documentations [8, 591] and specialized blogs can help [594]. Even official HID parser from Windows can present misunderstanding which leads to vulnerabilities [595].

For the sake of simplicity, we have only commented on the main lines and in particular those responsible for holding data. Nevertheless, an attempt will be made to explain the functioning of the lines that precede them. The first thing is to understand that the main tag items (input, output, feature, collection) are influenced by the lines that precede them. The first lines define a collection, such as different items will be mixed (input and output in our case). The first input (1) represents a single byte. Indeed, this one is defined with a `Report Count` equals to 8 objects by a `Report Size` equals to 1 bit. This is 8 bits and therefore one byte. The meaning of this byte is given by the line above. First, each value in the defined bits can take a minimum value (Logical Minimum) of 0 and a maximum value of 1 (Logical Maximum). It means that each bit has a special meaning. And this meaning is given by the lines uppermost with `Usage` tag. It defines the value represented by each bits, from *Keyboard Left Control* (`0xE0`) to *Keyboard Right GUI* (`0xE7`). It corresponds to modifier keys (such as control, shift, alt and so on) and unlike PS/2 keyboards, modifier keys are not managed as any other key codes. Instead, they are managed as a bitfield in a single byte since there are only 8 keys concerned. This specific modified code is merged by the host keyboard driver to represent a combination of several keys pressed or directly the scan code of the keystroke with the modified behavior included to be understandable by the operating system.

The second input (2) item is a constant of one byte full of zero. Details are provided in the line of the input tag and the size of the two above lines. No real meaning since it is a reserved byte (for OEM use but it should be

---

<sup>5</sup>Contrary to what has been written in [8], definitions of these values are given in section 7.1 and not 7.1.2 which describes `Set_Descriptor` request.





Item Tag (Value)	Raw Data	Description
Usage Page (Generic Desktop)	05 01	
Usage (Keyboard)	09 06	
Collection (Application)	A1 01	
Usage Page (Keyboard/Keypad)	05 07	
Usage Minimum (Keyboard Left Control)	19 E0	
Usage Maximum (Keyboard Right GUI)	29 E7	
Logical Minimum (0)	15 00	
Logical Minimum (1)	25 01	
Report Size (1)	75 01	
Report Count (8)	95 08	
Input (Data,Var,Abs,NWrp,Lin,Pref,NNul,Bit)	81 02	(1) One byte to define modifier keys (1 bit per key)
Report Count (1)	95 01	
Report Size (8)	75 08	
Input (Cnst,Var,Abs,NWrp,Lin,Pref,NNul,Bit)	81 03	(2) One byte padding (constant zero)
Report Count (3)	95 03	
Report Size (1)	75 01	
Usage Page (LEDs)	05 08	
Usage Minimum (Num Lock)	19 01	
Usage Maximum (Scroll Lock)	29 03	
Output (Data,Var,Abs,NWrp,Lin,Pref,NNul,NVol,Bit)	91 02	(3) 3-bit for binding LEDs on keyboard
Report Count (1)	95 01	
Report Size (5)	75 05	
Output (Cnst,Var,Abs,NWrp,Lin,Pref,NNul,NVol,Bit)	91 03	(4) 5-bit for padding LEDs (constant zero)
Report Count (6)	95 06	
Report Size (8)	75 08	
Logical Minimum (0)	15 00	
Logical Maximum (164)	26 A4 00	
Usage Page (Keyboard/Keypad)	05 07	
Usage Minimum (Undefined)	19 00	
Usage Maximum (Keyboard ExSel)	29 A4	
Input (Data,Ary,Abs)	81 00	(5) 6 bytes to buffer the current (most common) keystrokes
End Collection		

Table 4.12: Interface 0 HID Report Descriptor Keyboard.

the host should ignore input from the keyboard.

To summarize, the HID parser of the host is going to manage the inputs and outputs on this HID interface by automatically generating, on-the-fly, two structures that can be represented in C as given in table 4.13. This mechanism allows hosts to deal with HID devices to adapt themselves automatically to the data format used to exchanged information.

```
typedef struct _HID_REPORT_INTERFACE_0.INPUT {
    unsigned char ModifierKeys;
    unsigned char Reserved;
    unsigned char Keystrokes[6];
} HID_REPORT_INTERFACE_0.INPUT;
```

Code 4.4: HID report interface 0

```
typedef struct _HID_REPORT_INTERFACE_0.OUTPUT
{
    unsigned char LED_NUMLOCK : 1;
    unsigned char LED_CAPSLOCK : 1;
    unsigned char LED_SCROLL LOCK : 1;
    unsigned char LED_COMPOSE : 1;
    unsigned char LED_KANA : 1;
    unsigned char Reserved : 3;
} HID_REPORT_INTERFACE_0.OUTPUT;
```

Code 4.5: Hid report interface 1

Table 4.13: Structure used as Input and Output from HID report given in table 4.12.

### 4.2.3 HID standard requests

#### Key Point 4.17:

🔑 HID requests written as function names are technically standard USB SETUP commands on the Default Control Pipe (endpoint 0).

Once the host knows that the device has an HID interface (thanks to the `GetConfiguration` request which provides a list of the system interfaces), the host must retrieve HID descriptor. Indeed, the HID descriptor is not given directly in an usual `GetConfiguration` request. The host learns the HID descriptor by using a `GetDescriptor` request containing the HID interface. This is a standard USB SETUP command on the Default Control Pipe (endpoint 0). The `bmRequestType` is set to `GET_INTERFACE` (0b1000001) since it concerns an interface. The `bRequest` is equal to `GET_DESCRIPTOR` (0x06) since it concerns a descriptor. The `wValue` used in the request is equal to 0x21 for HID, 0x22 for report descriptor and 0x23 for physical descriptor [8, 589]. The `wIndex` corresponds to the interface number from where the HID descriptor is required. Once the HID descriptor has been retrieved, `GetDescriptor(report)` request is used to retrieve all HID report descriptor. From that point, the host has the ability to select one report descriptor to use with the `SetDescriptor` command, reusing same members than `GetDescriptor`. Note that, during initialization, the HID report descriptor 0 is preferred by Windows, especially if this one is indicated as boot device.

There are other class-specific requests that allow the host to inquire about the capabilities and state of a device. In addition, the host has the ability to set the state of the output and feature items. These class-specific requests are still SETUP commands on the default endpoint with a `bmRequestType` value where the fifth bit is always set to one (Class request). Technically, only 0b10100001 and 0b00100001 are valid values since communication is always performed at the interface level (bit 0) and exchange can be from device to host and vice versa (bit 7). Since the fifth bit is always set, `bRequest` field can be redefined to represent different requests. List of HID specific requests and `bRequest` field values with description is given in table 4.14.

bRequest	Request name	wValue	Boot only	Mandatory	Description
0x01	GET_REPORT	High byte: the report type - 0x1: Input - 03h: Feature Low byte: Report ID (default 0)	No	Yes	The host requests an Input or Feature report.
0x02	GET_IDLE	The Report ID where the request applies to (zero = all)	No	No	The host reads the current Idle rate (expressed in units of 4 ms)
0x03	GET_PROTOCOL	0x00	Yes	No	The host reads which protocol is currently active (boot or report).
0x09	SET_REPORT	High byte: the report type - 0x1: Input - 03h: Feature Low byte: Report ID (default 0)	No	No	The host sends an Output or Feature report (requires an OUT endpoint).
0x0A	SET_IDLE	The high byte sets the duration: - 0x00: only reporting when a change is detected. - Above 0x00: fixed duration is used (with a 4 millisecond resolution) Low byte: Report ID (default 0)	No	No	Limit the reporting frequency of an interrupt in endpoint.
0x0B	SET_PROTOCOL	- 0 : boot protocol. - 1 : report protocol.	Yes	No	Switches between the boot protocol and the report protocol.

Table 4.14: HID class-specific requests.

Apart `GetReport` which is mandatory (and useful for the host to ask data from the device), all other requests are for output/feature communication with the device (such as selecting displayed colors on the keyboard), boot protocol or for optimization such as idle requests. Regarding the last request in keyboard management, this one is not really used by Windows nowadays since Windows HID driver attempts to set the idle rate to zero [589]. In such a case, Windows is only *notified* when a key is stroke. There is no real necessity to save bandwidth for a keyboard device where the data stream exchanged is very low (few keystroke reports per second for a normal human).

## 4.2.4 HID and Windows kernel

### 4.2.4.1 HID kernel architecture

#### Key Point 4.18:

- ☞ All requests from devices that interface with HID are redirected to the HIDClass.sys driver.
- ☞ It is not necessary to be USB to use HID. The Bluetooth can also use it (hence USB/Bluetooth keyboard management compatibility).

Once Windows has retrieved an interface descriptor which references an HID interface, the system has to retrieve information from the device. Hence, a special API has been introduced in Windows 2000 and we are going to describe it. The central point of the HID driver stack in Windows is built on the class driver named HIDClass.sys. This one is a pivot point between requests sent by other drivers above HIDClass.sys to the device and below by transport mini-drivers which carry information from the device at destination to upper drivers. Both user-mode and kernel-mode components can interact with this driver, of course, with limited rights in the case of user-mode. Figure 4.35 [596] gives a simplified view of the HID architecture.

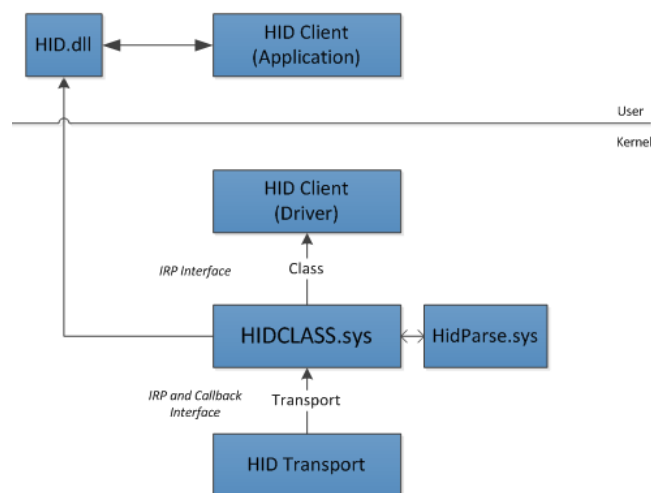


Figure 4.35: Simplified view of HID architecture on Windows.

On Figure 4.35, HID transport refers to any means of transport used to convey HID data. Technically, a HID mini-driver gives an abstraction from the device-specific operation of the input devices that it supports. In the most classic case, the HIDUSB.sys driver is the preferred means of transport for any USB device. But there are non-USB devices that also use a HID interface. Among the different technologies supported by Windows [597], Bluetooth and Bluetooth LE is the most notable with mini-drivers transport HIDBTH.sys and HIDBTHLE.dll. Indeed, Bluetooth technology uses HID interface to interact with host as any other USB device [598]. It explains why it is easy to configure and interact with a Bluetooth wireless keyboard since radio information broadcasting from the device is received by the host, transformed from radio to numeric signal via hardware and routed through HIDBTH.sys driver to HIDClass.sys driver.

Technically, the HID class driver is an *export driver*<sup>7</sup> that is linked to HID mini-drivers. The operation to bind a mini-driver to HID class driver is a registration by calling `HidRegisterMinidriver` [600] API routine [601]. The combined operation of the HID class driver and a HID mini-driver acts as a *Windows Driver Model* (WDM [602]) function driver for an input device and a bus driver for the child devices (different HID interfaces) that the input device supports. This reattachment design allows USB HID devices and non-USB input devices to be

<sup>7</sup>An *export driver* is a kernel-mode DLL that can be loaded by any driver to get access to its exported routines [599].

attached to ports or buses other than a USB bus. It allows communication with HIDClass.sys driver as if all these devices were regular USB HID devices. This is transparent for upper-drivers or user-mode applications when they communicate with these devices regardless the real communication protocol used.

#### 4.2.4.2 Registration of a HID class driver

##### Key Point 4.19:

- ☞ It is possible to register a HID driver by calling `HidRegisterMinidriver` routine from `HIDClass.sys` driver.
  - ✍ In practice, it hijacks the driver's IRP handler routines with generic routines from `HIDClass.sys` driver.
  - ✍ This behavior is undocumented by Microsoft.

How does this registration work in practice? Technically, this is `HIDClass.sys` driver which exports the routine `HidRegisterMinidriver` which is imported by the HID transport mini-driver. This routine takes a single `HID_MINIDRIVER_REGISTRATION` [603] structure in parameter. This one configures the HID version<sup>8</sup> that this mini-driver supports, everything to interact with the mini-driver (the main driver object and the registry path of the driver) and a field called `DevicesArePolled`. The last specifies that the devices on the bus must be polled or not in order to obtain data from the device. If not, device must notify the host with a report via some sort of interrupt. In practice [604], most devices will spontaneously generate a report whenever the end user does something at a predefined time interval except if an `Idle` request is sent.

Using reverse engineering on `HidRegisterMinidriver` allows us to understand the trick of the registration. Technically, drivers in a stack are notified via a mechanism of IRP [541, 605]. Hence, they register a set of callback routines for specific operations [606] and they handle them as expected [607]. The registration is done (Figure 4.36) by replacing (after saving) the original IRP callback routines of the mini-driver by a generic one able to reroute and manage HID notifications. Some lines of code in Figure 4.36 have been removed for the sake of clarity.

The replacement of IRP callback routines is undocumented by Microsoft but it remains that this behavior exists from a long time [604]. This is a *direct* way of doing such a registration, but perfectly efficient. Note that original pointers on mini-driver's routines are kept in a specific structure. Indeed, some of these routines will still be notified part of the HID management process. In addition, a custom transport mini-driver should be written only if a system-supplied HID mini-driver does not support a device's port or bus. This is far from being common. More information about specific requirements for HID mini-drivers is given in [608].

#### 4.2.4.3 HID class driver and how to interface with this one

##### Key Point 4.20:

- ☞ In practice, the freedom of HID devices with report descriptors is not as wide as we might think.
  - ✍ Windows must be able to adapt itself to this freedom.
  - ✍ Parsing operation is performed through an API exported by `HIDPARSE.sys` driver.
- ☞ Windows API allows to write HID Client driver (third party drivers) in a simplified way.
  - ✍ HID Client driver should be reserved for specific circumstances (already a lot of supported HID Clients).

Above the transport mini-driver, which makes an abstraction from the device-specific operations of the input

<sup>8</sup>This value is still equal to `0x01` despite HID is now 1.1 version.

```

1 NTSTATUS _stdcall HidRegisterMinidriver(PHID_MINIDRIVER_REGISTRATION MinidriverRegistration)
2 {
3     int v1; // er800
4     int v2; // er900
5     PHID_MINIDRIVER_REGISTRATION MiniDrvReg; // rbx@1
6     __int64 Revision; // rdx@3
7
8     PDRIVER_OBJECT DriverObject; // r14@20
9     int (__cdecl **MajorFunction)(DEVICE_OBJECT *, _IRP *); // rcx@20
10    _DRIVER_EXTENSION *v16; // rdx@20
11    int v17; // edx@20
12    int v18; // ecx@20
13    char v19; // r8@20
14    int v20; // er9020
15    _HIDEXTENSION *DriverObjectExtension; // [rsp+60h] [rbp+20h]@11
16
17    RtlCopyUnicodeString(AddrRegistryPathDevExtent, MiniDrvReg->RegistryPath); // Keep the driver's registry path.
18    HidpGetFastResumeDisableState(DriverObjectExtension); // Check and retrieve "FastResumeDisable" value in driver's registry.
19    DriverObject = MiniDrvReg->DriverObject; // Keep the structure representing the driver in memory.
20    MajorFunction = DriverObjectExtension->MajorFunction; // Retrieve the list of original major functions to save.
21    *(_QWORD *)MajorFunction = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_CREATE]; // Save original callbacks.
22    *(_QWORD *)MajorFunction + 1 = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_CLOSE];
23    *(_QWORD *)MajorFunction + 2 = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_WRITE];
24    *(_QWORD *)MajorFunction + 3 = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_SET_INFORMATION];
25    *(_QWORD *)MajorFunction + 4 = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_SET_EA];
26    *(_QWORD *)MajorFunction + 5 = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_QUERY_VOLUME_INFORMATION];
27    *(_QWORD *)MajorFunction + 6 = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_DIRECTORY_CONTROL];
28    MajorFunction += IRP_MJ_SHUTDOWN;
29    *(_QWORD *)MajorFunction - 1 = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL];
30    *(_QWORD *)MajorFunction = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_SHUTDOWN];
31    *(_QWORD *)MajorFunction + 1 = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_CLEANUP];
32    *(_QWORD *)MajorFunction + 2 = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_QUERY_SECURITY];
33    *(_QWORD *)MajorFunction + 3 = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_POWER];
34    *(_QWORD *)MajorFunction + 4 = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_DEVICE_CHANGE];
35    *(_QWORD *)MajorFunction + 5 = *(_QWORD *)&DriverObject->MajorFunction[IRP_MJ_SET_QUOTA];
36    DriverExtension = DriverObject->DriverExtension; // Keep original DriverExtension.
37    DriverObjectExtension->DevicesArePolled = MiniDrvReg->DevicesArePolled; // Following lines replace IRP callback routines to HIDMajorHandler.
38    DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL] = (int (__cdecl *) (DEVICE_OBJECT *, _IRP *))HidpMajorHandler;
39    DriverObject->MajorFunction[IRP_MJ_WRITE] = (int (__cdecl *) (DEVICE_OBJECT *, _IRP *))HidpMajorHandler;
40    DriverObject->MajorFunction[IRP_MJ_READ] = (int (__cdecl *) (DEVICE_OBJECT *, _IRP *))HidpMajorHandler;
41    DriverObject->MajorFunction[IRP_MJ_POWER] = (int (__cdecl *) (DEVICE_OBJECT *, _IRP *))HidpMajorHandler;
42    DriverObject->MajorFunction[IRP_MJ_PNP] = (int (__cdecl *) (DEVICE_OBJECT *, _IRP *))HidpMajorHandler;
43    DriverObject->MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL] = (int (__cdecl *) (DEVICE_OBJECT *, _IRP *))HidpMajorHandler;
44    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = (int (__cdecl *) (DEVICE_OBJECT *, _IRP *))HidpMajorHandler;
45    DriverObject->MajorFunction[IRP_MJ_CREATE] = (int (__cdecl *) (DEVICE_OBJECT *, _IRP *))HidpMajorHandler;
46    DriverObject->MajorFunction[IRP_MJ_CLOSE] = (int (__cdecl *) (DEVICE_OBJECT *, _IRP *))HidpMajorHandler;
47    DriverObjectExtension->AddDevice = DriverExtension->AddDevice;
48    DriverExtension->AddDevice = (int (__cdecl *) (DRIVER_OBJECT *, _DEVICE_OBJECT *))HidpAddDevice;
49    DriverObjectExtension->DriverUnload = DriverObject->DriverUnload;
50    DriverObject->DriverUnload = (void (__cdecl *) (DRIVER_OBJECT *))HidpDriverUnload;
51    DriverObjectExtension->Reserved1 = 0;
52    status = EnqueueDriverExt((__int64)DriverObjectExtension); // Register this extension in the HID mini-driver list.
53
54 }

```

Figure 4.36: Content of `HidRegisterMinidriver` shows us how registration is performed by hooking original IRP callback routines of caller mini-driver.

devices that it supports, there is the system-supplied *HID class driver* `HIDClass.sys`. This one is a WDM function driver and bus driver used to manage all HID device communications. The last relies on the `HIDPARSE.sys` driver, which provides supports routines to help to manage HID descriptors. As any device can write in its firmware the HID report descriptor it wants, Windows must be able to adapt itself to this freedom. The proper functioning of the devices with the system is at stake. This is why Windows parses HID descriptors to translate them into understandable internal structures for the system. In practice, this freedom is not so great, especially for keyboard or mouse devices. It is nevertheless recommended to follow the main lines of what is done from one device to another. Microsoft publishes some HID report descriptors [609, 604] that most devices manufacturers follow.

It is relevant to understand that `HIDClass.sys` is not a driver which has an active role by itself. Actually, its driver entry point does nothing and in its export routines table, only two routines are relevant: `HidRegisterMinidriver` and `HidNotifyPresence` [610]. The last is sufficiently documented to deter its use by being labelled as “reserved for the HID driver internal framework” only. But it is possible to explain these routine signals a device as being present on the system by modifying internal structures from `HIDClass.sys`. *De facto*, its role is more to provide an interface between the transport mini-drivers and the HID client (drivers or applications) by simplifying interactions for clients. Indeed, rerouting IRP callback routines allows it to receive, handle, parse, understand and formalize in a set of predefined structure HID descriptor reports so that HID clients always deal with a documented set of specific structures and functions.

This allows a HID Client to be written in an independent way from transports drivers. This level of abstraction allows clients to continue to work (with little to no modification) when a new standard or a third party transport is introduced. In the Windows documentation [596], the HID Clients are drivers, services or applications that communicate with `HIDClass.sys` and often represent a specific type of device (e.g. sensor, keyboard, mouse, etc.). Client drivers are also known under the name of function drivers [611]. Such drivers

are managed by the Plug and Play (PnP) which loads at most one function driver for a device to serve one or more devices. They identify the device via a hardware ID or a specific HID interface usage tag (usage page and usage ID) and communicate with the operating system's API [612]. On the first hand, in the case of user-mode drivers and applications, they use HIDClass support routines (`HidD_Xxx`) to obtain information about a HID collection. On the other hand, kernel-mode drivers use HID parsing support routines (`HidP_Xxx`) and HID class driver IOCTLs to handle HID reports [613]. In the last case, for most of *HidP* prefixed API is just a degree of convenient abstraction for developers. Indeed, these routines correspond to a call with a IOCTL request via an IRP [614, 615]. Note that, in the end, it finishes by using an USB Request Block (URB) [616] which is sent to the device.

Writing a specific HID client driver should be reserved for specific circumstances since it already exists a list of supported HID Clients [617]. And generally, a client driver — even if it is simpler to write — is not necessarily the right thing to do. Indeed, falling in the rare case where a given device is not in the list of supported device types by Windows (and the list [618] is rather large and even generic for transports [619]) is quite unlikely. Being unsupported would require the use of a particular or proprietary means of communication. In such a case, providing a transport mini-driver routed to HID is enough if the HID report descriptor is correctly formatted. The necessity to provide a client driver only happens in the case where a vendor would like to interact with its own type of device (clearly identified as a specific type of device — for instance a "Bluetooth Toaster"). Possibilities of architectures where there is an interaction with the HID drivers stack is provided in Figure 4.37 from [620]. Details about how to do and specific requirements or piece of advises are given also [621].

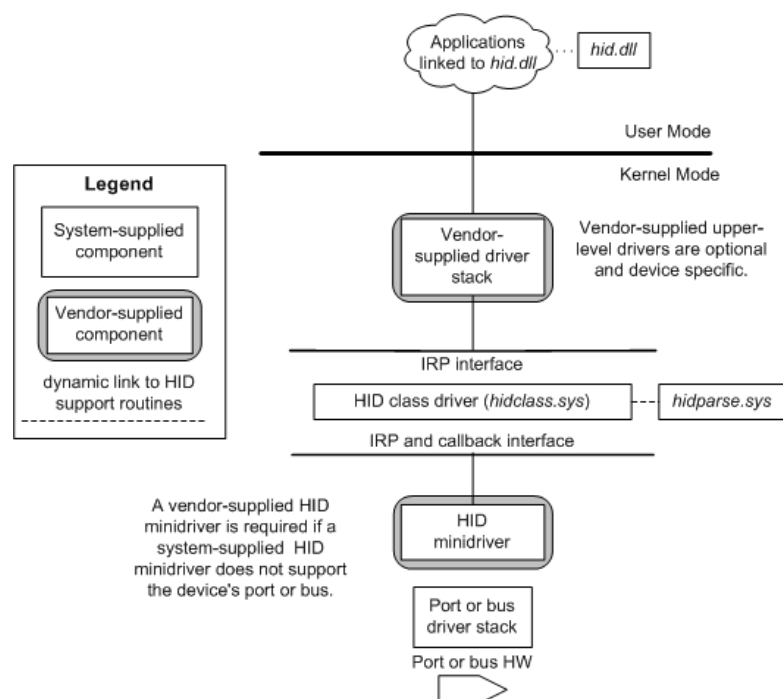


Figure 4.37: Driver stack for a generic HIDClass device with optional and required vendor-supplied components.

Usually, driver clients are not written but function or filter drivers which are written by vendors. Filter drivers [531] are optional drivers that add value to or modify the behavior of one or more device of a specific function. There are three types of filter drivers. *Bus Filter* Drivers which are the rarest ones since they add value to a bus and are supplied by Microsoft or a system OEM. *Lower-Level* and *Upper-Level* Filter Drivers are more common since the first typically modify the behavior of device hardware when the second provide added-value features for a device [622]. General architecture of device drivers is given in Figure 4.38 extracted from [532] and more information is provided about WDM Device Stack in [623].

Such drivers need to understand data coming from HID devices or be able to send data to these devices. This



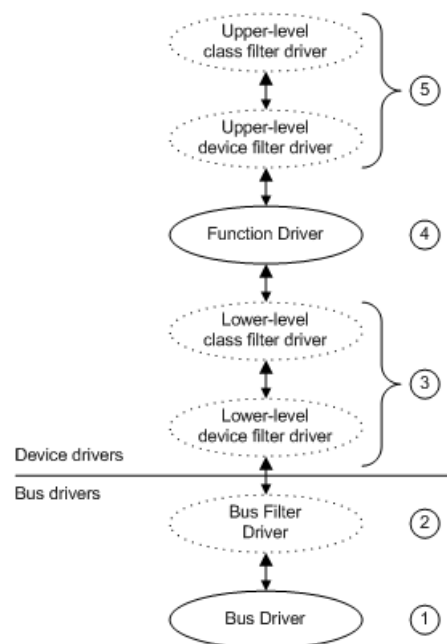


Figure 4.38: Types of WDM Drivers given in the device call stack. The lowest is the number on the picture, the closer to hardware the driver is.

is the role of the HID class driver which redefines specific HID concepts to fit its internal structure constraints.

#### 4.2.4.4 HID parsing in Windows kernel

##### Key Point 4.21:

- ☞ This subsection describes the Windows API structures for processing data parsed from an HID descriptor report.
  - 👉 Notions of *Top Level Collection*, *application collection* and *unnested collection* are presented.
  - 👉 Windows likes to use *button capability array* and *value capability array* to represent data from HID reports.
  - 👉 Equivalent structures from Windows API are provided.

The first thing that is striking when reading the Microsoft documentation on HID interface API is the re-definition of the vocabulary between the Microsoft documentation and the USB-IF documentation. Usage Page and Usage ID still remain the same but Aliased Usages do not. These ones refer to the HID tag item (input, output, feature) that could be shared between different reports in a collection. Far from being common, we will not them for the sake of simplicity. One of the most important concept redefined is the notion of *Top Level Collection* (shorted "TLC" in windows documentation) [624]. In practice, it does not cover the regular notion of item collection but more the one of *application collections* from HID documentation. Top level collection is a group of functionalities that targets a particular software consumer (or type of consumer) for a functionality. For instance, TLC may be described as Keyboard, Mouse, Consumer Control, Sensor, Display and so on. Since a report descriptor can include more than one top-level collection, the system must be able to interact with each. Thanks to HID report descriptor, HID device describes the purpose of each TLC with a specific usage tag. This allows the host to identify TLC in which they might be interested. HID Class driver creates a PDO for each TLC and it ensures the hardware ID associated with the TLC includes an identifier to represent each device object. Taking into account that a collection device provides a PDO for each interface of the device, each of this interface has an HID interface that receives as many PDO as it provides TLC by HID report descriptors.

Top level collection supposes, by its own name, that there are collections which are not *top level*. Indeed, as explained before, devices can embed several applications in a single HID collection. For instance, in a collection, a keyboard and a pointer can be defined if the keyboard device embeds the pointer. In this case, we have two applications in a single report HID collection. One could ask about HID report descriptor example provided in table 4.12. Actually, this report is considered as a TLC since controls should be grouped together if they are logically related or if they are functionally dependent on one another. For instance, a SHIFT key and a character key on a keyboard should not belong to separate collections. An *unnested* collection is always a top-level collection, regardless of its HID type.

But collections can have nested *sub-collections*, also called *link collections* in Microsoft documentation [625]. A top-level collection can have zero or more link collections. A link collection is represented by a `HIDP_LINK_COLLECTION_NODE` structure [626]. This one contains an Usage Page and an Usage ID but also the hierarchical order between all the different link collections. From parents (above link collection which is a TLC for the uppermost) to children, including closest sibling, this structure is used to keep relationships between different link collections. When a TLC contains several link collections, they are all in a *link collection array* composed of `HIDP_LINK_COLLECTION_NODE` structures for each link collection. By tracing through the nodes in the link connection array, it is possible to determine the organization and usage of all the link collections in a top-level collection. An example extracted from [625] illustrates a view of link collections array in Figure 4.39. We can see the different items (values, buttons) combined in different collections and sub-collections.

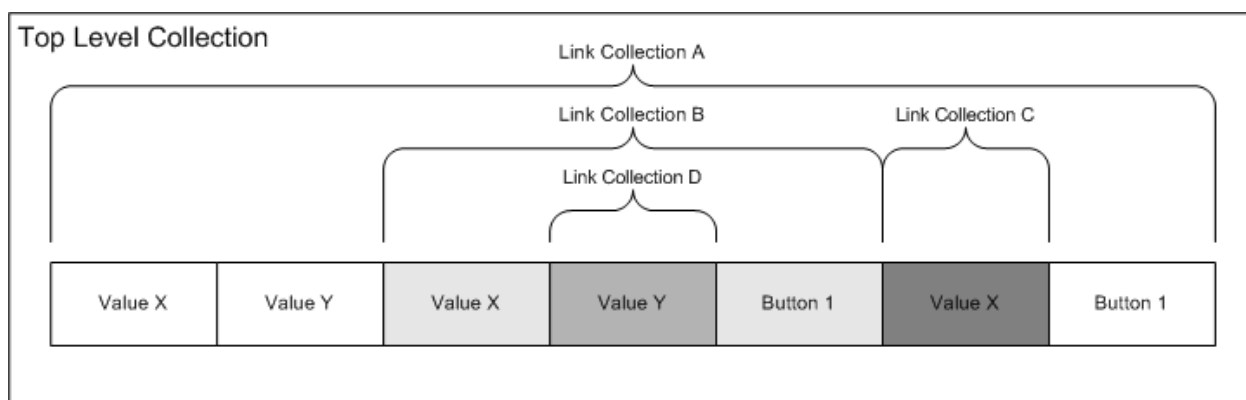


Figure 4.39: Illustration of a link collections array.

Information extracted from HID descriptors via the HID class driver is parsed to TLC and link collections structures so that a client driver can understand the shape of data coming from a device. Windows makes the difference between the button and the value. A button is a control or data item that has a discrete value. For instance, it can be "zero" or "one" for pressed on release but also a key-code (a unique Usage IDs) with Keyboard and LED Usage pages. Any report item that is not a button is a value usage. It remains to deal with values and buttons returned by a device. A *button capability array* [627] and a *value capability array* [628] contain information about the button and the value usages supported by a top-level collection for a specific type of HID report. Both have information about their capabilities contained in an `HIDP_CAPS` structure [629]. This structure gives the expected size from input, output and feature reports in addition to the number of link collections, including the number of values or buttons for input, output and feature. Each button or value is assigned with a data index [630] that uniquely identifies each usage described in a top-level collection. Conceptually, a data index is a zero-based array index that a user-mode application or kernel-mode driver can use to access individual control data in a report. The parser is in charge of assigning a unique set of data indices to each report type supported by each top-level collection.

#### 4.2.4.5 Access to HID parsed information for third-party components

##### Key Point 4.22:

- ☞ This sub-section explains how to use the Windows API (user-mode and kernel-mode) to interface with HID devices.
  - 👉 **User-mode:** After having access to a handle on the desired device (`CreateFile`), we can use the `HidD_Xxx` functions.
  - 👉 **Kernel-mode:** Driver can use `HidP_Xxx` routines or direct IOCTL interface with the device.

How do we technically access these information? First, we need to get access to the *HID Class driver* (`HID-Class.sys`) to operate the device's HID collections. All the details are given in [631].

For user-mode application, it is required to call device installation functions (`SetupDiXxx` functions) [632] to find and identify a HID collection. It is possible to select the class of the targeted device. Technically, there are a lot of device setup classes or device interface classes referenced by GUID values. For instance, it is possible to reference all HID instances present in the system with `GUID_DEVINTERFACE_HID` [633] or `GUID_DEVINTERFACE_KEYBOARD` (`{884b96c3-56ef-11d1-bc8c-00a0c91405dd}`) [634]. The last refers to all keyboards interacting with the system. Note that these GUID are reused in the registry to reference devices with their device name in `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceClasses\`. These GUID are different from those used to register a class driver [568]. Once the list of devices is retrieved thanks to `SetupDiEnumDeviceInterfaces` and `SetupDiGetDeviceInterfaceDetail` [635], it is possible to get access to the device using `CreateFile` [636] with the device name retrieved [637].

For a kernel-mode driver, there are two possibilities. If the driver is a function or filter driver and it is already attached to the device stack, it is automatic. It means it is directly notified for each device. The `AddDevice` routine or `IRP_MJ_CREATE` are two possible places to handle such notifications. If the driver is not attached to the device stack, the driver can use Plug and Play notifications [638] to know how to register and how to handle such notifications [639, 640, 640].

Once the access is guaranteed to the HID class driver, it makes sense to retrieve parsed data [641]. To processed, a user-mode application uses `HidD_GetPreparsedData` routine [642]. A kernel-mode driver needs to use IOCTLs to proceed. First by using `IOCTL_HID_GET_COLLECTION_INFORMATION` [643] to retrieve a `HID_COLLECTION_INFORMATION` structure [644] where the first member (`DescriptorSize`) provides the length, in bytes, of a collection's preparsed data. Once the size has been retrieved and a buffer allocated, it is possible to use a `IOCTL_HID_GET_COLLECTION_DESCRIPTOR` IOCTL [645] to retrieve a `HIDP_PREPARSED_DATA` [646]. This last structure is opaque and undocumented but used by other `HidPxxx` API routines as a parameter.

With the preparsed data, kernel-mode driver can call `HidP_GetCaps` [647] to retrieve a `HID_CAPS` structure that summarizes a top-level collection's capability [648]. Inside this structure, it is defined the number of buttons and values for each input, output, and feature report types. When we are looking for buttons from input report type, because we know the size of `HID_BUTTON_CAPS` [649] and the number of buttons thanks to `HID_CAPS`, it is possible to allocate the memory required to store the array of buttons. Then, by using `HidP_GetButtonCaps` [650], we have access to the list of buttons defined. It is possible to get an extended level of control to return only the buttons which meet a set of specific criteria with `HidP_GetSpecificButtonCaps` [651]. It is in `HID_BUTTON_CAPS` structure that all information about buttons defined in the HID report descriptor are present (range of values, size, bit-field, etc.). The same principles apply for values with `HIDP_VALUE_CAPS` structure [652] and `HidP_GetValueCaps` [653] or `HidP_GetSpecificValueCaps` [654] routines.

Once an application knows how to interact with buttons and values provided by a HID device, it is possible to handle HID reports. Technically, whether an application comes from the user-mode or the kernel-mode, the last has the possibility to obtain HID reports from, or send reports to, the device itself. For user-mode appli-

cations, the use of `HidD_GetInputReport` [655], `HidD_SetFeature` [656] and `HidD_SetOutputReport` [657] functions makes the operation quite easy. For further reading, it is possible to find examples and tutorials about driving specific HID devices vendors online [658].

For kernel-mode drivers, if the documentation exists, this one is partial and not easy to understand. An input report (from the device) is obtained by using an `IOCTL_HID_GET_INPUT_REPORT` code [659]. As input, driver initializes the IRP structure so that an output buffer is allocated to cover the length specified by the targeted report type in `HID_CAPS`. In addition, an extra byte can be included if the collection includes report IDs. In this case, the first byte of the report is set to this report ID by the calling driver. When the request has been successfully executed, the output buffer holds the input buffer provided by the device. When reports ID are used as inputs, the first byte is unchanged and the input report returned belongs right after that byte. Sending data to the device is feasible in both case of feature and output requests. The procedure is quite similar than the one used for input requests. Driver uses `IOCTL_HID_SET_FEATURE` [660] and `IOCTL_HID_SET_OUTPUT_REPORT` [661] codes to perform requests. The difference with the input request lies in the way the request is generated. As it is no longer a buffer that we receive but one that we send, the output (or feature) report is in the input buffer of the request. In the same way, if a report ID is used, it is written as the first byte in the buffer, followed by the report itself whose size has been provided in the `HID_CAPS` structure. Particularity of report requests (and for the sake of completeness), mini-drivers use a `HID_XFER_PACKET` structure [662] to format a report buffer request. As a facilitator when sending report to the device, `HidP_InitializeReportForID` [663] routine can be used to set all control data to zero or a control's null value, as defined by the USB HID standard. Then, driver can manipulate data in the report buffer. In simple case, with `HidP_SetUsages` [664] routine to set a button to 1. In most complex case, driver must use information retrieved from link collections (with `HidP_GetLinkCollectionNodes` [665] and `HIDP_LINK_COLLECTION_NODE` structure [666]) or top collection level to know how data is handled in the targeted report.

#### 4.2.5 Enforcing a Secure Read For a HID Collection

##### Key Point 4.23:

- It is possible to restrict access to HID devices only to processes acting with the same privilege as the operating system (*SeTcbPrivilege*).

For the security purpose, it could make sense to restrict access to a device by enforcing a secure read for a HID collection only to "trusted" clients [667]. For short, trusted clients in Windows are processes that have *SeTcbPrivilege* privilege [668]. This privilege allows them to act as part of the operating system. If a secure read is enabled for a given collection, only processes with this privilege can get access to inputs from that collection. Of course, kernel-mode drivers have *SeTcbPrivilege* privilege by default. The goal is to prevent user-mode applications that are not "trusted" (that is to say not having the required privilege) to get access to the input from a collection. This solution can be used during critical system operations. For instance, this could prevent a third-party application to get access to confidential information during an authentication operation with the device or an exchange of cipher keys. Hence, "trusted" clients use `IOCTL_HID_ENABLE_SECURE_READ` and `IOCTL_HID_DISABLE_SECURE_READ` device I/O requests to enable or disable a secure read for a collection. Without *SeTcbPrivilege* privilege, such requests are inoperative. When the secure read is active, no process interacting with the device is notified anymore about inputs from the collection.

#### 4.2.6 Data transfer between HID device keyboard and host's keyboard handler

##### Key Point 4.24:

- ☞ Selection of keyboard devices managed by the driver is done in `AddDevice` routine at system initialization time or when a device is plugged.
- ☞ There is a scan code set specific to HID devices (which is defined for keyboards at boot time).
- ☞ For backward compatibility purposes with the PS/2, Microsoft translates internally the codes coming from the HID keyboards into scan code set 1 coming from the PS/2.
- ☞ This operation is performed in `kbdhid.sys` driver with `HidP_TranslateUsageAndPagesToI8042ScanCodes` routine in two cases:
  - ☞ With `KbdHid_ReadComplete` routine when a key is read.
  - ☞ With `KbdHid_AutoRepeat` routine when the key pressed is repeated.
- ☞ To get access to the keystroke, HID driver must read from the device.
  - ☞ The reading operation is engaged by the driver which waits until a key is pressed (i.e. the reading order goes down to the device).
  - ☞ Access to the key code is only possible once the reading operation has been completed (i.e. the reading order is sent back to the driver).
- ☞ A read IRP is always pending due to `KbdHid_InitiateStartRead` routine which (re)-engages the reading IRP once a read operation has succeed.
  - ☞ During the re-initialization of the read IRP, `KbdHid_ReadComplete` routine is registered to be called once all underlying drivers in the device stack have finished to process the IRP.
  - ☞ When a read operation on the HID device is performed, `KbdHid_ReadComplete` gets access to the keystroke scan code.

How does the HID keyboard transfer data to the host? Technically, USB HID keyboards send information within HID reports thanks to HID report descriptors. But it remains that code values from keystrokes are sent through reports. Regardless specific vendor codes, codes used are different from those in PS/2 (section 3.2 and table 4.1). Indeed, keyboard manufacturers usually use the ones defined in HID documentation [591]. This code is defined for boot usages and with a North American Keyboard. Microsoft provides an official list of keystroke codes it supports [669]. This list is composed of the three scan codes sets previously exposed for PS/2 and the USB set from the HID definition [591]. But Windows deals with both PS/2 keyboards and USB HID keyboards. Since HID technology emerged after PS/2, Microsoft has chosen to maintain backward compatibility with PS/2 by translating HID data into scan code set 1 [669].

This operation is performed in `kbdhid.sys` driver, responsible to manage all HID keyboard devices. In practice, `kbdhid.sys` driver is linked to `hidclass.sys` (itself supplied by `usbhid.sys`) via its import address table. The driver `kbdhid.sys` imports from `hidparse.sys` `HidP_TranslateUsageAndPagesToI8042ScanCodes` routine [670]. This routine is quite undocumented with the notable exception of its prototype, which is nevertheless not very helpful. But thanks to its name, this routine is interesting since it provides to the HID driver a capacity to translate any HID code to a code that `i8042prt.sys` drive would have been able to use. But before diving in this routine, let us see how this one is recorded in `kbdhid.sys`. Routine `HidP_TranslateUsageAndPagesToI8042ScanCodes` is called from two undocumented routines: `KbdHid_AutoRepeat` and `KbdHid_ReadComplete` (Figure 4.40).

The `KbdHid_AutoRepeat` routine — as its name implies — manages repetition of a keystroke which would be continually pressed. Its code is provided in Figure 4.41. We can see first the use of `KbdHid_S0IdleStateUpdate` routine. For the sake of simplicity, this routine can be used to wait for repetition to occur. Then, after acquiring a lock (to prevent multiple access to a shared resource), it calls `HidP_TranslateUsageAndPagesToI8042ScanCodes`

Direction	Type	Address	Text
Up	p	KbdHid_ReadComplete+439	call cs:_imp_HidP_TranslateUsageAndPagesTol8042ScanCodes
Up	p	KbdHid_ReadComplete+4A6	call cs:_imp_HidP_TranslateUsageAndPagesTol8042ScanCodes
Up	p	KbdHid_AutoRepeat+B2	call cs:_imp_HidP_TranslateUsageAndPagesTol8042ScanCodes
Up	r	KbdHid_ReadComplete+439	call cs:_imp_HidP_TranslateUsageAndPagesTol8042ScanCodes
Up	r	KbdHid_ReadComplete+4A6	call cs:_imp_HidP_TranslateUsageAndPagesTol8042ScanCodes
Up	r	KbdHid_AutoRepeat+B2	call cs:_imp_HidP_TranslateUsageAndPagesTol8042ScanCodes
Do...	o	.idata:00000001C000A2F0	dd rva _imp_HidP_TranslateUsageAndPagesTol8042ScanCodes; Import Address Table

Figure 4.40: Calls to HidP\_TranslateUsageAndPagesTol8042ScanCodes routine in kbdhid.sys driver.

routine with parameters retrieved from the context with whom this routine was recorded for notification. Then comes the release of the lock and WPP tracing<sup>9</sup> [671]. What matters is the way the KbdHid\_S0IdleStateUpdate routine is recorded.

```

1|void __fastcall KbdHid_AutoRepeat(KDPC *Dpc, PUOID DeferredContext, PUOID SystemArgument1, PUOID SystemArgument2)
2|{
3|    PUOID LocalContext; // rbx@1
4|    struct _DEVICE_EXTENSION *DeviceContext; // rdi@1
5|    struct _USAGE_AND_PAGE *ChangedUsageList; // rcx@5
6|
7|    LocalContext = (PUOID)*((_QWORD *)DeferredContext + 14);
8|    DeviceContext = (struct _DEVICE_EXTENSION *)DeferredContext;
9|    if ( WPP_RECORDER_INITIALIZED != &WPP_RECORDER_INITIALIZED && LOWORD(WPP_GLOBAL_Control->DeviceType) )
10|        WPP_RECORDER_SF(WPP_GLOBAL_Control->DeviceExtension);
11|    KbdHid_S0IdleStateUpdate(DeviceContext, 1); // Update device power management context and use KbdHid_S0IdleStateChangeWorkItem.
12|    if ( KeTryToAcquireSpinLockAtDpcLevel((PKSPIN_LOCK)LocalContext + 17) == 1 ) // Get access to the lock.
13|    {
14|        ChangedUsageList = (struct _USAGE_AND_PAGE *)*((_QWORD *)LocalContext + 16); // Retrieve the UsageAndPage.
15|        if ( ChangedUsageList->Usage )
16|            HidP_TranslateUsageAndPagesToI8042ScanCodes(
17|                ChangedUsageList, // _In_ PUSAGE_AND_PAGE ChangedUsageList,
18|                *((_DWORD *)LocalContext + 18), // _In_ ULONG UsageListLength,
19|                HidP_Keyboard_Nake, // _In_ PUSAGE_AND_PAGE ChangedUsageList,
20|                (PHIDP_KEYBOARD_MODIFIER_STATE)LocalContext + 19, // _Inout_ PHIDP_KEYBOARD_MODIFIER_STATE ModifierState,
21|                (PHIDP_INSERT_SCANCODES)KbdHid_InsertCodesIntoQueue, // _In_ PHIDP_INSERT_SCANCODES InsertCodesProcedure,
22|                DeviceContext); // _In_ PUOID InsertCodesContext
23|        KeReleaseSpinLockFromDpcLevel((PKSPIN_LOCK)LocalContext + 17);
24|    } // Release the lock.
25|    if ( WPP_RECORDER_INITIALIZED != &WPP_RECORDER_INITIALIZED )
26|    {
27|        if ( LOWORD(WPP_GLOBAL_Control->DeviceType) )
28|            WPP_RECORDER_SF(WPP_GLOBAL_Control->DeviceExtension);
29|    }
30|}

```

Figure 4.41: Pseudo-code of KbdHid\_AutoRepeat routine from kbdhid.sys driver.

This routine is recorded in the KbdHid\_AddDevice routine. This one is directly linked to the DRIVER\_OBJECT [672] in the driver entry point of the driver, more precisely in the DriverExtension->AddDevice field from the DriverObject into which a driver's DriverEntry routine stores the driver's AddDevice routine [673]. The AddDevice routine creates one or more device objects (FDO or filter DO) representing the physical, logical, or virtual devices (enumerated by the Plug and Play manager) for which the driver carries out I/O requests. It also attaches the device object to the device stack, so that the device stack will contain a device object for each driver associated with the device. The PnP manager calls a driver's AddDevice routine at system initialization for each device controlled by the driver and when a new device is plugged in the system [674].

An AddDevice routine's primary responsibility is to call IoCreateDevice [675, 676] to create a device object. Then it calls IoAttachDeviceToDeviceStack [677] to attach the caller's device object to the highest device object in the device stack. Afterwards, an initialization of several components in the device object which are vendor or device specific. And finally, this is what the KbdHid\_AddDevice routine does. Its pseudo code given in Figure 4.42 keeps only relevant parts for us.

First, we can see calls to IoCreateDevice and IoAttachDeviceToDeviceStack routines, as expected. These ones are performed in order to create an object linked with a device type (FILE\_DEVICE\_KEYBOARD = 0x0b)

<sup>9</sup>WPP tracing is used to trace operations of a software component such as a kernel driver. For the sake of simplicity, this tracing is not including in our analysis.



```

1 |_int64 __fastcall KbdHid_AddDevice(PDRIVER_OBJECT IoObject, PDEVICE_OBJECT TargetDevice)
2 |{
3 |    _DEVICE_OBJECT *TargetDevice_1; // r14h
4 |    PDEVICE_EXTENSION DeviceExtension; // rdi
5 |    struct _DRIVER_OBJECT *IoObject_1; // rbx
6 |    void **v5; // r15
7 |    NTSTATUS status; // esi
8 |    signed __int64 v7; // rdx
9 |    _DEVICE_OBJECT *AttachedDevice; // rax
10 |    _DWORD *LogEntry; // rax
11 |    WORD NbDeviceLinked; // ax
12 |   _IRP *AllocatedIrp; // rax
13 |    WORD NbDevicesLinked; // ax
14 |    WORD NbDevicesLinked_2; // ax
15 |    PIO_WORKITEM IoWorkItem; // rax
16 |    _DEVICE_OBJECT *AttachedDevice_1; // rcx
17 |    _IRP *AllocatedIrp_1; // rcx
18 |    PDEVICE_OBJECT KeyboardDevice; // [rsp+80h] [rbp+40h]
19 |
20 |    TargetDevice_1 = TargetDevice;
21 |    DeviceExtension = 0i64;
22 |    KeyboardDevice = 0i64;
23 |    IoObject_1 = IoObject;
24 |    v5 = &WPP_RECORDER_INITIALIZED;
25 |    if ( WPP_RECORDER_INITIALIZED != &WPP_RECORDER_INITIALIZED && LOWORD(WPP_GLOBAL_Control->DeviceType) )
26 |        WPP_RECORDER_SF (WPP_GLOBAL_Control->DeviceExtension);
27 |    status = IoCreateDevice(IoObject_1, 0x328u, 0i64, FILE_DEVICE_KEYBOARD, 0, 0, &KeyboardDevice);
28 |    if ( status < 0 )
29 |    {
30 |        if ( WPP_RECORDER_INITIALIZED == &WPP_RECORDER_INITIALIZED )
31 |            goto LABEL_20;
32 |        goto LABEL_6;
33 |    }
34 |    DeviceExtension = (PDEVICE_EXTENSION)KeyboardDevice->DeviceExtension;
35 |    memset((KeyboardDevice->DeviceExtension, 0, 0x328ui64);
36 |    AttachedDevice = IoAttachDeviceToDeviceStack(KeyboardDevice, TargetDevice_1);
37 |    DeviceExtension->AttachedDevice = AttachedDevice;
38 |    if ( !AttachedDevice )
39 |    {
40 |        -----
41 |        NbDeviceLinked = InterlockedExchangeAdd((volatile signed __int32 *)&NbDevicesLinked, 1u) + 1; // How many keyboards are currently present.
42 |        DeviceExtension->OriginalTargetDevice = TargetDevice_1;
43 |        DeviceExtension->NbDeviceLinked = NbDeviceLinked;
44 |        KeInitializeSpinLock(&DeviceExtension->SpinLock);
45 |        AllocatedIrp = IoAllocateIrp(DeviceExtension->AttachedDevice->StackSize, 0); // Intermediate drivers -> Quota of memory not charged on the current process.
46 |        DeviceExtension->AllocatedIrp = AllocatedIrp; // Keep track of the allocated Irp for this device.
47 |        if ( AllocatedIrp )
48 |        {
49 |            KeInitializeEvent(&DeviceExtension->SynchroEvent, SynchronizationEvent, 0);
50 |            KeInitializeEvent(&DeviceExtension->NotificationEvent, 0, 1u);
51 |            IoInitializeRemoveLockEx(&DeviceExtension->RemoveLock, 'Hdbk', 1u, 0, 0x20u);
52 |            DeviceExtension->ReservedZero_1 = 0i64;
53 |            DeviceExtension->Const1 = 1;
54 |            DeviceExtension->ReservedZero_2 = 0i64;
55 |            DeviceExtension->ReservedZero_3 = 0;
56 |            KeInitializeEvent(&DeviceExtension->Event, SynchronizationEvent, 0);
57 |            NbDevicesLinked = DeviceExtension->NbDeviceLinked;
58 |            DeviceExtension->KeyboardNumberTotalKeysOverride = 101; // 101-keys keyboard by default.
59 |            -----
60 |            KeInitializeDpc(&DeviceExtension->KbdHid_AutoRepeat, (PKDEFERRED_ROUTINE)KbdHid_AutoRepeat, DeviceExtension); // Manage auto-repeat procedure.
61 |            KeInitializeTimer(&DeviceExtension->Timer);
62 |            DeviceExtension->Timer_DueTime_HighPart = 0xFFFFFFFF;
63 |            DeviceExtension->Timer_Period = 0x21;
64 |            DeviceExtension->Timer_DueTime_LowPart = 0xFFD9DA60;
65 |            KeInitializeDpc( // Initiate a start_read procedure from the device to the host (get input report).
66 |                &DeviceExtension->KbdHid_InitiateStartRead,
67 |                (PKDEFERRED_ROUTINE)KbdHid_InitiateStartRead, // Routine used to handle read (input) operations.
68 |                DeviceExtension);
69 |            KeInitializeTimer(&DeviceExtension->Timer_1);
70 |            -----
71 |            DeviceExtension->ISDTTERRY_KEYBOARDAlreadyLogged = 0;
72 |            DeviceExtension->PowerStateType = 1;
73 |            PoSetPowerState(KeyboardDevice, DevicePowerState, (POWER_STATE)1); // Manage power events.
74 |            DeviceExtension->ClibMgr.GuidCount = 2;
75 |            DeviceExtension->ClibMgr.GuidList = (PWNMIGUIDREGINFO)&KbdHid_WmiGuidList;
76 |            DeviceExtension->ClibMgr.QueryWmiRegInfo = (PWNM_QUERY_REGINFO)KbdHid_QueryWmiRegInfo;
77 |            DeviceExtension->ClibMgr.QueryWmiDataBlock = (PWNM_QUERY_DATABLOCK)KbdHid_QueryWmiDataBlock;
78 |            DeviceExtension->ClibMgr.SetWmiDataBlock = (PWNM_SET_DATABLOCK)KbdHid_SetWmiDataBlock;
79 |        }
80 |    }

```

Figure 4.42: Pseudo-code of KbdHid\_AddDevice routine from kbdhid.sys driver.

[678] representing the underlying hardware keyboard for the driver. Then comes the allocation of an IRP after having incremented the total number of keyboard devices linked. This operation is performed with `InterlockedExchangeAdd` routine [679] to ensure that the routine is thread-safe. Indeed, a call to `AddDevice` routine can be performed at the same time by different threads, meaning race-conditions could occur.

Subsequently, the allocated IRP will be used in order to engage the polling protocol used to communicate with an USB device (as explained in section 4.1.4). If this IRP is correctly allocated, the driver initializes different fields in a driver’s self-defined structure linked to the device. This structure owned by the driver is called `Device Extension` [680]. This one makes sense only for `kbdhid.sys` driver and subsequent drivers using the device object crafted (and where device extension has been linked). It is in the device extension that a *deferred procedure calls* (aka *DPC* routine) [526, 681] is used to record the `KbdHid_AutoRepeat` routine. Originally, DPC are used to postpone the completion of an interrupt until after the ISR returns (since ISR must return as soon as possible [518]). Therefore, the system provides the support for deferred procedure calls, which can be queued from ISRs and which are executed later on a lower IRQL than the ISR. Initialization of the DPC is performed via `KeInitializeDpc` [682] routine (line 95 on Figure 4.42). More information about completion routines and I/O system on Windows is given in [683].



As presented in `AddDevice` routine, one could think that the registration of the DPC uses a *CustomDpc* [684] to finish the servicing of an input or output operation. This is usual when the DPC is used to finish the servicing of an I/O operation. Actually it is a *CustomTimerDpc* [684] which is used to execute the routine after a timer object's time interval expires. Such DPC is used to be executed after a timer object's time interval expires. It fits perfectly the requirements to handle a polled input request for a device (repeating the request many times in a time-slice). Finally, several initialization operations will be performed on the device context in `AddDevice` routine. The first one is done within the power management context thanks to `DevicePowerState` routine which is recorded with `PoSetPowerState` [685] in order to help the driver to be notified when a change in the device power state for a device occurs (line 108 on Figure 4.42). Other actions are performed to record WMI in the driver [686] and there is another `KelInitializeDpc` call to record a *CustomDpc* with the `KbdHid_InitiateStartRead` routine (line 100 on Figure 4.42).

The routine `KbdHid_InitiateStartRead` has a central role since it is responsible to always engage an IRP to handle input interrupts from the device and to deal with those completed. Note that when a read IRP is engaged with nothing to read, `STATUS_PENDING` status is returned to notify that the operation is pending. Pseudo code of `KbdHid_InitiateStartRead` is not relevant since it almost only calls `KbdHid_StartRead` routine. Its code is given in Figure 4.43. First, `KbdHid_InitiateStartRead` reuses the IRP (with `IoReuseIrp` [687] line 28 in Figure 4.43) previously allocated in the context of the `AddDevice` procedure. It is not a big deal to reuse here the same IRP to handle requests from the device. On the first hand, because the code is protected by different locks to avoid deadlocks and a concurrent access to a single resource by several threads. On the other hand, because HID devices, driven by human, do not produce reports fast enough to stall the IRP procedure.

After having re-initialized the IRP, this one is rearmed by initializing its content for a read operation (the input request from the device, from USB point of view) on lines 29 to 35. Then comes at line 36 a call to `IoSetCompletionRoutineEx` [688] to register an *IoCompletion routine* [689, 690] which takes place when the next-lower-level driver has completed the requested operation for the given IRP. For short, the registered routine `KbdHid_ReadComplete` will be called once all underlying drivers in the device stack will have finished processing the IRP [691]. From another context, it can be seen as a post-callback routine [692] once the IRP has been completed.

From that point and after having resetting or setting different events (*NotificationEvent* is mostly used to broadcast that an input report has been received), a call to `IoCallDriver` [693] routine is performed line 56. This last one is responsible to send the IRP to the driver associated with a specified device object. In our case, it corresponds to the previous highest device object linked in the keyboard chain (that should be close to the transport driver, either USB or Bluetooth for the most common ones). This mechanism allows a notification of all subsequent drivers in the keyboard device stack. Once the IRP has been completed (which means that all members of the driver stack completed the I/O operation going down to the device [694]), an event notification is signaled line 63 to inform the system that a read operation has succeeded. From that point, a lock is removed and the IRP is rearmed<sup>10</sup>, the lock reset and the IRP re-engaged for read operation. This "infinite loop while success" is present to allow the notification mechanism to poll the UBS bus with IRP. Interesting fact, `KbdHid_StartRead` routine is used in `IRP_MJ_CREATE` operation (in `KbdHid_Create` routine) in a similar way as it is used in the context of `AddDevice`. In addition, in specific cases, it can be called from the completion routine `KbdHid_ReadComplete` to rearm the read IRP request from its completion routine.

Relevant point is that `KbdHid_ReadComplete` is mentioned in Figure 4.40 as using `HidPTranslateUsageAndPagesToI8042ScanCodes` routine too. Completion routine is interesting in a context of a read operation since it allows to get access to the HID report sent by the device. Indeed, `KbdHid_StartRead` is about to arm and send the IRP but not to get access to the content of the last. Hence, underlying drivers must complete the IRP when the device is about to send data. Processing a completion routine is far from being obvious and requires many actions [695]. This is why we detail only relevant parts of the pseudo code of `KbdHid_ReadComplete` which matters for handling keyboard devices.

---

<sup>10</sup>Technically speaking, there is an hidden *continue* statement at line 58 which is executed if the read operation succeed. We kept it hidden for the sake of simplicity in the pseudo-code presented in Figure 4.43.

```

1  int64 __fastcall KbdHid_StartRead(PDEVICE_EXTENSION Context, __int64 a2)
2  {
3  PIRP Irp; // rsi@1
4  PDEVICE_EXTENSION Context_1; // rbx@1
5  NTSTATUS status; // edi@1
6  __int64 InternalStruct; // r14@4
7  NTSTATUS (__stdcall *KbdHid_ReadComplete)(PDEVICE_OBJECT, PIRP, PUOID); // rbp@5
8  PIO_STACK_LOCATION v7; // rcx@7
9  struct _IO_STACK_LOCATION *StackLocation; // rax@8
10 char v9; // r8@11
11 PDEVICE_OBJECT InvokeOnError; // ST28_8@14
12 int v12; // edx@20
13 char v13; // r8@20
14 int v14; // er9@20
15 PDEVICE_OBJECT v15; // ST28_8@24
16 char InvokeOnSuccess; // [rsp+20h][rbp-38h]@7
17
18 Irp = Context->AllocatedIrp;
19 Context_1 = Context;
20 status = Irp->IoStatus.Status;
21 if...
22 InternalStruct = Context_1->ReservedC[0];
23 if ( _InterlockedExchange((volatile signed __int32 *)&Context_1->ReservedA, 1) == 2 )
24 {
25     KbdHid_ReadComplete = (NTSTATUS (__stdcall *) (PDEVICE_OBJECT, PIRP, PUOID))::KbdHid_ReadComplete;
26     while ( status >= 0 ) // Infinite loop while IRP operations success.
27     {
28         IoReuseIrp(Irp, 0);
29         v7 = (PIO_STACK_LOCATION)Irp->Tail.Overlay.CurrentStackLocation;
30         Irp->MdlAddress = *(PMDL *) (InternalStruct + 144);
31         v7[-1].Parameters.Read.Length = *(unsigned __int16 *) (InternalStruct + 12);
32         v7[-1].Parameters.Read.Key = 0;
33         v7[-1].Parameters.Read.ByteOffset.QuadPart = 0i64;
34         v7[-1].MajorFunction = IRP_MJ_READ; // Definitely read operation.
35         v7[-1].FileObject = (PFILE_OBJECT)Context_1->IrpFileObject;
36         if ( IoSetCompletionRoutineEx(Context_1->KeyboardDevice, Irp, KbdHid_ReadComplete, Context_1, 1u, 1u, 1u) < 0 )//
37             // Registers an IoCompletion routine, which is called when the
38             // next-lower-level driver has completed the requested operation
39             // for the given IRP.
40         {
41             StackLocation = (struct _IO_STACK_LOCATION *)Irp->Tail.Overlay.CurrentStackLocation;
42             StackLocation[-1].CompletionRoutine = KbdHid_ReadComplete;
43             StackLocation[-1].Context = Context_1;
44             StackLocation[-1].Control = -32;
45         }
46         KeResetEvent(&Context_1->NotificationEvent);
47         if ( !( (_DWORD *) &Context_1->PowerStateType + 1) || HIBYTE(Context_1->DeviceState) == 1 )
48         {
49             IoReleaseRemoveLockEx(&Context_1->RemoveLock, Context_1->AllocatedIrp, 0x200u);
50             status = HIBYTE(Context_1->DeviceState) != 0 ? STATUS_DELETE_PENDING : STATUS_UNSUCCESSFUL;
51             if...
52             KeSetEvent(&Context_1->NotificationEvent, 0, 0);
53             KeSetEvent(&Context_1->SynchroEvent, 0, 0);
54             goto __leave;
55         }
56         status = IoCallDriver(Context_1->AttachedDevice, Irp); // Sends an IRP to the driver associated with a specified device object.
57         KeSetEvent(&Context_1->NotificationEvent, 0, 0);
58         if...
59         goto __leave;
60     }
61     if...
62     KeSetEvent(&Context_1->SynchroEvent, 0, 0);
63     IoReleaseRemoveLockEx(&Context_1->RemoveLock, Context_1->AllocatedIrp, 0x200u);
64 }
65 __leave:
66 if...
67 return (unsigned int)status;
68 }

```

Figure 4.43: Pseudo-code of KbdHid\_StartRead routine from kbdhid.sys driver.

The first part of KbdHid\_ReadComplete (Figure 4.44) checks if the IRP has been completed successfully (with `Irp->status`) or if it has not been canceled for various reasons (including the keyboard device has been removed). If everything is correct, our completion routine will call `HidP_GetUsagesEx` [696] routine (line 68). This one is a HID command used to return a list of all the HID control button usages that are set to ON (i.e. they are equal to 1) in a HID report. That is to say, this routine, taking as parameters the *Prepared Data* [641] and the type of report (input report) from the top level collection returns a *Button List* which holds the usage and usage page identifiers for each button that is set to ON. Thanks to `hidparse.sys` driver, all information have been parsed and reports are correctly initialized in memory.

Interesting point following `HidP_GetUsagesEx` routine call, Figure 4.45 gives a specific condition for debuggers. Indeed, at initialization time, `kbdhid.sys` calls `Kbdhid_ServiceParameters` routine in its driver entry point. This routine reads several values in the driver's registry hive (`HKLM\SYSTEM\CurrentControlSet\Services\kbdhid\Parameters`). Among the relevant values read, we have `BreakOnSysRq`, `CrashOnCtrlScroll` and `DisableAutoRepeat`. When a special key is pressed and if `BreakOnSysRq` value is set to one in the Windows' registry, the driver is about to give the control to a debugger if one is connected to Windows.

```

39 | status_1 = Irp_1->IoStatus.Status;
40 | InternalStruct = (_USAGE_REPORT *)DeviceExtension->ReservedC[0];
41 | InternalStruct_1 = DeviceExtension->ReservedC[0];
42 | u9 = 3i64;
43 | Value3 = _InterlockedCompareExchange((volatile signed __int32 *)&DeviceExtension->ReservedA, 3, 1);
44 | IsValue3 = Value3 != 3;
45 | IsValue3_1 = Value3 != 3;
46 | if ( !*((_DWORD *)&DeviceExtension->PowerStateType + 1) )
47 | {
48 |     if...
49 |     goto LABEL_7;
50 | }
51 | if ( status_1 == STATUS_DELETE_PENDING
52 |     || status_1 == STATUS_PRIVILEGE_NOT_HELD
53 |     || status_1 == STATUS_DEVICE_NOT_CONNECTED
54 |     || status_1 == STATUS_CANCELLED )
55 | {
56 | LABEL_7:
57 |     if ( IsValue3 )
58 |     {
59 |         KeSetEvent(&DeviceExtension->SynchroEvent, 0, 0);
60 |         IoReleaseRemoveLockEx(&DeviceExtension->RemoveLock, DeviceExtension->AllocatedIrp, 0x20u);
61 |         IsValue3 = 0;
62 |     }
63 |     goto __leave_1;
64 | }
65 | if ( status_1
66 |     || !InternalStruct->UsageLength
67 |     || (UsageLength = InternalStruct->UsageLength,
68 |         HidP_GetUsagesEx(
69 |             HidP_Input,
70 |             0, // Returns information about all the buttons in the top-level
71 |             // collection associated with PreparedData.
72 |             InternalStruct->ButtonList,
73 |             &UsageLength,
74 |             InternalStruct->PreparedData,
75 |             InternalStruct->Report,
76 |             InternalStruct->ReportLength) < 0) )
77 | {
78 | __leave_1:
79 |     CurrentIrql = 0;
80 |     goto __leave;
81 | }

```

Figure 4.44: Pseudo-code of the beginning of KbdHid\_ReadComplete routine from kbdhid.sys driver.

Interestingly, information in *Button List* is accessed directly<sup>11</sup> to check if UsagePage is 0x07 (HID keyboard usage page) and Usage equal to 0x46. According to [697], such a combination of keystrokes corresponds to "Print Screen" key name. This behavior should be present to handle local debuggers when the boot procedure must be analyzed. Note that it is not possible to break before. Indeed, since the debugging is performed with a keyboard, any malfunction may make the keyboard unusable.

```

82 | if ( UsageLength == 1 )
83 | {
84 |     Usage = InternalStruct->ButtonList;
85 |     if ( 7 == Usage->UsagePage
86 |         && 0x46 == Usage->Usage // 0x07 0x46 => Print Screen.
87 |         && BreakOnSysRq
88 |         && (_BYTE)KdDebuggerEnabled
89 |         && !KdRefreshDebuggerNotPresent() )
90 |     {
91 |         DbgBreakPointWithStatus(2i64);
92 |     }
93 | }

```

Figure 4.45: Pseudo-code about debug purposes in KbdHid\_ReadComplete routine from kbdhid.sys driver.

The next part (Figure 4.46) of KbdHid\_ReadComplete is about to know which keys have been pressed. To

<sup>11</sup>Such behavior tends to show that there is not so much flexibility in the way to forge custom or exotic HID report descriptors while being perfectly supported by Windows.

proceed, `KbdHid_ReadComplete` uses `HidP_UsageAndPageListDifference` routine. This one is not documented and described as *“not implemented”* by official Windows documentation [698] but there is an old documentation about the routine [699] which helps a bit more to understand how it works. For the sake of simplicity and because it is prone to change in the future, we can say that the main purpose of this routine is to return the difference between two arrays of HID extended usages. It helps to observe any changes in button states between two usage lists returned by two `HidP_GetButtonsEx`. If there have been any changes, the routine calls `HidP_TranslateUsageAndPagesToI8042ScanCodes` to translate HID USB code to the scan code used by Windows. That way it is possible to see if the current state of keystroke has been changed from the last time the driver read from the keyboard device.

```

137 if ( HidP_UsageAndPageListDifference(
138     InternalStruct->PreviousUsageList,
139     InternalStruct->ButtonList,           // CurrentUsageList.
140     InternalStruct->BreakUsageList,
141     InternalStruct->MakeUsageList,
142     InternalStruct->UsagLength) < 0 )
143 {
144     IsValue3 = IsValue3_1;
145     v6 = &WPP_RECORDER_INITIALIZED;
146     goto __leave;
147 }
148 if ( (v21 = InternalStruct->MakeUsageList, !*v21) && !*InternalStruct->BreakUsageList
149     || (memmove(v21, InternalStruct->MakeUsageList_1, 4i64 * InternalStruct->UsagLength),
150         HidP_UsageAndPageListDifference(
151             InternalStruct->PreviousUsageList,
152             InternalStruct->ButtonList,           // CurrentUsageList.
153             InternalStruct->BreakUsageList_1,
154             InternalStruct->MakeUsageList_1,
155             InternalStruct->UsagLength) < 0)
156     || HidP_TranslateUsageAndPagesToI8042ScanCodes(
157         InternalStruct->BreakUsageList_1,       // ChangedUsageList
158         InternalStruct->UsagLength,
159         HidP_Keyboard_Break,
160         &InternalStruct->ModifierState,
161         &KbdHid_InsertCodesIntoQueue,         // InsertCodesProcedure
162         DeviceExtension) < 0
163     || (!*InternalStruct->ButtonList || (v22 = InternalStruct->MakeUsageList_1, *v22) ?
164         (status = HidP_TranslateUsageAndPagesToI8042ScanCodes(
165             InternalStruct->MakeUsageList_1,
166             InternalStruct->UsagLength,
167             HidP_Keyboard_Make,
168             &InternalStruct->ModifierState,
169             &KbdHid_InsertCodesIntoQueue,
170             DeviceExtension)) :
171         (status = HidP_UsageAndPageListDifference(InternalStruct->BreakUsageList_1, InternalStruct->MakeUsageList,
172             InternalStruct->BreakUsageList, v22, InternalStruct->UsagLength)),
173         status < 0) )
174 {
175 LABEL_75:
176     IsValue3 = IsValue3_1;
177     goto LABEL_70;
178 }
179 PreviousUsageList = InternalStruct->PreviousUsageList;
180 InternalStruct->PreviousUsageList = InternalStruct->ButtonList;
181 memset(PreviousUsageList, 0, 4i64 * InternalStruct->UsagLength);
182 InternalStruct->ButtonList = PreviousUsageList;

```

Figure 4.46: Pseudo-code about handling changes in keystrokes for `KbdHid_ReadComplete` routine from `kbdhid.sys` driver.

Finally, close to the end of `KbdHid_ReadComplete` routine, there is a specific management for auto repeat keystrokes (Figure 4.47). This one, after checking that auto-repeat feature has not been released (which is just about to set value `DisableAutoRepeat` in `kbdhid.sys` registry drivers’ parameters), transform the DPC `KbdHid_AutoRepeat` into a *CustomTimerDpc* [700]. A *CustomTimerDpc* is a DPC linked to a timer object which has been initialized previously. The system calls the *CustomTimerDpc* routine when the timer interval expires. It is perfect to manage an auto-repeat mechanism since Windows can configure the auto-repeat rate in its control configuration panel<sup>12</sup>. The value is read from the system’s configuration in order to initialize the timer linked to the DPC. In a specific case, `KbdHid_ReadComplete` can transform `KbdHid_InitiateStartRead` DPC routine into

<sup>12</sup>An illustration of the procedure (with the control panel and with the registry) is provided in the *Social Technet Microsoft* forum [701]. This illustrates that the configuration can be performed at different level in Windows and in the end the driver will take care to correctly interface with the hardware device. This principle has been described already by Raymond Chen with PS/2 keyboards [702]. We have documented it for USB/HID keyboards, illustrating new registry keys to use.

a *CustomTimerDpc* almost the same way it does with *KbdHid\_AutoRepeat*. Still, the due time is different (when the timer should expire) and this timer is a non-periodic timer (which does not automatically re-queue itself).

```

176 | PreviousUsageList = InternalStruct->PreviousUsageList;
177 | InternalStruct->PreviousUsageList = InternalStruct->ButtonList;
178 | memset(PreviousUsageList, 0, 4i64 * InternalStruct->UsageLength);
179 | InternalStruct->ButtonList = PreviousUsageList;
180 | if ( UsageLength <= 0 )
181 | {
182 |     cancel_auto_repeat_timer:
183 |     KbdHid_CancelAutoRepeatTimer(DeviceExtension, 1); // Disable AutoRepeat.
184 |     goto __leave_3;
185 | }
186 | if ( 0 != InternalStruct->BreakUsageList_1->Usage || 0 != *InternalStruct->MakeUsageList_1 )
187 | {
188 |     if ( UsageLength > 0 )
189 |     {
190 |         if ( !DisableAutoRepeat )
191 |         KeSetTimerEx(
192 |             &DeviceExtension->Timer, // Pointer to a timer object that was initialized with
193 |             // KeInitializeTimer or KeInitializeTimerEx.
194 |             DeviceExtension->Timer_DueTime, // Specifies the absolute or relative time at which the timer is
195 |             // to expire. If the value of the DueTime parameter is negative,
196 |             // the expiration time is relative to the current system time.
197 |             // Otherwise, the expiration time is absolute. The expiration
198 |             // time is expressed in system time units (100-nanosecond
199 |             // intervals). Absolute expiration times track any changes in the
200 |             // system time; relative expiration times are not affected by
201 |             // system time changes.
202 |             DeviceExtension->Timer_Period, // Specifies an optional recurring interval for the timer in
203 |             // milliseconds. Must be a value that is greater than or equal to
204 |             // zero. If the value of this parameter is zero, the timer is a
205 |             // nonperiodic timer that does not automatically re-queue itself.
206 |             &DeviceExtension->KbdHid_AutoRepeat); // Pointer to a DPC object that was initialized by
207 |             // KeInitializeDpc. This parameter is optional.
208 |         goto __leave_3;
209 |     }
210 |     goto __cancel_auto_repeat_timer;
211 | }

```

Figure 4.47: Pseudo-code about auto-repeat in *KbdHid\_ReadComplete* routine from *kbdhid.sys* driver.

Last but not least, as a bonus, there is a message in the *KbdHid\_ReadComplete* routine (Figure 4.48). It seems that Microsoft has to deal with talkative keyboards which were responsible for potential issues. We can suppose that too many useless reports were notified to the HID driver. Hence, it could flood and hang the system as a potential result. If the driver is able to recognize one of these devices or if it is directly set in the registry path of the driver, there is a log entry which is reported to the system. In addition, a debugger would be notified with a nice message, explaining that it is the user's responsibility to contact such keyboards manufacturers so that they fix issues with their devices. Up to the reader to consider this as a joke or a serious advice...

```

212 | if ( !DeviceExtension->IsCHATTERY_KEYBOARDAlreadyLogged )
213 | {
214 |     DeviceExtension->IsCHATTERY_KEYBOARDAlreadyLogged = 1;
215 |     DbgPrint("*****\n***** CHATTERY_KEYBOARD : Keyboard is sending useless reports. Tell 'em to fix it.\n*****\n");
216 |     LODWORD(DeviceExtension->ReservedC[1]) |= 1u;
217 |     v25 = IoAllocateErrorLogEntry(DeviceExtension->KeyboardDevice->DriverObject, 0x30u);
218 |     v26 = v25;
219 |     if ( v25 )
220 |     {
221 |         memset(v25, 0, 0x30ui64);
222 |         v26[3] = 0x80050001;
223 |         v26[5] = 0x80050001;
224 |         *(v26 + 2) = 0;
225 |         IoWriteErrorLogEntry(v26);
226 |     }
227 | }

```

Figure 4.48: Pseudo-code about talkative keyboards in *KbdHid\_ReadComplete* routine from *kbdhid.sys* driver.

#### 4.2.7 From USB HID code to Windows custom scan code set

##### Resume 24:

- ☞ In this subsection, we explain how the translation of the HID scan code set is done.
  - ☞ For the sake of simplicity, it is a system of callback arrays that perform the translation with tables of corresponding values (as a chart).
- ☞ In this subsection, thanks to the reverse-engineering, we prove two points:
  - ☞ The translation of the HID scan code set is done in scan code set 1 from PS/2.
  - ☞ This translation does not conform to the one provided in scan code set 1 as defined by the PS/2 standard.

##### Key Point 4.25:

- ☞ The scan code used by Windows is in fact an extension of the scan code set 1 (to manage new keystrokes used on modern keyboards).

No matter how the read request is processed on the device, `HidP_TranslateUsageAndPagesToI8042ScanCodes` routine remains central in the processing of the received data. Pseudo-code of this routine is given in Figure 4.49. The first part of this routine is to detect if the usage page type in the report received is about keyboard or consumer page information. In the last case, it corresponds to buttons one might find on small devices (microphone, headphone, dvd player and so on) or specific buttons on keyboards such as Menu, Menu up, Menu down, Power, Reset, Sleep, Pause, Stop, Eject [591]... This information about the usage page allows the routine to select a specific set of callback routines and scan code tables associated to perform the translation. This is with this set of callbacks and tables that the system is able to perform a conversion between USB/HID code and the scan code set used by Windows. We note that debug symbols provided by Microsoft give variable names relative to keyboard. The same thing applies with consumer usage page.

Selection of the set is useful in the call of `HidP_TranslateUsage` routine which is undocumented. This routine is a generic way to translate and transfer information to the keyboard handler driver. Since it is a set of callbacks and scan codes tables selected before the call, the routine acts in a generic way with the arguments provided. As described in its pseudo-code given in Figure 4.50, line 10 is responsible to call the callback lookup provided with the set of scan code tables. Two possibilities: `HidP_AssociativeLookup` or `HidP_StraightLookup`. Both routines are similar in their way of doing things (Figures 4.51 and 4.52). They process two parameters. The first is the scan code table to use and the second is the usage code read from the device's report. Routine `HidP_StraightLookup` is the easiest to explain (but logic remains the same for `HidP_AssociativeLookup`). Usage is checked to be less than `0xFF` (maximal value supported by USB HID code) and if it is the case, usage value is used as an index offset to get access to a value in the scan code table provided (`HidP_KeyboardToScanCodeTable` for keyboards).

Taking into account the content of the `HidP_KeyboardToScanCodeTable` in Figure 4.53, it is possible to explain how the translation is performed. Usage is used as an offset from the base address of the `HidP_KeyboardToScanCodeTable` which is an array of `DWORDs` (32-bit values). Reading directly in this array is enough to perform the linear transformation from one code to another. Looking for Windows supported codes table [669], usage value `0x04` is translated with `HidP_KeyboardToScanCodeTable` (Figure 4.53<sup>13</sup>) into `0x1E`. It turns out that `0x04` in USB HID code corresponds to the character 'a' and that character is encoded by the value `0x1E` in the scan code set 1. Our reverse engineering process confirmed that Windows translates HID keyboard device's codes into scan code set 1 (from PS/2 protocol), as stated in [669] but without any detail.

Coming back to `HidP_TranslateUsage`, the result returned by callbacks `HidP_AssociativeLookup` or `HidP_StraightLookup` is checked in line 11 (Figure 4.50). If no scan code is returned (value 0), `HidP_TranslateUsage` returns

<sup>13</sup>HID code values which corresponds to offsets have been added for the sake of readability in Figure 4.53.



```

1 NTSTATUS __stdcall HidP_TranslateUsageAndPagesToI8042ScanCodes(PUSAGE_AND_PAGE ChangedUsageList,
2 ULONG UsageListLength, HIDP_KEYBOARD_DIRECTION KeyAction, PHIDP_KEYBOARD_MODIFIER_STATE ModifierState,
3 PHIDP_INSERT_SCANCODES InsertCodesProcedure, PVOID InsertCodesContext)
4 {
5     struct _HIDP_KEYBOARD_MODIFIER_STATE *ModifierState_1; // rbp@1
6     HIDP_KEYBOARD_DIRECTION KeyAction_1; // er14@1
7     ULONG UsageListLength_1; // esi@1
8     PUSAGE_AND_PAGE ChangedUsageList_1; // rbx@1
9     NTSTATUS result; // eax@1
10    ULONG indexUsageList; // edi@1
11    USAGE CurrentUsagePage; // ax@3
12    void *callback_hidp_lookup; // r9@5
13    __int64 *ScanCodeTable; // rax@5
14    _QWORD *SubTables; // rcx@5
15
16    ModifierState_1 = ModifierState;
17    KeyAction_1 = KeyAction;
18    UsageListLength_1 = UsageListLength;
19    ChangedUsageList_1 = ChangedUsageList;
20    result = HIDP_STATUS_SUCCESS;
21    indexUsageList = 0;
22    if ( UsageListLength )
23    {
24        while ( 0 != ChangedUsageList_1->Usage )
25        {
26            CurrentUsagePage = ChangedUsageList_1->UsagePage;
27            if ( CurrentUsagePage == 7 ) // Keyboard Page (0x07).
28            {
29                callback_hidp_lookup = HidP_StraightLookup;
30                ScanCodeTable = &HidP_KeyboardToScanCodeTable;
31                SubTables = &HidP_KeyboardSubTables;
32            }
33            else
34            {
35                if ( CurrentUsagePage != 0xC ) // Consumer Page (0x0C) - they affect a specific
36                // device, not the system as a whole.
37                return HIDP_STATUS_I8042_TRANS_UNKNOWN;
38                callback_hidp_lookup = HidP_AssociativeLookup;
39                ScanCodeTable = &HidP_ConsumerToScanCodeTable;
40                SubTables = &HidP_ConsumerSubTables;
41            }
42            result = HidP_TranslateUsage(
43                ChangedUsageList_1->Usage,
44                KeyAction_1,
45                ModifierState_1,
46                callback_hidp_lookup,
47                (__int64)ScanCodeTable,
48                SubTables,
49                (__int64)InsertCodesProcedure,
50                (__int64)InsertCodesContext);
51            if ( result == HIDP_STATUS_SUCCESS )
52            {
53                ++indexUsageList;
54                ++ChangedUsageList_1;
55                if ( indexUsageList < UsageListLength_1 )
56                    continue;
57            }
58            return result;
59        }
60    }
61    return result;
62 }

```

Figure 4.49: Pseudo-code of `HidP_TranslateUsageAndPagesToI8042ScanCodes` from `hidparse.sys` driver.

the internal error code `HIDP_STATUS_I8042_TRANS_UNKNOWN`. In line 13 and 14, in case of specific modifier state key would be active (`HIDP_KEYBOARD_MODIFIER_STATE` [670] — where 0x11 corresponds to *LeftControl* and *RightShift*) and the returned code would be 0x451DE1 (HID Usage 0x48), result 0xE046 must be returned. Technically, entry 0xE046 in scan code set 1 and HID usage 0x48 value both correspond to Break (Ctrl-Pause) key code. This may be a way to generate this key for specific keyboard devices...

The condition given in line 15 is quite interesting. This one checks that the code returned holds 0xF0 value. Actually, there is no way in official scan code set 1 to return such value. But Microsoft uses that trick to extend



```

1 HIDP_STATUS __fastcall HidP_TranslateUsage(USAGE Usage, HIDP_KEYBOARD_DIRECTION KeyAction, PHIDP_KEYBOARD_MODIFIER_STATE ModifierState,
  PVOID callback_hidp_lookup, __int64 ScanCodeTable, PVOID SubTables, __int64 InsertCodesProcedure, __int64 InsertCodesContext)
2 {
3   HIDP_KEYBOARD_DIRECTION KeyAction_1; // edi@1
4   PHIDP_KEYBOARD_MODIFIER_STATE ModifierState_1; // rbx@1
5   unsigned int result_callback; // er10@1
6   PHIDS_SUBTABLES v11; // rcx@6
7
8   KeyAction_1 = KeyAction;
9   ModifierState_1 = ModifierState; // callback_hidp_lookup(ScanCodeTable, Usage);
10  result_callback = _guard_dispatch_icall_fptr(ScanCodeTable, Usage, ModifierState, callback_hidp_lookup);
11  if ( !result_callback )
12  {
13    return HIDP_STATUS_I8042_TRANS_UNKNOWN;
14  }
15  if ( ModifierState_1->ul & 0x11 && result_callback == 0x451DE1 )
16  {
17    result_callback = 0x46E0;
18    if ( (result_callback & 0xF0) == 0xF0u )
19    {
20      SetSubTables = (PHIDS_SUBTABLES)((char *)SubTables + 0x10 * (result_callback & 0xF));
21      if ( !SetSubTables->Callback // This is this callback which is called.
22          || !(unsigned __int8)_guard_dispatch_icall_fptr( // callback_hidp_subtables(CodesSubTable, (ResultCallback >> 8),
23              SetSubTables->CodesSubTable, InsertCodesProcedure, InsertCodesContext)
24              BYTE1(result_callback),
25              InsertCodesProcedure,
26              InsertCodesContext) )
27      {
28        return HIDP_STATUS_I8042_TRANS_UNKNOWN;
29      }
30    }
31  }
32  else
33  {
34    HidP_KbdPutKey(result_callback, KeyAction_1, (PVOID)InsertCodesProcedure, InsertCodesContext);
35  }
36  return HIDP_STATUS_SUCCESS;
37 }

```

Figure 4.50: Pseudo-code of HidP\_TranslateUsage from hidparse.sys driver.

```

1 __int64 __fastcall HidP_AssociativeLookup(LPDWORD ScanCodeTable, USAGE Usage)
2 {
3   __int64 offset; // r8@1
4
5   offset = 0i64;
6   while ( Usage != ScanCodeTable[offset] )
7   {
8     offset = (unsigned int)(offset + 2);
9     if ( (unsigned int)offset >= 0xF )
10    {
11      return 0i64;
12    }
13    return ScanCodeTable[(unsigned int)(offset + 1)];
14 }

```

Figure 4.51: Pseudo code of HidP\_AssociativeLookup routine.

```

1 __int64 __fastcall HidP_StraightLookup(LPDWORD ScanCodeTable, USAGE Usage)
2 {
3   __int64 result; // rax@2
4
5   if ( Usage <= 0xFFu )
6     result = ScanCodeTable[Usage];
7   else
8     result = 0i64;
9   return result;
10 }

```

Figure 4.52: Pseudo code of HidP\_StraightLookup routine.

the scan code set 1 to hold value undefined at the time the scan code set 1 has been released — mainly HID usage page codes. Most of these keys are written as "UNDEFINED" in [669] but they correspond to a value key for USB keyboards. Note that most of these undefined keys cover in practice some specific keys linked to a feature (such as "Keyboard Copy", "Keyboard Paste", "Keyboard Find", "Keyboard Mute", "Keyboard Volume Up", "Keyboard Volume Down"...). Transformation is performed with the help of the sub-table provided. A sub-table is a structure composed of a set of callback routines linked to a scan code table. Dealing with keyboard in HidP\_TranslateUsageAndPagesToI8042ScanCodes routine, the sub-table selected is HidP\_KeyboardSubTables (Figure 4.54). A sub-table is an array of members, where each member is composed of a callback routine pointer and a context (a scan code table) provided to the callback when called. On Figure 4.54, we underline in green callback routines and in red related contexts.

The sub-table linked to the custom scan code defined by Microsoft is selected according to the least significant byte. That one is used as an offset in the array of members in the sub-table (line 17 from Figure 4.50). Once the member in the sub-table has been selected, the callback is executed with four provided parameters (line 19). The first is the context linked to the callback (which corresponds, in practice, to a scan code table), the second is about the resulting code returned by the previous callback call and the third and the fourth parameters are a callback routine responsible to transfer the code to the keyboard driver and a context address related to this callback.

With HidP\_KeyboardKeypadCode callback routine and its scan code table context HidP\_XlateKbdPadCodesSubTable, we observe the same philosophy than the one met with HidP\_StraightLookup and HidP\_KeyboardToScanCodeTable. The goal is to translate an existing code into a new one. It is still a linear transform which is done in HidP\_KeyboardKeypadCode (but the same applies with small differences with other possible callbacks routines).

```

; __int64 HidP_KeyboardToScanCodeTable
.data:00000001C0008190 HidP_KeyboardToScanCodeTable dd 0FFh
.data:00000001C0008190 HID USB Code Scan code set 1
.data:00000001C0008194 001h dd 0FFh
.data:00000001C0008198 002h dd 0FFh
.data:00000001C000819C 003h dd 0FFh
.data:00000001C00081A0 004h dd 1Eh
.data:00000001C00081A4 005h dd 30h
.data:00000001C00081A8 006h dd 2Eh
.data:00000001C00081AC 007h dd 20h
.data:00000001C00081B0 008h dd 12h
.data:00000001C00081B4 009h dd 21h
.data:00000001C00081B8 00ah dd 22h
.data:00000001C00081BC 00bh dd 23h
.data:00000001C00081C0 00ch dd 17h
.data:00000001C00081C4 00dh dd 24h
.data:00000001C00081C8 00fh dd 25h
.data:00000001C00081CC 010h dd 26h
.data:00000001C00081D0 011h dd 32h
.data:00000001C00081D4 012h dd 31h
.data:00000001C00081D8 013h dd 18h
.data:00000001C00081DC 014h dd 19h

```

Figure 4.53: Beginning of the content of HidP\_KeyboardToScanCodeTable.

```

; _QWORD HidP_KeyboardSubTables
HidP_KeyboardSubTables dq offset HidP_KeyboardKeypadCode
dq offset HidP_XlateKbdPadCodesSubTable
dq offset HidP_ModifierCode
dq offset HidP_XlateModifierCodesSubTable
dq offset HidP_VendorBreakCodesAsMakeCodes
dq offset HidP_BreakCodesAsMakeCodesTable
dq offset HidP_PrintScreenCode
dq offset HidP_XlatePrtScrCodesSubTable
align 100h

```

Figure 4.54: Content of the sub-table HidP\_KeyboardSubTables.

The relevant point in HidP\_KeyboardKeypadCode is the use of HidP\_KbdPutKey routine (lines 15, 16 and 19 from Figure 4.55).

The undocumented routine HidP\_KbdPutKey takes four parameters. The first is the translated code value that should be transferred to the system. The second parameter is a HIDP\_KEYBOARD\_DIRECTION [613] value. This one is a collection of two possible values HidP\_Keyboard\_Break and HidP\_Keyboard\_Make. The two values are undocumented and the names are confusing on purpose to avoid a use by a third-party software. But old documentation [703] gives that HidP\_Keyboard\_Break indicates a key press and HidP\_Keyboard\_Make indicates a key release, which is quite useful. The last two parameters are a callback and its context responsible to transfer data to the system. Indeed, HidP\_KbdPutKey is just an intermediary transfer routine. Its pseudo code is given in Figure 4.56. The first part with the loop has the objective aims at setting the most significant bit<sup>14</sup> for each key code to one when the key is pressed.

<sup>14</sup>This bit could explain why information will be given with the same shape with GetAsyncKeyState [704], for instance. Technically, Microsoft uses its own scan code modified to improve the quantity of information transferred from the device to the system.

```

1  BOOLEAN __fastcall HidP_KeyboardKeypadCode(LPDWORD ScanCodeTable, BYTE ConvertedCode, PHIDP_INSERT_SCANCODES InsertCodesProcedure,
2  PVOID InsertCodesContext, HIDP_KEYBOARD_DIRECTION KeyAction, PHIDP_KEYBOARD_MODIFIER_STATE ModifierState)
3  {
4  PVOID InsertCodesContext_1; // rdi@1
5  PHIDP_INSERT_SCANCODES InsertCodesProcedure_1; // rsi@1
6  BYTE Converted_1; // bp@1
7  LPDWORD ScanCodeTable_1; // r15@1
8  ULONG ModifierState_1; // eax@4
9  InsertCodesContext_1 = InsertCodesContext;
10 InsertCodesProcedure_1 = InsertCodesProcedure;
11 Converted_1 = ConvertedCode;
12 ScanCodeTable_1 = ScanCodeTable;
13 if ( ModifierState->ul & 0x400 && KeyAction == HidP_Keyboard_Make )// 0x400 -> NumLock.
14     // 0xE02A is undefined in all codes used (internal code?).
15     HidP_KbdPutKey(0x2AE0164, HidP_Keyboard_Make, InsertCodesProcedure, (__int64)InsertCodesContext); // Use self defined code.
16 HidP_KbdPutKey(ScanCodeTable_1[Converted_1], KeyAction, InsertCodesProcedure_1, (__int64)InsertCodesContext_1); // Use converted code.
17 ModifierState_1 = ModifierState->ul; // Retrieve the modifier state.
18 if ( _bittest(const signed int *)&ModifierState_1, 0xAu) && KeyAction == HidP_Keyboard_Break )// RightShift | RighthGUI
19     HidP_KbdPutKey(0x2AE0164, HidP_Keyboard_Break, InsertCodesProcedure_1, (__int64)InsertCodesContext_1); // Internal code too.
20 return TRUE;
21 }

```

Figure 4.55: Pseudo-code of HidP\_KeyboardKeypadCode routine from hidparse.sys.

```

1  __int64 * __fastcall HidP_KbdPutKey(__int64 ScanCodeConverted, HIDP_KEYBOARD_DIRECTION KeyAction,
2  PHIDP_INSERT_SCANCODES InsertCodesProcedure, __int64 InsertCodesContext)
3  {
4  __int64 *result; // rax@1
5  __int64 Index; // r8@1
6  __int64 UshortResult; // [rsp+30h] [rbp+8h]@1
7  LODWORD(UshortResult) = ScanCodeConverted;
8  result = &UshortResult;
9  Index = 0i64;
10 do
11 {
12     if ( !*( _BYTE *)result )
13         break;
14     if ( KeyAction == HidP_Keyboard_Break )
15         *( _BYTE *)result |= 0x80u; // Set the most significant bit when a key is pressed.
16     Index = (unsigned int)(Index + 1);
17     result = (__int64 *)((char *)result + 1);
18 }
19 while ( (unsigned int)Index < 4 );
20 if ( (_DWORD)Index ) // InsertCodesProcedure(InsertCodesContext, &UshortResult);
21     result = (__int64 *)_guard_dispatch_icall_fptr(InsertCodesContext, &UshortResult, Index, InsertCodesContext);
22 return result;
23 }

```

Figure 4.56: Pseudo-code of HidP\_KbdPutKey routine from hidparse.sys.

At the end, this is the *inserting scan codes callback* which is called. This one is an undocumented PHIDP\_INSERT\_SCANCODES routine [613] which takes three parameters. The first is the context registered with the callback. The second parameter is the scan code or the array of scan codes to transfer to the system. The last is the length of the scan codes transferred in the second parameter. Just as a note, if there was no modifier state in HidP\_TranslateUsage, the HidP\_KbdPutKey routine is directly call from that point with the provided callback. The role of this callback is to transfer the scan code translated from the HID driver to the driver in charge of the keyboard.

#### 4.2.8 Data transfer from the HID driver to the keyboard driver in Windows

##### Key Point 4.26:

- ☞ How does the HID driver (`kbdhid.sys`) transfer the code received from the keyboard to the generic keyboard driver (`kbdclass.sys`)?
  - ☞ In practice, `kbdhid.sys` uses a callback exported by `kbdclass.sys` is called `KeyboardClassServiceCallback`.
  - ☞ This is the same procedure as `i8042prt.sys` driver (PS/2 keyboards) which shows that both methods converge toward the same result.
  - ☞ This also explains why the HID technology, although more modern (but appeared after) than PS/2, is translated into scan code set 1 from PS/2.
- ☞ Notification from `kbdclass.sys` to `kbdhid.sys` to take into account the `KeyboardClassServiceCallback` callback is performed through `IOCTL_INTERNAL_KEYBOARD_CONNECT` operation.
- ☞ We provide an illustration of the level of routines called when a key is pressed (Figure 4.57) as well as a summary of the HID architecture of the keyboard (Figure 4.58).

We need to study the callback provided to understand how the scan code value is given to the rest of the system. This callback is `KbdHidInsertCodesIntoQueue` provided by `kbdhid.sys` driver. This routine is linked to a lot of internal structures and values in `kbdhid.sys` driver where most of them are not documented. But on principle, after checking and updating the content of scan codes provided, the routine raise the level of IRQL to `DISPATCHLEVEL` and call an internal callback it owns. This callback is provided by `kbdclass.sys` to `kbdhid.sys` driver when `IOCTL_INTERNAL_KEYBOARD_CONNECT` notification [705] is sent to the `kbdhid.sys` driver.

This notification happens at initialization time when the driver is loaded at boot-time. In this case, an IOCTL code is sent for each existing keyboard device. Note that other notifications could happen at running time when a new device is plugged into the system. The `kbdhid.sys` driver handles the notification in its callback routine `KbdHid_IOCTL` registered for `IRP_MJ_INTERNAL_DEVICE_CONTROL` operation. During the notification, an input buffer is provided, holding a `CONNECT_DATA` structure [706]. The first member of the structure is a pointer to an upper-level class filter device object (filter DO). The second one is a pointer which specifies the *class service routine*. This *class service routine* is a generic callback defined by a `PSERVICE_CALLBACK_ROUTINE` routine [707]. In the case of the keyboard, this routine is the modern and generic version of what was known under the name of `KeyboardClassServiceCallback` routine [529].

The `KeyboardClassServiceCallback` routine transfers input data from the input buffer of the device to the keyboard class data queue. This routine is also called by the ISR dispatch completion routine of the function driver when dealing with a PS/2 keyboard. This callback can be superseded by a filter service callback that is provided by an upper-level keyboard filter driver. This is a way to hook in the keyboard chain. This is not a real hack since it is the standard way to proceed [705]. This callback allows a filter service to filter the keyboard data that is transferred to the class data queue. For example, the filter service callback can delete, transform or insert data. This is how `kbdclass.sys` driver is notified by `kbdhid.sys` driver after it retrieves translated values from HID devices. A simplified resume of the keyboard HID notification for a pressed key is given in Figure 4.57.

Finally, the `kbdhid.sys` driver is a communication link between `hidclass.sys` and `kbdclass.sys`, where the first represents HID class devices when the second represents keyboard devices. The registration of `kbdhid.sys` driver in the device stack is performed via a Plug and Play notification callback routine. This one is handle specifically through `IRP_MJ_PNP` [708] operation with the `KbdHid_PnP` routine. This IRP Pnp code is divided in sub-codes called *Plug and Play Minor IRPs* [709], where a non exhaustive list is provided:

- `IRP_MN_START_DEVICE` ;
- `IRP_MN_REMOVE_DEVICE` ;

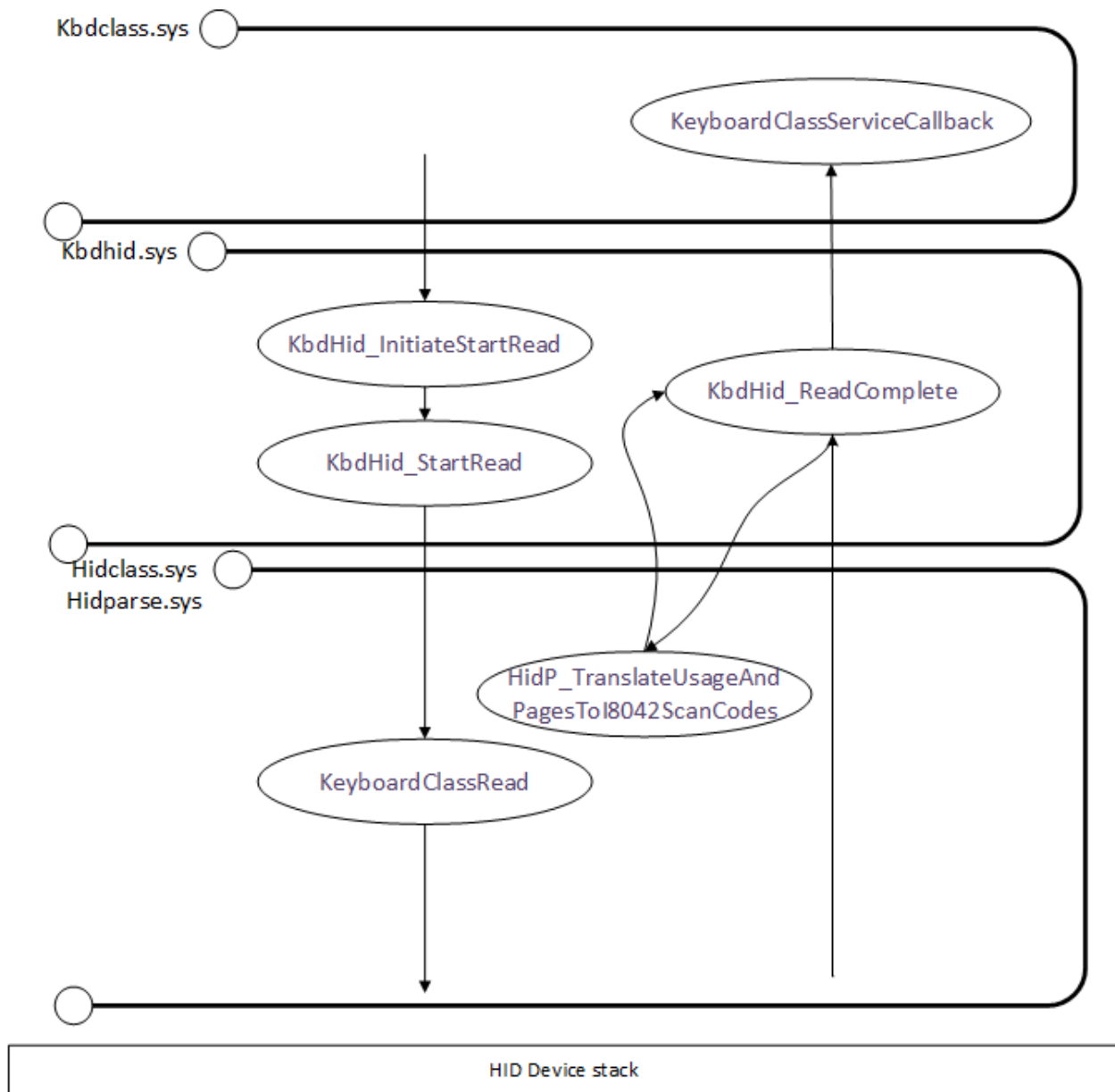


Figure 4.57: Representation of the internal routines notifications in the keyboard HID stack for a pressed key.

- IRP\_MN\_STOP\_DEVICE ;
- IRP\_MN\_SURPRISE\_REMOVAL

In case of a start operation notification (IRP\_MN\_START\_DEVICE is given when a device is plugged during run-time or at boot-time in enumeration phase), KbdHid\_StartDevice is called by kbdhid.sys driver. First, this routine is responsible to call KbdHid\_CallHidClass twice to retrieve<sup>15</sup> data about the device via IOCTL\_HID\_GET\_COLLECTION\_INFORMATION and IOCTL\_HID\_GET\_COLLECTION\_DESCRIPTOR sent to HID class driver. Then, it calls HidP\_GetCaps [647] before reading registry keys related to the device, especially the value *KeyboardTypeOverride*. Once the initialization of several internal structures in memory has been performed (with content from HidP\_MaxUsageListLength [711] which returns the maximum

<sup>15</sup>Via IoBuildDeviceIoControlRequest [710] routine which allocates and sets up an IRP for a synchronously processed device control request.

number of HID usages [712] that `HidP_GetUsages` can return for a specified type of HID report), completion routine `KbdHid_PresenceNotificationCompleted` is armed to be completed once IOCTL 0xB0267 internal code would be executed by next-lower-level driver. This IOCTL code is guessed to be a presence notification for `hidclass.sys` driver to check that the keyboard device is currently taken into account by `kbdhid.sys` driver and it is ready to be notified for that device.

To be fully operational and notified, the link with `kbdclass.sys` must be crafted in `KbdHid_AddDevice` routine, as explained before. A resume of the HID keyboard architecture (with the associated exported API for each driver) is given in Figure 4.58. Technically, it corresponds to the block "HIDUSB.sys" in Figure 4.28 which can be detailed as given in Figure 4.58. For more information, there is a possibility to retrieve the device relations on the device stack by using a device IOCTL code `IRP_MN_QUERY_DEVICE_RELATIONS` [713].

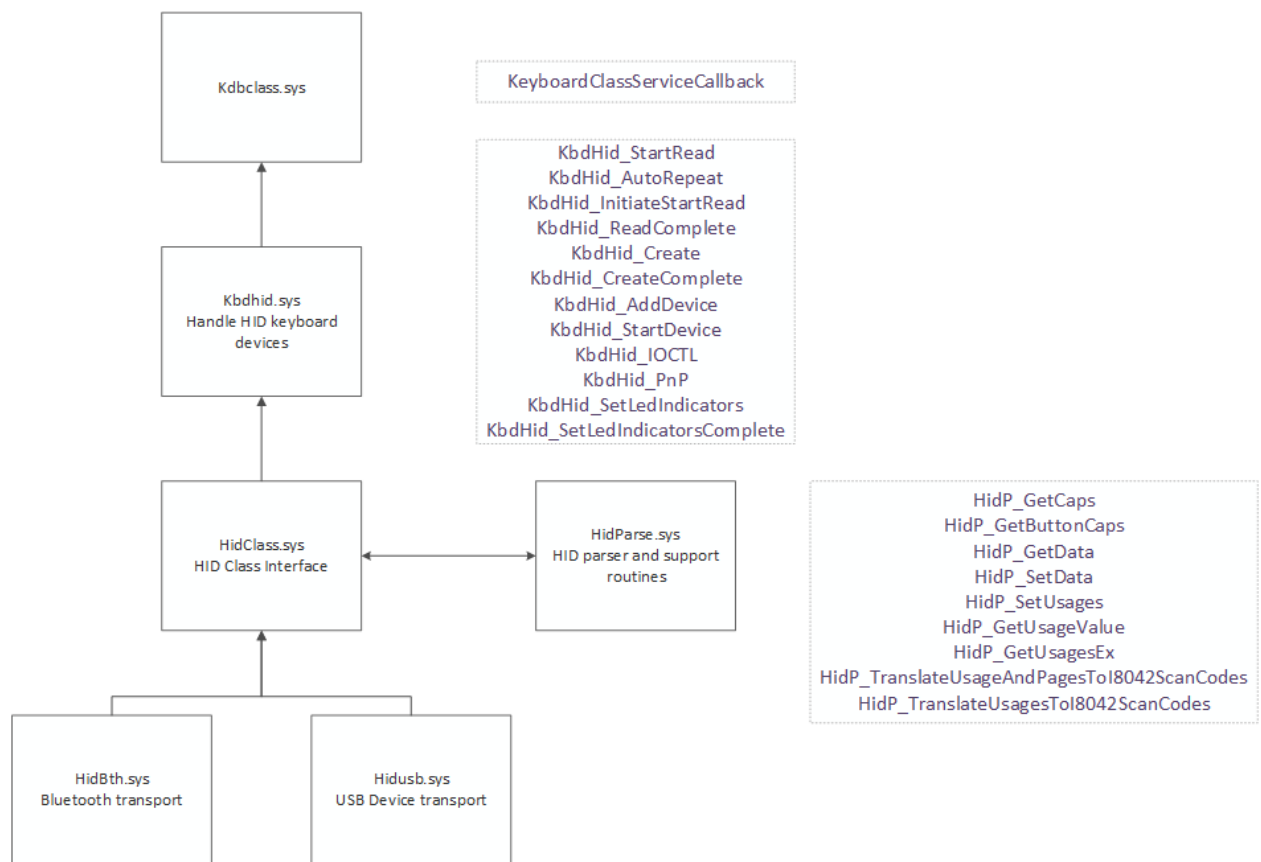


Figure 4.58: Representation of the architecture of HID keyboard drivers with exported API.

### 4.2.9 HID device representation in Windows

#### Key Point 4.27:

- ☞ The device name provided by Windows to a USB keyboard is composed of several letters and numbers (as a path).
- ☞ These numbers owe nothing to chance but with the USB and HID characteristics given by the device descriptors at initialization.
- ☞ We explain how HID device names are composed and how to find them in Windows.

The same way we did it with USB devices, it is possible to visualize the device ID representing the HID device in the system. In our system, using the Windows device manager [714], we have the possibility to view the keyboard, as given in Figure 4.59 and in Figure 4.60. Figure 4.59 looks similar with Figure 4.27. Indeed, it uses the same logic to name objects. Technically, it comes from the Plug and Play support and the inherent enumeration process on the Windows USB stack. But there are specificities for HID devices [715]. The hardware IDs name is generated by the HID class driver. For a given device node, this name depends on the number of functions supported by the underlying transport and from the number of Top Level Collections in the Report Descriptor. Names retrieve most of information from regular USB devices [578, 576, 716]. The differences lie in the use of "HID" prefix and the extended number of lines used. This one describes the same object but from different point of views. If the two first lines still match the requirements used for regular USB devices, the following lines are specific to HID devices. They are not supposed to be documented but we can see that the third line follows the pattern:

```
HID_DEVICE_UP:p(4)_U:u(4)
```

where p(4) corresponds to "Usage Page" number for TLC and u(4) corresponds to "Usage" number of TLC. Note that combination (p = 1 and u = 6) matches HID\_DEVICE\_SYSTEM\_KEYBOARD [715].

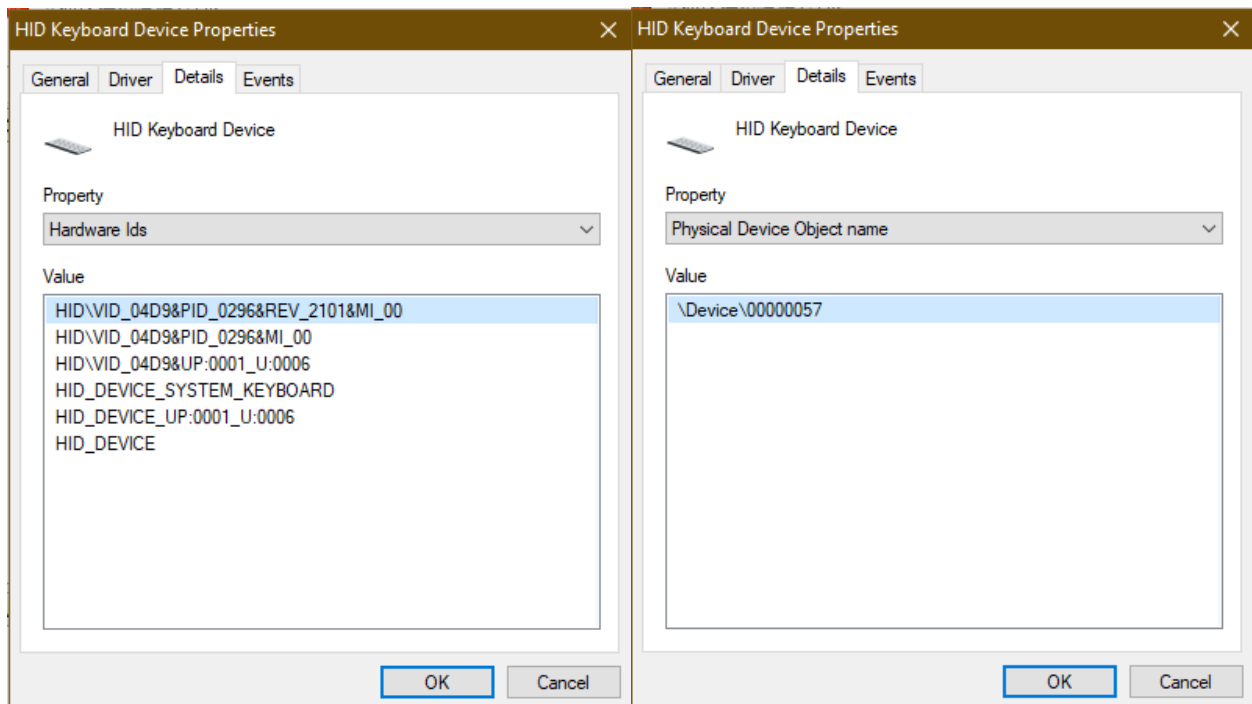


Figure 4.59: Hardware identifiers linked with the key-board device. Figure 4.60: Physical device name associated by the keyboard.



Figure 4.60 corresponds to the physical device name linked with our keyboard device. This device name is used internally by the plug and play manager for each device in the system (whatever the type of the device is). Looking for this device in the NT Object Manager’s name space [717] via WinObj software [718], we find it on Figure 4.61 which gives us all required information too. First, we can see that many lines for a single USB device mean that we are dealing with a composite device. The selected line on Figure 4.61 corresponds to our USB keyboard device. At the difference with USB object’s name, we have at the end of the name a GUID concatenated as a suffix. This GUID corresponds to GUID\_DEVINTERFACE\_KEYBOARD [634] referencing a keyboard object. In front of the name, we have the device name matching the one in the device manager’s view (Figure 4.60). Note the hierarchical representation with the line above the selected one where only the final GUID is different. This GUID corresponds to GUID\_DEVINTERFACE\_HID [633] meaning we are dealing with a HID device.






	HID#VID_048D&PID_8297&Col01#6&1d60bfb1&0&0000#{4d1e55b2-f16f-11cf-88cb-001111000030}	SymbolicLink	\\Device\00000050
	HID#VID_048D&PID_8297&Col02#6&1d60bfb1&0&0001#{4d1e55b2-f16f-11cf-88cb-001111000030}	SymbolicLink	\\Device\00000051
	HID#VID_04D9&PID_0296&MI_00#8&1a54ccd0&0&0000#{4d1e55b2-f16f-11cf-88cb-001111000030}	SymbolicLink	\\Device\00000057
	HID#VID_04D9&PID_0296&MI_00#8&1a54ccd0&0&0000#{884b96c3-56ef-11d1-bc8c-00a0c91405dd}	SymbolicLink	\\Device\00000057
	HID#VID_04D9&PID_0296&MI_01#8&291408b&0&0000#{4d1e55b2-f16f-11cf-88cb-001111000030}	SymbolicLink	\\Device\00000058

Figure 4.61: View from WinObj software where our one of our HID keyboard interface is selected.

#### 4.2.10 Debugging HID components

In some circumstances, it may be useful to be able to debug a USB or HID device. To that end, there are extensions [719] that can be used with Windbg [345, 720, 721]. Information presented here is fully documented. But based on our own experience, dealing with HID and device objects can be a bit complex at the beginning, especially when we do not have tools to visualize what is happening in the system. To help further analysis and debug operations when interacting with this technology, we propose a dedicated section which helps to master the debugger and its HID interface. This part can also be seen as a forensic tutorial for USB/HID devices.

To debug, we need to access to the keyboard device first. Hence, we can use the “!devstack” extension [722] with the keyboard device’s name. The result is given in Code 4.6. Note that in this section, all captures were made on a fresh installed Windows system in a Virtual Box virtual machine. We plugged a USB keyboard in the virtual machine for illustration purposes. Debugging outputs therefore reflect the minimal situation for each user. Virtual box can modify or filter on-the-fly some information coming from the device (notably device descriptor interface). It is a simplified version but it allows us to illustrate the principles seen previously.

```

kd> !devstack \device\keyboardclass0
2  !DevObj          !DrvObj          !DevExt          ObjectName
>  fffffae0a36673c90 \Driver\kbdclass fffffae0a36673de0 KeyboardClass1
4  fffffae0a36e71040 \Driver\kbdhid   fffffae0a36e71190
   fffffae0a36241060 \Driver\HidUsb  fffffae0a362411b0 0000002e
6  !DevNode fffffae0a32f2ccb0 :
   DeviceInst is "HID\VID_1631&PID_5000&MI_00\7&36d65ab7&0&0000"
8  ServiceName is "kbdhid"

```

Code 4.6: “Device stack’s view from a keyboard.”

The device stack gives us the device instance name (as explained in section 4.2.9) and the service name “kbdhid” that handles the device. Knowing the device node address is perfect to get extended information with “!DevNode” extension [723] as given in Code 4.7. This one holds the last states of the device (started, enumerate, assigned, pending and so on) and a lot of information given from USB architecture.

```

kd> !DevNode fffffae0a32f2ccb0
2 DevNode 0xffffae0a32f2ccb0 for PDO 0xffffae0a36241060
   Parent 0xffffae0a32f18cb0 Sibling 0000000000 Child 0000000000

```

```

4 InstancePath is "HID\VID_1631&PID_5000&MI_00\7&36d65ab7&0&0000"
  ServiceName is "kbdhid"
6 TargetDeviceNotify List — f 0xffffc002056ac5a0 b 0xffffc002056ac5a0
  State = DeviceNodeStarted (0x308)
8 Previous State = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory [12] = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory [11] = DeviceNodeEnumeratePending (0x30c)
10 StateHistory [10] = DeviceNodeStarted (0x308)
  StateHistory [09] = DeviceNodeEnumerateCompletion (0x30d)
  StateHistory [08] = DeviceNodeEnumeratePending (0x30c)
12 StateHistory [07] = DeviceNodeStarted (0x308)
  StateHistory [06] = DeviceNodeStartPostWork (0x307)
14 StateHistory [05] = DeviceNodeStartCompletion (0x306)
  StateHistory [04] = DeviceNodeStartPending (0x305)
16 StateHistory [03] = DeviceNodeResourcesAssigned (0x304)
  StateHistory [02] = DeviceNodeDriversAdded (0x303)
18 StateHistory [01] = DeviceNodeInitialized (0x302)
  StateHistory [00] = DeviceNodeUninitialized (0x301)
20 StateHistory [19] = Unknown State (0x0)
  StateHistory [18] = Unknown State (0x0)
22 StateHistory [17] = Unknown State (0x0)
  StateHistory [16] = Unknown State (0x0)
24 StateHistory [15] = Unknown State (0x0)
  StateHistory [14] = Unknown State (0x0)
26 StateHistory [13] = Unknown State (0x0)
  Flags (0x2c000130) DNF_ENUMERATED, DNF_IDS_QUERIED,
30 DNF_NO_RESOURCE_REQUIRED, DNF_NO_LOWER_DEVICE_FILTERS,
  DNF_NO_LOWER_CLASS_FILTERS, DNF_NO_UPPER_DEVICE_FILTERS
  UserFlags (0x00000008) DNUF_NOT_DISABLEABLE
32 CapabilityFlags (0x00001e83) DeviceD1, DeviceD2,
  SilentInstall, SurpriseRemovalOK,
34 WakeFromD0, WakeFromD1,
  WakeFromD2
36 DisableableDepends = 1 (including self)

```

Code 4.7: "Device node's view from a keyboard."

And if we are looking for the device object represented by the driver "kbdhid", it is possible to see the hierarchy between the different drivers (Code 4.8). To proceed, we are using the "!devobj" extension [724].

```

1 kd> !devobj fffffae0a36e71040
  Device object (ffffae0a36e71040) is for:
3   \Driver\kbdhid DriverObject fffffae0a36e886b0
  Current Irp 00000000 RefCount 0 Type 0000000b Flags 00002000
5 SecurityDescriptor fffffc001fe4e1ba0 DevExt fffffae0a36e71190 DevObjExt fffffae0a36e714b8
  ExtensionFlags (0x00000800) DOE_DEFAULT.SD_PRESENT
7 Characteristics (0000000000)
  AttachedDevice (Upper) fffffae0a36673c90 \Driver\kbdclass
9 AttachedTo (Lower) fffffae0a36241060 \Driver\HidUsb
  Device queue is not busy.

```

Code 4.8: "Device driver object's view for kbdhid driver."

But it could be easier to directly interact with HID debugging extension and its commands implemented in Hidkd.dll [725]. This extension allows to list, view and interact with HID devices. Generally, the first command to use is "!hidkd.hidtree" to get a view of the tree representation of the HID devices plugged in the system. The output of hidtree command can be very talkative (in Code 4.9, parts have been removed for the sake of readability) but it holds a lot of useful information. For instance, we have the USB view with the content of the device descriptor (vendor id, product id, version — those displayed corresponds to the virtual USB interface provided by VirtualBox) and USB device path given by Windows. Then we have the HID representation of each instance. Using the TLC parsed by Windows thanks to hidparse.sys, our keyboard device is a collection device (with three collections: Consumer, Vendor-defined and Generic Desktop Controls). Finally, we have the keyboard device with address of structures used by the kernel to hold HID devices.

```

kd> !hidkd.hidtree
2 HID Device Tree
-----
4 FDO VendorID:0x1631(Focus Enhancements) ProductID:0x5000 Version:0x0600
!hidfdo 0xfffffae0a36235060
6 PowerStates: S0/D0 | 0n0
dt FDO_EXTENSION 0xfffffae0a362351d0
8 !devnode 0xfffffae0a33019cb0 | DeviceNodeStarted (0n776)
InstancePath: USB\VID_1631&PID_5000&MI_01\6&148150c0&0&0001
10 IFR Log: !rcdrlogdump HIDCLASS -a 0xFFFFFAE0A3702C000

12 PDO Consumer (0x0C) | Consumer Control (0x01)
(...)
14 PDO Vendor-defined (0xFF00) | 0x01
(...)
16 PDO Generic Desktop Controls (0x01) | System Control (0x80)
(...)
20
-----
22 FDO VendorID:0x1631(Focus Enhancements) ProductID:0x5000 Version:0x0600
!hidfdo 0xfffffae0a368312c0
24 PowerStates: S0/D0 | 0n0
dt FDO_EXTENSION 0xfffffae0a36831430
26 !devnode 0xfffffae0a32f18cb0 | DeviceNodeStarted (0n776)
InstancePath: USB\VID_1631&PID_5000&MI_00\6&148150c0&0&0000
28 IFR Log: !rcdrlogdump HIDCLASS -a 0xFFFFFAE0A36E9E000

30 PDO Generic Desktop Controls (0x01) | Keyboard (0x06)
!hidpdo 0xfffffae0a36241060
32 Power States: S0/D0 | COLLECTION_STATE_RUNNING (0n3)
dt PDO_EXTENSION 0xfffffae0a362411d0
34 !devnode 0xfffffae0a32f2ccb0 | DeviceNodeStarted (0n776)
Instance Path: HID\VID_1631&PID_5000&MI_00\7&36d65ab7&0&0000

```

Code 4.9: "HID device tree view of the different components."

Using the "!hidpdo" command helps to get access to detailed information about our HID keyboard device. In Code 4.10, we have address to pre-parsed data report and details about report lengths.

```

1 !hidpdo 0xfffffae0a36241060
PDO 0xfffffae0a36241060 (!devobj/!devstack)
3
-----
5 Collection Num : 1
Name : \Device\_HID00000001#COLLECTION00000001
7 FDO : !hidfdo 0xfffffae0a368312c0
Per-FDO IFR Log : !rcdrlogdump HIDCLASS -a 0xFFFFFAE0A36E9E000
Usage Page : Generic Desktop Controls (0x01)
9 Usage : Keyboard (0x06)
Report Lengths : 0x9(Input) 0x2(Output) 0x0(Feature)
11 Pre-parsed Data : 0xfffffae0a35e941c0
Position in HID tree

13 dt PDO_EXTENSION 0xfffffae0a362411d0
15
Power States
-----
17 Power States : S0/D0
19 Wait Wake IRP : none

21 !devnode 0xfffffae0a32f2ccb0
-----
23 State : DeviceNodeStarted (0n776)
Instance Path : HID\VID_1631&PID_5000&MI_00\7&36d65ab7&0&0000

```

Code 4.10: "HID keyboard device information."

From information in Code 4.10 one can retrieve detailed information about the functional device (FDO) represented here. Output from the command `!hidfdo` with the address of the HID FDO is given in Code 4.11. We have the power states and capabilities of our device but we also have access to its report descriptor.

```

kd> !hidfdo 0xfffffae0a368312c0
2 FDO 0xfffffae0a368312c0 (!devobj/!devstack)
-----
4 Name : \Device\_HID00000001
6 Vendor ID : 0x1631 (Focus Enhancements)
8 Product ID : 0x5000
10 Version Number : 0x0600
12 Is Present? : Y
14 Report Descriptor : !hidrd 0xfffffae0a36a02d10 0x3b
16 (...)
18 Device States
-----
20 Power States.....: S0/D0
22 State Machine State...: 0n0
24 Idle IRP.....: !irp 0xfffffae0a36730b20 (completed with status code 0x0)
26 Idle PDOs.....: 0
28 WaitWake IRP.....: none
30 Power-delayed IRPs....: 0
32 PDO WaitWake IRPs....: 0
34 Open Count.....: 2
36 Last INT Report Status: 0x0
38 Last INT Report Time..: 09/02/2020-17:45:49.262 (Romance Daylight Time)
40 Device Capabilities
-----
42 Support D1 : Y
44 Support D2 : Y
46 Removable : N
48 SurpriseRemovalOK : N
50 Wake from D0 : Y
52 Wake from D1 : Y
54 Wake from D2 : Y
Wake from D3 : N
Device states : S0=>D0, S1=>D3, S2=>D3, S3=>D3, S4=>D3 S5=>D3
SystemWake : S0
DeviceWake : D2
(...)
42 Collections (1 Total)
-----
44 Collection Num.....: 1
46 Collection.....: dt HIDCLASS_COLLECTION 0xfffffae0a332b28b0
48 Collection PDO.....: !hidpdo 0xfffffae0a36241060
50 UsagePage.....: Generic Desktop Controls (0x01)
52 Usage.....: Keyboard (0x06)
54 Report Lengths.....: 0x9(Input) 0x2(Output) 0x0(Feature)
Prepared Data.....: !hidppd 0xfffffae0a35e941c0
Open Count.....: 1 (Read:1|Write:0 Restriction:[Read ])
Pending Reads.....: 1
Cumulative # of INT Reports..: 8
Last INT Report Time.....: 09/02/2020-17:45:49.262 (Romance Daylight Time)

```

Code 4.11: "HID functional device information."

If we need to visualize the content of the HID report for that device, we can use the `!hidrd` command. The output result is given in Figure 4.12. We have first the raw data as provided by the device and then a parsed view of the content of this report. It is valuable help to debug and to understand reports provided by the device. Technically, with this HID report, we have a structure which is very close to the one given in table 4.12. It makes sense since the two HID reports presented are those coming from interfaces supporting boot devices.

Such boot interfaces follow requirements previously exposed from [591].

```

kd> !hidrd 0xffffae0a36a02d10 0x3b
2 Report Descriptor at 0xffffae0a36a02d10
4 Raw Data
6 -----
6 0x0000: 05 01 09 06 A1 01 05 07-19 E0 29 E7 15 00 25 01
7 0x0010: 75 01 95 08 81 02 95 08-81 01 95 03 05 08 19 01
8 0x0020: 29 03 91 02 95 05 91 01-95 06 75 08 15 00 26 FF
9 0x0030: 00 05 07 19 00 2A FF 00-81 00 C0
10 -----
10 Parsed
12 -----
12 Usage Page (Generic Desktop Controls)..0x0000: 05 01
14 Usage (Keyboard).....0x0002: 09 06
14 Collection (Application).....0x0004: A1 01
16 .. Usage Page (Keyboard/keypad).....0x0006: 05 07
16 .. Usage Minimum (0xE0).....0x0008: 19 E0
18 .. Usage Maximum (0xE7).....0x000A: 29 E7
18 .. Logical Minimum (0).....0x000C: 15 00
20 .. Logical Maximum (1).....0x000E: 25 01
20 .. Report Size (1).....0x0010: 75 01
22 .. Report Count (8).....0x0012: 95 08
22 .. Input (Data,Var,Abs).....0x0014: 81 02
24 .. Report Count (8).....0x0016: 95 08
24 .. Input (Cnst,Ary,Abs).....0x0018: 81 01
26 .. Report Count (3).....0x001A: 95 03
26 .. Usage Page (LEDs).....0x001C: 05 08
28 .. Usage Minimum (Num Lock (0x01)).....0x001E: 19 01
28 .. Usage Maximum (Scroll Lock (0x03))...0x0020: 29 03
30 .. Output (Data,Var,Abs).....0x0022: 91 02
30 .. Report Count (5).....0x0024: 95 05
32 .. Output (Cnst,Ary,Abs).....0x0026: 91 01
32 .. Report Count (6).....0x0028: 95 06
34 .. Report Size (8).....0x002A: 75 08
34 .. Logical Minimum (0).....0x002C: 15 00
36 .. Logical Maximum (255).....0x002E: 26 FF 00
36 .. Usage Page (Keyboard/keypad).....0x0031: 05 07
38 .. Usage Minimum (0x00).....0x0033: 19 00
38 .. Usage Maximum (0xFF).....0x0035: 2A FF 00
40 .. Input (Data,Ary,Abs).....0x0038: 81 00
40 End Collection ().....0x003A: C0

```

Code 4.12: "HID report from our keyboard."

If the parsed view of the HID report is too confusing for the reader, it is possible to use `!hidppd` command to get a view from the pre-parsed structure used by `hidparse.sys` driver. This view given in Code 4.13 is much more user-friendly since it is close to the structure definition.

```

1 kd> !hidppd 0xffffae0a35e941c0
2 Reading preparsed data...
3 Preparsed Data at 0xffffae0a35e941c0
5 Summary
7 -----
7 UsagePage : Generic Desktop Controls (0x01)
7 Usage : Keyboard (0x06)
9 Report Lengths : 0x9(Input) 0x2(Output) 0x0(Feature)
9 Link Collection Nodes : 1
11 Button Caps : 2(Input) 1(Output) 0(Feature)
11 Value Caps : 0(Input) 0(Output) 0(Feature)
13 Data Indices : 264(Input) 3(Output) 0(Feature)
15 Input Button Capability #0
17 -----
17 Report ID : 0x0

```

```

19  Usage Page      : Keyboard/keypad (0x07)
    Alias         : No
    Link Collection : 0
21  Link Usage Page : Generic Desktop Controls (0x01)
    Link Usage    : Keyboard (0x06)
23  Usage Range    : 0xE0 (min) 0xE7 (max)
    Data Index Range : 0x0 (min) 0x7 (max)
25  String Index Range: 0x0 (min) 0x0 (max)
    Designator Range : 0x0 (min) 0x0 (max)
27  Is Absolute   : Yes
29
31  Input Button Capability #1
    -----
33  Report ID      : 0x0
    Usage Page    : Keyboard/keypad (0x07)
    Alias         : No
35  Link Collection : 0
    Link Usage Page : Generic Desktop Controls (0x01)
37  Link Usage    : Keyboard (0x06)
    Usage Range    : 0x00 (min) 0xFF (max)
39  Data Index Range : 0x8 (min) 0x107 (max)
    String Index Range: 0x0 (min) 0x0 (max)
41  Designator Range : 0x0 (min) 0x0 (max)
    Is Absolute   : Yes
43
45  Output Button Capability #0
    -----
47  Report ID      : 0x0
    Usage Page    : LEDs (0x08)
49  Alias         : No
    Link Collection : 0
51  Link Usage Page : Generic Desktop Controls (0x01)
    Link Usage    : Keyboard (0x06)
53  Usage Range    : Num Lock (0x01) (min) Scroll Lock (0x03) (max)
    Data Index Range : 0x0 (min) 0x2 (max)
55  String Index Range: 0x0 (min) 0x0 (max)
    Designator Range : 0x0 (min) 0x0 (max)
57  Is Absolute   : Yes

```

Code 4.13: "HID pre-parsed report from our keyboard."

All of these commands and extensions are very useful tools to understand the design of the system, but also for forensic purposes, reverse engineering or driver development. There are other possibilities which are more specific and useful in given contexts. For the sake of simplicity, we only cover those that make sense in terms of what has been presented in this state of the art.

### 4.3 Research Contributions

#### Contribution 3: Protection of analyzed executable files

- ☞ State-of-the-art about different techniques used by keyboard devices to communicate.
- ☞ About PS/2 protocol:
  - ✍ We have documented how the PS/2 protocol works with keyboard device.
  - ✍ We have explained how Windows works with the devices that are still connected via PS/2.
- ☞ About USB:
  - ✍ We proposed a synthesis of the USB protocol focused on the use of keyboard devices.
  - ✍ We explained how a USB keyboard is recognized by the system.
  - ✍ We explained how Windows initializes and configures a USB keyboard device.
  - ✍ We explained how works the USB architecture from the Windows kernel with unpublished details.
- ☞ About USB/HID:
  - ✍ We showed how the read IRP polling system works via the Windows HID driver.
  - ✍ We have documented that the read IRP is activated since the initialization of the driver and after each key is pressed.
  - ✍ We have documented the device driver stack mechanism which show how the content of a keystroke is retrieved in a completion routine.
  - ✍ We document new registry keys used to modify the behavior of the HID driver (both on debugging operations and key repetition).
  - ✍ We have shown that the scan code set used by Microsoft for translation is actually an extension of the original scan code set 1 from PS2.
  - ✍ We have proved that for backward compatibility reasons, Microsoft translates the modern USB/HID technology into the old PS/2 one.
  - ✍ This exact knowledge of the scan code set used will be useful, later, for our security solution (Chapter 6).
- ☞ We document unpublished Windows features thanks to reverse engineering.
  - ✍ This provides us a complete knowledge about how the system works.
  - ✍ This allows us to understand what the possibilities are for malware.
  - ✍ Hence, we can design and develop accurate and efficient solutions against them.
  - ✍ Solutions that can be developed at different levels (PS/2, USB, HID) simply by reading this chapter.



## 5 Kbdclass and Windows subsystem

### Resume 25:

- ☞ This section explains how broadcasting keyboard inputs works from the generic keyboard driver (kbdclass.sys) to system applications.

Once we have explained how the data was retrieved from the keyboard device, it remains to know how this information is broadcasted to the rest of the system and how the system is designed to manage this information. Let us focus first on the design of the system to explain how this design is used to broadcast information from the device.

### 5.1 Transition from kernel to user mode architecture

#### Resume 26:

- ☞ This subsection explains how Windows kernel initiates read operations on the keyboard from csrss.exe with the *Raw Input Thread* (RIT).
  - ☞ We briefly explain the history of the message system in Windows.
  - ☞ From the creation to the operations of the RIT in csrss.exe, we see how it initiates and processes the keyboard content.
  - ☞ We observe the role of the RIT is to transfer input data from the kernel to the user-mode.
- ☞ Vocabulary clarifications:
  - ☞ The word "Windows" with a capital letter at the beginning refers to the Microsoft operating system.
  - ☞ The word "window(s)" in lowercase refers to *Graphical User Interface* (GUI).
  - ☞ An "*application*" is a user-mode software and a "*driver*" is a kernel-mode software.

As explained in previous chapters, retrieving the code of a keystroke is performed by reading the input data from the keyboard device driver. Technically, the system engages an IRP linked to the kernel keyboard driver and waits until this one is complete, mostly when key is pressed. The good point is that it perfectly complies with USB norm requirements since this one is host driven by IRP (Key Point 4.10). This IRP is then translated and transferred to kbdclass.sys driver (Key Point 4.26). Former technology keyboard PS/2 are direct interrupt devices but the system handles them the same way (Key Point 4.5). As a result, from our point of view, now, the two types of keyboard devices are undifferentiated since they both rely on kbdclass.sys driver.

A great question is to know which component has built the initial keyboard read IRP and how that IRP has been built. Indeed, it is the insertion of this IRP that starts reading operations of the keyboard and thus it authorizes its usage by the system. As we can see in the HID (and USB) driver, an IRP is engaged at each level. But these ones are generic IRPs, which are there for all devices that the drivers support. There is no "keyboard" specificity here. More directly, if there is no request by the top of system itself, such intermediate IRP will remain useless for handling keyboard specifically.

The first thing to consider is the fact that keystrokes are usually read by user-mode applications. For short, the kernel is in charge of managing the hardware, that is to say the keyboard device by itself. Everything about PS/2 or USB protocol, HID and IRP are kernel-mode concepts which are not accessible to user-mode applications. Indeed, the latter are less privileged. But they must be able to read the content of the information provided by the keyboard. Technically, there are several ways [726, 727] to do it and we are going describe them. For the sake of clarity, we can assume, at that point, that keystrokes are broadcast through a specific system of messages application's read by applications [728, 729] or directly by a dedicated thread in an application which waits for events informing that a given key has been pressed.

### 5.1.1 Raw input thread

#### Key Point 4.28:

- ☞ Management of the keyboard is performed through the *raw input thread* with a message system used for GUI windows.
  - ☞ RIT initializes the read IRP processed by kbdclass.sys driver before broadcasting information retrieved through messages for windows.
  - ☞ The System of messages used for GUI windows is asynchronous nowadays (it was synchronous on 16-bit Windows).
- ☞ The *Raw Input Thread* (RIT) is a *centralized* system that manages keystrokes to distribute them through messages in an *asynchronous* way.

System of messages used for windows is typically a technology related to *Graphical User Interface* (GUI). The principle is that every application has a dedicated thread holding messages from the rest of the system. Messages are very diverse [729] since they cover how application is display on the screen (dialogue box, painting and so on) to input provided from devices.

In the old days of 16-bit Windows, input system was synchronous [730]. It meant that input messages were stored into a system-wide input queue and they were dispatched in chronological order. It was the application's responsibility to hold input messages and release them to let other applications to access to the list (Figure 4.62). This strategy ensured that the user saw input being processed in the order where events had been generated. But the disadvantage is that if an application stopped to process input events, it could potentially block the entire system since cooperation would not be guaranteed anymore.

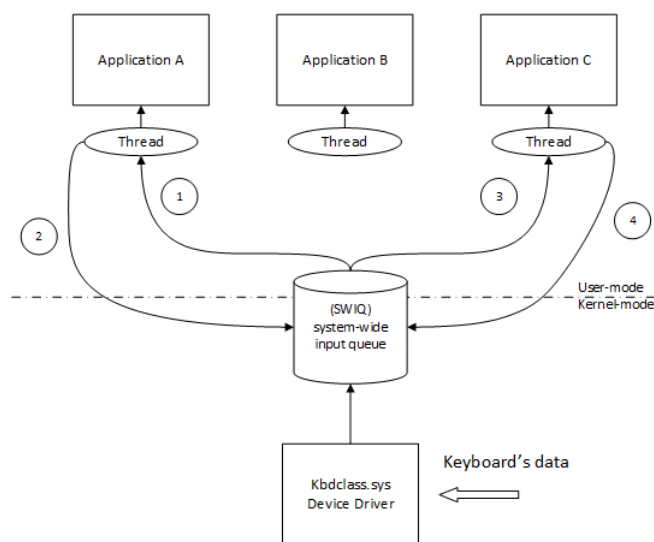


Figure 4.62: Illustration of the 16-bit Windows synchronous message system.

Starting with Windows 32-bit (Win32), input queues were made asynchronous. It means that each thread on the system has its own input queue. Technically speaking, different devices (by the mean of their driver stack) post events into the *system hardware input queue*<sup>16</sup> (SHIQ) so that a dedicated thread, called *raw input thread* (RIT) [731], distributes them to the appropriate applications' input queues. This new strategy does not allow the distribution to be performed at the time the hardware devices post the event, because that happens asynchronously. Technically speaking, messages coming from hardware devices are posted into the SHIQ from

<sup>16</sup>According to Raymond Chen [730], "system hardware input queue" is not the official term but we are using it for the sake of simplicity.

the bottom half (Figure 4.63). Then, the raw input thread processes them from the top half. The disadvantage of the delay induced is nevertheless completely imperceptible for a user. But the advantage of this strategy is that if one thread stops managing inputs, other threads are not impacted because the input queues of two threads are separated and do not dependent on each other. This property of asynchronous input system has been called the *Holy Grail* [732] at the time of OS/2 development.

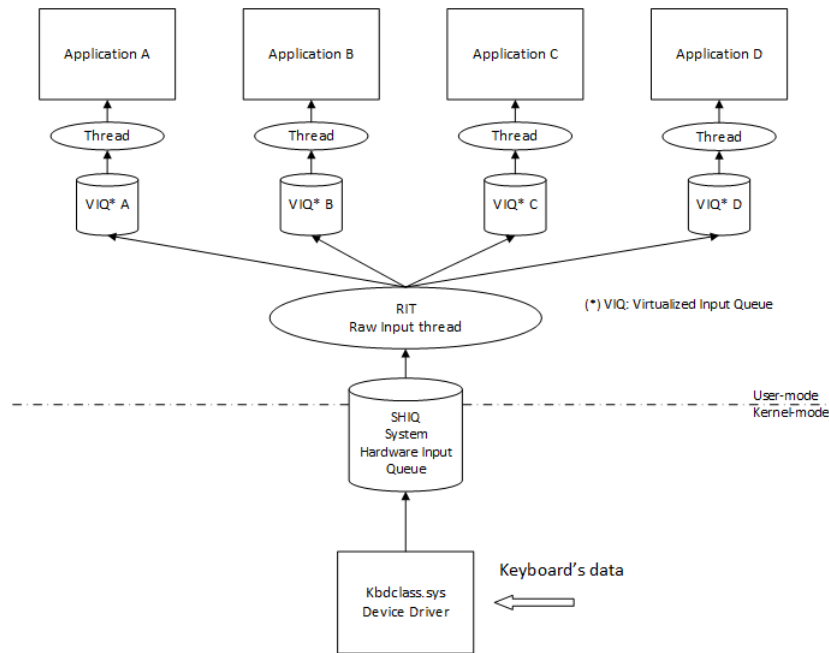


Figure 4.63: Illustration of the Win32 asynchronous message system.

It is the raw input thread which is at the origin of the read IRP initiated on the kbdclass.sys driver. In practice, this is an IRP\_MJ\_READ [733, 734] operation. This can be checked with Windbg debugger. By breaking in KeyboardClassServiceCallback routine (from kbdclass.sys), we can check the structure of the IRP processed as first parameter from the routine IoCompleteRequest [735, 695] right before the end. The content of the IRP can be trusted but not the current thread (in the current process) which handles the IRP request. The reason is that IRPs are rarely executed in the context of the calling thread but in an arbitrary thread context [736]. Note the fact that this IRP contains the keystroke(s) provided by the keyboard (in the form of a table of KEYBOARD\_INPUT\_DATA structure(s) [737] [734]. In the IRP [530] structure, there is a Tail member. This one references a "tail.overlay" structure (real symbol name is unknown) which gives an access to the thread (and by consequence the process where the thread belongs) holding the original IRP. This structure is given in Code 4.14 from a debugging session with Windbg.

```

1 dx -r1 (*((ntdll!_IRP *)0xffff8e8405769b00)).Tail.Overlay
2 (*((ntdll!_IRP *)0xffff8e8405769b00)).Tail.Overlay [Type: <anonymous-tag>]
3 [+0x000] DeviceQueueEntry [Type: _KDEVICE_QUEUE_ENTRY]
4 [+0x000] DriverContext [Type: void * [4]]
5 [+0x020] Thread : 0xffff8e8403354080 [Type: _ETHREAD *]
6 [+0x028] AuxiliaryBuffer : 0x0 [Type: char *]
7 [+0x030] ListEntry [Type: _LIST_ENTRY]
8 [+0x040] CurrentStackLocation : 0xffff8e8405769dc8 : IRP_MJ_READ / 0x0 for Device for "\
9 Driver\kbdclass" [Type: _IO_STACK_LOCATION *]
10 [+0x040] PacketType : 0x5769dc8 [Type: unsigned long]
11 [+0x048] OriginalFileObject : 0xffff8e840332d650 [Type: _FILE_OBJECT *]
12 [+0x050] IrpExtension : 0x0 [Type: void *]

```

Code 4.14: "Parsing of IRP.tail.overlay structure."

If we are looking for the content of the `_ETHREAD` structure referenced in Code 4.14, we have access to the `_EPROCESS` structure [525]<sup>17</sup> describing the current process holding the thread. Extract of the `_KTHREAD` structure is given in Code 4.15. From that code, we can make three relevant observations.

- `PreviousMode` member is set to one (user-mode), confirming the requesting thread belongs to a user-mode application.
- `BasePriority` member is equal to 16 which corresponds to the lowest level of *realtime priority class* [739]. This priority is rather high and it requires to get administrators rights to run a thread at that level.
- `ThreadName` member holds the thread's name which is a string with "Win32k Raw Input Thread".

This last characteristic is quite interesting and rather rare for system threads. In practice, this member has been set by the `RawInputThread` routine (from `win32kfull.sys`) itself with the help of `SetThreadName` (from `win32kbase.dll`) [740]. Routine `SetThreadName` is a call to `ZwSetInformationThread` [741] with `ThreadNameInformation` (0x26) and a `UNICODE_STRING` [742] as parameter initialized with the name of the thread to set.

```

1 kd> dx -r1 (*((ntdll!_ETHREAD *)0xffff8e8403354080)) .
  (*((ntdll!_ETHREAD *)0xffff8e8403354080)) [Type: _KTHREAD]
3   [+0x000] Header [Type: _DISPATCHER_HEADER]
  (...)
5   [+0x220] Process : 0xffff8e84032e5080 [Type: _KPROCESS *]
  [+0x228] UserAffinity [Type: _GROUP_AFFINITY]
7   [+0x228] UserAffinityFill [Type: unsigned char [10]]
  [+0x232] PreviousMode : 1 [Type: char]
9   [+0x233] BasePriority : 16 [Type: char]
  (...)
11  [+0x7e0] ThreadName : "Win32k Raw Input Thread"

```

Code 4.15: "Part of the `_EPROCESS` structure where the IRP comes from."

### 5.1.2 Csrss process

#### Key Point 4.29:

- ☞ `Csrss.exe` is the process holding the *raw input thread*.
  - 👉 This is a critical process, the lack of the one would cause the system to crash.
  - 👉 It is launched more than once for security reasons, once per user's session.
  - 👉 It is firstly launched in Session 0 used to host most of critical system's services.
- ☞ Since `csrss.exe` is launched once per session, there is one *raw input thread* per session holding the keyboard.
  - 👉 Hence the messages (and therefore the keyboard content) processed by the RIT of a session are specific to each session (and they cannot be listened to from another session).

By knowing the thread which has initiated the IRP allows us to know in which process this thread has been attached. Checking in the `_EPROCESS` structure provided by the `_ETHREAD` structure gives at offset `+0x450`<sup>18</sup> the member `ImageFileName` with "csrss.exe" [743, 744]. *Client/Server Runtime Subsystem* (aka

<sup>17</sup>Both of these kernel structures are partially opaque structures [525]. It means they are partially documented and prone to change at any time. But it is possible to *understand* them with the help of debugging symbols [738] from Microsoft.

<sup>18</sup>This offset could be updated in future versions of Windows. If it is provided for pedagogical purposes to illustrate how to use debugging information in order to understand how Windows operating system works. Of course, you should never rely on this means of access to develop software that would seek to use this information.

csrss.exe) process had had different roles in the history<sup>19</sup> of Windows operating system. One of them is to expose environment subsystems as a strictly-defined subset of native functions provided to user-mode applications. Microsoft's operating system supports two of them: Windows and POSIX (Portable Operating System Interface for Unix). But csrss is not involved in every API call being issued by an user-mode application. Indeed, most of the API is housed in dedicated DLLs loaded at runtime by applications. This process is only required when dealing with mechanisms implemented by the subsystem process, for instance with `CreateProcess` [745] or `CreateRemoteThread` [746]. Interaction by an application with csrss is largely covered by the Windows API where internal functioning is undocumented (mostly through *advanced local procedure call* ALPC [747]).

Since csrss is responsible for different operations which are a the heart of the interaction between applications and the operating system, Windows is unable to work correctly without this process running in the background. This process is marked to be a Subsystem process and protected. In other words, in case of a crash of csrss.exe, Windows is going to have a kernel panic (full crash of the system) also known as a blue screen of death (BSOD) [748].

From Windows Vista, there are more than one instance of csrss instance running on the system. Indeed, csrss is created at boot time, by the *Session Manager* (smss.exe) process, right after loading the different graphical modules (including win32k.sys) into kernel memory. Due to the fact that it is launched at system start time, csrss inherits its parent's security token, making it running with the highest possible privileges (SYSTEM). Before Microsoft Windows Vista [749], all services were running in the same session as the first user who logs on to the console. This session still exists and it is called Session 0. This was a serious security risk because services running at elevated privilege are perfect targets for malicious agents who are looking for a mean to elevate their own privilege level. Windows Vista fixed this issue by isolating services in Session 0 and making Session 0 non-interactive [750]. The user logs on to Session 1<sup>20</sup> and other users log in Session 2, 3 and so on. This means that services never run in the same session as users' applications and they are therefore protected from attacks coming from application in session 1. This is why csrss is launch more than once on the system: once per session. The first launch is for Session 0. With no interface for devices (it is even hard to interact with Session 0 for debug purposes [751]), it does not directly deal with keyboard and even if it initializes a raw input thread, this one is limited in its activities. Indeed, csrss in Session 0 is present for handling the requests coming from session-0 modules without any real graphical interface. This is why it does not need all the window manager's capabilities. But in the case where csrss is launched in Session 1, this one creates a real raw input thread able to deal with all different events.

Note that csrss process is dealing with win32k.sys driver. This driver has evolved over time to take different forms depending on the constraints of the environment in which it has been executed. Indeed, Windows 10 is currently executed on Xbox, phones or small tablets, and on IoT devices. In these cases, Windows does not need many of the graphical features it has on desktop version because there is only one window in the foreground and this one cannot be minimized or resized [743]. In case of IoT, there is sometimes no screen on the device which means that there is nothing to display at all. This is why win32k.sys has been split among several kernel modules, to only use the required modules on the different systems where Windows may run. The result is by eliminating many of legacy pieces of code or irrelevant parts for some devices, it reduces the attack surface. This is why we are dealing with routines coming from different *parts* of win32k. On full desktop systems, Windows 10 loads win32k.sys which loads win32kbase.sys and win32kfull.sys drivers. Driver win32kbase.sys can be used on certain IoT systems in addition to phone sub-systems. Driver win32kfull.sys is generally used by desktop versions since it provides more *advanced graphical features* than win32kbase.sys. To help the reader understand where the analyzed routines are located, we will specify the driver file in which they reside when necessary. Note that there is no obligation for a routine to belong forever to one of the drivers composing win32k. Locations

---

<sup>19</sup>According to [744], at former time and before Windows NT4, csrss was also responsible to handle Window Manager and Graphic Services by itself. It was a very slow design since every graphical request needed to pass through csrss which resulted in a bottleneck and a loss of performance as a result of the change of context between the requesting process and csrss.exe. Coming with Windows NT4, the window management had been implemented in a kernel-mode driver called win32k.sys. This one is available to user-mode applications through the standard system-call mechanism as regular NT API. It remains that there still exists connections between win32k.sys kernel driver and csrss.exe to deal with window management.

<sup>20</sup>Note that, if it is mentioned that the disappearance of csrss will result in a BSOD, it will not be the case if csrss is closed properly part of the disconnection from a user's session procedure or in the procedure of shutdown. Diverse events are created by csrss to be notified in case of session changing or shutdown signal received.

of routines can change over time. The presentation provided here is intended to facilitate reverse engineering if the reader would like to verify the explanations given or to go further.

### 5.1.3 How the raw input thread is created by csrss.exe?

#### Resume 27:

☞ This subsection explains how the RIT is initialized from csrss.exe, via a rather complex but detailed procedure.

The application csrss.exe has evolved a lot between the different versions of Windows operating system. Nowadays, this one is almost empty. Indeed, it is composed by few functions where the entry point `main` (Figure 4.64) is just about setting different attributes on the current process (mostly about priority of execution since csrss must be very responsive to avoid to clog up the whole system by not responding enough fast) and calling `CsrServerInitialization` stored in `csrssrv.dll` library.

```

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     const char **argv_1; // rbx@1
4     unsigned int argc_1; // edi@1
5     unsigned int error; // eax@1
6     unsigned int error_1; // ebx@1
7     __int64 HardErrorMode; // [rsp+20h] [rbp-18h]@3
8     __int64 v9; // [rsp+28h] [rbp-10h]@1
9     __int64 ProcessInformation; // [rsp+50h] [rbp+18h]@1
10    __int64 BasePriority; // [rsp+58h] [rbp+20h]@1
11
12    LODWORD(ProcessInformation) = 1;
13    argv_1 = argv;
14    argc_1 = argc;
15
16    NtSetInformationProcess((HANDLE)-1i64, ProcessPriorityBoost|0x20, &ProcessInformation, 4u);
17    RtlSetUnhandledExceptionFilter(CsrUnhandledExceptionFilter);
18    LODWORD(BasePriority) = 13; // HIGH_PRIORITY_CLASS | THREAD_PRIORITY_NORMAL
19    NtSetInformationProcess((HANDLE)-1i64, ProcessBasePriority, &BasePriority, 4u);
20    RtlSetHeapInformation(0i64, 1i64, 0i64);
21    v9 = 0x100000002i64;
22    NtSetInformationProcess((HANDLE)-1i64, ProcessHandleCount|0x20, &v9, 8u);
23    error = CsrServerInitialization(argc_1, argv_1);
24    error_1 = error;
25    if ( (error & 0x80000000) != 0 )
26        NtTerminateProcess(-1i64, error);
27    LODWORD(HardErrorMode) = 0;
28    NtSetInformationProcess((HANDLE)-1i64, ProcessDefaultHardErrorMode, &HardErrorMode, 4u);
29    NtTerminateThread(-2i64, error_1);
30    return 0;
31 }

```

Figure 4.64: Entry point of csrss.exe application on Windows 10.

In practice, the real entry point of csrss.exe is in `CsrServerInitialization` function since it retrieves the same parameters provided in `main`. One of the first action performed is to retrieve the current number of session from where csrss process is running with `RtlGetCurrentServiceSessionId` (from `ntdll.dll`). Then, csrss is removing unnecessary rights it could have inherited from `smss.exe` with internal functions `CsrRemoveUnneededPrivileges` and `CsrSetProcessSecurity`. It allocates a tagged "CSRSS!" "TMP" heap manager for itself with `RtlCreateTagHeap` (from `ntdll.dll`). This heap will be intensively used subsequently by different functions to allocate memory. Csrss service is driven with a couple of central structures holding most of information the service could need. One of them is allocated with `CsrInitializeProcessStructure` and stored in a global value called `CsrRootProcess` to be accessible in any point of the service.

After all initialization and security adjustments, there is a call to `CsrParseServerCommandLine`. This function is central since it drives the way csrss is running thanks to the command line parameters provided by `smss.exe`. These parameters, on a standard Windows 10 operating system, under Session 1 look like:



```
%SystemRoot%\system32\csrss.exe ObjectDirectory=\Windows SharedSection=1024,20480,768 Win-
dows=On SubSystemType=Windows ServerDll=basesrv,1 ServerDll=winsrv:UserServerDllInitialization,3
ServerDll=sxssrv,4 ProfileControl=Off MaxRequestThreads=16'
```

Command line parsing is just a piece of the task of `CsrParseServerCommandLine` function. Two symbolic objects are created "`\Sessions\s_(0)\BaseNamedObjects`" (where `s_(0)` references the current session number) and "`\DosDevice`". But parsing and acting according to the command line is the main job of this function. For instance, `ObjectDirectory` parameter and its value (here "`\Windows`") are used to create a directory object in memory of the shape "`\Sessions\s_(0)\Windows`", with specific rights so that it can be used as a symbolic object to communicate with `csrss`. This procedure is hold by `CsrServerCreateApiPort` function which is more or less a wrapper to `CsrApiPortInitialize` function. Technically, communication is performed via an ACPL port (crafted with `NtAlpcCreatePort`) in the directory previously created with the full path name of the symbolic device "`\Session\s_(0)\ApiPort`". This one is accessible by anyone since it is created with an Access Control List (ACL) [752] generated by the function `CsrpGenerateWorldAccessSD`. Then, a thread is created via `RtlCreateUserThread` to make run `CsrApiRequestThread`. But this one is not executed directly since it is created to be *suspended*. It means there is a thread structure in memory initialized correctly but the thread itself is in pause. It will be resumed to be executed itself. This thread is finally registered in the global `CsrRootProcess` structure by `CsrAddStaticServerThread` function. Note that the handle referencing this created thread will be returned at the end by `CsrParseServerCommandLine` to `CsrServerInitialization`.

Among the parameters, we have `SharedSection`. This one is responsible to create dedicated shared memory sections (whose sizes are provided as values) under the name of "`\Sessions\s_(0)\SharedSection`" via a call to `NtCreateSection` (from `ntdll.dll`). This one is accessible to anyone thanks to `CsrpGenerateWorldAccessSD` function. Note that another heap is specially crafted (under the tag names "`CSRSHR!`" and "`!CSRSHR!`") to perform the different allocations requested in parameter. At the end of `SharedSection` parameter management, `CsrParseServerCommandLine` calls `CsrLoadServerDll` with "`CSRSS`" as first parameter. It will result in loading and initializing `csrss` in the list of the different modules hold by `csrss.exe` service.

The relevant point of `csrss` service is that it holds different *internal modules*. These ones are referenced through `ServerDll` parameter. This parameter can occur more than once in the command line. Its shape is "`ServerDll=<dllname>:<entrypoint_name>,<index>`". `CsrParseServerCommandLine` function parses all the different values linked to `ServerDll`. Only the entry point name of the Dll is optional. This one corresponds to the first function to call in the Dll carrying the module that `csrss` must load in memory. This one is usually used for initialization purposes. Technically `CsrParseServerCommandLine` calls `CsrLoadServerDll` function with the Dll name, the entry point function name and the index thus constituting the three parameters of this function.

First, `CsrLoadServerDll` function loads the Dll provided via `LdrLoadDll` which can be seen as an internal function used by `LoadLibrary` [753]. Then, `CsrLoadServerDll` allocates and initializes a structure which could be documented as given in Code 4.16. After this structure initialization, there is a call to `LdrGetProcedureAddress` which is close in its philosophy from `GetProcAddress` function [754]. This one looks for an "entry point function" in a Dll by knowing the function's name. If no name has been provided (since this value is optional), `csrss` uses `ServerDllInitialization` as default entry point name. If no entry point has been defined to the module Dll, `csrss` use its own default (and minimal) function `CsrServerDllInitialization` to initialize the module for its main structures in memory. To finish the values list, `index` corresponds to an index entry in an array in `csrss` memory where to store information about the module. Information stored are those set in the structure given in Code 4.16. This structure is both initialized by `csrss` in `CsrLoadServerDll` function and in the entry point function called from the loaded Dll.

```
1 typedef struct _PARAM_INITIALIZE_SRVDLL
2 {
3     ANSLSTRING DllName;
4     PVOID hModuleDll;
5     DWORD IndexInCsrLoadedServerDll;
6     DWORD Aligment2;
7     DWORD Const1024;
8     DWORD Const1031;
```



```

9   PVOID ApiDispatchTable;
   PVOID ApiServerValidTable;
11  PVOID Reserved1 [2];
   PVOID ClientConnectRoutine;
13  PVOID ClientDisconnectRoutine;
   PVOID UserHardError;
15  PVOID ServerSectionHeap;
   PVOID Reserved2;
17  PVOID UserClientShutdownProcesses;
   WCHAR DllNameBuffer [1];
19 } PARAM_INITIALIZE_SRVDLL, *PPARAM_INITIALIZE_SRVDLL;

```

Code 4.16: "Possible documentation for the parameter provided to the entry point function of the module loaded by CsrLoadServerDll."

When there is an entry point for that module, this one is called. We would like to focus on winsrv.dll with UserServerDllInitialization and index 3. This one is loaded by CsrLoadServerDll as explained before. But its entry point of the function UserServerDllInitialization (Figure 4.65) is about to look for an extended version of its own Dll. Indeed, it first calls IsUserServerDllInitializationExtPresent and then, if winsrvext.dll exists on the system, loads it as a delayed loaded Dll [755]. In this case, the entry point of winsrv.dll calls UserServerDllInitialization from winsrvext.dll. Usually, on Windows 10, this is the default configuration. Note that, in the case of winsrvext.dll would be missing, the module tries to use a minimal version and by default itself.

```

1 | int64 __fastcall UserServerDllInitialization(struct _PARAM_INITIALIZE_SRVDLL *Param)
2 | {
3 |     struct _PARAM_INITIALIZE_SRVDLL *Param_1; // rbx@1
4 |     __int64 result; // rax@2
5 |
6 |     Param_1 = Param;
7 |     if ( (unsigned __int8)IsUserServerDllInitializationExtPresent() )
8 |         return UserServerDllInitializationExt(Param_1);
9 |     if ( (unsigned __int8)IsUserServerDllInitializationMinPresent() )
10 |         return UserServerDllInitializationMin(Param_1);
11 |     MinSrvDllTag = RtlCreateTagHeap(*(_QWORD *)(&MK_FP(_GS_, 96i64) + 48i64), 0i64, L"MINSRV!", L"TMP");
12 |     Param_1->UserHardError = UserHardError;
13 |     Param_1->ApiDispatchTable = &MinUserServerApiDispatchTable;
14 |     Param_1->ApiServerValidTable = &MinUserServerApiServerValidTable;
15 |     Param_1->UserClientShutdownProcesses = UserClientShutdown;
16 |     result = 0i64;
17 |     Param_1->Const1024 = 1024;
18 |     Param_1->Const1031 = 1026;
19 |     LODWORD(Param_1->Reserved2) = 8;
20 |     return result;
21 | }

```

Figure 4.65: Entry point called for winsrv.dll module by csrss.

The function UserServerDllInitialization from winsrvext.dll is quite interesting. It is about to create several events: *EventCancel*, *EventCancelled*, *EventRitExited*, *PowerRequestEvent*, *MediaRequestEvent*, *NlsEvent*, *FontChangeEventHKLM* (if GdiSupportsFontChangeEvent returns true), and *FontRegistryHKLM*. It is able to manage fonts in the system and it records a callback NtUserNotifyProcessCreate (from ntdll.dll) with BaseSetProcessCreateNotify (from basesrv.dll). This function is used to notify NtUserNotifyProcessCreate each time a process is created in the system. Such undocumented callback is usually reserved for drivers with PsSetCreateProcessNotifyRoutine routine [756], for instance, but it is interesting to see such feature usable in user-mode. About callbacks, UserServerDllInitialization will create and launch (with RtlCreateUserThread) in different threads the following functions: GdiAddInitialFontsThread, NotificationThread and PowerNotificationThread. All of them will be registered in internal structures of csrss via CsrAddStaticServerThread function.

But the most interesting part is in the way UserServerDllInitialization initializes its provided parameter (Code 4.16). The extract from UserServerDllInitialization is given in Figure 4.66. We can see that there are callbacks from communication management (UserClientConnect, UserHardError and UserClientShutdownProcesses). But the most relevant list of callbacks lies in UserServerApiDispatchTable provided in "ApiDispatchTable" member of the provided parameter. This table of functions lists several functions used in what seems to be the configuration of a (communication) server. One relevant among others is SrvCreateSystemThreads since it is

responsible to initialize the raw input thread procedure.

```

30 Paramter->ApiDispatchTable = UserServerApiDispatchTable; // UserServerApiDispatchTable:
31                                     // -----
32                                     // [0] -> SrvExitWindowsEx
33                                     // [1] -> SrvEndTask
34                                     // [2] -> SrvLogon
35                                     // [3] -> SrvActivateDebugger
36                                     // [4] -> SrvCreateSystemThreads <<--
37                                     // [5] -> SrvRecordShutdownReason
38                                     // [6] -> SrvCancelShutdown
39 Paramter->Const1031 = 0x407;
40 Paramter->ApiServerValidTable = &UserServerApiServerValidTable;
41 Paramter->ClientConnectRoutine = UserClientConnect;
42 Paramter->UserHardError = UserHardError;
43 Paramter->UserClientShutdownProcesses = UserClientShutdownProcesses;

```

Figure 4.66: Part of UserServerDllInitialization in winsrvext.dll, initialization of the provided parameter with useful callbacks.

All the modules listed in parameters and which have been loaded with `CsrLoadServerDll` are referenced in a global structure called `CsrLoadedServerDll`. This structure in memory is nothing else than an array of pointers, each holding the parameter (Code 4.16) of the module loaded. The index where the module is stored is given by the index value provided in the command line parameters for each module. In the case of `winsrv.dll`, the index value is 3 meaning it is stored in the 4<sup>th</sup> entry of the array.

Once all the different modules have been loaded and the last parameters have been parsed, `CsrParseServerCommandLine` function returns to `CsrServerInitialization` function. The initialization function tries to initialize the different modules directly in memory if some values have been set previously as parameter. This is not the case on a clean Windows 10 setup. Initialization continues with `CsrSbApiPortInitialize` which initializes a communication port thanks to `NtCreatePort` (from `ntdll.dll`) under the name `"\Session\s_(0)\SbApiPort"`. This one is a more restrictive symbolic object device to communicate with `csrss` service since its access is configured by `CsrCreateLocalSystemSD` function. A thread is created and run with `RtlCreateUserThread` and `NtResumeThread` (which is close to `ResumeThread` [757]) to execute the code of the function `CsrSbApiRequestThread`. This thread is — as all threads created by `csrss` — added to internal structures via `CsrAddStaticServerThread` function call. Close to the end of the initialization procedure, `RtlConnectToSm` from `ntdll.dll` is called. This function is not documented but supposed to be used to connect to the NT Session Manager (`smss.exe`) which launched `csrss.exe`.

Finally, `NtResumeThread` is used on the thread's handle initiated by `CSRSERVERCREATEAPIPORT`. This thread is created with `CsrApiRequestThread` function to be executed. This function is quite complex since it does a lot of things and called from multiple places with different parameters. First, this function checks on which session it is running. When it is executed in the context of Session 1, it is enough to launch device interface handling. An event is set and after a call to `AlpcInitializeMessageAttribute` function (from `ntdll.dll`), a loop is engaged. This one starts by calling `NtAlpcSendWaitReceivePort` with `CsrApiPort` which references a previously created `APIPort`. This function is undocumented and the detailed behavior is out of the scope of this section. But, for the sake of simplicity, this one is used to communicate with a system client using ALPC API. From that part, there are two possibilities. Either there is a client with whom to communicate and the function does not wait to continue its execution. Either, the function waits until there is connection from a client.

Once there is a possible interaction with a client, `Alpc GetMessageAttribute` function is called, followed by `CsrLocateProcessByClientId`, `CsrProcessLazyRegister` or `CsrRegisterThread` and finally `NtAlpcAcceptConnectPort`. The first time this procedure is called, there is no wait for `NtAlpcSendWaitReceivePort` which results in an execution of the loop. At the second occurrence of the loop, `RtlCreateUserThread` (and thereafter `CsrAddStaticServerThread`) is called `CsrApiRequestThread` itself as the function to execute with the thread. But the provided parameter is changing from one loop execution to another. If everything is correct, the thread is run with `NtResumeThread`. But right after this thread has been resumed, there is a call to one function pointer for each occurrence of the loop. Among the pointers of function, we have the second occurrence of `CsrSrvClientConnect` (from `csrsrv.dll`)

but more important, there is a call to `SrvCreateSystemThreads` from `winsrvext.dll` at the fourth occurrence. In order to be complete, the remaining of the loop occurrence when a thread is created calls `CsrReleaseCapturedArguments` and `CsrReplyToMessage` among the most relevant functions to mention.

The function `SrvCreateSystemThreads` from `winsrvext.dll` is given in Figure 4.67. This function calls `CsrExecServerThread` which is an exported function from `csrsrv.dll`. This function is nothing else than the regular create thread procedure used by `csrss` service. It means that we have `RtlCreateUserThread` function called with a function pointer to be executed as an independent thread. Then, it comes a sort of inline version of `CsrAddStaticServerThread` function.

```

1  int64 SrvCreateSystemThreads()
2  {
3      NTSTATUS status; // ebx@1
4
5      status = CsrExecServerThread(StartCreateSystemThreads, 0i64);
6      if ( status < 0 )
7          NtUserCallNoParam(0xFi64);
8      return status;
9  }

```

Figure 4.67: Callback `SrvCreateSystemThreads` from `winsrvext.dll` and registered by `UserServerDllInitialization`.

Finally, `StartCreateSystemThreads` from `winsrvext.dll` (Figure 4.68) is executed in a dedicated thread. This function first calls `CsrConnectToUser` from `csrsrv.dll`. This last function checks first if `ClientThreadSetupRoutine` global value has been registered via `CsrRegisterClientThreadSetup`. In this case, the function recorded at `ClientThreadSetupRoutine` is called and its output returned to `StartCreateSystemThreads`. Otherwise, the function tries to retrieve the address of CSR client thread from the caller. Finally, there is a call to `NtUserCallNoParam` (from `win32u.dll`).

```

1  void StartCreateSystemThreads()
2  {
3      __int64 ConnectedUser; // rbx@1
4
5      ConnectedUser = CsrConnectToUser();
6      NtUserCallNoParam(5i64);
7      if ( ConnectedUser )
8          CsrDereferenceThread(ConnectedUser);
9      RtlExitUserThread(0i64);
10 }

```

Figure 4.68: Function `StartCreateSystemThreads` from `winsrvext.dll`.

Function `NtUserCallNoParam` is particularly interesting for us. Contrary to what its name is suggesting, this one does not belong to `ntdll.dll` but to `win32u.dll`. This library is the user-mode interface for `win32k.sys` driver, the same way `ntdll` is the interface for `ntoskrnl.exe`. Indeed, it provides native system calls interface to interact with `win32k.sys`. That way, its code is composed of a single `syscall` instruction initialized with `rax` equals to `0x109E` in our Windows 10 version. Actually, even if historically it was `win32k.sys` which was interfaced by `win32u.dll`, we are not really interacting with `win32k.sys` driver only. Indeed, from Windows 10, `win32k.sys` has been split in different *sub-drivers* to reduce the attack surface of the window manager by reducing the complexity of the code and eliminating many of its legacy pieces [758]. Technically `win32k.sys` still exists but it is just a giant stub to original routines hold by `win32kfull.sys` and `win32kbase.sys` mostly.

In our version of Windows 10, `NtUserCallNoParam` is exported by `win32kfull.sys`. Note that there are other routines such as `NtUserCallOneParam`, `NtUserCallHwndParamXxx`, and `NtUserCallTwoParam` in `win32kfull.sys` (with the last not exported as `syscall` via `win32u.dll`). They all work more or less the same way, with the

notable exception of the parameter(s) passed as arguments. Despite its name, `NtUserCallNoParam` takes only a single parameter that corresponds to an index. This one is stored first in `rdi` register (Figure 4.69) and used in `"rax, [rcx+rdi*8]"` where `rcx` is the base address of `apfnSimpleCall` global value.

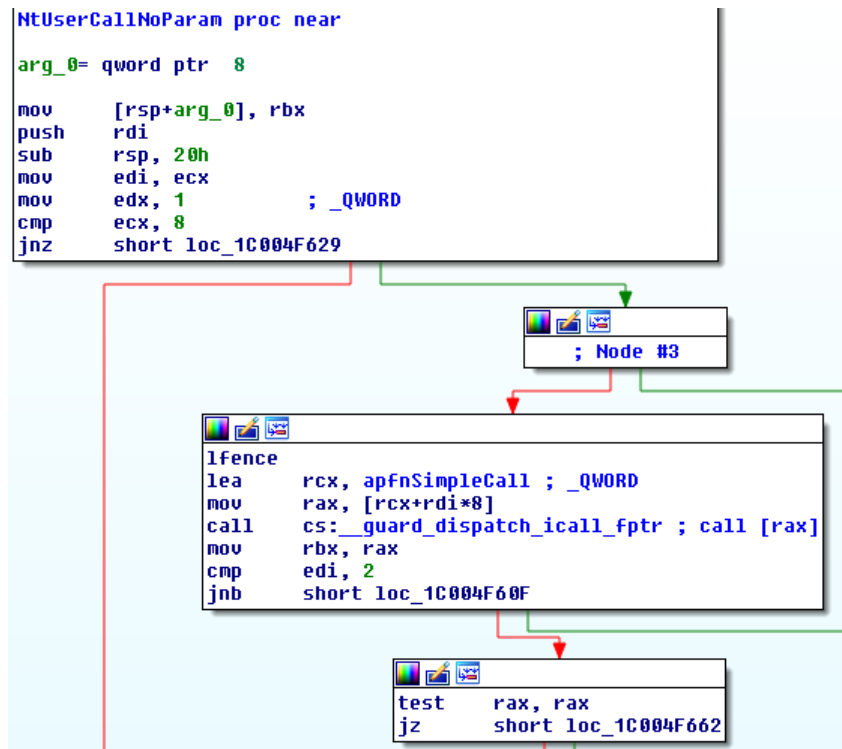


Figure 4.69: Part of `NtUserCallNoParam` from `win32kfull.dll`.

This last instruction extracts a function pointer from `apfnSimpleCall` which is a huge table of routine pointers. An extract from that list is given in Figure 4.70. This table has around 44 routines referenced. An application can perform a call to one of these routines by calling `NtUserCallNoParam`. That way, the application provides as first parameter the value of the index corresponding to the entry in the `apfnSimpleCall` table. In our case, index provided is `0x05` which corresponds to `xxxCreateSystemThreads_0` routine in `win32kfull.sys`.

```

apfnSimpleCall dq offset _CreateMenu
               dq offset _CreatePopupMenu
               dq offset _AllowForegroundActivation
               dq offset _CancelQueueEventCompletionPacket
               dq offset xxxClearWakeMask
               dq offset xxxCreateSystemThreads_0
               dq offset zzzDestroyCaret
               dq offset _DisableProcessWindowsGhosting
               dq offset _DrainThreadCoreMessagingCompletions
               dq offset xxxGetDeviceChangeInfo
               dq offset _GetIMEShowStatus
               dq offset _GetInputDesktop
               dq offset _GetMessagePos

```

Figure 4.70: List of routines which can be used via `NtUserCallXxx` routines selected via an index provided as first parameter.

This last routine is just a wrapper which directly jumps to `xxxCreateSystemThreads` in `win32base.sys`. This

routine (extract is given in Figure 4.71) retrieves parts of its parameters through `CSTPop` routine call. That one is based on global structures initialized at different points in the driver. Specifically, it allows to retrieve and to initialize a "Reason" value. If this one is equal to 2 (which is our case in Session 1 with that call to `NtUserCallNoParam`), `RawInputThread` routine (from `win32kfull.sys`) is called. This is the start of the *raw input thread* procedure.

```

1 NTSTATUS __stdcall xxxCreateSystemThreads()
2 {
3     LARGE_INTEGER *v0; // rbx@8
4     struct tagTHREADINFO *v1; // rbx@12
5     struct tagTHREADINFO **v2; // rax@13
6     -----
35  if ( (PVOID)PsGetCurrentProcess() == gpepCSRSS && (unsigned int)CSTPop((unsigned int *)&Reason, (void **)&param) )
36  {
37      if ( (_DWORD)gdwInAtomicOperation && gdwExtraInstrumentations & 1 )
38          KeBugCheckEx(0x160u, (unsigned int)gdwInAtomicOperation, 0i64, 0i64, 0i64);
39      UserSessionSwitchLeaveCrit();
40      switch ( (_DWORD)Reason )
41      {
42      case 1:
43          if ( (signed int)IsxxxDesktopThreadSupported() >= 0 )
44              xxxDesktopThread(param);
45          break;
46      case 2:
47          if ( (signed int)IsRawInputThreadSupported() >= 0 )
48              RawInputThread(param);
49          break;
50      case 4:
51          VideoPortCalloutThread(param);
52          break;
53      }
54      v0 = (LARGE_INTEGER *)PsGetCurrentThreadWin32Thread();
55      if ( v0 )
56          v0[1] = KeQueryPerformanceCounter(0i64);

```

Figure 4.71: Extract from the code of `xxxCreateSystemThreads` routine in `win32kbase.sys`.

Among different possibilities taken by `xxxCreateSystemThreads`, this one is about to launch two other threads. These threads are similar in their architecture to the raw input thread. The first is `xxxDesktopThread` (from `win32kfull.sys`) named "Win32k Desktop Thread (NOIO-DT)". That one is responsible to handle and dispatch general window management messages to the rest of the applications running. The last thread is `VideoPortCalloutThread` (from `win32kbase.sys`) unnamed but quite complex. It seems that it is responsible, among other things, to handle remote desktop connection (use of routines such as `UserRemoteConnectedSessionUsingWddm`, `xxxWaitForVideoPortCalloutReady`, `UserRemoteConnectedSessionUsingWddm` and general display with `DrvProcessDxgkDisplayCallout` where "Wddm" stands for Windows Display Driver Model [759]). Note the call to `xxxSetCsrssThreadDesktop` routine from `win32kfull.sys` is the heart of the video port communication thread with the use of `xxxSetCsrssThreadDesktop`, `xxxInternalGetMessage`, `xxxDispatchMessage` and `xxxSetThreadDesktop` to manage the video threads' interface.

#### 5.1.4 What is the purpose of the raw input thread?

##### Resume 28:

- ☞ In this section and its subsections, we will describe all the actions (from initialization to final execution) of the Raw Input Thread.
- ☞ In the following sub-sections, an arbitrary division has been made of the features associated with the RIT, for the sake of simplicity.

The behavior of the raw input thread is quite complex since it handles a lot of different things for different purposes (keyboard mainly, but mouse too). First thing to note is its original context of execution. Indeed, if this thread belongs to the user-mode `csrss.exe` process, the thread remains a kernel-mode thread executing privileged ring-0 code. This is not an issue since `csrss` service is highly privileged meaning it has all required rights to interact with system threads. But it remains that `RawInputThread` is a large routine. For the sake of clarity, we propose to split the analysis in different phases. In addition, we propose to represent the main actions done by the raw input thread in different figures (since a general one would be too big to be readable).

##### 5.1.4.1 General initialization of Raw input Thread

##### Key Point 4.30:

- ☞ Beginning of *raw input thread* is about events initialization and hot-keys (keyboard shortcuts) registration.

The beginning of `RawInputThread` routine is given in Figure 4.72. At the beginning, the raw input thread sets its internal name with `SetThreadName` and its priority to 16 (which is a real time class priority observed in Code 4.15) [739]. A call to `InitKeyboard` routine is responsible to initialize to zero a set of global values in `win32kfull.sys`. A call to `GetBiosNumLockStatus` routine is performed to retrieve information from keyboard devices connected to the system. Information are retrieved via `IoQueryDeviceDescription`<sup>21</sup> which is an obsolete routine [760]. This obsolete routine is called with a callback routine `KeyboardDeviceSpecificCallout` which used to handle information for each keyboard. The call to `IoQueryDeviceDescription` is performed in a loop where the first parameter is incremented. In fact, this browses all possible values of `INTERFACE_TYPE` enumeration values [761]. Technically, this operation is performed to handle all bus driver types on the machine. The callback is only notified in the context of the Session 1. This one targets information with ISA bus type on:

```
"HKLM\HARDWARE\DESCRIPTION\SYSTEM\MultifunctionAdapter\0\KeyboardController\0\KeyboardPeripheral\0"
```

Then, the RIT initializes internal data in its `Win32Thread` structure (which is not documented). Then comes specific mouse and keyboard initialization. Note that in the case of keyboard, this operation is performed only on interactive sessions (not session 0). Its goal is to register specific callbacks on different key combinations. `SetDebugHotKeys` is a routine used to read in registry of Windows "HKLM\Software\Policies\Microsoft\Windows\UserDebuggerHotkey" to know how to bind these key combinations. This registry configuration is undocumented and not present by default on a fresh Windows 10 installation.

The case of `SetWinlogonHotKeys` routine aims to manage `Winlogon.exe` interactions. The list of hot keys currently implemented is given in Figure 4.73 and callbacks names (second parameter provided to `RegisterHotKey` routine) are a good clue to the activities targeted by these recorded keyboard shortcuts.

The call to `SetWindowArrangementHotKeys` routine aims to record an internal list of keys linked to `WindowArrangementHotKeyCallback` callback (Figure 4.74). This one posts directly its messages to the thread's message queue via a call to `PostEventMessageEx` routine which finally calls `CTouchProcessor::ReferenceMsgDataExternal`

<sup>21</sup><http://www.codewarrior.cn/ntdoc/winnt/io/IOQueryDeviceDescription.htm>

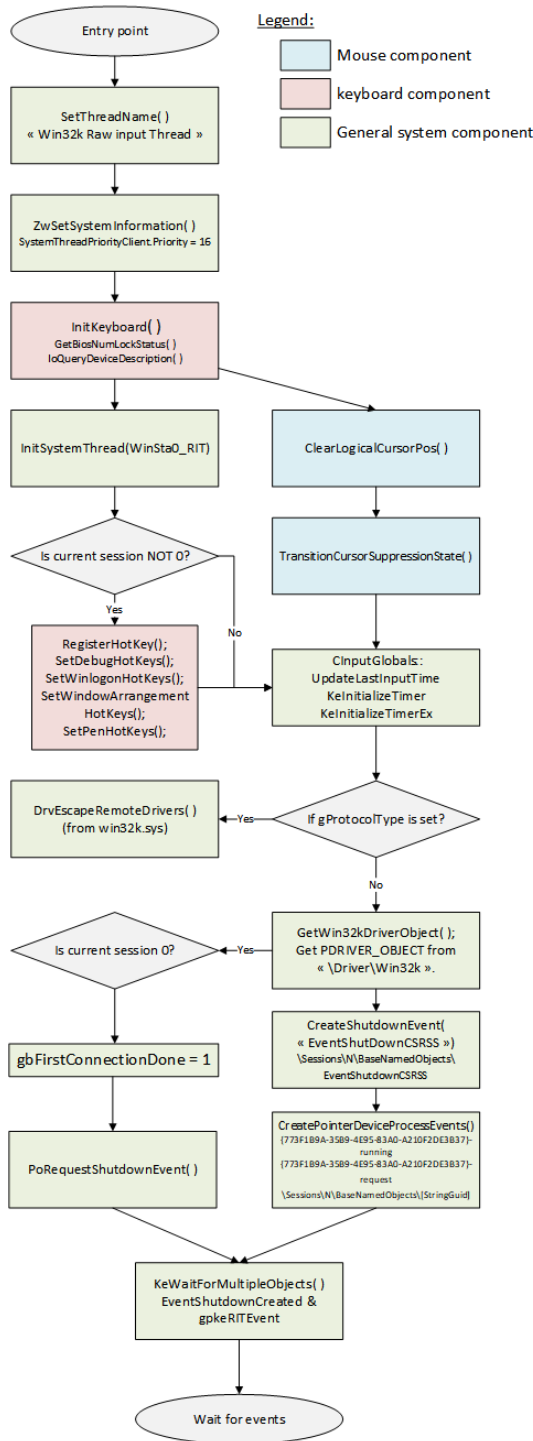


Figure 4.72: Beginning of RawInputThread routine from win32kfull.sys — Initialization and wait for notification events.

routine in win32kbase.sys. This last routine is calling ultimately CTouchProcessor::ReferenceMsgData routine.

Then, to finish the beginning of the raw input thread procedure, there is the initialization of different callbacks or events, notably to be notified in case of a shutdown. This makes sense since the RIT is very close to the system and any breakout of that thread could paralyze the whole system. Shutdown procedure is handled



```

1 void SetWinlogonHotKeys(void)
2
3 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 0, 0x8003, 0x2Eu);
4 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 4u, 6, 0x1Bu);
5 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 5u, 0x8008, 0x4Cu);
6 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 6u, 0x2008, 0x55u);
7 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 8Du, 0x6000, 0x0u);
8 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 0xEu, 0x6008, 0x0u);
9 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 0xCu, 0x6008, 0x8Bu);
10 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 0xCu, 0x6008, 0x6Bu);
11 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 0xFu, 0x2007, 0x46u);
12 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 0x8u, 0x6000, 0x70u);
13 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 7u, 0x2008, 0x50u);
14 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 8u, 0x2008, 0x50u);
15 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 9u, 0x200C, 0x50u);
16 RegisterHotKey(0164, (ULONG_PTR)RotationLockCallback, 0xFFFFFFFF, 0x2008, 0x4Fu);
17 RegisterHotKey(0164, (ULONG_PTR)WinlogonWinSpaceCallback, 0xFFFFFFFF8, 0x2008, 0x20u);
18 RegisterHotKey(0164, (ULONG_PTR)WinlogonWinSpaceCallback, 0xFFFFFFFF7, 0x2008, 0x20u);
19 RegisterHotKey(0164, (ULONG_PTR)WinlogonWinSpaceCallback, 0xFFFFFFFF6, 0x200C, 0x20u);
20 RegisterHotKey(0164, (ULONG_PTR)WinlogonWinSpaceCallback, 0xFFFFFFFF5, 0x200C, 0x20u);
21 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 7u, 0x2008, 0x35u);
22 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 8u, 0x2008, 0x85u);
23 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 9u, 0x200C, 0x85u);
24 RegisterHotKey(0164, (ULONG_PTR)RotationLockCallback, 0xFFFFFFFF3, 0x6008, 0x70u);
25 RegisterHotKey(0164, (ULONG_PTR)WinlogonHotkeyCallback, 0x8u, 0x6008, 0x7Fu);
26 RegisterHotKey(0164, (ULONG_PTR)PenHotkeyCallback, 0xFFFFFFFF2, 0x4800, 0x87u);
27 RegisterHotKey(0164, (ULONG_PTR)DisplayDiagHotkeyCallback, 0xFFFFFFFF0, 0x6808, 0x80u);
28 }

```

Figure 4.73: Content of SetWinlogonHotKeys routine.

```

1 void SetPenHotKeys(void)
2 {
3     if ( (unsigned int)Feature_PenTailDockEvents__private_IsEnabled() )
4     {
5         RegisterHotKey(0164, PenHotkeyCallback, 0x1Eu, 0x4208, 0x83u);
6         RegisterHotKey(0164, PenHotkeyCallback, 0x1Fu, 0x4208, 0x82u);
7         RegisterHotKey(0164, PenHotkeyCallback, 0x20u, 0x4208, 0x81u);
8         RegisterHotKey(0164, PenHotkeyCallback, 0x21u, 0x4208, 0x83u);
9         RegisterHotKey(0164, PenHotkeyCallback, 0x22u, 0x4208, 0x82u);
10    }
11 }

```

Figure 4.74: Content of SetPenHotKeys routine.

carefully by the RIT to stop everything is a clean way. Finally, the RIT waits for a set of different events. Some of them are linked to shutdown procedure, others are initialized in diverse points of the driver are specific RIT events. Finally the RIT calls KeWaitForMultipleObjects routine [762] to wait for one of the event to be signaled. Most of the notification for this wait procedure are linked to initialization time or shutdown time by the operating system. The rest of operations in RawInputThread is more relevant.

#### 5.1.4.2 First part of devices initialization by *Raw Input Thread*

##### Key Point 4.31:

- ☞ The RIT acts according to the current session (session number 0 or another one).
- ☞ RIT is able to manage *desktop* switching in a session.
- ☞ Desktops provides a security with message system isolation and by resetting keyboard state whenever switching.

In Figure 4.75 is given the next part of the RIT procedure. This one is still in the general initialization of the thread. Regarding to one of the possibilities to wake up, the procedure has three possibilities. The first is to reset some timers and finish the RIT. This option only happens at the initialization phase. But it is not a big deal since a new RIT is reengaged soon after. Among the remaining two other possibilities, the first is about setting an event according to the output of GetRITWakeReason (which is just about a global value test). The second is about desktop switching.

According to the Microsoft's documentation [763], a desktop has a logical display surface and contains user interface objects such as windows, menus, and hooks. More directly, it is used to create and manage windows. This way, window messages are only processed between processes that are on the same desktop. By default, there are three desktops in the interactive window station: *Default*, *ScreenSaver*, and *Winlogon*.

The *Default* desktop is the one with whom the user interacts with. It is created by Winlogon and activated once the user is logged. The *ScreenSaver* desktop [764] is activated when the secure screen saver functionality has been activated on the system [765]. The system automatically switches to the *ScreenSaver* desktop when the machine is about to go in sleep mode. This is done in order to prevent processes on the default desktop from unauthorized user's access. Finally, the *Winlogon* desktop is active while a user logs on. This is the interface requiring a user name and optionally a password to get access to the user's session before displaying the user's desktop on classic Windows client. This desktop is used when a user is going to run an application with administrator rights<sup>22</sup>.

<sup>22</sup>User Account Control (UAC) management should be always required to enter a password to be securely efficient. But UAC is not a security feature. As Larry Osterman noted [766], it is a convenience feature that acts as a forcing function to get software developers to get their act together. UAC is not part of the *Windows Integrity Mechanism Design* [767].

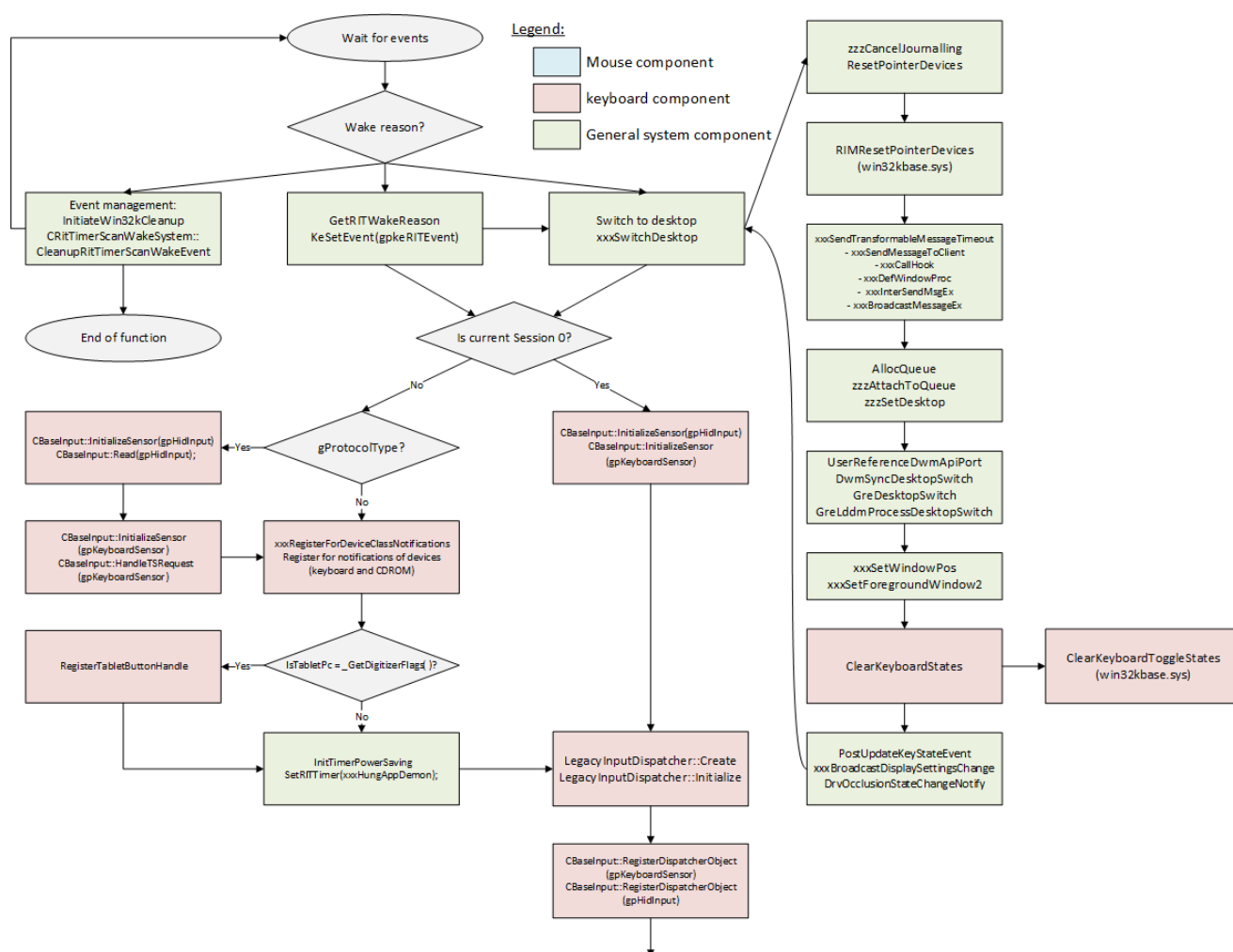


Figure 4.75: Setup of RawInputThread routine from win32kfull.sys — Initialization of keyboard devices and desktop.

To ask the administrator password, *User Account Control* interface is displayed as given in Figure 4.76<sup>23</sup>. The password is entered by the user in another desktop preventing the desktop of the current application to be notified by keystrokes from that different desktop (since messages of a session are private to other sessions). Note that this security is far from being perfect since applications from other desktops are still running (they are just starved from graphics resources) [769] and they can still get access to the keyboard by other means.

<sup>23</sup>Elevation interface has evolved from Windows 8 [768]. At the beginning, the image in background was a snapshot of the user's desktop, including all open windows. But from Windows 8, the background is now the current user's wallpaper. This change happened when Microsoft observed the screen capture performed by the elevation prompt often looked ugly because there was a good chance it caught an animation mid-stream. To avoid that and because they had other things to perform with higher priority, they decided to simplify the process.

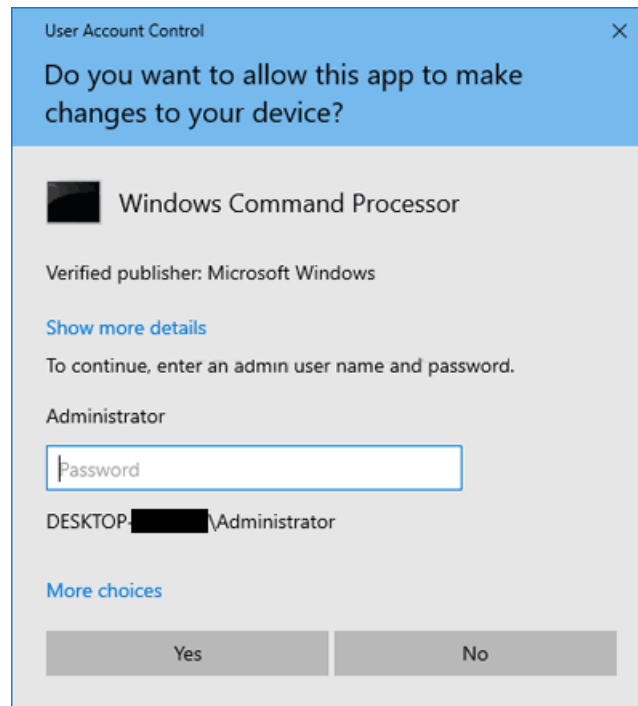


Figure 4.76: Winlogon desktop used to handle UAC.

#### Key Point 4.32:

- ☞ The raw input thread deals with different *desktops* for a given user's session.
  - ☞ System of messages for GUI windows are private for each desktop.
  - ☞ It is used for *User Account Control* interface purposes and CTRL+ALT+DEL key sequence management.
  - ☞ This is not a perfect security (because applications from another desktop could get access the keyboards by privileges means) but this constitutes a first level of security.

Contrary to the *ScreenSave* desktop, the *Winlogon* desktop is active while a user logs on. Indeed, whenever the user pressed the CTRL+ALT+DEL key sequence, the system switches to the *Winlogon* desktop the same way it does with the UAC dialog box. Once the desktop has been switched, the call to `ClearKeyboardStates` (from `win32kfull.sys`) and finally to `ClearKeyboardToggleStates` (from `win32kbase.sys`) resets the internal structure in `win32k` representing the current state of every key of the keyboard. This structure is particularly used by asynchronous requests to know or update the state of different keys in the keyboard.

#### Key Point 4.33:

- ☞ The current keyboard internal state is cleared when switching to another desktop.
  - ☞ This reduces the means of access to the keyboard for a keylogger (section 5.3.1).

Once the desktop initialization has been performed, the raw input thread is about to initialize keyboard input notifications. The procedure is different for the RIT running in Session 0 than from the one running in Session 1. The difference lies in the fact that Session 0 has normally no interaction. But it remains that the RIT still follows device activity to be notified if a device is added or removed. To proceed, it initializes two *Sensors* with `CBaseInput::InitializeSensor` routine (from `win32kbase.sys`). The sensor initialization is always performed in the context of Session 0 and sometimes in the context of another session. It depends on the `gProtocolType` value

which is imported from `win32kbase.sys` by `win32kfull.sys`. This value is initialized by `InitVideo` and `SetProtocolType` (called by the `xxxRemoteConnect` routine only). It is hard to explain what this value could exactly mean since its name is not really meaningful and it is manipulated in the context of totally undocumented structures.

From our observations based on Windows 10 installed in a virtual machine (VirtualBox), `gProtocolType` is equal to zero, meaning the procedure of sensors initialization is not directly performed by the raw input thread. It is not a big deal since this initialization is performed later in the `xxxRegisterForDeviceClassNotifications` routine (section 5.1.4.5), no matter what happened before. But for the sake of completeness and because we are observing this procedure now and it will be used later, we propose to explain what is happening with the `CBaseInput::InitializeSensor` routine.

### 5.1.4.3 Initialization and read from sensors

#### Key Point 4.34:

- ☞ There are a differences in initialization for read procedure between session 0 and other sessions.
- ☞ RIT manages differently three types of devices (called *sensors*): *mouse*, *keyboard* and *HID* devices.
  - ☞ HID devices represent all device which can provide data except keyboard (for instance, a *gamepad*).
- ☞ The read procedure with the RIT is implemented with a partially oriented object representation.
  - ☞ There is a rather complex embedding of object oriented routines and classical ones.
- ☞ Once the sensor initialization is performed, the read operation is engaged for each sensor (i.e. device).
  - ☞ To simplify, the read operation call stack is: `CBaseInput::Read`, `RIMReadInput`, `rimIssueReads`, `RIMStartDeviceRead` and finally `rimCompleteReads` routines.
  - ☞ The prefix *RIM* used at the beginning of some routines' names refers to *Raw Input Manager*.
- ☞ The read procedure is technically engaged by `RIMStartDeviceRead`.
  - ☞ This routine calls `ZwReadFile` with an APC routine `rimInputApc` in an undocumented way to read from the targeted device. The call to `ZwReadFile` is enough to engage a read IRP to lower drivers.
  - ☞ The `rimInputApc` routine is supposed to handle the keystroke content and to reengage the read procedure through `rimProcessDeviceBufferAndStartRead`.
  - ☞ Thus, after each reading, the reading operation is always reengaged.
  - ☞ To deal with cancel orders or any error, `rimInputApc` tries to read from the device five times before giving up.
- ☞ Once the read operation handled by `RIMStartDeviceRead` is done, `rimCompleteReads` can be notified.
  - ☞ As any completion routine, `rimCompleteReads` is called once keystroke content from keyboard device has been read.
  - ☞ Internally, `rimProcessDeviceBufferAndStartRead` is used to manage the content from the keyboard.
  - ☞ This is one via `rimProcessHidInput` and `rimProcessKeyboardInput` transfers buffers from kernel-mode to user-mode in `csrss.exe`'s memory.
  - ☞ The read operation is finally reengaged.
- ☞ This implementation, which may seem redundant, depends in fact on the nature of the device to be read.
  - ☞ More directly, Windows implements here all possible cases to interface with various devices.

The initialization of sensors by `CBaseInput::InitializeSensor` routine calls different callbacks from the provided structure (`gpHidInput`) and it initializes the sensor via `CRIMBase::CreateHandles`. The initialization of sensor is based on different routines where we note a call to `ApiSetEditionHidAutoRepeatTimeout` (calling `IsEditionHidAutoRepeatTimeoutSupported` and `EditionHidAutoRepeatTimeout`) and another to `RawInputManagerObjectResolveHandle` routine.

But the most interesting routine called by `CRIMBase::CreateHandles` is `RIMRegisterForInput` (from `win32kbase.sys`) called with `CBaseInput::RIMCallback`. This one is a wrapper to the `RIMRegisterForInputWithCallbacks` (from

win32kbase.sys) routine called with the Win32k's *DriverObject* [770, 771]. The description of *RIMRegisterForInputWithCallbacks* is given in Figure 4.77. This one aims to register internal structures with the callback provided for each input device referenced in Win32k. The central point of the operation is performed in the *RIMDiscoverDevicesOfInputType* routine. This one lists the devices matching the device type referenced as input type and it initializes a long list of internal and undocumented structures with the different parameters provided, including the callback *CBaseInput::RIMCallback*.

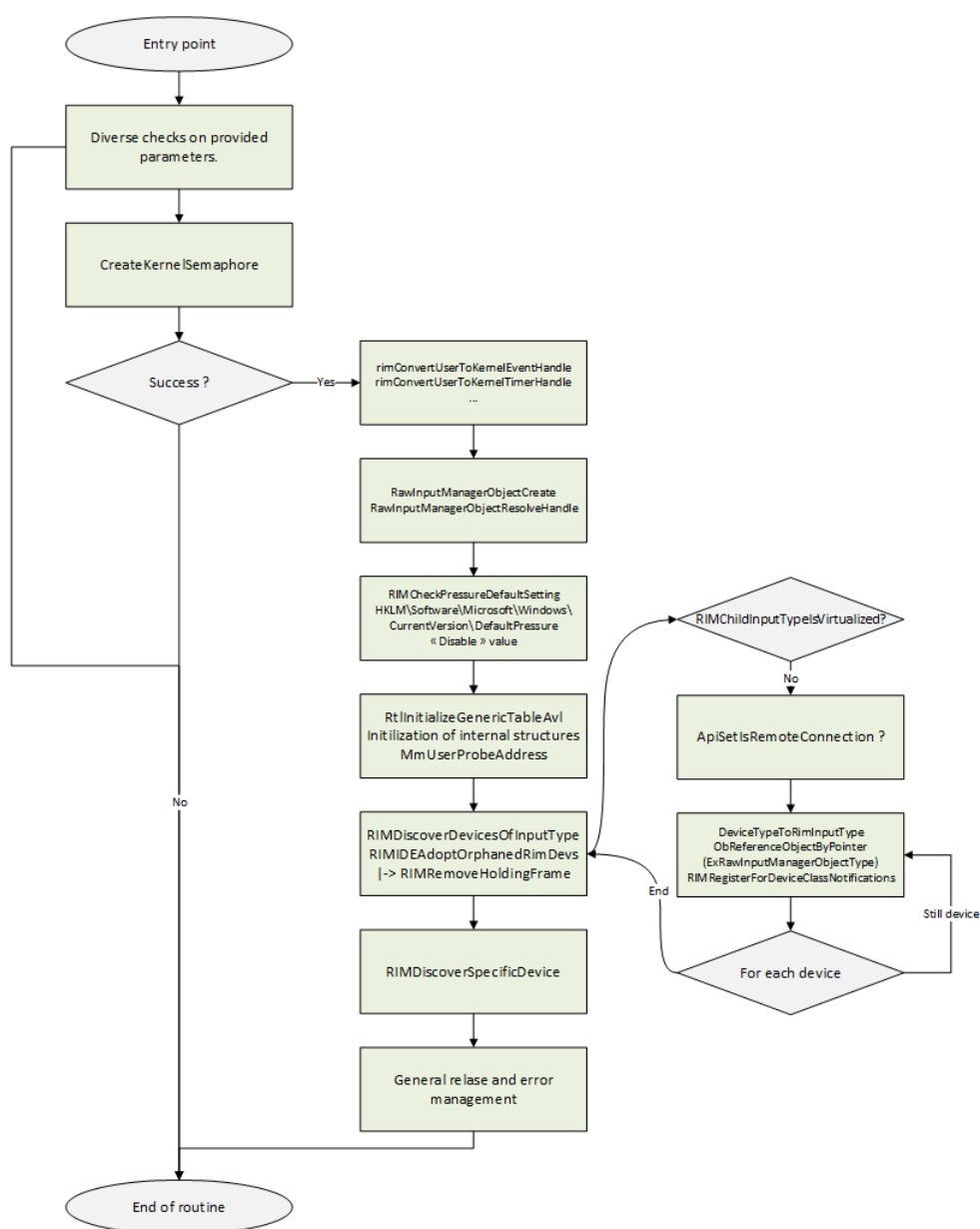


Figure 4.77: Simplified pseudo-code of *RIMRegisterForInputWithCallbacks* routine (from win32kbase.sys).

The callback *CBaseInput::RIMCallback* matters since it interacts with the plug and play (PnP) manager. Its simplified pseudo-code is given in Figure 4.78. The role of *CBaseInput::RIMCallback* is to be notified during the life of the device it represents. This callback is mainly focused on interacting with the PnP manager in Win32k architecture. But it is also useful since it communicates with different components of the keyboard device (*gkeyboardInfo* structure or HID related structures). The callback is called with a special structure where the first member is a number where value is set between 0 and 5. It corresponds to a callback in an undocumented

table of pointer of routines. To give to this table a name, we call it `CBaseInput_RIMDeviceCallback_Xxx` in Figure 4.78. All the callbacks are present to manage device's internal states (created, opened, closed, destroyed and reset). Once the selected callback from the table has been executed, information is gathered in a global structure which holds device's information. Finally, information is broadcasted to the PnP manager via an ALPC port.

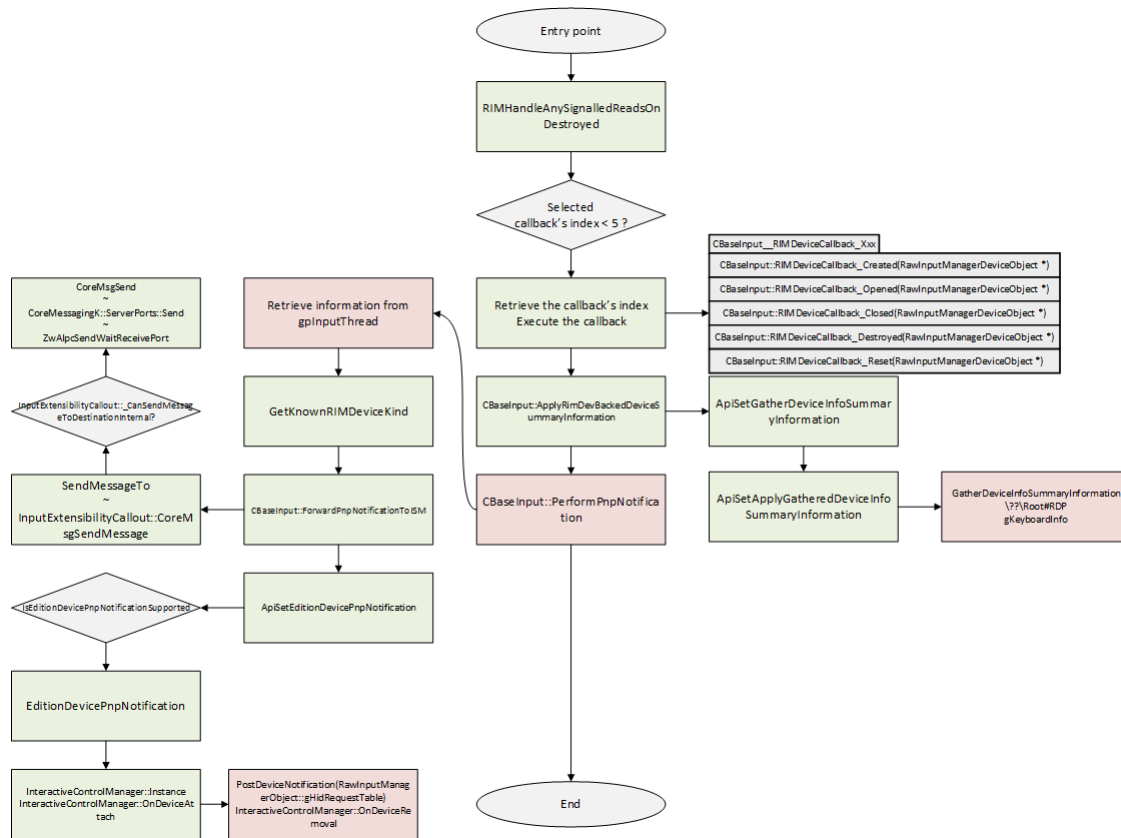


Figure 4.78: Simplified pseudo-code of `CBaseInput::RIMCallback` routine (from `win32kbase.sys`).

Coming back to the `RawInputThread` routine described in Figure 4.75, in the context of Session 0, `CBaseInput::InitializeSensor` routine is called on both `gpHidInput` and `gpKeyboardSensor`. This helps the RIT from Session 0 to follow device's activity without having to manage direct event coming from the device interaction. The case of Session 1 is an extended procedure from Session 0 when `gProtocolType` is set. Sensor `gpKeyboardSensor` is initialized with `CBaseInput::InitializeSensor` but it is directly handled by `CBaseInput::HandleTSRequest` routine. This one is more about logging activity that performing a real work, without mentioning `CRIMBase::SensorDoWorkAndWait` which calls `CRIMBase::SensorDispatcherObject::GetWorkAndWait`.

Sensor `gpHidInput` is also initialized with `CBaseInput::InitializeSensor`. But this one is directly engaged to a read operation with `CBaseInput::Read` (from `win32kbase.sys`) routine. This routine is relevant since it follows requirements of HID (and by consequence USB devices) where a read IRP operation must always be engaged. The pseudo-code of `CBaseInput::Read` is not very complex (Figure 4.79) since it retrieves first the `dispatcher` handle from the "CRIMBase" class and then it calls `RIMReadInput` from `win32kbase.sys`. Note that a `dispatcher` is an object that manages a queue of work items and runs (i.e. *dispatches*) them in some order, usually to a dedicated thread.

Routine `RIMReadInput` is a bit more complex than `CBaseInput::Read`. This routine is called with the first parameter initialized to the device's handle (`gpHidInput` in our case). The first action is to convert that handle to the kernel object representing it. This is performed through `RawInputManagerObjectResolveHandle` which



```

1  __int64 __fastcall CBaseInput::Read(CBaseInput *this)
2  {
3      CBaseInput *this_1; // rbx@1
4      void *DispatcherHandle; // rdi@1
5
6      this_1 = this;
7      DispatcherHandle = (void *)CRIMBase::GetDispatcherHandleByName((__int64)this, 1u, 0);
8      if ( !DispatcherHandle )
9          MicrosoftTelemetryAssertTriggeredNoArgsKM();
10     return RIMReadInput(
11         *((_QWORD *)this_1 + 1),
12         (__int64)this_1 + 64,
13         *((_DWORD *)this_1 + 14),
14         DispatcherHandle,
15         (__int64)this_1);
16 }

```

Figure 4.79: Code from CBaseInput::Read routine (from win32kbase.sys).

internally uses `ObReferenceObjectByHandle` [772]. This routine is called to retrieve an `ExRawInputManager-ObjectType` object type. Since this type is undocumented, it is hard to understand the underlying structure which drives a `Raw Input Manager` (RIM) object. Thereafter, `RIMReadInput` routine calls `rimHandleAnyPnpRemovePendingDevices` to remove any plug and play pending notification in order to read efficiently from the device. This operation is performed through a set of callbacks which are not really relevant here.

`RIMReadInput` continues by checking if the member at offset `+0x50` in the RIM is initialized. If it is initialized, `rimIssueReads` routine is called. Otherwise, the RIM is initialized with some parameters with whom `RIMReadInput` routine has been called. Then, `rimIssueReads` is called followed by a call to `rimCompleteReads` routine. The first call is to read from the device, the second is to finalize once the read operation has been completed. Finally, the routine finishes by a cleanup procedure releasing unnecessary memory and previously acquired locks. A general view of the simplified pseudo-code of `RIMReadInput` is given in Figure 4.80.

Let us first focus on `rimIssueReads` routine. First, the routine calls `DeviceTypeToRimInputType` and `RimInputTypeToDeviceType` routines to convert the type of the device stored in the RIM to the type it needs. But the most relevant part is the call to `rimStartDeviceReadIfAllowed` routine in a loop. After checking if different bits have been set in its second parameter, this one calls `RIMStartDeviceSpecificRead` routine. This routine called with two parameters (RIM Object and RIM Device) is responsible to initiate the read operation from the device. Notwithstanding the code responsible to perform a lot of logging or debugging operations, the relevant operation is to call `RIMStartDeviceRead`. An illustration of such sequence of routine calls is given in Code 4.17. This one has been produced from a driver in the keyboard stack inserting a breakpoint when a read operation is performed. Note that we are *breaking in the context* of `csrss.exe` which holds the RIT, as expected.

	1: kd> kn			
	#	Child-SP	RetAddr	Call Site
3	01	fffff004 'fd35c930	fffff806 '743f2a95	nt!IoCallDriver+0x59
	02	fffff004 'fd35c970	fffff806 '743ef9ef	nt!IopSynchronousServiceTail+0x1a5
5	03	fffff004 'fd35ca10	fffff806 '73fd5355	nt!NtReadFile+0x59f
	04	fffff004 'fd35cb00	fffff806 '73fc7940	nt!KiSystemServiceCopyEnd+0x25
7	05	fffff004 'fd35cd08	ffffbbd6 'fdbf93d4	nt!KiServiceLinkage
	06	fffff004 'fd35cd10	ffffbbd6 'fdbf92b0	win32kbase!RIMStartDeviceRead+0x48
9	07	fffff004 'fd35cd70	ffffbbd6 'fdbfc98b	win32kbase!RIMStartDeviceSpecificRead+0xc8
	08	fffff004 'fd35ce40	ffffbbd6 'fdbf9680	win32kbase!rimOnPnpArrived+0x1cb
	09	fffff004 'fd35cf50	ffffbbd6 'fdbfe592	win32kbase!RIMDoOnPnpNotification+0x60
11	0a	fffff004 'fd35cf90	fffff806 '74535ccd	win32kbase!RIMDeviceClassNotify+0x2f2
13	0b	fffff004 'fd35d210	fffff806 '74520ba6	nt!PnpNotifyDriverCallback+0x95
	0c	fffff004 'fd35d2c0	ffffbbd6 'fdc00cb9	nt!IoRegisterPlugPlayNotification+0x2f6
15	0d	fffff004 'fd35d380	ffffbbd6 'fdbfd15x4d	win32kbase!RIMRegisterForDeviceClassNotifications+0x4d
	0e	fffff004 'fd35d3d0	ffffbbd6 'fdbfa65e	win32kbase!RIMDiscoverDevicesOfInputType+0xf9
17	0f	fffff004 'fd35d420	ffffbbd6 'fdbf9be4	win32kbase!RIMRegisterForInputWithCallbacks+0xa6e
	10	fffff004 'fd35d580	ffffbbd6 'fdbfa966	win32kbase!RIMRegisterForInput+0x64

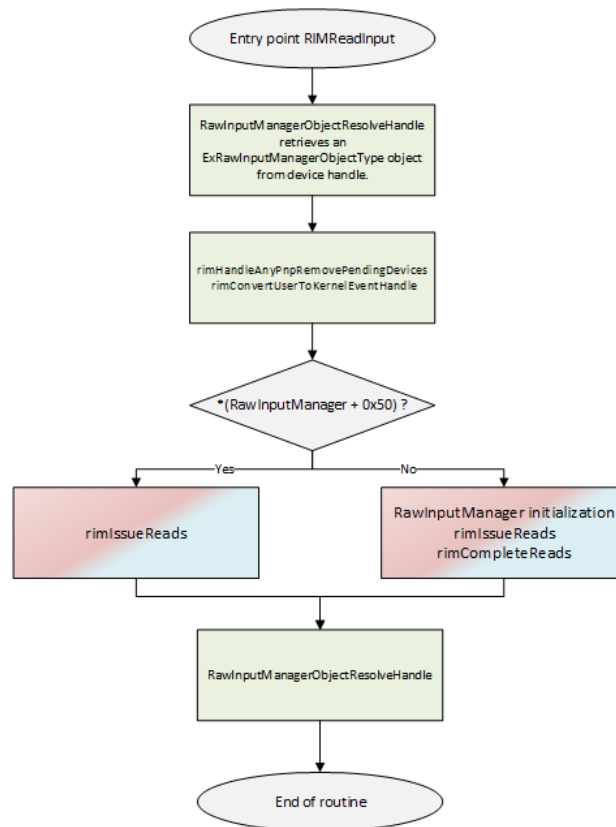


Figure 4.80: Simplified pseudo-code of RIMReadInput routine (from win32kbase.sys).

```

19 11 fffff004 'fd35d600 fffffb00 'fdbf9b4d win32kbase!CRIMBase::CreateHandles+0x126
12 12 fffff004 'fd35d660 fffffb00 'fd20a7c7 win32kbase!CBaseInput::InitializeSensor+0x8d
21 13 fffff004 'fd35d700 fffffb00 'fd2d7e84 win32kfull!xxxRegisterForDeviceClassNotifications+0
    x12b
14 14 fffff004 'fd35d740 fffffb00 'fdc43afe win32kfull!RawInputThread+0x804
23 15 fffff004 'fd35d9c0 fffffb00 'fd215ea0 win32kbase!xxxCreateSystemThreads+0x9e
16 16 fffff004 'fd35dad0 fffff806 '73fd5355 win32kfull!NtUserCallNoParam+0x70
25 17 fffff004 'fd35db00 00007ffc '256d1144 nt!KiSystemServiceCopyEnd+0x25
18 18 000000b4 '6c7ff758 00007ffc '248232da win32u!NtUserCallNoParam+0x14
27 19 000000b4 '6c7ff760 00007ffc '27a6d4df winsrvext!StartCreateSystemThreads+0x1a
1a 000000b4 '6c7ff790 00000000 '00000000 ntdll!RtlUserThreadStart+0x2f
  
```

Code 4.17: "Callstack from a keyboard driver when notified for handling a read IRP operation."

The content of the routine RIMStartDeviceRead is given in Figure 4.81. This routine mainly corresponds to a single call to ZwReadFile routine [773]. But this read operation is particular since it is not performed on a regular file but directly on the handle representing our device (in our case, HID keyboard, HID mouse or any other HID device — i.e. InputTraceLogging::RimDevTypeToString routine). In our context, it corresponds to a read IRP [774] issued for a read operation. This IRP is armed for HID devices and used through the whole device drivers call stack for the device. In the case of the keyboard, this IRP allows to complete the IRP that has been transmitted to kbdclass.sys by HID drivers stack.

Another particular point is the presence of an APC routine provided with ZwReadFile. According to the documentation of ZwReadFile [773], this callback is reserved for system use. In fact, such a callback can have context parameter supplied thanks to ZwReadFile routine. In RIMStartDeviceRead, the supplied context is the RIM device object. Technically, this APC routine rimInputApc is called every time a read operation is about to be performed. In such a case, the callback is notified in the context of the System process (Pid = 0) when ZwReadFile is able to read from the device. For short, it means that every time a key is pressed, it is the job of rimInputApc to take care of it. From a conceptual point of view, rimInputApc routine is really the heart of the

```

1 NTSTATUS __fastcall RIMStartDeviceRead(PVOID ApcContext, __int64 a2, void *Buffer, ULONG Length)
2 {
3     DWORD *ApcContext_1; // rbx@1
4     NTSTATUS result; // eax@1
5
6     ApcContext_1 = ApcContext;
7     result = ZwReadFile(
8         *((HANDLE *)ApcContext + 28),
9         0i64,
10        (PIO_APC_ROUTINE)rimInputApc,
11        ApcContext,
12        (PIO_STATUS_BLOCK)ApcContext + 16,
13        Buffer,
14        Length,
15        &gZero,
16        0i64);
17     ApcContext_1[0x44] = result;
18     if ( result >= 0 )
19     {
20         // Global variable which is related to the uptime.
21         *((_QWORD *)ApcContext_1 + 0x10F) = MEMORY[0xFFFFF78000000014];
22         result = ApcContext_1[0x44];
23     }
24     return result;
25 }

```

Figure 4.81: Content of RIMStartDeviceRead routine from win32kbase.sys.

raw input thread for HID devices.

The rest of the RIMStartDeviceRead routine is about updating the RIM device structure with the content of what is stored at address "0xFFFFF78000000014". This section of memory is not documented but it has been already reversed in other contexts [775]. This address corresponds to the time since the machine has started. We can conclude that this action is done to set up a kind of *timestamp* each time a key is stroke on the keyboard.

Before analyzing rimInputApc routine, we should observe the rimCompleteReads routine. This routine follows a call to rimIssueReads in RIMReadInput routine to complete the read operation once the content of the device has been retrieved. Technically, after checking members in the RIM provided in parameter, rimCompleteReads calls rimFindPausedDeviceAndCompleteRead. This routine is essentially a call to rimProcessDeviceBufferAndStartRead which is a complex routine. Its goal is to process the device buffer returned by the device. Retrieving operation can be performed from the RIM or via a *first in - first out* (FIFO) structure (stored in RIM Device object at offset +0x30) thanks to RIMTransferInjectionDeviceDataFifoToDataBuffer routine. This FIFO structure is usually present for keyboard devices but not for mouse ones <sup>24</sup>.

Once the data has been retrieved, a call to rimObsRouteInputAndCheckForExclusiveObservers is performed to check whether we are an *observer* or not. If not, the routine ends immediately. Otherwise, the procedure acts according to the notification context. This one can be: *DropInput*, *PauseDevice*, *ResumeDevice*, *SignalReadComplete*, *ProcessHidInput*, *ProcessKeyboardInput* and *ProcessMouseInput*. The three first contexts are not really relevant for us. The fourth *SignalReadComplete* is driven by rimSignalReadComplete and rimProcessInjectedDeviceBuffers routines. This context has three possible outcomes. The first is to act as *DropInput* if RIMIsInputSuppressed returns true or — similarly to what is performed in *SkipReadComplete* routine — if the Raw Input Manager object has no member at offset +0x248. Otherwise, in case of success, there is a call to ZwUpdateWnfStateData, a totally undocumented routine.

Routine rimProcessHidInput is called in *ProcessHidInput* context. This one matters for us. In this routine,

<sup>24</sup>In the mouse case, the routine first checks different points with rimObsRouteInputAndCheckForExclusiveObservers and rimIsPointerInputAllowed. In the case of the first check routine, among the different things checked, this one seeks to know if an "Observer" is allowed. This is done with rimObsIsRegisteredObserverAllowed routine which results to true if rimIsProcessLocalSystem or RIMIsTestSigningOn return true. Test signing mode [776] corresponds to a specific boot option of Windows operating system where it is easier to load drivers since security has been removed. Once the checks have been performed, the routine tries to retrieve information from HID mouse device directly with rimFreeAutoRepeatCompleteFrame and RIMProcessAnyPointerDeviceInput routines. The last one is interesting, especially the part where it checks which device is sending information with ApiSetEnsurePointerDeviceHasMonitor routine. But this analysis is beyond our scope focused on keyboard.

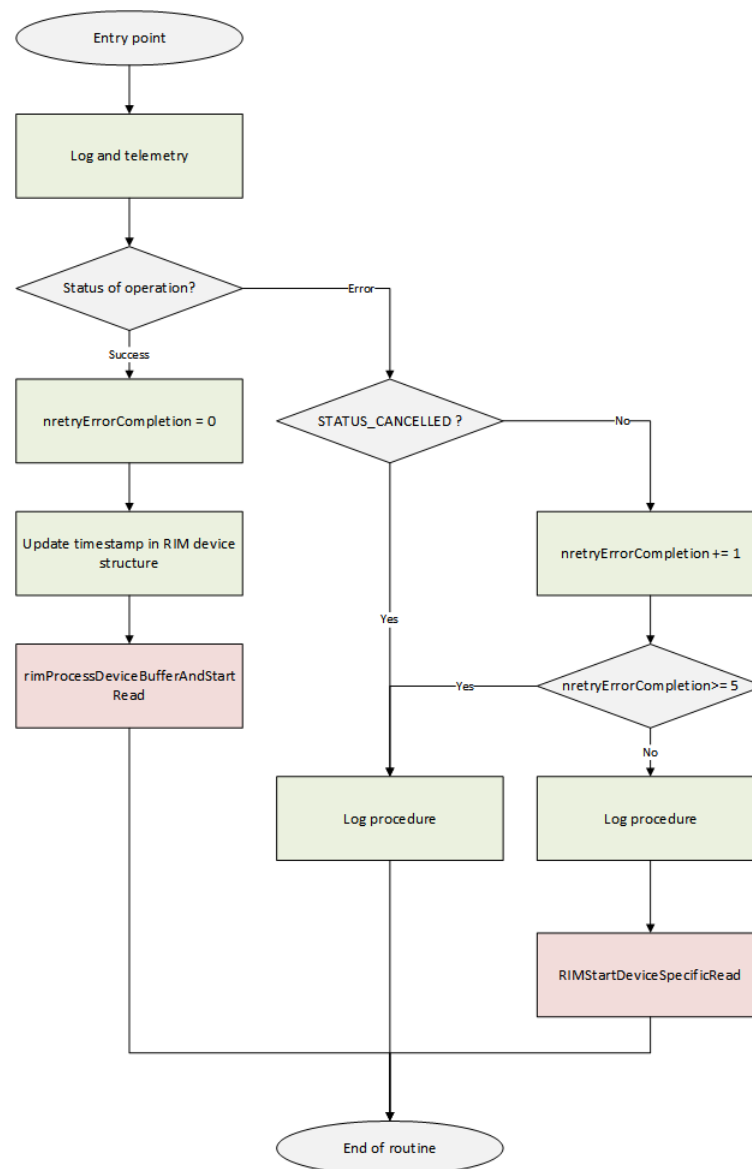
after checking internals in RIM devices and RIM object structures for telemetry purposes, it calls `rimStackAttachAndProcessInput` and `ApiSetProcessHidRawInput` routines. The first routine is responsible to call `rimProcessInput` after making sure that it is correctly attached to the address space (the *working set* [777]) of the current process via a call to `KeStackAttachProcess` [778]. Once the attachment to the user-mode process's memory is performed, `rimProcessInput` allocates memory in the target process thanks to `ZwAllocateVirtualMemory` [779] and `MmSecureVirtualMemory` [780]. Once the allocation is performed, data from the device is transferred to the user-mode memory with all required security when dealing with user-mode memory (especially with `RIMFixUpCompleteFrame` routine) from kernel-mode. The goal of this information transferred in user-mode memory is to allow a future broadcast to the rest of the system. About the use of `ApiSetProcessHidRawInput` routine call in `ProcessHidInput`, this one calls `ProcessHidRawInput` (from `win32kfull.sys`) if `IsProcessHidRawInputSupported` routine returns true. `ProcessHidRawInput` routine is based on `xxxProcessHidInput` which uses `PostHidInput` (internally `PostInputMessage` routine which is about to call `PostUpdateKeyStateEvent` in our keyboard context). Here we are interacting directly with what allows applications to read from the keyboard.

The case of *ProcessKeyboardInput* (routine `rimProcessKeyboardInput`) is similar to `ProcessHidInput` routine since it will call `rimStackAttachAndProcessInput`. The same applies with *ProcessMouseInput* (routine `rimProcessMouseInput`) which calls `rimStackAttachAndProcessInput` after having called `rimUpdateLatestMouseState` routine.

Once `rimFindPausedDeviceAndCompleteRead` has been executed correctly, the content of the read data from the device has been mapped to the memory in `csrss` process. To continue, `rimCompleteReads` calls `rimProcessAnyQueuedCompleteFrames`. This one calls `rimProcessAnyQueuedCompleteFrames` which has the ability to call `rimDispatchCompleteFrame`. The last can be seen as a combination of `rimStackAttachAndProcessInput` and the `SignalReadComplete` procedure. At the end, this is the `rimProcessDeviceBufferAndStartRead` which is called. This one will call the `RIMStartDeviceSpecificRead` routine which ensures that there is always one IRP armed by the raw input thread to manage HID devices.

We can now focus on the `rimInputApc` routine which is the routine set by `RIMStartDeviceRead` to issue an IRP. This routine is notified each time a read operation is possible from the device. Its main goal is to deal with the content of the buffer transferred. To proceed, the APC routine reuses most of the procedures previously introduced. An illustration of its procedure is given in Figure 4.82. This one starts with log and telemetry routines call. Then it checks the status of the read operation performed. If the operation was a success, the APC routine acts on different points. First, it resets an internal counter in the RIM Device object called internally `nretryErrorCompletion`. Then, it updates the *timestamp* in the RIM Device object the same way it does with `RIMStartDeviceRead`. Finally, it executes `rimProcessDeviceBufferAndStartRead` (which can call `rimProcessHidInput`, `rimProcessKeyboardInput` or `rimProcessMouseInput` among different possibilities) to process the device buffer and to re-initiate a read operation. If the status of the read operation is not a success, the APC routine checks if the operation has been canceled. In such a case, this information is logged and the routine simply returns. In the case where the error would be different, the APC tries to renew the operation by calling `RIMStartDeviceSpecificRead` and increasing the `nretryErrorCompletion` counter's value. If this counter is above or equal to the maximum number of retry (which is 5), the procedure is logged and stopped by the routine.

It is obvious that the role of `rimInputApc` routine is to process the content of the information sent by the device and to keep the reading IRP always armed and engaged. The content of data provided by the device still follows the same path as a regular read operation. It is just another way to proceed.

Figure 4.82: Simplified pseudo-code of `rimInputApc` routine (from `win32kbase.sys`).

### 5.1.4.4 Global view on the read IRP operation engaged from the RIT

Taking a little break in our analysis of the raw input thread, we propose to understand consequences with the read IRP engaged by the RIT and with the architecture of the keyboard device stack. An interesting point to note is that the reading operation, concerning the HID keyboard device, is controlled through at least two IRPs. The first is initialized in HID call stack in kbdhid.sys (as explained in section 4.2.6). The second is initialized by the raw input thread in win32kfull.sys. The transition between the two IRPs (where both of them require reading from the device) is handled by kbdclass.sys. The transition from the HID driver reading from the keyboard is performed through KeyboardClassServiceCallback routine. Then, it is the kbdclass.sys driver's responsibility to complete the IRP original engaged by the raw input thread at initialization time to notify the raw input thread at running time. This particular procedure is illustrated in Figure 4.83.

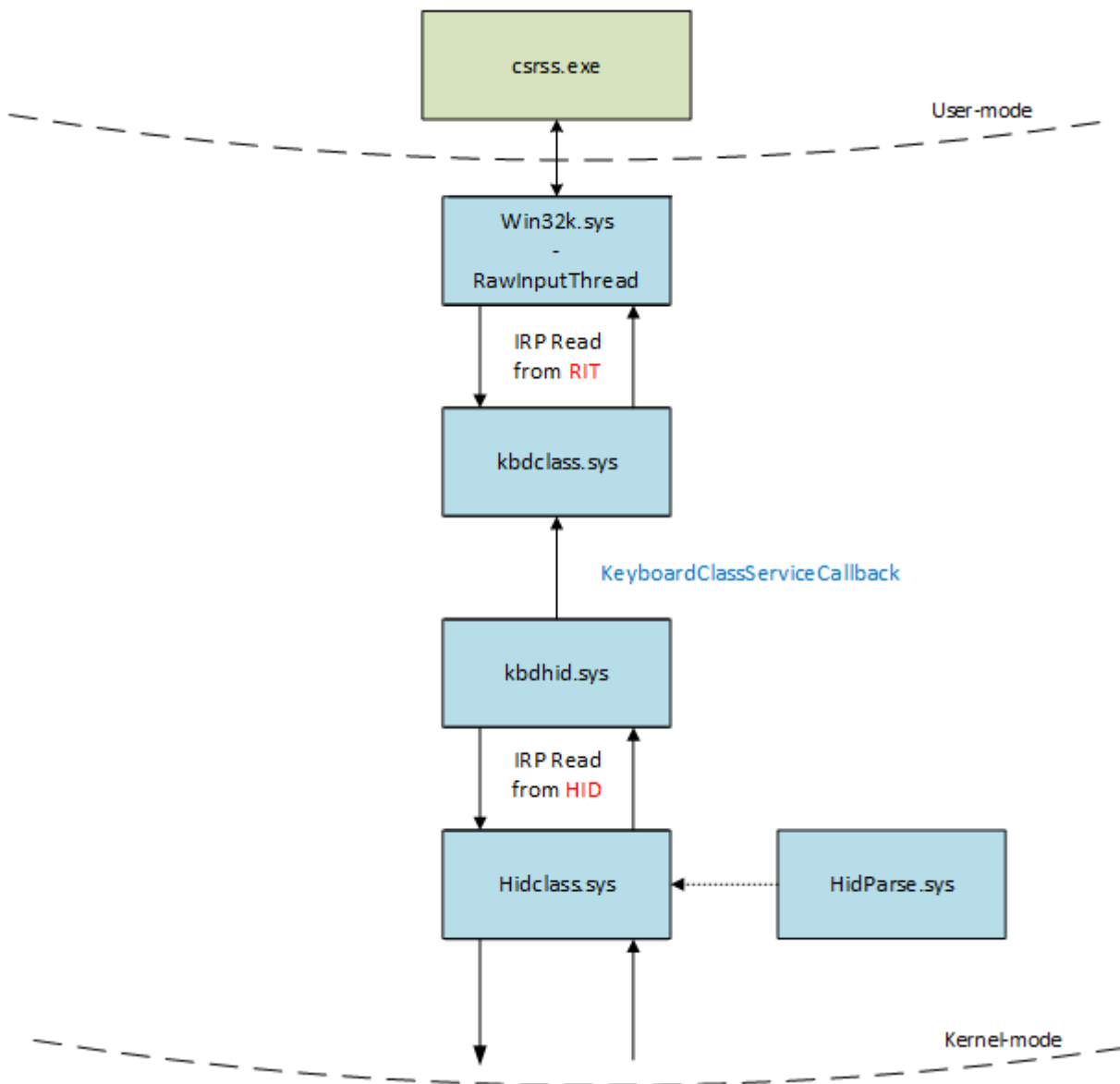


Figure 4.83: General illustration of read operation with a HID keyboard device. Note the double IRP procedure engaged from each part of kbdclass.sys.

#### 5.1.4.5 Second part of devices initialization by Raw input Thread

##### Key Point 4.35:

- ☞ The end of the RIT initialization procedure focuses on setting up callbacks to handle the various events managed by the RIT.
  - ✍ The RIT is notified when a CDROM is inserted. The reason of this monitoring remains mysterious (only this type of devices with rather limited actions).
  - ✍ The RIT is able to adapt to tablets that often have extra buttons. These ones are processed in HID and directly taken into account by the kernel (there is no specific notification of applications with the button managing the brightness on the screen, for instance).
  - ✍ The RIT manages applications that hang on the screen (by taking care of their detection and displaying them as grayed).
- ☞ At the end of the initialization, the RIT uses `LegacyInputDispatcher` class.

To continue on the second part of the raw input thread given in Figure 4.75, once the initialization of sensors has been performed (if required), the `RawInputThread` routine calls the `xxxRegisterForDeviceClassNotifications` routine. In case of initialization would have not been performed previously, notifications for keyboards are recorded via `CBaseInput::InitializeSensor` for `gpHidInput` and `gpKeyboardSensor` sensor objects. The `CBaseInput::Read` routine is called for each sensor once the initialization is finished. Description of this initialization procedure is given in Figure 4.84. Initialization is performed thanks to a loop where a local value (called *Choice* in our illustration in Figure 4.75) is incremented. It is the result of an optimization by a compiler in order to factorize similar parts of the code written, since the two initialization procedures are similar.

Once the initialization has been performed (if required), `xxxRegisterForDeviceClassNotifications` routine is able to call `RegisterCDROMNotify` which uses `IoRegisterPlugPlayNotification` [781] to handle device interface change about `GUID_DEVINTERFACE_CDROM` [782]. This procedure allows the `DeviceClassCDROMNotify` callback routine to be notified by the PnP during the arrival (enabling) or the removal (disabling) of a CDROM device interface instance. Technically, `DeviceClassCDROMNotify` acts only when a device is added. In this case, `IoGetDeviceObjectPointer` is used to retrieve the `FILE_OBJECT` [783] from the CDROM device which is newly instanced by the system. Then, a dedicated context callback structure is allocated to hold different information about the device (including its name). Thereafter, there is a new call to `IoRegisterPlugPlayNotification` targeting the new device thanks to the file object retrieved. Notifications are selected to be about "*Target Device Change*", which means PnP notifies the callback `DeviceCDROMNotify` when there are events related to removing the device (those documented [784] are: *query remove*, *remove complete* or *remove canceled*). Technically speaking, the callback `DeviceCDROMNotify` is able to go further in the notifications than those indicated by the Microsoft documentation by adding the arrival of devices and their readiness. Whenever a new device arrives or when an existing one is removed, the context callback structure of the device previously allocated is copied and inserted into `gMediaChangeList` list.

This list is used to keep an history of insert and remove of CDROM devices. Once a device has been removed from the list, its callback is released with `IoUnregisterPlugPlayNotification` routine [785] as a response to cancel the previous call to `IoRegisterPlugPlayNotification` routine. When a device becomes ready, the `DeviceCDROMNotify` callback calls `ShowAutorunCursor` routine which is beyond the scope of our subject since it seems to be related to mouse device exploitation ("*autorun cursor*"). Since `IoRegisterPlugPlayNotification` must be canceled at the end, `UnregisterDeviceClassNotifications` routine (called either by `xxxRemoteDisconnect` or `RawInputThread` when specific shutdown events are signaled after the wait procedure at the end of Figure 4.72) is responsible to remove the callback set in `RegisterCDROMNotify` routine.

From our research, as far as we have gone, there is no real indisputable explanation on the role of this monitoring of CDROM devices. From what we observed, we only have a notification in Session 1 on `KeyboardDeviceSpecificCallout` callback for device:



```

1 __int64 xxxRegisterForDeviceClassNotifications()
2 {
3     MACRO_STATUS_ABANDONED status; // edi@1
4     unsigned int Choice; // ebx@1
5     _QWORD *Selected; // rcx@6
6     ULONG_PTR IsAtomicOperation; // rdx@10
7
8     status = STATUS_INVALID_PARAMETER;
9     Choice = 1;
10    do
11    {
12        if ( !*( _QWORD *)gpWin32kDriverObject )
13            goto __next;
14        if ( *( _DWORD *)gdwInAtomicOperation && *( _DWORD *)gdwExtraInstrumentations & 1 )
15            KeBugCheckEx(0x160u, *(unsigned int *)gdwInAtomicOperation, 0i64, 0i64, 0i64);
16        UserSessionSwitchLeaveCrit();
17        if ( Choice != 2 )
18        {
19            if ( Choice != 1 )
20                goto __entercrit_next;
21            status = (unsigned int)CBaseInput::InitializeSensor(*( _QWORD *)gpKeyboardSensor);
22            if ( (status & 0x80000000) != 0 )
23                goto __entercrit_next;
24            Selected = gpKeyboardSensor;
25            goto __read_entercrypt_next;
26        }
27        status = (unsigned int)CBaseInput::InitializeSensor(*( _QWORD *)gpHidInput);
28        if ( (status & 0x80000000) == 0 )
29        {
30            Selected = gpHidInput;
31        }
32        __read_entercrypt_next:
33        status = (unsigned int)CBaseInput::Read(*Selected);
34        |
35        __entercrit_next:
36        EnterCrit(0i64, 1i64);
37        next:
38        ++Choice;
39    }
40    while ( Choice <= 2 );

```

Figure 4.84: Beginning of the xxxRegisterForDeviceClassNotifications routine in win32kfull.sys.

```

\??\SCSI#CdRom&Ven_VBOX&Prod_CD-ROM#4&3554261f&0&010000#{53f56308-b6bf-11d0-94f2-00a0c91efb8b}

```

This one corresponds to the CDROM device provided by Virtual Box. One explanation might lie in legacy backward compatibility from the old time when the raw input thread was doing more in the system. Another explanation would propose that this area of the code had been used to house interactions with CDROM devices when they were critical to the installation of system drivers (such as keyboard or mouse device drivers). Today, this code still allows to trace the activities from the point of view of the plug and play manager concerning the CDROMs devices.

The next step in the system is the check of the computer type where Windows is running. Indeed, Windows 10 is designed to be running on different types<sup>25</sup> of computers [786]. Tablets PC are particular since they are designed to be tactile and present extra buttons (embedded in the tablet itself). To manage this type of device correctly, the raw input thread has to check whether it is running on a tablet or on a computer. This check is performed through `_GetDigitizerFlags` where the name does not correspond to actions performed by the routine. To proceed, this routine checks in the registry key "HKLM\Software\Microsoft\Windows\Tablet PC" the value `IsTabletPC`. If this one is present and the content is different from zero, it means that we are running a Windows 10 on a tablet.

In such a case, the routine `RegisterTabletButtonHandler` (from win32full.sys) is called. This routine calls first

<sup>25</sup><https://www.microsoft.com/fr-fr/windows/view-all-devices>

`ReadTabletButtonSettings` routine which retrieve information in the registry. This information is stored in the key `"HKLM\SYSTEM\CurrentControlSet\Control\TabletPC\TabletButtons\n_(0)"` where `"n_(0)"` refers to an index which is incremented from 0 until this is not possible to access to a key anymore. For each key, the value `"ButtonId"` is read by `ReadTabletButtonIndex` and it is used as an offset to initialize `TabButtonConfig` internal structure. This offset is used as a base offset to retrieve and store information with `ReadTabletButtonConfig` routine from the same key on values `"PrimaryLandscape"`, `"PrimaryPortrait"`, `"SecondaryLandscape"`, and `"SecondaryPortrait"`. Once the configuration from registry is read, `RegisterTabletButtonHandler` calls `RegisterRawInputDevices` with `TabletButtonHandler` callback. The registration routine is based on HID. To proceed, it uses `AllocateProcessHidTable`, `HidRequestValidityCheck`, `SearchProcessHidRequest`, `SetProcDeviceRequest`, `FreeHidProcessRequest`, and `CHidInput::HandleDirectStartStopDeviceReadRequest` routines, the last on `gpHidInput` object. Routines `HidRequestValidityCheck` and `SearchProcessHidRequest` are dedicated to operations parsing while `SetProcDeviceRequest` is more about processing HID operations. Indeed — without going into details because they are complex and not really relevant — this one deals with the current process' GUI window displayed on the screen (thanks to `ValidateHwnd`), and it has the ability to allocate a HID request with `AllocateHidProcessRequest` routine to communicate through `InsertProcRequest`. The last, using `gpHidInput` with `CBaseInput::TmpGetDeviceList` (from `win32kbase.sys`, it only returns the value `CBaseInput::spDevList`), applies `PostDeviceNotification` on each HID device. This last routines uses `PostHidNotification` which calls `PostInputMessage` previously mentioned and it can use `PostMessage` which, in the end, calls routines such as `xxxBroadcastMessageEx` and `xxxSendMessageCallback` to broadcast messages.

Still about tablet management, the callback `TabletButtonHandler` deals with HID Usages from the tablet. Using `HidP_MaxUsageListLength` [711] and `HidP_GetUsages` [787], it deals with a list of all the HID control button usages that are on an usage page 0 and which are set to ON in the HID report provided to the callback. From the usages extracted, the callback routines potentially use `TabletRetrieveDevMode` to retrieve information including how the tablet's display is set-up. But the central point of the callback is the use of `xxxTabletButtonExecuteAction` routine. The last is responsible for several specific actions linked with the tablet, including display management. To do so, it uses `TabletAdjustBrightness` and `xxxTabletSetDisplayOrientation` routines where the last routine uses `TabletRetrieveDevMode` and `xxxUserChangeDisplaySettings` routines. Finally, there are timers managed with `SetRITTimer` routine which is called on the `gtmridTabletButtonTimer` object with `xxxTabletButtonTimerCallback` callback. This one is still about tablet management with the use of `xxxTabletButtonExecuteAction`.

At the end of the procedure described in Figure 4.75, there is a call to the `InitTimerPowerSaving` routine. This routine calls `FastGetProfileDword` routine twice (from `win32kbase.sys`) to read and complete global values `gdwRITdemonTimerPowerSaveElapse` and `gdwRITdemonTimerPowerSaveCoalescing` from Windows' registry values. Technically, `FastGetProfileDword` routine calls `FastGetProfileDwordEx` routine (still from `win32kbase.sys`) which uses `OpenCacheKeyEx` (from `win32kbase.sys`) to select a specific registry path according to provided parameters from the call to `FastGetProfileDword`. Internally, it corresponds to tables of predefined path in registry where required information could be located. In our case, path for both values is `"HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows"` on values `"RITdemonTimerPowerSaveElapse"` and `"RITdemonTimerPowerSaveCoalescing"`.

To conclude the description of the *Raw Input Thread* with Session 1, as given in Figure 4.75, there is a call to `SetRITTimer` with `gnRITdemonTimerId` object to register a callback `xxxHungAppDemon` to be notified every second. This callback handles hang GUI windows in the system [788]. A user-mode GUI window can hang when there is too much delay in processing messages from the system [789, 790, 791]<sup>26</sup>. This is `xxxHungAppDemon` routine which is responsible to manage hung GUI windows by painting them in a special way when they are frozen (`zzzCalcStartCursorHide`, `ProcessHungWindow` and `xxxRedrawHungWindow`) or providing ways to interact with them from user-mode (such as with `IsHungAppWindow` [793] in user-mode or `IsHungWindow` (from `win32full.sys`) in kernel-mode where the last uses `IsPumpingInputMsgs` in its testing procedure to check if the targeted window still process messages).

At the end of the procedure given by Figure 4.75, both Session 0 and Session 1 raw input threads call `LegacyInputDispatcher::Create` and `LegacyInputDispatcher::Initialize` routines. Before studying the end of the raw input thread procedure, it could make sense to observe the real execution path taken by the raw input thread

<sup>26</sup>Consequences of a hung window explains some design choices in Windows' explorer [792].

during its initialization phase, for both Session 0 and Session 1.

#### 5.1.4.6 Observing Raw input Thread with breakpoints in debugger

##### Key Point 4.36:

- ☞ We use Windbg debugger to analyze (and confirm) some of our previous observations.
  - ☞ There is indeed a difference in RIT behavior between session 0 and the other sessions.
- ☞ The RIT is notified for a read operation twice for each keystroke. Once when the key is pressed and another time when this one is released.
- ☞ Using a debugger, we can see two relevant points when a key is pressed:
  - ☞ Only *raw input threads* (from *csrss.exe*) which do not belong to session 0 are notified when a key is pressed.
  - ☞ The RIT manages keystrokes in the procedure starting with *LegacyInputDispatcher* routines class (called *legacy procedure* by us).

To understand how the raw input thread works without having to hardly reverse anything, it might make sense to use dynamic analysis. Windbg debugger is the perfect tool to proceed. We propose to set a breakpoint at the entry of *RawInputThread* routine as given in Code 4.18.

```
bp win32kfull!RawInputThread
```

Code 4.18: "Set a breakpoint to *RawInputThread* in *win32kbase.sys*."

Once the breakpoint is set, we let the debugger running with a `g0` order<sup>27</sup>. The breakpoint is raised the first time *csrss.exe* is executed under Session 0. In this case, the list of routines called by the raw input thread only is provided in Code 4.20. Routines called which are specific to the execution of the raw input thread in the context of Session 0 are highlighted.

```
1 0: kd> !process -1
  \newlinePROCESS fffffd48f00acd140
3   SessionId: 0  Cid: 019c  Peb: 2c63aa1000  ParentCid: 0194
   DirBase: 1c664002  ObjectTable: fffff9d02a49b7b00  HandleCount: 111.
5   Image: csrcss.exe
7   (...)
```

<sup>27</sup>In practice, the procedure is a bit more complex since there is an exception handler to manage and early boot breakpoints can be a bit special to take into account. Our exact procedure used at initial breakpoint proposed by the debugger is provided in code 4.19. Note that this one could evolve with future versions of Windows.

```
1 $h1 clearAll
  !gflag +ksl
3 sxe ld csrcss.exe
  g:g
5
  kn
7 .process
  !process -1 f
9 bp csrcss!main
11 bp win32kbase!RawInputThread
```

Code 4.19: "Full Windbg procedure used for debug purposes."

```

9 0: kd> wt (...)
win32kbase!GreGetRemoteContext (ffff87c0 '96b54070)
11 win32kfull!memset (ffff87c0 '96965700)
win32kbase!SetThreadName (ffff87c0 '96b541b8)
13 ntoskrnl!PsGetThreadProcessId (ffff87c0 '96b53df8)
ntoskrnl!PsGetThreadId (ffff87c0 '96b53d48)
15 ntoskrnl!ZwSetSystemInformation (ffff87c0 '96b53a28)
ntoskrnl!RtlInitUnicodeString (ffff87c0 '96b53ea0)
17 win32kbase!EnterCrit (ffff87c0 '96b56e70)
win32kfull!InitKeyboard (ffff87c0 '968dfa50)
19 win32kbase!UserSessionSwitchLeaveCrit (ffff87c0 '96b56ea8)
win32kbase!Win32AllocPoolZInit (ffff87c0 '96b56df8)
21 win32kfull!InkProcessor::InkProcessor (ffff87c0 '9683e970)
win32kbase!InitSystemThread (ffff87c0 '96b541c0)
23 ntoskrnl!PsGetProcessWin32Process (ffff87c0 '96b531b8)
win32kbase!GetDispInfo (ffff87c0 '96b56ad0)
25 win32kbase!ClearLogicalCursorPos (ffff87c0 '96b541c8)
win32kbase!EnterCrit (ffff87c0 '96b56e70)
27 win32kfull!TransitionCursorSuppressionState (ffff87c0 '96889680)
win32kbase!UserSessionSwitchLeaveCrit (ffff87c0 '96b56ea8)
29 win32kbase!EnterCrit (ffff87c0 '96b56e70)
win32kbase!CInputGlobals::UpdateLastInputTime (ffff87c0 '96b569a0)
31 win32kbase!UserSessionSwitchLeaveCrit (ffff87c0 '96b56ea8)
win32kbase!Win32AllocPoolNonPaged (ffff87c0 '96b567c8)
33 win32kbase!KeInitializeTimer (ffff87c0 '96b53a20)
win32kbase!Win32AllocPoolNonPaged (ffff87c0 '96b567c8)
35 win32kbase!KeInitializeTimerEx (ffff87c0 '96b53a18)
win32kbase!DrvEscapeRemoteDrivers (ffff87c0'96b541e8)
37 ntoskrnl!PoRequestShutdownEvent (ffff87c0'96b53a10)
win32kfull!W32GetThreadWin32Thread (ffff87c0 '9686a674)
39 ntoskrnl!KeSetEvent (ffff87c0 '96b53c98)
ntoskrnl!ObReferenceObjectByPointer (ffff87c0 '96b53c70)
41 ntoskrnl!KeWaitForMultipleObjects (ffff87c0 '96b53ca0)
ntoskrnl!ObfDereferenceObject (ffff87c0 '96b53ee0)
43 win32kfull!GetRITWakeReason (ffff87c0 '968e18ac)
win32kbase!EnterCrit (ffff87c0 '96b56e70)
45 win32kfull!xxxSwitchDesktop (ffff87c0 '9688a4d8)
ntoskrnl!KeSetEvent (ffff87c0 '96b53c98)
47 win32kbase!CBaseInput::InitializeSensor (ffff87c0'96b54208)
win32kbase!CBaseInput::InitializeSensor (ffff87c0'96b54208)
49 win32kbase!UserIsUserCritSecIn (ffff87c0 '96b55ba0)
win32kbase!UserSessionSwitchLeaveCrit (ffff87c0 '96b56ea8)
51 win32kbase!LegacyInputDispatcher::Create (ffff87c0 '96b54218)
win32kbase!LegacyInputDispatcher::Initialize (ffff87c0 '96b54220)
53 win32kbase!CBaseInput::RegisterDispatcherObject (ffff87c0 '96b541b0)
win32kbase!CBaseInput::RegisterDispatcherObject (ffff87c0 '96b541b0)
55 win32kbase!LegacyInputDispatcher::WaitAndDispatch (ffff87c0 '96b54228)
win32kbase!KeClearEvent (ffff87c0 '96b53cb0)
57 win32kfull!GetRITWakeReason (ffff87c0 '968e18ac)
win32kbase!ProcessMouseEvent (ffff87c0 '96b54110)
59 win32kfull!GetRITWakeReason (ffff87c0 '968e18ac)
win32kbase!LegacyInputDispatcher::WaitAndDispatch (ffff87c0 '96b54228)
61 ntoskrnl!KeClearEvent (ffff87c0 '96b53cb0)
win32kfull!GetRITWakeReason (ffff87c0 '968e18ac)
63 win32kbase!ProcessMouseEvent (ffff87c0 '96b54110)
win32kfull!GetRITWakeReason (ffff87c0 '968e18ac)
65 win32kbase!LegacyInputDispatcher::WaitAndDispatch (ffff87c0 '96b54228)

```

Code 4.20: "List of routines called by RawInputThread in Session 0."

We can compare the same trace captured for `csrss.exe` launched from Session 1. This trace is given in Code 4.21. If the beginning is similar, the end is quite different. Indeed, the final loop executes a subset of routines which are notified either by timers either by events from devices. The trace has been captured at boot time and it could be a little bit different when arriving to user's session. Differences with execution in Session 0 are highlighted in Code 4.21.

```
1 1: kd> !process -1
PROCESS ffff98872c73a140
3   SessionId: 1  Cid: 0200  Peb: 4829db5000  ParentCid: 01ec
   DirBase: 1edb2002  ObjectTable: fffffd40a57edc380  HandleCount: 114.
5   Image: csrss.exe
7   (...)
9 0: kd> wt (...)
win32kbase!GreGetRemoteContext (ffffa9de'7ff54070)
11 win32kfull!memset (ffffa9de'7fd65700)
win32kbase!SetThreadName (ffffa9de'7ff541b8)
13 ntoskrnl!PsGetThreadProcessId (ffffa9de'7ff53df8)
ntoskrnl!PsGetThreadId (ffffa9de'7ff53d48)
15 ntoskrnl!ZwSetSystemInformation (ffffa9de'7ff53a28)
ntoskrnl!RtlInitUnicodeString (ffffa9de'7ff53ea0)
17 win32kbase!EnterCrit (ffffa9de'7ff56e70)
win32kfull!InitKeyboard (ffffa9de'7fcdfa50)
19 win32kbase!UserSessionSwitchLeaveCrit (ffffa9de'7ff56ea8)
win32kbase!Win32AllocPoolZInit (ffffa9de'7ff56df8)
21 win32kfull!InkProcessor::InkProcessor (ffffa9de'7fc3e970)
win32kbase!InitSystemThread (ffffa9de'7ff541c0)
23 win32kbase!PsGetProcessWin32Process (ffffa9de'7ff531b8)
win32kbase!GetDispInfo (ffffa9de'7ff56ad0)
25 win32kbase!ClearLogicalCursorPos (ffffa9de'7ff541c8)
win32kbase!EnterCrit (ffffa9de'7ff56e70)
27 win32kfull!TransitionCursorSuppressionState (ffffa9de'7fc89680)
win32kbase!UserSessionSwitchLeaveCrit (ffffa9de'7ff56ea8)
29 win32kbase!EnterCrit (ffffa9de'7ff56e70)
win32kfull!RegisterHotKey (ffffa9de'7fc908dc)
31 win32kfull!SetDebugHotKeys (ffffa9de'7fcdfb5c)
win32kfull!SetWinlogonHotKeys (ffffa9de'7fc8d694)
33 win32kfull!SetWindowArrangementHotKeys (ffffa9de'7fcdfbf0)
win32kfull!SetPenHotKeys (ffffa9de'7fd62bd4)
35 win32kbase!CInputGlobals::UpdateLastInputTime (ffffa9de'7ff569a0)
win32kbase!UserSessionSwitchLeaveCrit (ffffa9de'7ff56ea8)
37 win32kbase!Win32AllocPoolNonPaged (ffffa9de'7ff567c8)
win32kbase!KeInitializeTimer (ffffa9de'7ff53a20)
39 win32kbase!Win32AllocPoolNonPaged (ffffa9de'7ff567c8)
ntoskrnl!KeInitializeTimerEx (ffffa9de'7ff53a18)
41 win32kbase!EnterCrit (ffffa9de'7ff56e70)
win32kbase!GetWin32kDriverObject (ffffa9de'7ff541f0)
43 win32kbase!UserSessionSwitchLeaveCrit (ffffa9de'7ff56ea8)
win32kfull!CreateShutdownEvent (ffffa9de'7fce5930)
45 ntoskrnl!ObReferenceObjectByHandle (ffffa9de'7ff53c80)
win32kfull!CreatePointerDeviceProcessEvents (ffffa9de'7fcdcf44)
47 win32kfull!W32GetThreadWin32Thread (ffffa9de'7fc6a674)
ntoskrnl!KeSetEvent (ffffa9de'7ff53c98)
49 ntoskrnl!ObReferenceObjectByPointer (ffffa9de'7ff53c70)
ntoskrnl!KeWaitForMultipleObjects (ffffa9de'7ff53ca0)
51 ntoskrnl!ObDereferenceObject (ffffa9de'7ff53ee0)
win32kfull!GetRITWakeReason (ffffa9de'7fce18ac)
53 win32kbase!EnterCrit (ffffa9de'7ff56e70)
win32kfull!xxxSwitchDesktop (ffffa9de'7fc8a4d8)
55 ntoskrnl!KeSetEvent (ffffa9de'7ff53c98)
win32kfull!xxxRegisterForDeviceClassNotifications (ffffa9de'7fc0aa2c)
57 win32kfull!_GetDigitizerFlags (ffffa9de'7fce0088)
win32kfull!InitTimerPowerSaving (ffffa9de'7fce01c0)
59 win32kfull!SetRITTimer (ffffa9de'7fd1def0)
win32kbase!UserIsUserCritSecIn (ffffa9de'7ff55ba0)
61 win32kbase!UserSessionSwitchLeaveCrit (ffffa9de'7ff56ea8)
win32kbase!LegacyInputDispatcher::Create (ffffa9de'7ff54218)
63 win32kbase!LegacyInputDispatcher::Initialize (ffffa9de'7ff54220)
win32kbase!CBaseInput::RegisterDispatcherObject (ffffa9de'7ff541b0)
65 win32kbase!CBaseInput::RegisterDispatcherObject (ffffa9de'7ff541b0)
win32kbase!LegacyInputDispatcher::WaitAndDispatch (ffffa9de'7ff54228)
67 win32kbase!EnterCrit (ffffa9de'7ff56e70)
win32kfull!TimersProc (ffffa9de'7fce1440)
69 win32kbase!UserSessionSwitchLeaveCrit (ffffa9de'7ff56ea8)
```

```

win32kbase!LegacyInputDispatcher::WaitAndDispatch (ffffa9de'7ff54228)
71 ntoskrnl!KeClearEvent (ffffa9de'7ff53cb0)
win32kfull!GetRITWakeReason (ffffa9de'7fce18ac)
73 win32kbase!ProcessMouseEvent (ffffa9de'7ff54110)
win32kfull!GetRITWakeReason (ffffa9de'7fce18ac)
75 win32kbase!LegacyInputDispatcher::WaitAndDispatch (ffffa9de'7ff54228)
ntoskrnl!KeClearEvent (ffffa9de'7ff53cb0)
77 win32kfull!GetRITWakeReason (ffffa9de'7fce18ac)
win32kbase!ProcessMouseEvent (ffffa9de'7ff54110)
79 win32kfull!GetRITWakeReason (ffffa9de'7fce18ac)
win32kbase!LegacyInputDispatcher::WaitAndDispatch (ffffa9de'7ff54228)
81 win32kbase!KeClearEvent (ffffa9de'7ff53cb0)
win32kfull!GetRITWakeReason (ffffa9de'7fce18ac)
83 win32kbase!ProcessMouseEvent (ffffa9de'7ff54110)
win32kfull!GetRITWakeReason (ffffa9de'7fce18ac)
85 win32kbase!LegacyInputDispatcher::WaitAndDispatch (ffffa9de'7ff54228)
win32kbase!KeClearEvent (ffffa9de'7ff53cb0)
87 win32kfull!GetRITWakeReason (ffffa9de'7fce18ac)
win32kbase!ProcessMouseEvent (ffffa9de'7ff54110)
89 win32kfull!GetRITWakeReason (ffffa9de'7fce18ac)
win32kbase!LegacyInputDispatcher::WaitAndDispatch (ffffa9de'7ff54228)
91 win32kbase!EnterCrit (ffffa9de'7ff56e70)
win32kfull!TimersProc (ffffa9de'7fce1440)
93 win32kbase!UserSessionSwitchLeaveCrit (ffffa9de'7ff56ea8)
win32kbase!LegacyInputDispatcher::WaitAndDispatch (ffffa9de'7ff54228)
95 win32kbase!EnterCrit (ffffa9de'7ff56e70)
win32kfull!TimersProc (ffffa9de'7fce1440)
97 win32kbase!UserSessionSwitchLeaveCrit (ffffa9de'7ff56ea8)
win32kbase!LegacyInputDispatcher::WaitAndDispatch (ffffa9de'7ff54228)
99 (...)

```

Code 4.21: "List of routines called by RawInputThread in Session 1."

Few remarks from the differences between Code 4.20 and 4.21. We observe some specificities taken by the raw input thread according to its session's context. Main differences are from shutdown notification management and initialization of input devices. In Session 0, the initialization is basic and notifications about shutdown events are registered directly from the kernel. In the case of Session 1, initialization is more advanced on specificities (hotkeys and tablet management) but the initialization of input devices is performed later within `xxxRegisterForDeviceClassNotifications`. Indeed, if we trace (Code 4.22) the routines called by `xxxRegisterForDeviceClassNotifications` in Session 1, we can observe initialization of keyboard devices via `CBaseInput::InitializeSensor` and `CBaseInput::Read` routines.

```

1 win32kbase!UserSessionSwitchLeaveCrit (ffffcf01'ca946ea8)
win32kbase!CBaseInput::InitializeSensor (ffffcf01'ca944208)
3 win32kbase!CBaseInput::Read (ffffcf01'ca944210)
win32kbase!EnterCrit (ffffcf01'ca946e70)
5 win32kbase!UserSessionSwitchLeaveCrit (ffffcf01'ca946ea8)
win32kbase!CBaseInput::InitializeSensor (ffffcf01'ca944208)
7 win32kbase!CBaseInput::Read (ffffcf01'ca944210)
win32kbase!EnterCrit (ffffcf01'ca946e70)
9 win32kbase!UserSessionSwitchLeaveCrit (ffffcf01'ca946ea8)
win32kfull!RegisterCDROMNotify (ffffcf01'ca730570)
11 win32kbase!EnterCrit (ffffcf01'ca946e70)

```

Code 4.22: "List of routines called by xxxRegisterForDeviceClassNotifications in Session 1."

The end of the procedure of the RIT between Session 0 and Session 1 is different. In the case of Session 0, the routine ends after calling twice `ProcessMouseEvent` when it continues in an endless loop in the case of Session 1, dealing with extra events compared to those observed in Session 0.

More than observing the initialization of the raw input thread in both sessions, it would make sense to check what is happening when a keystroke signal is received by the system. To proceed, we propose to set several breakpoints in the different routines previously studied. For instance, we can set breakpoints on `CBaseInput::Read`, `RIMReadInput`, `rmlIssueReads` or `RIMStartDeviceSpecificRead` since they are all supposed to be notified



whenever a key is pressed on the keyboard. When these ones are set and a key is pressed, we observe that notification of routines is performed twice for each. Technically speaking, one notification is performed when a key is pressed and one other during the release of the key.

But if we are checking the call stack of the thread issuing `RIMReadInput`, this routine is not called in the context of one of the routines described in the initialization of the raw input thread (Figure 4.75). Indeed, since we are dealing with `csrss.exe` in Session 1 (as expected since Session 0 does not directly manage input devices), when we watch the current thread's call stack where the breakpoint has been hit, we have the output provided in Code 4.23.

```

1 1: kd> kn
# Child-SP          RetAddr           Call Site
3 00 fffffb284`ad93a2a8 fffffb4a7`707d2795 win32kbase!RIMReadInput
01 fffffb284`ad93a2b0 fffffb4a7`70928908 win32kbase!CBaseInput::Read+0x55
5 02 fffffb284`ad93a300 fffffb4a7`7080b36d win32kbase!CBaseInput::OnReadNotification+0x4c8
03 fffffb284`ad93a440 fffffb4a7`7080b472 win32kbase!CBaseInput::OnDispatcherObjectSignaled+0
  x291
7 04 fffffb284`ad93a5b0 fffffb4a7`707d37bf win32kbase!CBaseInput::_OnDispatcherObjectSignaled
  +0x12
05 fffffb284`ad93a5e0 fffffb4a7`707d35c2 win32kbase!LegacyInputDispatcher::Dispatch+0x53
9 06 fffffb284`ad93a610 fffffb4a7`704e0e49 win32kbase!LegacyInputDispatcher::WaitAndDispatch+0
  x102
07 fffffb284`ad93a740 fffffb4a7`7085377e win32kfull!RawInputThread+0x959
11 08 fffffb284`ad93a9c0 fffffb4a7`70415ea0 win32kbase!xxxCreateSystemThreads+0x9e
09 fffffb284`ad93aad0 fffff802`195d4255 win32kfull!NtUserCallNoParam+0x70
13 0a fffffb284`ad93ab00 00007ffb`0d881144 nt!KiSystemServiceCopyEnd+0x25
0b 0000000e`8ccbfb00 00007ffb`0d4732da 0x00007ffb`0d881144
15 0c 0000000e`8ccbfbf0 00000000`00000000 0x00007ffb`0d4732da

```

Code 4.23: "Call stack from `RIMReadInput` routine when a key has been pressed."

From Code 4.23, we observe that the notification has been performed from `RawInputThread` with an offset of `+0x959` from the base address of the routine. At that point, there is a call to `LegacyInputDispatcher::WaitAndDispatch` which is called at the end of the raw input thread. This part is called *legacy procedure* by us.

#### 5.1.4.7 Dispatcher object link to the device: data processing

##### Key Point 4.37:

- ☞ When a key is received, it is converted from the keyboard scan code to an universal key representation code in Windows called *virtual key code* thanks to the `CKeyboardSensor::ProcessInput` routine.
- ☞ The conversion is a two-steps procedure acting as a post-processing on the read data from devices.
  - ☞ It first normalize the scan code received (with `MapScanCode`).
  - ☞ Then, the scan code is converted in a virtual key code (with `VKFromVSC` and `InternalMapVirtualKeyEx` routines).
  - ☞ If one of the operations fails, it means that the key is invalid. In consequence, the keystroke is dropped and ignored by the system.

Before starting to explore the end of the raw input thread procedure, it must be noted the use of the `CBaseInput::RegisterDispatcherObject` routine on both `gpKeyboardSensor` and `gpHidInput` objects. This routine `CBaseInput::RegisterDispatcherObject` is interesting since it links a callback routine called `CBaseInput::_OnDispatcherObjectSignaled` to the object provided in parameter. This last routine is just a direct wrapper to `CBaseInput::OnDispatcherObjectSignaled`. This one is a dispatcher routine able to call the relevant callback routines linked to the provided object. Remember, a *dispatcher* manages a queue of work items to be executed in the order they happen. This set of callbacks routines is based on internal tables predefined composed of undocumented structures. In the case of



the keyboard, there is a call to `CBaseInput::OnReadNotification` callback routine when a key is pressed. This last routine is relevant for us. Indeed, after usual lock access and log procedure and checking if it is dealing with specific processes (`DWM.exe` or `csrss.exe`) to avoid them, it retrieves information from the class object provided. In this object, configured for each device, it is able to retrieve the routine referenced to be used for read operations with this device. For the keyboard, the read operation routine referenced is `CKeyboardSensor::ProcessInput` (from `win32kbase.sys`).

In the `CKeyboardSensor::ProcessInput` routine, there is a call to `CBaseInput::FindDeviceInfo` to retrieve information about the current keyboard device, followed by a call to `CKeyboardProcessor::ProcessInput`. This routine is not really relevant except for its final call to `CKeyboardProcessor::ProcessInputNoLock`. For the sake of simplicity, we can consider that the `CKeyboardProcessor::ProcessInputNoLock` has two possibilities. The first is to call `ProcessKeyboardInjectedInputViaRim` routine which calls `ProcessKeyboardInjectedInput`. This last routine converts the internal key code revived before broadcasting it with `xxxProcessKeyEvent` routine.

The second possibility is to call `ProcessKeyboardInputWorker`. This routine is quite a big routine whose behavior can be split in different parts. The first is to handle the code of the key provided by the keyboard device to convert it in a virtual key code (further information in sections 5.2.5 and 5.2.7) after checking the provided code is valid. In case where the keyboard would not be able to manage keystrokes correctly (*Phantom key* with HID devices), there is a call to `ApiSetEditionUserBeep` routine to ring a *beep* sound to the user.

The conversion from keystroke scan code to virtual key code is performed thanks to `MapScanCode`, `VK-FromVSC` and `InternalMapVirtualKeyEx` routines. The first is used to normalize the scan code received when the two others are used to provide the virtual key code. If the normalization does not work, it means the scan code of the keystroke is unknown for Windows, hence this one is not able to convert it. In this case, the keystroke is dropped and not broadcast to the rest of the system. Once all the conversions have been performed, there is memory manipulation to keep trace of modifier key(s) (control, shift, alt).

If the global value `gdwPUDFlags` has its ninth bit set, a very interesting procedure is launched. First, `AccessTimeOut` routine is called. This one calls `ApiSetEditionSetAccessibilityTimer` to register a potential timer callback routine called `xxxAccessTimeOutTimer`. This callback routine seems to reset a lot of global values while calling `xxxTurnOffStickyKeys` to turn off *sticky keys* [794] (these ones are keys that do not need to be continuously pressed to be considered as pressed — CAPS LOCK is a good example). Once `AccessTimeOut` routine has been called, `AccessProceduresStream` is called. This routine is quite simple since it uses a list of routines to call each of them, one after the other. If one of the routine called returns zero, the loop notification procedure is ended (Figure 4.85). All these routines called in the loop seem to be related to accessibility for the user [795, 796].

```

1 |__int64 __fastcall AccessProceduresStream(struct tagKE *KeyEvent_1, unsigned int ExtraInformation_1, unsigned int index)
2 |{
3 |    unsigned int i; // ebx@1
4 |    unsigned int ExtraInformation; // esi@1
5 |    struct tagKE *KeyEvent; // rbp@1
6 |    __int64 (__fastcall **TableEntry)(struct tagKE *, unsigned __int32, int); // rdi@2
7 |    __int64 (__fastcall *callback)(struct tagKE *, unsigned __int32, int); // rax@3
8 |
9 |    i = index;
10 |    ExtraInformation = ExtraInformation_1;
11 |    KeyEvent = KeyEvent_1;
12 |    if ( index >= 5 )
13 |        return 1i64;
14 |    TableEntry = &AccessRoutines[index];
15 |    while ( 1 )
16 |    {
17 |        callback = *TableEntry;
18 |        if ( !(unsigned int)guard_dispatch_icall_fptr(KeyEvent, ExtraInformation, ++i) )
19 |            break;
20 |        ++TableEntry;
21 |        if ( i >= 5 )
22 |            return 1i64;
23 |    }
24 |    return 0i64;
25 |}

```

<code>AccessRoutines</code>	<code>dq offset HighContrastHotKey</code>
<code>dq offset FilterKeys</code>	<code>dq offset xxxStickyKeys</code>
<code>dq offset MouseKeys</code>	<code>dq offset ToggleKeys</code>

Figure 4.85: Pseudo-code of the `win32kbaseAccessProceduresStream` routine in `win32kbase.sys`.

The rest of the procedure of `ProcessKeyboardInputWorker` is focused on memory manipulation, keeping the last key code received from the keyboard, optionally calling `RemoteSyncToggleKeys` routine in a specific context and, in the end, whatever is the result, calling `xxxProcessKeyEvent`. Note that, if the conversion from scan code to virtual key code did not success, the routine does not execute everything. It only keeps the common lines of code to finish in a clean way the routine.

The important point about dispatchers is that they link callbacks on the objects representing the devices. The goal is to allow them to perform post-processing on the read data from devices. In the context of the keyboard, it means to translate the internal code of the keyboard to a code used by Windows. In addition, it is able to manage sticky keys and other accessibility features. For the sake of clarity, it should be noted that when a key is read, it is first `RIMReadInput` and thereafter `rimIssueReads` which are called to retrieve the content of the key. Once it has been done, it is about `CKeyboardSensor::ProcessInput` and `VKFromVSC` to be called *in fine* to process data provided by the keyboard.

#### 5.1.4.8 End of Raw Input Thread: legacy procedure

##### Key Point 4.38:

- ☞ The legacy procedure of the raw input thread is usually waiting for specific events (with `LegacyInputDispatcher::WaitAndDispatch` routine).
- ☞ When the event is associated with a device, the RIT calls `LegacyInputDispatcher::Dispatch` routine
  - ☞ This routine uses the dispatch callback routines table of the device to perform the required action.
- ☞ But there are other actions not specially linked to a device or which are global for many devices.
  - ☞ For most of events, it is a reaction justifying a reaction from the system.
  - ☞ It could involve a GUI reaction at a mouse event, or to manage shutdown event, reset properties, specific display features, laptop's TouchPad configuration...
  - ☞ Only one general reason called *RitTakeover* is about keyboard device handling.
  - ☞ The *RitTakeover* reasons seems to be linked to sleep and wake mode able to be used by HID devices (for energy consumption issues).
  - ☞ It could allow the system to avoid reading from the keyboard device if this one is in sleep mode.
  - ☞ The RIT legacy procedure is awakened every second by a timer used to compute internal statistics.
- ☞ The RIT can react to specific GUI windows messages (such as clipboard management).
- ☞ At the end of an awaking reason, the legacy procedure loops to wait again for a new event to be signaled.

The *legacy procedure* is not the official name but an extrapolation from the prefix used by the routines called at the end of the procedure. This procedure starts at the end of Figure 4.75 with the call to `LegacyInputDispatcher::Create` (from `win32kbase.sys`) on a local structure which is a *legacy dispatcher* owned by the `RawInputThread` routine. This routine allocates a structure of 64 bytes and it initializes few of its members. The main initialization is performed by `LegacyInputDispatcher::Initialize` which is about basic memory manipulations. Once this action has been performed, the raw input thread calls `CBaseInput::RegisterDispatcherObject` (from `win32kbase.sys`) on `gpKeyboardSensor` and `gpHidInput` with the previously allocated legacy dispatcher. The goal is to link these dispatcher objects to the legacy dispatcher structure. This procedure is the beginning to handle input in what Microsoft considers to be a *legacy way*.

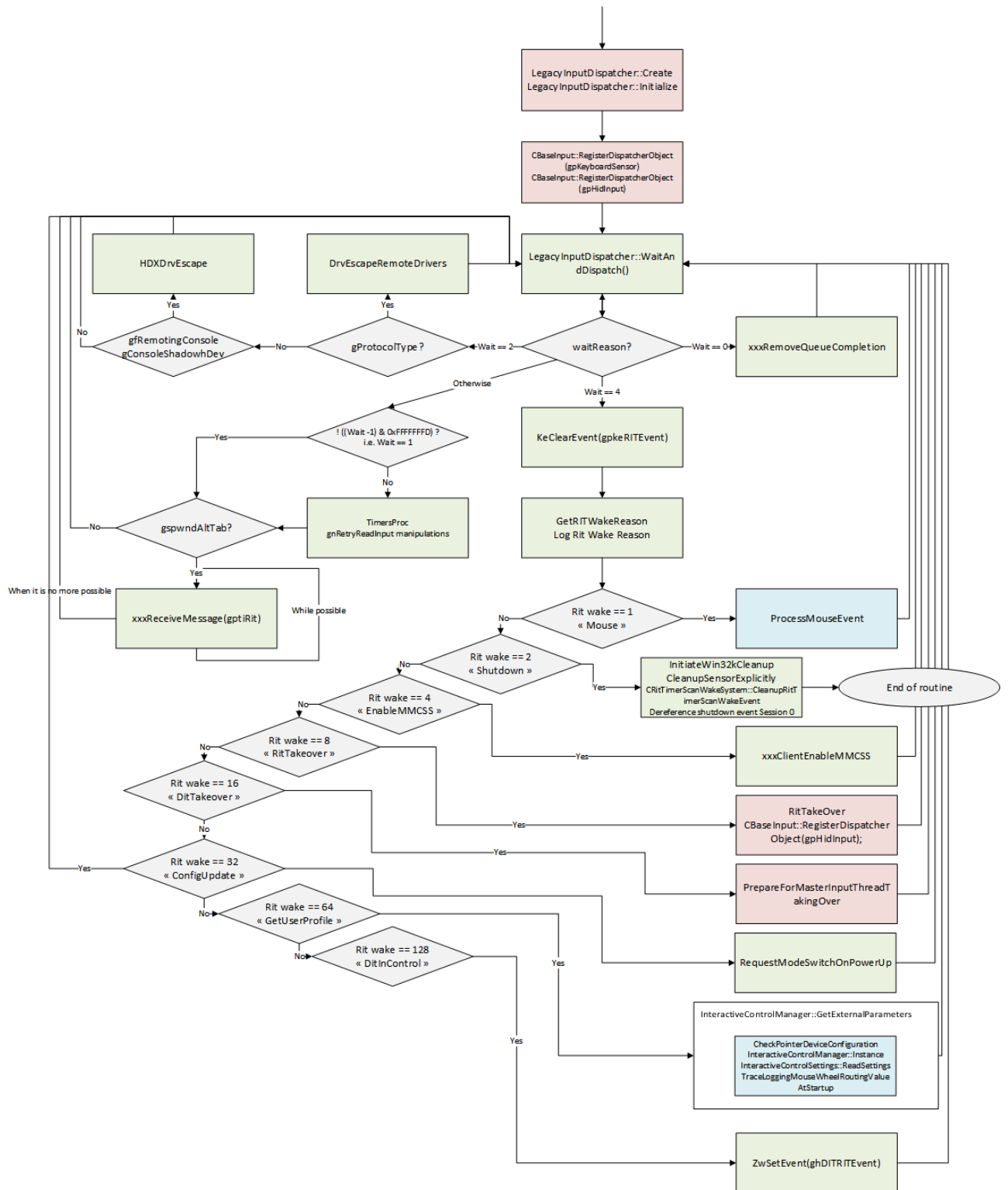


Figure 4.86: End of RawInputThread routine from win32kfull.sys — legacy procedure.

The rest of the procedure is quite complex and a simplified version is given in Figure 4.86. The legacy procedure consists in nested loops around the `LegacyInputDispatcher::WaitAndDispatch` routine (from `win32kbase.sys`). This routine uses a lot of things but is focused on a call to `KeWaitForMultipleObjects` [762] to wait for events to be signaled. These events are part of the legacy dispatcher structure previously allocated and initialized.

If one of them happens, the wait routine is notified and the RIT executes the remaining of `LegacyInputDispatcher::WaitAndDispatch` routine.

In general case, it consists to a call to `LegacyInputDispatcher::Dispatch` routine (from `win32kbase.sys`). This routine has the ability to call dispatcher routines from `gpKeyboardSensor` or `gpHidInput`, depending on the notification received. This explains why whenever we set a breakpoint in `RIMReadInput` (from `win32kbase.sys`), the call stack given in Code 4.23 shows that read operation is handled from the `LegacyInputDispatcher::Dispatch` routine. More directly, this is how the *legacy procedure* read keystroke from keyboard, by waiting for devices' activity and handling them with a dispatcher callback routines table associated to each device.

Note that the `LegacyInputDispatcher::WaitAndDispatch` performs a lot of other operations. For instance, it manages directly LEDs on keyboard with `NeedsUpdateKeyboardLEDs` and `CKeyboardSensor::UpdateKeyboardLED` routines. Indeed, this is the host machine which manages LED on the keyboard and not the keyboard device itself (Key Point 4.2.2). There is also a lot of telemetry and log procedures in `LegacyInputDispatcher::WaitAndDispatch`. But the main point of this routine is the call to `KeWaitForMultipleObjects` routine. This one is called with a list of *waitable* objects composed of devices and events previously initialized by the RIT. Finally, the return value of `KeWaitForMultipleObjects` is a `NTSTATUS` [797] value which can represent an error code or the number of the object which has been signaled. This value is used by the RIT to know the reason of the awakening.

Once the dispatcher has been called, this is the wait index which is returned by the `LegacyInputDispatcher::WaitAndDispatch` routine. This one is used to select the appropriate action to perform. If the wait reason is zero, the routine `xxxRemoveQueueCompletion` is called. This one is based on `ZwRemoveIoCompletionEx`, `ZwAssociateWaitCompletionPacket` and `xxxHandleQueueCompletion`. The last routine is focused on `xxxHandleCoreMessagingQueueCompletion` routine which has the ability to use `PostMessageExtended` and `xxxSendMessage` to broadcast messages. In summary, it can be assumed that the purpose of `xxxRemoveQueueCompletion` routine is to remove orders from IO-controls in order to propagate information to the rest of the system.

If the wait reason is equal to two, there is a test performed to check if `gProtocolType` holds something or not. In the case where `gProtocolType` is not null, there is a call to `DrvEscapeRemoteDrivers`. This routine has the ability, depending on the calling context, to use `HDXDrvEscape`. From the prefix "*HDX*", we might guess it is used in the context of HD video media drivers [798]. If there is nothing in `gProtocolType` but both `gfRemotingConsole` and `gConsoleShadowhDev` are set, there is still a call to `HDXDrvEscape` routine.

When the result is neither 0, nor 2, it can be 4. In such a case, the raw input thread retrieves the reason which woke up the RIT with `GetRITWakeReason` routine. The reason is store in the global value `gdwRITWakeReason` which is modified by `WakeRIT` routine. Once the modification has been set, there is a call to `KeSetEvent` [799] on `gpkeRITEvent` event to signal it. This signal is used to wake up from `LegacyInputDispatcher::WaitAndDispatch` routine and this is why the raw input thread clears the event with `KeClearEvent` [800] once the RIT's wake reason is equal to 4.

Once the wake reason of the RIT has been retrieved, the raw input thread reacts according to the returned value. From what we can observe, the reason is encoded as a bit field, i.e. each bit gives a reason. That way, the RIT reasons look to be exclusive (one at a time). There are multiple reasons which are quite obscure to justify the wake. But thanks to the log procedure in the routine, we can translate most of the reasons justifying the RIT to wake up. These reasons are reported in Figure 4.86 in the conditions with names such as: "*Mouse*", "*Shutdown*", "*RitTakeover*" and so on...

If the RIT reason is equal to one, this is the `ProcessMouseEvent` routine (from `win32kbase.sys`) which is called. This one calls `GetMouseProcessor` which is based on an internal callback coming from an undocumented local value which is managed by mouse routines. Then, there is a call to `CMouseProcessor::ProcessMouseEvent` to process the mouse. It implies to retrieve data from the device and to manage part of the display of the mouse on the screen. This routine is quite complex since it evolves with internal mouse routines such as `CMouseProcessor::CMouseQueue::Dequeue`, `IsValidGuiThreadContext`, `CMouseProcessor::ComputeUIPIForMouseEvent`, `ApiSetEditionForegroundQAccessibleToMouseProducer` and `CMouseProcessor::ContainerMouseInputBuffer::CommitStagedChunkInput` to just name the most relevant ones.

The second RIT reason lies in managing the shutdown event. This one calls a set of routines to remove callbacks and events (`CleanupSensorExplicitly`, `CRitTimerScanWakeSystem::CleanupRitTimerScanWakeEvent`, `InitiateWin32kCleanup`), close handles (`ZwClose`) and release memory (`LegacyInputDispatcher::'scalar deleting destructor'`). This is the only case which allows the routine to return properly. All other RIT wake reasons are destined to loop with `LegacyInputDispatcher::WaitAndDispatch` routine.

The third reason is about enabling *MMCSS* (*Multimedia Class Scheduler Service*) with `xxxClientEnableMMCSS` routine. According to MMCSS's documentation [801], MMCSS is a service that enables multimedia applications to ensure that their time-sensitive processing receives prioritized access to CPU resources. It allows a maximal use of the CPU without denying CPU resources to lower-priority applications. Operations performed by `xxxClientEnableMMCSS` manage internal components which are undocumented and irrelevant in our context.

The fourth reason is one of the most relevant for us. This one is described by *RitTakeover* label. It is responsible to manage the keyboard by a call of two routines `RitTakeOver` and `CBaseInput::RegisterDispatcherObject` on `gpHidInput` and the legacy dispatcher. The `RitTakeOver` routine engages the read operating on the `gpHidInput` object with `CBaseInput::Read` after having initialized it with `CBaseInput::InitializeSensor`. Then it signals the event `ghDITRITEvent` (initialized by `InitDwmInputProcessing` routine) with `ZwSetEvent`. The pseudo code of the routine is given in Figure 4.87.

```

1 NTSTATUS __stdcall RitTakeOver()
2 {
3     if ( CBaseInput::InitializeSensor(*(PVOID *)gpHidInput) >= 0 )
4         CBaseInput::Read(*(_QWORD *)gpHidInput);
5     EtwTraceRitReEngaged();
6     ZwSetEvent(ghDITRITEvent, 0i64);
7     return 1;
8 }

```

Figure 4.87: Pseudo-code of the `RitTakeOver` routine in `win32kfull.sys`.

This first action of *takeover* implies that the RIT has the ability to restart the HID sensor which drives the keyboard in our case. Once this setup operation has been done, the raw input thread calls `CBaseInput::RegisterDispatcherObject` as it did already at the beginning of the legacy part (Figure 4.86). This operation is far from being common but it might be performed in a case of reset operation or after having slept and woke up the HID keyboard device (which has capacity to go in sleep mode).

The fifth RIT reason is about "*DitTakeover*" where *Dit* is not clearly defined from our researches. In this case, this is the routine `PrepareForMasterInputThreadTakingOver` which is called. Its pseudo-code is given in Code 4.88 for illustration purposes. The first part of `PrepareForMasterInputThreadTakingOver` is to reset the threads priorities of `gplnputThread` through `SetThreadPriority` (from `win32kbase.sys` routine) and `gptManipulationThread` via `SetThreadBasePriority` if needed. The `gplnputThread` structure is created by `InputInitialize` when the driver `win32kbase.sys` is loaded in memory. Once the threads priorities have been set to 16, there is a call to `ResetPointerDevices`. This routine calls `RIMResetPointerDevices` on a handle referencing `gpHidInput`. This operation essentially reset internal events and components in `gpHidInput`. The call to `CleanupSensorExplicitly` routine is in the same vein than the call to `ZwSetEvent` on `ghDITRITEvent`. This one will be used thereafter to remove IO completion queue from the RIT object. Finally, `LegacyInputDispatcher::PurgeInputDispatcherObjects` routine is called to reset and remove events and reset and internal members from the legacy dispatcher object. `CBaseInput::RegisterDispatcherObject` is called to register dispatcher for `gpKeyboardSensor` before setting to 1 the global value `gbDIT` indicating that the *Dit takeover* operation has succeeded. For short, this operation is a good way to reset and reinitialize keyboard handlers by the RIT.

Other wake reasons for RIT are less relevant in our case. But just for the sake completeness, we can describe them briefly. The case where the RIT wake reason is equal to 32 is labeled as "*ConfigUpdate*". This one is based on a call to `RequestModeSwitchOnPowerUp`. This routine first calls `IsPrecisionTouchPadEnabled`

```

1 void __fastcall PrepareForMasterInputThreadTakingOver(struct LegacyInputDispatcher *a1)
2 {
3     struct LegacyInputDispatcher *a1_1; // rbx@1
4
5     a1_1 = a1;
6     EnterCrit(0i64, 1i64);
7     SetThreadPriority();
8     if ( *(_QWORD *)gptiManipulationThread )
9         SetThreadBasePriority(**(_QWORD **)gptiManipulationThread, 16i64); // Idle in realtime priority.
10    ResetPointerDevices();
11    gbPendRecreateTouchInjectionDevices = 1;
12    CleanupSensorExplicitly(2i64);
13    EtwTraceRitDisEngaged();
14    ZwSetEvent(ghDITRITEvent, 0i64);
15    LegacyInputDispatcher::PurgeInputDispatcherObjects(a1_1);
16    CBaseInput::RegisterDispatcherObject(*(_QWORD *)gpKeyboardSensor, a1_1);
17    gbDIT = 1;
18    UserSessionSwitchLeaveCrit();
19 }

```

Figure 4.88: Pseudo-code of the PrepareForMasterInputThreadTakingOver routine in win32kfull.sys.

which returns a global value from win32kbase.sys called `gPTPEnabled`. This last value is set by `EnablePTPDevices` routine called from `NtUserEnableTouchPad`. Touch pad [802] is a specific hardware which is used as a mouse on laptop [803]. This one is identified as a HID device (Page 0x0D, Usage 0x05) by Windows operating system [804]. For short, it could be a good way for the system to *guess* whether it is running on a laptop since this type of device is specific to laptop computers. Once the laptop check has been performed, `RequestModeSwitchOnPowerUp` calls `RIMOnPowerNotification` (from win32kbase.sys). This last routine is based on `RIMDeliverConfigRequest` and `RIMSendLatencyMgtDeviceRequest` which both use HID requests to handle hardware device usages (`rimHidP_GetSpecificButtonCaps`, `rimHidP_GetSpecificValueCaps`) and send information to HID device (`rimHidP_SetUsageValue` and `ZwDeviceIoControlFile`). For short, this operation of *config update* seems to be related to the type of computer used (laptop or desktop) and used to setup the underlying hardware device appropriately about power management, among other things.

Wake reason value 64 labeled as "*GetUserProfile*" is essentially a log procedure (internally, there are some strings using "*InteractiveControlManager::GetExternalParameters*") which correspond to a call to `CheckpointDeviceConfiguration` followed by `InteractiveControlSettings::ReadSettings` and `TraceLoggingMouseWheelRoutingValueAtStartup`, all intertwined with logging procedures. The first routine matters for laptops, the second reads information from registry thanks to `InteractiveControlSettings::OpenDeviceKey` and `ZwQueryValueKey` [805]. Routine `TraceLoggingMouseWheelRoutingValueAtStartup` is just about internal log procedure.

Finally, the last reason of wake up for the RIT (number 128) is "*DitInControl*". This one only calls `ZwSetEvent` on `ghDITRITEvent`. This is the same event which is manipulated in the case of the *DitTakeover* RIT wake reason.

At the end, the procedure always loops on `LegacyInputDispatcher::WaitAndDispatch()` to wait for another reason and react accordingly. The last reason to go out from `LegacyInputDispatcher::WaitAndDispatch()` routine is when the wake reason is different from all other reasons. In such a case, there is a specific test to check specific bits of the wake reason. For short, in our case, it is true if and only if the wake reason is equal to one. This event is by far the most signaled one since it is about to be notified every second in a classic use of a Windows session. Notification is performed through a timer object set to be triggered close to every second. In this case, there is a call to `TimersProc`. This routine computes statistics and configures internal structures before resetting `gptmrMaster` timer for the next notification (in one second) with `KeSetTimer` [806] or `KeSetCoalescableTimer`.

After having calling the `TimersProc` routine, there is a check on `gspwndAltTab`. If this value is set, the RIT loops based on internal value `gptiRit` and a call to `xxxReceiveMessage` routine with `gptiRit` as parameter. Routine `xxxReceiveMessage` is a very complex routine. This one has multiple possibilities to act according to its calling context. In addition, this routine can call callback routines from `gapfnScSendMessage`. The list of routines exposed in Figure 4.89 and the names of routines seem to suggest that these callback routines are supposed to handle messages of windows structure. For instance, when reading names of routines such as `SfnINDESTROYCLIPBRD`, `SfnINPAINTCLIPBRD` or `SfnINSIZECLIPBRD`, they all refer to Clipboard [807, 808].



Clipboard is a mechanism used to copy and paste information from one application to another. Clipboard is managed both with different types of messages, which are more or less complex to handle [809]. For instance, there is WM\_PAINTCLIPBOARD message [810] and the callback routine called SfnINPAINTCLIPBRD seems to be related to this message.

```

.rdata:00000001C02D8930 gapfnScSendMessage dq offset SfnDWORD
.rdata:00000001C02D8930
.rdata:00000001C02D8938 dq offset SfnNCDESTROY
.rdata:00000001C02D8940 dq offset SfnINLPCREATESTRUCT
.rdata:00000001C02D8948 dq offset SfnINSTRINGNULL
.rdata:00000001C02D8950 dq offset SfnOUTSTRING
.rdata:00000001C02D8958 dq offset SfnINSTRING
.rdata:00000001C02D8960 dq offset SfnINOUTLPPPOINTS
.rdata:00000001C02D8968 dq offset SfnINLPDRAWITEMSTRUCT
.rdata:00000001C02D8970 dq offset SfnINOUTLPMEASUREITEMSTRUCT
.rdata:00000001C02D8978 dq offset SfnINLPDELETEITEMSTRUCT
.rdata:00000001C02D8980 dq offset SfnINMPARACHAR
.rdata:00000001C02D8988 dq offset SfnINLPHLPSTRUCT
.rdata:00000001C02D8990 dq offset SfnINLPCOMPAREITEMSTRUCT
.rdata:00000001C02D8998 dq offset SfnINOUTLPWINDOWPOS
.rdata:00000001C02D89A0 dq offset SfnINLPWINDOWPOS
.rdata:00000001C02D89A8 dq offset SfnCOPYGLOBALDATA
.rdata:00000001C02D89B0 dq offset SfnCOPYDATA
.rdata:00000001C02D89B8 dq offset SfnINLPHELPIFOSTRUCT
.rdata:00000001C02D89C0 dq offset SfnGETWINDOWDATA
.rdata:00000001C02D89C8 dq offset SfnINOUTSTYLECHANGE
.rdata:00000001C02D89D0 dq offset SfnINOUTNCALCSIZE
.rdata:00000001C02D89D8 dq offset SfnWORDOPTINLPHSG
.rdata:00000001C02D89E0 dq offset SfnINOUTLPSIZE
.rdata:00000001C02D89E8 dq offset SfnINOUTLPPRECT
.rdata:00000001C02D89F0 dq offset SfnOPTOUTLPWORDOPTOUTLPWORD
.rdata:00000001C02D89F8 dq offset SfnOUTLPPRECT
.rdata:00000001C02D8A00 dq offset SfnINCENTOUTSTRING
.rdata:00000001C02D8A08 dq offset SfnPOPTINLPPOINT
.rdata:00000001C02D8A10 dq offset SfnINOUTLPSCROLLINFO
.rdata:00000001C02D8A18 dq offset SfnINCBBOXSTRING
.rdata:00000001C02D8A20 dq offset SfnOUTCBBOXSTRING
.rdata:00000001C02D8A28 dq offset SfnINLBOXSTRING
.rdata:00000001C02D8A30 dq offset SfnOUTLBOXSTRING
.rdata:00000001C02D8A38 dq offset SfnPOUTLPOINT
.rdata:00000001C02D8A40 dq offset SfnOUTDWORDINDWORD
.rdata:00000001C02D8A48 dq offset SfnINOUTNEXTMENU
.rdata:00000001C02D8A50 dq offset SfnINDEVICHANGE
.rdata:00000001C02D8A58 dq offset SfnINLPHDICREATESTRUCT
.rdata:00000001C02D8A60 dq offset SfnINOUTDRAG
.rdata:00000001C02D8A68 dq offset SfnINDESTROYCLIPBRD
.rdata:00000001C02D8A70 dq offset SfnINPAINTCLIPBRD
.rdata:00000001C02D8A78 dq offset SfnINSIZECLIPBRD
.rdata:00000001C02D8A80 dq offset SfnINCENTOUTSTRINGNULL
.rdata:00000001C02D8A88 dq offset SfnDWORD
.rdata:00000001C02D8A90 dq offset SfnDWORD
.rdata:00000001C02D8A98 dq offset SfnSENTDEMSG
.rdata:00000001C02D8AA0 dq offset SfnGETBCSTEXTLENGTHS
.rdata:00000001C02D8AA8 dq offset SfnOPTOUTLPWORDOPTOUTLPWORD
.rdata:00000001C02D8AB0 dq offset SfnDWORD
.rdata:00000001C02D8AB8 dq offset SfnINMPARAMBCSCHAR
.rdata:00000001C02D8AC0 dq offset SfnOPTOUTLPWORDOPTOUTLPWORD
.rdata:00000001C02D8AC8 dq offset SfnIMECONTROL
.rdata:00000001C02D8AD0 dq offset SfnINOUTMENUOBJECT
.rdata:00000001C02D8AD8 dq offset SfnPOWERBROADCAST
.rdata:00000001C02D8AE0 dq offset SfnINLPKDRAWSWITCHWND
.rdata:00000001C02D8AE8 dq offset SfnOUTLPCOMBOBOXINFO
.rdata:00000001C02D8AF0 dq offset SfnOUTLPSCROLLBARINFO
.rdata:00000001C02D8AF8 dq offset SfnINLPUAHDRAWMENU
.rdata:00000001C02D8B00 dq offset SfnINLPUAHDRAWMENUITEM
.rdata:00000001C02D8B08 dq offset SfnINLPUAHDRAWMENUITEM
.rdata:00000001C02D8B10 dq offset SfnINOUTLPUAHDRAWMENUITEM
.rdata:00000001C02D8B18 dq offset SfnINLPUAHDRAWMENUITEMPOPUP
.rdata:00000001C02D8B20 dq offset SfnOUTLPTITLEBARINFOEX
.rdata:00000001C02D8B28 dq offset SfnTOUCH
.rdata:00000001C02D8B30 dq offset SfnGESTURE
.rdata:00000001C02D8B38 dq offset SfnINMPGESTURENOTIFYSTRUCT
.rdata:00000001C02D8B40 dq offset SfnDWORD
.rdata:00000001C02D8B48 dq offset SfnDWORD
.rdata:00000001C02D8B50 dq offset SfnTOUCHHITTESTING
.rdata:00000001C02D8B58 dq offset SfnKEYBOARDCORRECTIONCALLOUT
.rdata:00000001C02D8B60 dq offset SfnSHELLWINDOWMANAGEMENTCALLOUT
.rdata:00000001C02D8B68 dq offset SfnSHELLWINDOWMANAGEMENTNOTIFY
.rdata:00000001C02D8B70 dq offset SfnEMPTY
.rdata:00000001C02D8B78 align 20h

```

Figure 4.89: List of routines held by gapfnScSendMessage value in win32kfull.sys.

This type of callback management is also performed in xxxReceiveMessage with the help of gServerHandlers table (Figure 4.90). The routines stored in this table are more about GUI input management [811], but the logic is similar than the one used with gapfnScSendMessage table.

```

;_int64 (*const near * const gServerHandlers)(struct tagWND *, unsigned int, unsigned __int64, __int64)
?gServerHandlers@@@3QBQ6A_JPEAUTagWND@@@I_KJ@ZB dq offset xxxDefWindowProc
; DATA XREF: xxxSendMessageCallback+3BEftr
; xxxDispatchMessage+406ftr ...
dq offset xxxDesktopWndProc
dq offset xxxSwitchWndProc
dq offset xxxMenuWindowProc
dq offset xxxSBWndProc
dq offset xxxTooltipWndProc
dq offset xxxEventWndProc
align 20h

```

Figure 4.90: List of routines held by ServerHandlers value in win32kfull.sys.

Among different tasks vested to xxxReceiveMessage, one is to manage mouse's messages with LogicalToPhysicalDPIPoint and PhysicalToLogicalDPIPoint in addition to manage mouse low level hooks with xxxCallHook2 routine. According to Microsoft's documentation [812], a hook is a point in the system message-handling mechanism where an application can install a subroutine to monitor the message traffic in the system and process certain types of messages before they reach the target window procedure. Other types of hooks can also be called via xxxCallHook.

Messages are dispatched from xxxReceiveMessage with xxxSendMessageToClient which itself uses gapfnScSendMessage callbacks list and xxxDefWindowProc routine. Also, it uses xxxInterSendMsgEx routine to communicate. This last routine is incredibly complex because it is based on a whole bunch of different cases whose



selection is given by the second parameter. This parameter corresponds to a message ID type. We note the use of `MSGSQMAddMessage`, `TransformMessageBetweenCoordinateSpaces`, `xxxRealSleepThread`, and `MakeUpKeyboardCorrectionCalloutContents` routines internally by this routine. To sum up, this routine does an inter-thread send message. In our context, it seems to process the various messages issued from the RIT and destined to the rest of the system.

To conclude, we observe that the raw input thread has to deal with multiple tasks in addition to take care of keyboard and mouse devices. This is due to the history of the raw input thread which dealt with different components of the display interface. But its main purpose is to manage input coming from input devices (mouse, keyboard, HID) through IRP to translate them into messages in order to broadcast them to the rest of the system. This complex system finally loops on different events which are notified and handled accordingly. This thread belongs in kernel-mode but it is part of `csrss.exe` user-mode process. It is the *interface* between the kernel that manages the hardware, the rest of the operating system and applications that need to have information from this hardware.

---

## 5.2 Broadcast of keystrokes by the system with Window Messages

### Resume 29:

- ☞ In section 5.1 we explained how the contents of the keystroke pass from the kernel to the user mode (via the RIT and csrss.exe which broadcast messages).
- ☞ We will see in this subsection how an application can retrieve this information.
  - ☞ We will present the known methods to interact with Windows messages.
  - ☞ First we will focus on the message system (which is the guideline for keystroke management).
  - ☞ We will document the internal Windows mechanisms that drive this keyboard management from application's point of view.

Now that we know how the contents of the keystroke pass from the kernel to the user mode (via the RIT and csrss.exe), we will see in this subsection how an application can retrieve this information.

### 5.2.1 Foreground thread

#### Key Point 4.39:

- ☞ To receive keystroke hardware events, the RIT is about broadcasting information to a single type of thread per application.
  - ☞ The *foreground thread* corresponds to the default thread created by the system when a GUI window is created.
  - ☞ The thread creating a window owns the window and its associated message queue.
  - ☞ This thread has to deal with messages while its associated window is foreground on the screen.
  - ☞ A window that is in the foreground and active for the user is said to have *focus* property.
- ☞ Property of *foreground thread* belongs to a single thread at time while *focus* belongs to a single GUI window.
- ☞ The system posts keyboard messages to the message queue of the *foreground thread* that created the window with the *focus*.
- ☞ The current window that has the *focus* receives (from the message queue of the *foreground thread*) all keyboard messages until the focus changes to a different window [9].
- ☞ This procedure of foreground thread association is quite complex (Figure 4.93).
  - ☞ There are many reasons for a thread to become a *foreground thread*.
  - ☞ There are also reasons why a thread cannot have *foreground thread* attribute (not enough rights, locked screen...).
  - ☞ This subsection describes the main reasons in detail.
  - ☞ Generally speaking, there is a strong relationship between the *focus* of a window and the *foreground thread* property for the thread associated with the window.

Once the raw input thread has intercepted data from the device, especially keyboard and mouse, it broadcasts it to the rest of the system. There are multiple ways to broadcast information for the raw input thread as we tried to show it previously (but this is not the heart of our subject). For short, the RIT routes the user's keyboard input to a thread's virtualized input queue.

According to literature [731], the way the RIT knows which thread's virtualized input queue to supply depends on the type of message. For mouse messages, the RIT simply determines which window is under the

mouse cursor. It corresponds to what we observed with mouse routines. For keystroke hardware events, the principle is different. Indeed, there is only one thread which is "connected" by the RIT anytime. This thread is called the *foreground thread* [813] and it corresponds to the default thread created by the system when an application window is created [814]. Indeed, using `CreateWindowEx` [815] function for a user-mode application, it forces the thread that creates the window to own the window and its associated message queue. By consequence, it has to deal with messages while the created window keeps the user's focus<sup>28</sup>, meaning it is foreground on the screen. This makes this particular property of *foreground thread* an important notion for finding keystrokes subsequently. In practice, the notion of *foreground thread* is a property (more directly an *attribute*) that is given (or withdrawn) to a thread.

This procedure of foreground thread association is quite complex but it can be checked in the compiled routines in `win32kbase.sys` driver. Two routines are good candidates to hold the change of foreground thread. The first is `CitProcessForegroundChange` and the second is `ApiSetEditionIsAppForeground`. About `CitProcessForegroundChange` routine, this one is called by `CitProcessForegroundChange` and `CitModerncoreProcessForegroundChange`. From the two names of routines, we can suppose there is a legacy version and a modern one. About the legacy version, this one seems to be the one used today since `CitModerncoreProcessForegroundChange` is an exported routine from `win32kbase.sys` but it does not seem to be imported by any driver. It might be used in next versions of Windows as a new procedure. But routine `CitProcessForegroundChange` is highly used by the system.

The call context of `CitProcessForegroundChange` routine is given in Figure 4.93. This routine is called by `xxxSetActiveWindowWithWindowHint` which is itself called by multiple routines. Exhaustive hierarchy and details of these calls are given in Figure 4.93. For the sake of clarity, some sets of specific routines called in specific contexts (for instance, menu manager routines in GUI) are grouped in blocs (in green on Figure 4.93) and user-mode API interface (from `ntdll.dll`) is in red. Example of a set of routines is given in Figure 4.91 about the different routines able to call `xxxActivateWindowWithOptions` in Figure 4.93.

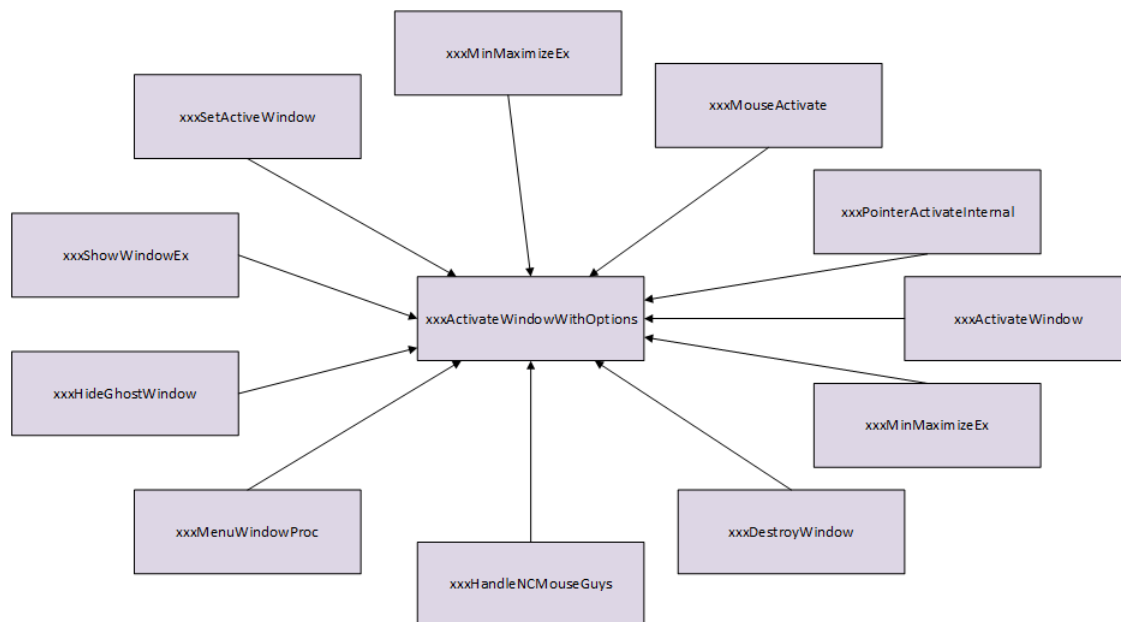


Figure 4.91: View of the different routines able to call `xxxActivateWindowWithOptions` in `win32kfull.sys`.

From this big architecture, it is relevant to keep the user-mode API interface functions which allow a change to the current foreground thread. The set of all<sup>29</sup> functions is given in table 4.15. A documented version of the

<sup>28</sup>In this subsection, the notion of focus is kept deliberately vague for the sake of simplicity. It will be specified in section 5.2.3.

<sup>29</sup>Technically speaking, there are more functions to cite here, for instance by including extended or simplified versions of given functions and their derivatives. But for the sake of simplicity, only those relevant and directly used in Figure 4.93 have been listed

user-mode interface is provided when it is possible.

Nt function name	Documented function	Description
NtUserSwitchDesktop	SwitchDesktop [267]	Makes the specified desktop visible and activates it.
NtUserRealInternalGetMessage	<i>UNDOCUMENTED</i>	Internal version of <code>GetMessage</code> reserved for system use.
NtUserGetMessage	GetMessage [816]	Retrieves a message from the calling thread's message queue.
NtUserPeekMessage	PeekMessage [817]	Dispatches incoming sent messages, checks the thread message queue for a posted message, and retrieves the message (if any exist).
NtUserShowWindow	ShowWindow [818, 819]	Sets the specified window's show state (without waiting for the operation to complete).
NtUserSetFocus	SetFocus [820]	Sets the keyboard focus to the specified window.
NtUserDestroyWindow	DestroyWindow [821]	Destroys the specified window. Give the hand to another window.
NtUserMinMaximize	ShowWindow with <code>SW_MINIMIZE</code>	Same as <code>ShowWindow</code> function.
NtUserCreateWindowEx	CreateWindowEx [815]	Creates an overlapped, pop-up, or child window.
NtUserCreateDesktopEx	CreateDesktopEx [266]	Creates a new desktop and assigns it to the calling thread.
NtUserRemoteStopScreenUpdates	<i>UNDOCUMENTED</i>	Stop operation of update screen on remote desktop.
NtUserSetForegroundWindowForApplication	SetForegroundWindow [822]	Brings the thread that created the specified window into the foreground and activates the window. Keyboard input is directed to the window, and various visual cues are changed for the user. . The system assigns a slightly higher priority to the thread that created the foreground window than it does to other threads.

Table 4.15: List of functions used to update the foreground threads.

From a general point of view, it is possible to update the foreground thread whenever a window takes the *focus* on user's screen. To proceed, there are different possibilities at different levels. From the highest levels, it is possible to switch (or create) from a desktop to another, which implies to change the windows displayed on the screen, giving the foreground thread attribute to another one. Remote desktop operations are involved too. Indeed, by interacting with the window holding a remote desktop connection [823], the keyboard focus is provided to that window (from the client point of view) and removed from the remote desktop window (from the server point of view). This is performed through `xxxRemoteNotify`, `xxxRemoteReconnect`, and `xxxRemoteDisconnect` routines.

At a lowest level, we have the creation and the management of GUI windows. Creation is done with `CreateWindow(Ex)` function [815] while destruction is done with `DestroyWindow` function [821]. In both case, the new window (or one of the remaining) gives to the thread that manages it the property to be the foreground thread. More generally, every action which aims to put an application at the foreground is about to give it this property.

In addition, events to change the foreground thread can come from kernel. This is possible through `xxxProcessActivationEvent`, `xxxDoHotKeyStuff`, `xxxSwitchToThisWindow`, and `xxxApplyArrangeAction` since all manage the display of all windows on current user's desktop or specific hot-key shortcuts to rearrange window display (such as `ALT+TAB` [824]) [825]. Note that the mouse is involved in the procedure of selecting the foreground thread with `xxxMouseActivate` routine from `xxxScanSysQueue` routine. By default, when there is no window to display on the screen, the focus is given to `explorer.exe`. If `explorer.exe` is killed, the access to the raw input thread is free.

Of course, there are other routines able to deal with the foreground thread. `ApiSetEditionIsAppForeground` (`ApiSetIsProcessForeground` only returns the result of a test) (from `win32kfull.sys`) is based on `CoreWindow-Prop::CompositeAppHasForeground` (from `win32kbase.sys`) which calls `GetTopLevelWindow` which checks internals from `Win32k` structures. In `win32kfull.sys`, there are routines able to deal with foreground thread such as:

- `CWindow::ForceForeground` (which internally uses `xxxSetForegroundWindowWithOptions`) to force a thread to get foreground attribute if it is able to get it<sup>30</sup> ;
- `HasRawInputForegroundTarget` to check if the raw input thread as a foreground target (based on `gpqForeground` value) ;
- `ForegroundInputOwnerMatch` to check if a given thread is the one owning the foreground property ;
- `NtUserClearForeground` to remove access to the raw input thread for all windows ;

in table 4.15.

<sup>30</sup>There are different reasons to refuse a thread to get foreground attribute. We can try to name a few. For instance, the thread owns it already, it is executed on a locked screen (checked with `IsWindowUnderActiveLockScreen`), it does not have enough rights (`CheckAccess` from `win32kbase.sys`), the last thread owner is being debugged and so on...

- `NtUserGetForegroundWindow` to retrieve the window linked to the foreground thread ;
- `xxxSetForegroundThread` which is a wrapper for `xxxSetForegroundThreadWithWindowHint` routine.

In the case where there is a window which is driven by the foreground thread, we can observe the raw input thread is checking if there is are GUI which is displayed on the screen. Indeed, during the call to `CBaseInput::OnReadNotification`<sup>31</sup> (from `win32base.sys`), there is a call to `IsValidGuiThreadContext` to check if the current thread has display abilities (Figure 4.92). Technically speaking, this routine checks if the *Win32 Thread* member (undocumented) of the current `_ETHREAD` structure (although undocumented) has specific flags and pointers initialized.

```
::gptiCurrent = (PVOID)gptiCurrent;
gbValidateHandleForIL = 1;
if ( (unsigned int)IsValidGuiThreadContext() )
{
    DomainLock = (PERESOURCE *)GetDomainLockRef(12);
    if ( (_UNKNOWN *)DomainLock == &gDomainDummyLock )
        MicrosoftTelemetryAssertTriggeredNoArgsKM();
    if ( ExIsResourceAcquiredExclusiveLite(*DomainLock) == 1 )
        MicrosoftTelemetryAssertTriggeredNoArgsKM();
    ExEnterCriticalRegionAndAcquireResourceExclusive(*DomainLock);
    gpducstulHead = ::gpducstulHead;
    if ( ::gpducstulHead )
```

Figure 4.92: Part of pseudo code of `OnReadNotificationIsValidGuiThreadContext` routine in `win32kfull.sys`.

---

<sup>31</sup>`CBaseInput::OnReadNotification` is the routine which calls `CBaseInput::Read` when a key is pressed on released on the keyboard. This one is given in the call stack provided for read operation in Code 4.23.

---



## 5.2.2 Window Messages

### Key Point 4.40:

- ☞ The communication component between one application and the rest of the system (the operating system and third-party applications) is called *message*.
  - ☞ Technically speaking, a message is a value, but some messages may have data associated with them.
  - ☞ This is the case with keyboard input messages.
- ☞ The subsection presents the correct procedure to handle messages for a user-mode application.
  - ☞ A *Window Procedure* callback function is necessary to fully cover received messages.
  - ☞ The two main messages about keystrokes are WM\_SYSKEYDOWN and WM\_KEYDOWN.
  - ☞ Using `TranslateMessage` function, it is possible to have WM\_CHAR message holding the corresponding displayable character from keystrokes (whenever it makes sense).

Since we know that the content of keyboard is transmitted through message architecture in Windows, we have to understand how the message architecture works from user-mode application point of view<sup>32</sup> Windows messages [728] are the heart of the GUI subsystem of Windows. A GUI application must respond to events from the user (mouse clicks, key strokes, touch-screen gestures, and so on) and from the operating system (plug and play notification, lower-power state such as sleep or hibernate). These events can occur at any time while the program is running, in almost any order. This is why Windows uses a message-passing model since Windows 2.0 which became fully preemptive with Windows 3.1. The communication component between one application and the rest of the system (the operating system and third-party applications) is the *message*. A message is simply a numerical value that describes a particular event. For example, if the user presses the left mouse button, the window receives a message that has the following message code (Code 4.24).

```
#define WMCOMMAND      0x0111    // Menu or button was pressed.
#define WMLBUTTONDOWN  0x0201    // Left button of mouse was pushed.
#define WMCHAR         0x0301    // A keyboard character was entered.
```

Code 4.24: Example of window messages.

Of course, some messages may have data associated with them. This is the case with keyboard character message or cursor coordinates when a mouse click is done. Let us illustrate our procedure with pieces of code handling message from a Window GUI. First, we need to create a window with `CreateWindowEx` [815, 826]. This window can have a title, a position, a style, a parent window and so on but especially a *window class* [827].

A window class defines a set of behaviors that several windows might have in common. For example, different buttons can share the same function handler to manage each of them. Of course, each button is different (text, position) but they share a similar behavior since it is the same function callback which is linked to each of them. A window class is not a "class" in the C++ sense but it is a data structure used internally by the operating system. Technically, a class is a `WNDCLASSA` structure [828] with many fields. The field `lpszClassName` is one of the most interesting since it holds a string that identifies the window class (the name is local to the process). In addition, `lpfnWndProc` is a pointer to an application-defined function called the window procedure or "*window proc*". The window procedure defines most of the behavior of the window.

To pass a message to a window, the operating system calls the *window procedure* registered for that window. Registration of the class with the operating system is done thanks to `RegisterClass`<sup>33</sup> function [830]. Usually, the

<sup>32</sup>Kernel-mode point of view could be relevant but it is beyond the scope of our document. Nevertheless, it remains that a large part (which is the most relevant in our case) has already been presented here with the explanations about direct memory write operation in processes' memory or *send messages* procedures in the kernel (with `InputExtensibilityCallout::CoreMsgSendMessage` for instance).

<sup>33</sup>The process must destroy all windows using a class before the DLL or the executable are unloaded. To proceed, a call to



registration of the class is performed before calling `CreateWindowEx` so that the last can use the registered class. Once the window is created, calling `ShowWindow` helps to display the window to the user's screen in addition to provide it an access to the keyboard.

```

1 // Register the window class.
  const wchar_t CLASS_NAME[] = L"Sample Window Class";
3 WNDCLASS wc = { };

5 wc.lpfnWndProc = WindowProc;
  wc.hInstance = hInstance;
7 wc.lpszClassName = CLASS_NAME;
  RegisterClass(&wc);

9 // Create the window.
11 HWND hwnd = CreateWindowEx(
    0, // Optional window styles.
    CLASS_NAME, // Window class
    L"Learn to Program Windows", // Window text
    WS_OVERLAPPEDWINDOW, // Window style

    // Size and position
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

    NULL, // Parent window
    NULL, // Menu
    hInstance, // Instance handle
    NULL // Additional application data
  );
25 if (hwnd == NULL) {
    return 0;
27 }

29 // Display the window.
  ShowWindow(hwnd, TRUE);

```

Code 4.25: Creation of a basic window extracted from [826].

A user application is about to receive thousands of messages while it runs, especially if it is dealing with many windows. To handle all messages, the application needs a loop to retrieve the messages and to dispatch them to the correct windows if there is more than one. For each thread creating a window, the operating system creates a queue for window messages (part of the Win32 thread member of `ETHREAD` structure [525], undocumented). This queue holds messages for all the windows that are created on that thread. In literature [731], this queue is sometime called the *virtual input queue* and it would be stored in a structure called `THREADINFO` linked to the *system hardware input queue*. These names are wrong [831] and the `THREADINFO` structure does not appear anymore nowadays (maybe its name has changed). Today, such a list should be present in the `_TEB` (*thread execution block*) [832] in "User32Xxx" filed members. But this structure is largely undocumented and prone to change by Microsoft at any time. One or many lists of posted and input messages can be present but the result for the application is exactly the same at the end [833] (the only difference is psychological). Management between one or different lists could have few differences [834, 835]. Since the queue itself is hidden from the program's eyes and it cannot be manipulated directly, `GetMessage` function [816] must be used instead.

```

MSG msg;
GetMessage(&msg, NULL, 0, 0);

```

Code 4.26: Retrieve message for a GUI thread.

This function removes the first message from the head of the queue. In Code 4.26, `GetMessage` fills in the `MSG` structure [836] with information about the message. This is where the message code and the target window intended are stored. Other parameters from `GetMessage` can be used to filter messages from the queue, but it is not relevant in most of the case. Normally, `GetMessage` returns a nonzero value when a message is

---

`UnregisterClass` function [829] is mandatory.

pumped from the queue. Note that if the queue is empty, the function waits until another message is queued. During that time, other threads in the process of the application can handle other tasks (but the *message thread* is stunk, which could lead to unresponsive applications [837]).

Even if MSG holds all relevant information, it is not usually handled directly. Instead, `TranslateMessage` [838] and `DispatchMessage` [839] functions are used by developers. `TranslateMessage` is related to keyboard input and it translates keystrokes (when a key is down or up) into characters. This one is called before `DispatchMessage` and it does not modify the message provided in parameter but it produces `WM_CHAR` messages [840] only for keys that are mapped to ASCII or Unicode characters by the keyboard driver. This translation function can be used for system messages with `TranslateAccelerator` [841].

The `DispatchMessage` function tells the operating system to call the window procedure of the window that is the target of the message [728]. In other words, the operating system looks up the window handle in its table of windows, finds the function pointer associated with the window, and invokes the function. This operation routes the message to the appropriate window which is supposed to know how to handle it (or to ignore it).

When the window procedure called to handle the message returns, it returns back to where `DispatchMessage` function was called. Of course, there may have other messages in the message queue, which forces to reuse that procedure. This is why messages handling is often implemented in a loop called the *message loop*. As long as the program is running, messages will continue to arrive on the queue and the loop continually pulls messages from the queue and dispatches them. A good way to proceed (so that it allows message loop to quit when the window is destroyed) is given in Code 4.27.

```
MSG msg = {0};
while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

Code 4.27: Message loop correctly implemented (from [826]).

Indeed, if the loop would be an infinite loop, there would be no way to quit it in a clean way. Instead, we propose to use this architecture so that, when the application needs to quit and break out of the message loop, it calls the `PostQuitMessage` function [842]. This last function puts a `WM_QUIT` message [843] on the message queue. This is a special message which forces `GetMessage` to return zero, signaling the end of the message loop in Code 4.27. Bonus, with this construction, the window procedure never receives a `WM_QUIT` message since `GetMessage` ends the loop without reaching `DispatchMessage`, avoiding to handle this type of message.

For general knowledge, there are two ways of handling messages in the system. In this section, we dealt with messages going onto a queue. This is what we call *posting a message*. In such a case, messages go on the message queue and they are dispatched through the message loop (`GetMessage` and `DispatchMessage` as minimal requirement). But there is a way for the operating system to bypass the queue and to call the window procedure directly. Such procedure is called *sending a message*. The difference is not really important except if the application communicates between windows or if it matters to synchronize virtual message generated by application and not by devices [835, 831]. More information are given in [729].

Finally, we must explain how to write the Window Procedure [837] to fully cover the window message subsystem of Windows. Technically speaking, the window procedure targeted by the message is called by `DispatchMessage` function in the message loop. The prototype of the window procedure is called `WindowProc`<sup>34</sup> [844] and it is given in Code 4.28.

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
```

Code 4.28: Prototype of `WindowProc` function callback.

<sup>34</sup>`WindowProc` is a placeholder for the application-defined function name.

This function callback takes four parameters. The first `hwnd` is a handle to the targeted window. The second `uMsg` is the message code (as given in Code 4.24). And finally, `wParam` and `lParam` contain additional data that pertains to the message. The exact meaning of these two last parameters depends on the message code. This is the developer's responsibility to check in Microsoft messages' documentation [726] to cast the parameters to the correct data type according to the message type provided to the callback. Usually, data is either a numeric value or a pointer to a structure when there are messages that do not have any data. The callback can return a code which is an integer value which holds the window's response to a particular message. A typical window procedure is simply a large switch statement that switches on different functions according to the message code.

In the case where an application does not want to deal with specific message types, this one must call `DefWindowProc` [845] with provided parameters to the window procedure. `DefWindowProc` calls the default window procedure to provide default processing for any window messages. This function performs the default actions for the message, which varies by message type. This operation ensures that every message is processed and windows' application are responsive, respecting GUI standards [846].

A full commented sample of messages processing with a newly created window is provided by the Microsoft's documentation [847] for further information. Usually, when dealing with keyboard in the windows procedure, we are dealing mainly with four types of messages: `WM_KEYDOWN` [848], `WM_SYSKEYDOWN` [849], `WM_CHAR` [840], and `WM_HOTKEY` [850]<sup>35</sup>. The first two messages are given for key pressed but there are similar with `WM_SYSKEYUP` [852] and `WM_KEYUP` [853] that correspond to messages notified when keys are released to the windows procedure. The `WM_HOTKEY` message is quite specific since it happens when specific hot keys have been registered through `RegisterHotKey` function [854]. That one helps to define system-wide hot keys, which are combinations of keystrokes involving at least one of the special key: `ALT`, `CONTROL`, `SHIFT` or `WINDOW`. It allows registration (or overwrite) of shortcuts [855] in the system. This is the operating system's responsibility to check if a key pressed matches one combination previously registered as a hot key. In case of match, the system posts the `WM_HOTKEY` message to the message queue of the window with which the hot key is associated and, in case of the windows would be missing, the thread associated with the hot key. Therefore, we can see here that the logic is very different from the strategy of the foreground thread used by the raw input thread.

The difference between `WM_SYSKEYDOWN` and `WM_KEYDOWN` belongs in the fact that the first one is only posted in two different contexts. On the first hand, a window has the keyboard input from the raw input thread and the user presses the F10 key (which activates the menu bar) without the `ALT` key. The same way, this message is sent when the user holds down the `ALT` key and then presses another key (the same way a hot key could be registered for). On the other hand, when no window currently has the keyboard focus, the `WM_SYSKEYDOWN` message is sent to the active window. This message could happen by the use of the "ALT+TAB" combination<sup>36</sup> [857, 858] which is both relevant for `WM_SYSKEYDOWN` message. Indeed, it would happen for the window currently managed by the foreground thread and to the new one which gets the foreground thread while the window switch operation has been performed. The difference between a message to activate a window and one coming to a window managing the keyboard lies in the 29<sup>th</sup> bit of `lParam` from the Window Procedure callback (Code 4.28).

From a practical point of view, the last two parameters are used differently according to the type of message received by the Window Procedure callback. In the case of both `WM_SYSKEYDOWN` or `WM_KEYDOWN` (and `WM_SYSKEYUP` or `WM_KEYUP`), the third parameter of the callback `wParam` corresponds to the *virtual-key code* [516] of the key pressed. The virtual-key codes is a normalized value of the scan code which is universal from the application point of view, whatever is the configuration of the active keyboard. The fourth parameter is a bit-field where specific sections of bits correspond to different information. The bit-field is given in Code 4.29 and in Figure 4.94.

```
typedef struct _LPARAMKEY {
    unsigned int RepeatCount : 16;    // 00 – 15: The repeat count for the current message.
                                     // The value is the number of times the keystroke
```

<sup>35</sup>An exhaustive list of keyboard notifications is given in [851].

<sup>36</sup>Further information about how this key combination's management has involved in [856].

```

        // is autorepeated as a result of the user holding
        // down the key. If the keystroke is held
        // long enough, multiple messages are sent. However,
        // the repeat count is not cumulative.
unsigned int ScanCode : 8;    // 16 – 23: The scan code. The value depends on the OEM.
unsigned int ExtendedKey : 1; // 24: Indicates whether the key is an extended key,
        // such as the right – hand ALT and CTRL keys
        // that appear on an enhanced 101 – or 102 – key keyboard.
        // The value is 1 if it is an extended key; otherwise, it is 0.
unsigned int Reserved : 4;    // 25 – 28: Reserved; do not use.
unsigned int ContextCode : 1; // 29: The context code. The value is always 0 for a
        // WMKEYDOWN message.
unsigned int PreviousKeyState : 1; // 30: The previous key state. The value is 1 if the key
        // is down before the message is sent, or it is zero if the key is up.
unsigned int TransitionState : 1; // 31: The transition state. The value is always 0 for a
        // WMKEYDOWN message.
} LPARAMKEY, *PLPARAMKEY;

```

Code 4.29: Definition of the bit-field used as fourth parameter when a key pressed message is sent to the Window procedure callback.

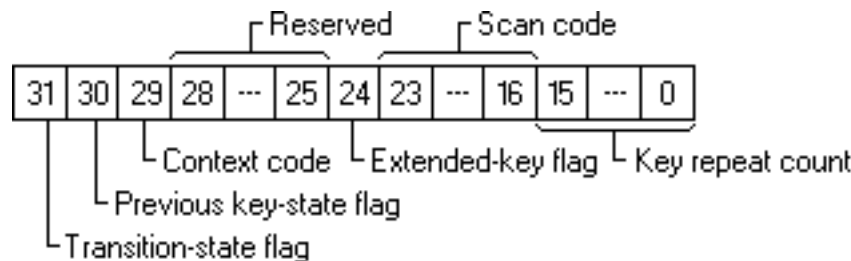


Figure 4.94: Locations of the flags and values used in the lParam parameter when a key down message is received (from [9]).

For the bit-field structure given in Code 4.29 (Figure 4.94) and extracted from [852, 853, 848, 849], we have two observations. The first is that the original scan code provided by the keyboard (which is device manufacturer defined even if in practice they all follow standards from PS/2 scan codes lists or HID keyboard values) and the RepeatCount field are the most relevant information provided. The second observation lies in the difference between WM\_KEYDOWN and WM\_SYSKEYDOWN message where the 29<sup>th</sup> bit is defined in the case of WM\_SYSKEYDOWN and not for WM\_KEYDOWN message (bit-field structure given in Code 4.29 is for WM\_KEYDOWN message). In the case where a key would be released, a *up* message will be sent and the main difference of bit-field structure given in Code 4.29 is the use of “*up*” prefix in keyword instead of “*down*”.

A last message which matters when dealing with keyboard is WM\_CHAR [840]. This one follows a WM\_KEYDOWN message when a key is pressed and TranslateMessage function [838] is called in the context of the message loop. In practice, TranslateMessage function reengages a new message in the message loop with the translated scan code to procedure the character from the pressed key. This property is relevant only for displayable ASCII characters sent by keyboard. When it is not ASCII characters, the function does not issue a WM\_CHAR message. The message translated holds the character provided in Unicode Transformation Format (UTF)-16 to be easily displayed by a GUI window.

The utility is direct in the case of a window requiring text input from the user (word, notepad, text-widget or even password to read it). There is not necessarily a one-to-one correspondence between keys pressed and character messages generated. Indeed, characters modifiers such as SHIFT or *Alt codes*<sup>37</sup> with right ALT plus a number can provide for a large set of keystrokes combination a single character on the screen. Moreover, INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and ENTER to mention just a few have the ability to modify what could be displayed on the screen. With the evolution of Windows operating system comes also

<sup>37</sup>An exhaustive version can be found at: <https://www.alt-codes.net/>.

the evolution of the character sets it supports. Modern applications should better supports WM\_UNICHAR message [859] which is similar to WM\_CHAR but it uses Unicode Transformation Format (UTF)-32, whereas WM\_CHAR uses UTF-16. This message is designed to send or post Unicode characters to ANSI windows and it can handle Unicode Supplementary Plane characters. More information about interacting with keyboard messages is given in [860].

### 5.2.3 Focus on keyboard

#### Key Point 4.41:

- ☞ In fact, when a window has the *focus* on the user's screen, it mostly has the *keyboard focus* property.
  - ☞ This is a temporary property, because the focus can be gained or lost depending on user's interactions.
  - ☞ This property is used to dispatch keyboard messages to visible applications which require direct interaction.
  - ☞ The goal is to get keyboard input without disturbing other applications in background.
  - ☞ In practice, the *keyboard focus* is not be shared with another window.
- ☞ To be accurate: *focus* is a notion reserved for user-mode applications when the notion of *foreground thread* is reserved for thread and only used internally by Microsoft for kernel-mode code.
  - ☞ But in practice it is the same thing for us: it gives an exclusive access to messages coming from the keyboard.
- ☞ The focus can be obtained in three different ways.
  - ☞ From user's interaction (mouse, keyboard shortcuts...).
  - ☞ From Windows API with a function such as `SetFocus` (but not only).
  - ☞ From an application point of view, the focus is message driven with WM\_KILLFOCUS, WM\_SETFOCUS, and WM\_ACTIVATE messages.
- ☞ In general, it is easy to manipulate the focus, which makes this information unreliable from a security or stability point of view.

We have to mention that in MSDN documentation [727] for developers, the term of *keyboard focus* [9] is used. This property (which is a temporary property, because the focus can be gained or lost depending on user's interactions) is used to only dispatch keyboard messages to visible applications which require direct interaction. Such architecture allows one process to get keyboard input without disturbing other applications in background. The result is that, if Microsoft Word application is displayed at foreground, text typed inside the latter is not typed inside another application. If it would be the case, it would result in an unexpected duplication of the input. Hence, it would mean that all applications would handle the keyboard input, resulting in a text broadcast to all applications at the same time, which is not acceptable. Note that in practice, we are using the keyboard on a single application only, with almost no case where the input from the keyboard device is shared and synchronized between more than one window displayed on the screen — even if such specifications are possible.

Even if it can constitute an abuse of language, the *focus* is a notion reserved to user-mode applications (linked to a displayed window) when the notion of *foreground thread* (linked to a thread) is reserved internally by Microsoft for kernel-mode. But both notions describe the same reality: having an exclusive access on the keyboard. This way, it allows a use of the keyboard on the window holding the focus so that the input is only used by the application's foreground thread. It results that other background applications do not receive the input from the keyboard, which is a good thing.

The notion of *focus* is used to describe internally what corresponds to *foreground thread* property. This is



clearly not by chance since the two notions are synonymous. Indeed, any call to `SetFocus` [820] function from user-mode application sets the keyboard focus to the specified window given as parameter. And to proceed, according to the function's documentation, the targeted window must be attached to the calling thread's message queue, which means to be eligible for foreground thread ownership. If we debug the call to `SetFocus` function, this logic appears. In fact, using a debugger, we observe that `SetFocus` is exported by `user32.dll` Dll which is responsible to hold any user responsive API. This one is exported as a direct jump function pointing on `NtUserSetFocus` in `win32u.dll` which is the Dll responsible to interface `win32k.sys` in user-mode. This function is nothing but a classic syscall [861] instruction call to give the hand<sup>38</sup> to the kernel routine which interfaces `NtUserSetFocus` function. The interface kernel-mode routine is `NtUserSetFocus` from `win32kfull.sys` driver. This routine is not unknown from us since it calls `xxxSetFocus` routine and *in fine* `CitProcessForegroundChange` routine as referenced in Figure 4.93. This construction proves the relation between the *focus of the keyboard* and the *foreground thread*.

It is possible to know which window holds the keyboard focus through `GetFocus` function [862]. That one is based on `NtUserGetThreadState` (from `win32u.dll` in user-mode and `win32kfull.sys` in kernel-mode) which uses `gppqForeground` global object to retrieve that information. This object is updated each time a window holds the keyboard focus.

From an application point of view, the *focus* is message driven. Indeed, a call to `SetFocus` function forces the system to send a `WM_KILLFOCUS` message [863] to the window that has lost the focus and a `WM_SETFOCUS` message [864] to the one which now holds it. These messages are sent to the different windows in case of they would need to process specific actions when such events occur. But these last two messages are a legacy procedure coming from the old 16-bit days [865], where there was only one active window, only one focus window and only one keyboard state. At that time, the use of `SetFocus` forced the caller to wait until the previous focus window responded to the `WM_KILLFOCUS` message to see the function returning. With asynchronous input in 32-bit system and above, these types of operations are now local to the thread's input queue. A call to `SetFocus` steals the focus from windows that belong to the caller's thread input queue. Windows which belong to other input queues are unaffected. But there are techniques to deal across different input queues.

The focus is not exclusively managed by the system through the thread's input queue but with the notion of *foreground thread* too. It is possible to interact directly with GUI windows and to change the *active window* which references the foreground thread and hence that holds the focus. Such operation can be driven by the use of a function such as `SetActiveWindow` [866] which generated a `WM_ACTIVATE` message [867] (`GetActiveWindow` function [868] can be used to retrieve the current active window).

The result is quite similar to a user clicking on a given window with the mouse. The difference lies in the `wParam` value of the Window Procedure callback holding the message and the reception of an exclusive `WM_MOUSEACTIVATE` additional message for mouse interaction only via `DefWindowProc`. Note that `WM_ACTIVATE` is also used in the case where the window would be deactivated (`wParam` is used to provide this information through a specific value). This activation of window is possible if the application is not in the background. The difference between foreground and background windows' applications lies in the Z-Order [869] list. According to Microsoft's documentation [869], the z-order of a window indicates the window's position in a stack of overlapping windows. This window stack is oriented along an imaginary axis, the z-axis, extending outward from the screen. This is a single list of windows. The window at the top of the z-order overlaps all other windows. The window at the bottom of the z-order is overlapped by all other windows. When an application creates a window, the system puts it at the top of the z-order for windows of the same type. It is possible to use the `BringWindowToTop` function [870] to bring a window to the top of the z-order for windows of the same type. In practice, we can rearrange the z-order by using the `SetWindowPos` [871] and `DeferWindowPos` [872] functions.

But operations driven with `SetActiveWindow` are limited to the local area of window, linked to the foreground thread's message queue. In other words, application's windows must depend in a way of another to the virtualized input queue of foreground thread that created the windows. To express the "really global active

---

<sup>38</sup>In fact, when a syscall instruction call occurs, the kernel routine `nt!KiSystemServiceCopyEnd` is notified and that one reads the *System Service Descriptor Table* (SSDT) [861] to call the pointer referenced at the index given in `rax` register previously set by `NtUserSetFocus` in `win32u.dll`. In our current version of Windows 10, the index is `0x1054` but this one is prone to change on another version.

window”, there is `SetForegroundWindow` [822] function. This function has limitations of use and restrictions but it is used in many situations by developers. As Raymond Chen explained it [865], this mechanism is supposed to be used only in emergencies since it violates the isolation of input queues, but as seen on other examples [873], eventually nothing is special anymore, and what used to be the special function for stepping outside the box has become the function used every day for getting things done. In fact, `SetForegroundWindow` is still subject to the rules on synchronous input. In the case where the previous foreground window also belongs to the application thread’s input queue, any call to `SetForegroundWindow` will wait until the previous foreground window processes its `WM_ACTIVATE` (with `WA_INACTIVE` value in `wParam`) message.

To avoid limitations and restrictions implied by `SetForegroundWindow` function, it is possible to use `AttachThreadInput` function [874]. In this case, the logic to read keystroke is different. Instead of hijacking the focus of the application’s window, a dedicated thread will be including in the virtual input queue of messages for another thread. It means a thread will be attached to the input processing mechanism of another thread (if the specified thread does not have a message queue). This mechanism is far from being perfect to handle messages. On the first hand, it cannot attach a thread to a thread in another desktop for security reasons. On the other hand, it ties the attached thread’s fate to the targeted thread [865]. It is the targeted thread that still responds to messages and if this one stops, the attached thread is going to stop responding to messages, since the two are sharing the same input queue and operations within an input queue in a synchronous way. If `AttachThreadInput` function can allow to manipulate windows of other programs or bypass the usual rules about focus and activation, it is more likely to rise issues because the two threads are sharing the same input queue and attached operations to interact synchronously. This is particularly true when attachment is performed on a third-party process which does not expect to synchronize access to a queue that is supposed to be handled asynchronously [875].

Last technique about keyboard focus is to fiddle with the parent/child relationship or owner/owned relationship between windows [865]. Indeed, when a window creates another window, their input queues are automatically attached [847, 729]. To proceed, we can use the `SetParent` function [876] to change the parent window of the specified child window. There is no limitation to get access to a window managed by another thread. But when the parent and child windows are in different threads, we have to synchronize the two threads since they are now sharing their input queues. Even if it is technically legal [877], it is very difficult to manage it correctly since all the input queues are attached to each other. More generally, queues of all windows related by a chain of parent/child or owner/owned or shared-thread relationships are attached to each other. Such construction should be avoided since they are prone to error.



#### 5.2.4 Direct raw input access to the keyboard device

##### Key Point 4.42:

- ☞ It is possible to directly interact with the keyboard, bypassing the *raw input thread* with *direct raw input access*.
  - ☞ We gain in access what we lose with the smooth interface provided by the RIT.
  - ☞ This bypass is no more possible with modern versions of Windows (for security reasons).
- ☞ Microsoft allows to do it in another way (and securely) with the *raw input API*.
  - ☞ We get access to device's raw input content thanks to WM\_INPUT messages.
  - ☞ In the context of WM\_INPUT message in a Window procedure, functions `GetRawInputData` and `GetRawInputBuffer` can be used to get a more direct access to the HID content.
  - ☞ With the keyboard context at least, the data provided is nevertheless parsed through the RIT (access is not totally direct).

Technically, there is only one *real* raw input thread in the system. This one aims to retrieve data from input devices in order to broadcast them as messages. But in certain circumstances, it might be necessary to directly interact with the content of what is sent by the device. For instance, one might think about custom data provided by a specific device such as extended information. Such direct content from the device is called *raw input* and there is an official API to interact with it [878]. This direct access mostly concerns HID devices. From a conceptual point of view, we are about to craft our own raw input thread to deal with the raw input data provided by the device. But this is just a conceptual view. Indeed, we cannot interact directly with the keyboard device since keyboard device is opened exclusively by the system [617, 879]. It means we cannot directly access to `"\Device\KeyboardClassN"` (where the final "N" is the keyboard device number) with `CreateFile` function [636].

Protection of the keyboard device can be performed using two different ways. The technical objective is to enable an exclusive access to a device such as only one handle to the device can be open at a time. The first is the standard procedure by setting the exclusive property for the named device object in the device stack [880]. This can be set for WDM device stack that has a both a PDO and a FDO, only by configuring the .inf file with an INF `AddReg` directive [881]. Drivers whose device objects are not stacked, such as non-WDM drivers and devices that operate in raw mode, can use the `IoCreateDeviceSecure` routine [882] to set the exclusive property for their named device object.

The second way is a hand made one used by Windows 10 to manage the keyboard. The procedure used is given in [883]<sup>39</sup> as a day to day evolution of the implementation of the protection for HID devices. Actually, the protection is first set in `HidpRegisterDeviceInterface` routine from `hidclass.sys` driver. Beginning of the routine is given in Figure 4.95. We can see `HidpRegisterDeviceInterface` checking if the HID device registered is a HID keyboard (`UsagePage = 0x01` and `UsageID = 0x06`) [712]. In this case, the device interface class is registered through `IoRegisterDeviceInterface` [884], if it has not been previously registered, to create a new instance of the interface class with the aim that another driver can subsequently enable it to be used by applications or other system components (with `IoSetDeviceInterfaceState` routine [885]). The call to `HidpRegisterDeviceInterface` routine — in addition to having registered a HID device with `GUID_DEVINTERFACE_HID` [633] — is different for HID keyboard devices since it includes an optional third parameter `ReferenceString`. That one is not especially used by regular drivers. But it remains useful when an instance of an interface is opened. In this case, the I/O manager passes the instance's reference string to the driver. The string becomes part of the interface instance's name (as an appended path component). The driver can use the reference string to make a difference between two interface instances of the same class for a single device. The string must not contain any path separator characters (`"/"` or `"\"`) since one would be added automatically.

The rest of the security belongs to `KbdHid_Create` routine from `kbdhid.sys` driver. This routine is called when

<sup>39</sup>This source references bugs in the original implementation and how it has been fixed by Microsoft.

```

32 | HidClassCollection = GetHidclassCollection(v1 + 32, v2);
33 | CollectionDesc = GetCollectionDesc(v1 + 32, *(DWORD*)(v3 + 8));
34 | v6 = (WORD*)CollectionDesc;
35 | if ( HidClassCollection && CollectionDesc ) // If we are dealing with a HID device.
36 | {
37 |     InterfaceClassGuid = GUID_DEVINTERFACE_HID;
38 |     if ( *(WORD*)(HidClassCollection + 8) != 1 // UsagePage = 0x01 -> Generic Desktop Controls.
39 |         || *(WORD*)(HidClassCollection + 10) != 6 // UsageID = 0x06 -> Keyboard.
40 |         || (RtlInitUnicodeString(&DestinationString, L"KBD"), *(WORD*)(HidClassCollection + 8) != 1)
41 |         || *(WORD*)(HidClassCollection + 10) != 6 )
42 |     {
43 |         ReferenceString = 0i64; // If it is not a HID keyboard device.
44 |     }
45 |     else
46 |     {
47 |         ReferenceString = &DestinationString; // If it is a HID keyboard device.
48 |     }
49 |     status = IoRegisterDeviceInterface(
50 |         *(PDEVICE_OBJECT*)(v3 + 48),
51 |         &InterfaceClassGuid,
52 |         ReferenceString,
53 |         (PUNICODE_STRING)(HidClassCollection + 232));

```

Figure 4.95: Beginning of the HidpRegisterDeviceInterface routine in hidclass.sys.

CreateFile<sup>40</sup> is trying to get a direct access to the HID keyboard. In this case, there is a check at the beginning of the create handler given in Figure 4.96. The file object of the targeted device is checked to match “\KBD” string. This string is inherited in the file object via the registration by hidclass.sys with HidpRegisterDeviceInterface routine (Figure 4.95). If it is the keyboard which is targeted, the routine checks if there is a read access (through the ACCESS\_MASK [887]) to the device. If it is the case, a STATUS\_SHARING\_VIOLATION status error is returned.

```

56 | pIoStackLocation = Irp_1->Tail.Overlay.CurrentStackLocation; // IoGetCurrentIrpStackLocation(Irp);
57 | DeviceExtension = DeviceObject_1->DeviceExtension;
58 | DesiredAccess = pIoStackLocation->Parameters.Create.SecurityContext->DesiredAccess;
59 | ShareAccess = pIoStackLocation->Parameters.Create.ShareAccess;
60 | if...
61 | FileObject = pIoStackLocation->FileObject;
62 | if ( !FileObject || !FileObject->FileName.Length )
63 |     goto LABEL_18;
64 | ConstStringKbd = aKbd[4]; // unicode 0, <\KBD>,0
65 | KbdString.Buffer = &LocalBuffer;
66 | LocalBuffer = *L"\KBD";
67 | *&KbdString.Length = 0xA0008i64;
68 | if ( RtlEqualUnicodeString(&FileObject->FileName, &KbdString, 1u) )
69 | {
70 |     if ( DesiredAccess & 1 )
71 |     {
72 |         status = STATUS_SHARING_VIOLATION;

```

Figure 4.96: Beginning of the KbdHidCreate routine in kbdhid.sys.

Note that this procedure is not specific to HID keyboard only. For other type of keyboards, the access procedure is managed directly in kbdclass.sys driver KeyboardClassCreate routine. This routine is registered through the DriverObject [770] inside its *dispatch table* [888] (technically a MajorFunction array of routine pointers) in the DriverEntry routine. It is called every time an IRP\_MJ\_CREATE [889] is notified to the keyboard class driver. When such notification occurs (with CreateFile called directly on the keyboard device name), the KeyboardClassCreate routine checks if the access can be guaranteed (Figure 4.97). This one is always provided for kernel-mode requests but stringent checks are carried out for user-mode ones. In this last case, the required access mode is checked to be as minimal as possible. Otherwise, the routine returns STATUS\_ACCESS\_DENIED status error code, refusing access to the device. This procedure follows general recommendations to control the access to a device driven by a WDM driver [890].

<sup>40</sup>In case of a user-mode application, otherwise, it would be ZwCreateFile [886] routine which would have been used from a kernel-mode driver.

```

16 | Irp_1 = Irp;
17 | DeviceObject_1 = DeviceObject;
18 | v5 = &WPP_RECORDER_INITIALIZED;
19 | if...
20 | StackLoc = (struct _IO_STACK_LOCATION *)Irp_1->Tail.Overlay.CurrentStackLocation;
21 | DeviceExt = DeviceObject_1->DeviceExtension;
22 | JUMPOUT(StackLoc->Flags & 1, 0, sub_1C0003F75);
23 | RequestorMode = Irp_1->RequestorMode;
24 | if ( RequestorMode != UserMode
25 |     || (DesiredAccess = StackLoc->Parameters.Create.SecurityContext->DesiredAccess, !(DesiredAccess & 1))
26 |     || StackLoc->Parameters.Create.Options & 1 )
27 | {
28 |     status = IoAcquireRemoveLockEx((PIO_REMOVE_LOCK)DeviceExt + 1, Irp_1, File, 1u, 0x20u);
29 |     if...
30 | }
31 | else
32 | {
33 |     status = STATUS_ACCESS_DENIED;
34 |     if ( WPP_RECORDER_INITIALIZED != v5 )
35 |     {
36 |         WPP_RECORDER_SF_qqdDdd(WPP_GLOBAL_Control->DeviceExtension, DesiredAccess, (char)v5, v2);
37 |         RequestorMode = Irp_1->RequestorMode;
38 |     }
39 |     if ( !RequestorMode )
40 |     {
41 |         sub_1C0003F7D(0, (__int64)StackLoc);
42 |         return result;
43 |     }
44 | }
45 | Irp_1->IoStatus.Status = status;
46 | Irp_1->IoStatus.Information = 0i64;
47 | IofCompleteRequest(Irp_1, 0);
48 | if...
49 | return (unsigned int)status;
50 | }

```

Figure 4.97: Pseudo-code of the KeyboardClassCreate routine in kbdclass.sys.

In the old days of Windows, it was possible to directly interact with the keyboard. Such functionality still exists [879]. The Win32 subsystem opens legacy devices by name (for example, "\Device \KeyboardLegacy-Class0"). Note that once the Win32 subsystem successfully opens a legacy device, it cannot determine if the device is later physically removed. Once access has been guaranteed, a dedicated thread must be available (or created if required) to allow dispatch processing by managing the shared mode and periodically read the device or set up the I/O completion port, and so forth. This operation is usually performed in kernel-mode and tends to disappear by an extinction of devices requiring such management nowadays (mostly PS/2 devices). Access from user-mode application is strictly supervised and it is *de facto* a kind of information rerouting from the raw input thread (and in no case a direct access).

To avoid implementing a dedicated driver to handle raw input of a given device, Windows proposes a raw input API [878, 891]. The management is still similar to the message system previously presented with a Windows Procedure handling message notifications such as WM\_CHAR, WM\_MOUSEMOVE, and WM\_APPCOMMAND. But if an application registers itself to get access to the raw input of a device (not especially a keyboard), it is about to receive WM\_INPUT messages [892]. For short, the raw input notification still leans on the raw input thread which transfer messages, pumped in a message loop in the application and handled in the Window Procedure callback. In such a case, the third parameter wParam of the WindowProc callback [844] (Code 4.28) clarifies if an input occurred while the application was in the foreground or not. The fourth parameter lParam is a HRAWINPUT handle to the RAWINPUT [893] structure that contains the raw input from the device. To get the raw data from that handle, a call to the GetRawInputData function [894] with that handle allows to retrieve a pointer on a specific structure holding the raw data from the HID device. Technically, a pre-parsing operation is performed in the RAWINPUT structure for mouse [895], keyboard [896], and other HID devices [897]. In practice, information is retrieved from these structures after a call to GetRawInputData function since it is easier to manage for standard HID devices (mouse and keyboard). The true raw data from the device, to be parsed by the application, is generally reserved to highly customized devices. Afterwards, information about the HID device targeted by the raw input operation can be retrieved through GetRawInputDeviceInfo function call [898]. Note that dealing with specific devices, the buffered method to retrieve HID raw input gets an array of RAWINPUT structures RAWINPUT at a time. In this case, the application calls GetRawInputBuffer [899]

to get an array of RAWINPUT structures (practical example in [900]).

Since we know how to handle HID raw input notifications, it matters to know how to register for such notifications [891]. By default, no application receives raw input which means that it is necessary to be a volunteer to receive raw input. To register raw input notification from devices, an application first creates an array of RAWINPUTDEVICE structures [901] that specify the top level collection (TLC) for the targeted devices the application wants. The TLC is defined to be selected by its Usage Page (the class of the device) and its Usage ID (the device within the class). In the case of the keyboard, Usage Page is equal to 0x01 and Usage ID is equal to 0x06. With this configuration set in a RAWINPUTDEVICE structure, the application calls `RegisterRawInputDevices` function to register raw input from keyboard. Example of registration is given in Code 4.30.

```
RAWINPUTDEVICE Rid[1];
// (...)
Rid[0].usUsagePage = 0x01; // HID_USAGE_PAGE_GENERIC;
Rid[0].usUsage = 0x06; // HID_USAGE_GENERIC_KEYBOARD;
Rid[0].dwFlags = RIDEV_INPUTSINK;
Rid[0].hwndTarget = hWnd; // From CreateWindowEx function call.
if (RegisterRawInputDevices(Rid, 1, sizeof(Rid[0])) == FALSE) {
    _tprintf(_T("[-] Error: RegisterRawInputDevices failed with status 0n%03d.\n"), GetLastError());
    __leave;
}
_tprintf(_T("[+] RegisterRawInputDevices succeed.\n"));
```

Code 4.30: Registration for keyboard raw input notifications.

One relevant point in Code 4.30 is the use of `dwFlags` in the RAWINPUTDEVICE structure. This flag can be used to select the devices to listen to and also those to ignore. Example of such flags are given in [900]. Note that an application can register a device that is not currently attached to the system. When this device is attached, the Windows Manager will automatically send the raw input to the application. In addition, it is possible to get a list of raw input devices on the system. This can be done through `GetRawInputDeviceList` [902] to obtain a handle to the list and therefore to call `GetRawInputDeviceInfo` function [903] with that handle in a loop to retrieve each device's information. A keyboard will be identified with `RIM_TYPEKEYBOARD` type flag in the RAWINPUTDEVICELIST structure [904] provided by `GetRawInputDeviceList` function. An example is given in [905].

### 5.2.5 Virtual key codes and keyboard layout

#### Key Point 4.43:

- ☞ Virtual key code is a standardized set of values to represent each key with a virtual value.
  - ☞ This code is used to avoid applications to deal with all layouts of keyboard devices.
  - ☞ Virtual-key codes are device-independent.
- ☞ To proceed, Windows offers capabilities to translate from keyboard scan code to virtual key code.
  - ☞ This translation depends on the *keyboard layout* (English, French, German, Japanese...).
  - ☞ In Windows' documentation, *input local identifier* is the new name for *keyboard layout*.
  - ☞ Most of translation operations can be performed with `MapVirtualKeyEx` function.
- ☞ The system records a list of Keyboard Identifiers.
  - ☞ Keyboard Identifiers are language identifiers referenced as numbers.
  - ☞ U.S. English layout is named "00000409", for instance.
  - ☞ In the registry of Windows, *Layout File* references a Dll that manages specificities of a given keyboard layout.
  - ☞ This Dll is called in kernel-mode context part of the *raw input thread* procedure.
  - ☞ Despite being partially documented, it is possible to create our own Dll layout (requires administrator rights).
  - ☞ Loading by default a layout a start-up does not require specific rights (its is part of user's preference).
- ☞ At creation time, each process inherits the keyboard layout from the current user's desktop.

Historically, there are many types of keyboards, each with its own layout or hardware shape. Generally, it depends from the world region from where the user comes from. Indeed, there are differences of key arrangement between German, French, and English keyboards to name a few. If applications would have to deal with each of these configurations, interacting with keyboard would be a nightmare. To solve this issue, Windows proposes from a long time a standardized set of values to represent each key with a virtual value. This set is called *virtual key codes* [516]. With such a code, an application will almost never have to interact directly with scan codes [727]. The keyboard driver translates scan codes into virtual-key codes (section 5.1.4.7 with Key Point 4.37). More directly, virtual-key codes are device-independent. For instance, pressing the "A" key on any keyboard generates the same virtual-key code on every Windows operating system.

In general, virtual-key codes do not correspond to ASCII codes or any other character-encoding standard. It makes sense when we think that multiple keys could be pressed to generate a specific character, like an accented character. Instead, it represents the individual keys used on any keyboard, whatever is the combination of keystrokes used and whatever is the layout of the physical keys on the keyboard.

It is possible to perform the translation manually from scan code to virtual key code (and vice-versa). Function `MapVirtualKeyEx` [906] translates the code given in first parameter from one code type to another thanks to the second parameter used to select the translating mode (for instance, `MAPVK_VK_TO_VSC` or `MAPVK_VSC_TO_VK`). The third parameter is an input locale identifier, that is to say the specific *language* layout used by the keyboard. This input locale identifier is the value returned by `LoadKeyboardLayout` function [907]. This last function loads a driver layout thanks to its name. This name is a string composed of the hexadecimal value of the Language Identifier (low word) and a device identifier (high word). For instance, U.S. English has a language identifier of 0x0409, so the primary U.S. English layout is named "00000409". Variants of U.S. English layout (such as the Dvorak layout) are named "00010409", "00020409", and so on.

The list of *Keyboard Identifiers* is managed in the registry of Windows under the key "HKLM\SYSTEM\ControlSet001\Control\Keyboard Layouts" [908]<sup>41</sup>. For each keyboard layout setup on the system, there is a sub-key. Each sub-key is named according to the keyboard identifier value linked to a specific keyboard. Each key has three values. The value "*Layout File*" references a Dll which holds the code to manage specificities for the keyboard while "*Layout Display Name*" is a string<sup>42</sup> providing the name of the keyboard identifier and a *Layout Text* which is a real string text given the name of the keyboard type. This configuration is used to select the current keyboard used by the user. More directly, during the initial setup of Windows operating system, the user first identifies the time and date format and Windows uses that information to configure the input profiles (or input locales) for that user [911]. Technically, the first user logging in Windows identifies its region and initializes the input profile that describes the language and the keyboard associated to the input entered. This configuration is going to be used each time the user logs to the system. Thereafter, it is possible to update at running time the configuration of keyboard layout through a procedure given in [911].

Such architecture allows to use different *input local identifiers* at different scales. Note that in Windows' documentation, *input local identifiers* is the new name for what was known as *keyboard layouts*. Technically, there is one general layout sets for the current user's desktop which is inherited by default for each process at creation time (and provided to the thread handling window messages). This layout is a bit different (even if it is part of it) from the general layout provided by the system to a process which can be retrieved through `GetProcessDefaultLayout` [912]. This layout is used to defines how text and Windows *Graphics Device Interface* (GDI) objects are laid out in a window or device context. Some languages, such as English, French, and German, require a left-to-right layout. Other languages, such as Arabic and Hebrew, require right-to-left layout [869]. The general list of layouts is accessible through `GetKeyboardLayoutList` function [913]. This general layout can be changed using `SetProcessDefaultLayout` [914].

It must be noted that the natural layout from a hardware keyboard device is not provided with HID description. In a way, it would be possible to use the *bCountryCode* field in the USB descriptor data (table 4.11 in section 4.2.2) to carry this information. But it is not used by devices manufacturers. By the way, the keyboard layout is not part of the HID information. It is more a notion that is specific to the operating system. Technically speaking, the key's codes represent a physical key position on the keyboard. It means that keys on a keyboard device are usually in the same place, whatever is the symbol they represent. More directly, on all keyboards, each key will send the same code no matter what the layout (english, german, korean, etc.) is. In the context of HID, it is written in the documentation that there is no key mappings for every language [591]. Instead of, the keys should always map the same code with the closest equivalent key position than the one given in HID documentation [915] (section 4.2.1). This strategy allows that a keyboard may be modified for a different language by simply using a keyboard driver translating from one universal code (HID Usage ID table) into a language specific one, based on the selected keyboard layout. This driver is installed when the keyboard device is plugged for the first time or manually by the user [916, 917]. Note this part is also driven through the input local identifiers with the associated layout Dll file, as presented previously.

Coming back to the keyboard layout specifically, it is possible to get the one used by the system via `GetKeyboardLayoutName` function [918] which completes a string buffer with the name of the input locale identifier. Retrieving one thread's keyboard layout is done via `GetKeyboardLayout` function [919] which takes in argument the thread ID from which to return the active input locale identifier. It is also possible to change the keyboard layout at thread level. Function `ActivateKeyboardLayout` sets the input locale identifier for the calling thread or the current process. The input local identifier specifies a local as well as the physical layout of the keyboard.

When a keyboard layout is changed for an application, that one receives a `WM_INPUTLANGCHANGE` message [920]. This message is sent to the topmost affected window after an application's input language has been changed. It is recommended to make any application-specific settings in the function handling this message before passing it with `DefWindowProc` function. This function passes the message to all first-level child windows. Thereafter, these child windows can pass the message via `DefWindowProc` to pass the message to their child windows, and so on. This helps to broadcast the information to all windows in an application. Note that a change of layout can be performed by the user either with the specific hotkey (specified in the Keyboard

---

<sup>41</sup>A list of keyboard identifier values is given in [908].

<sup>42</sup>This string is usually referenced as a resource export [909, 910] from a Dll.



control panel application — by default Alt+Shift) or from the indicator on the system taskbar. In this case, applications are notified through `WM_INPUTLANGCHANGEREQUEST` message [921]. An application can accept the change by passing the message to the `DefWindowProc` function or to reject the change (hence preventing it from taking place) by returning immediately. If there is no rejection of the input language layout to change, the application is then notified with a regular `WM_INPUTLANGCHANGE` message.

Internally, the change of the keyboard layout (via `ActivateKeyboardLayout` function) results in a syscall in `NtUserActivateKeyboardLayout` from `win32u.dll`. It results in the execution of `NtUserActivateKeyboardLayout` from `win32kbase.sys`. This routine deals with members from the undocumented `Win32Thread` structure (retrieved via a call to `PsGetCurrentThreadWin32Thread` routine). Thereafter, it checks if the calling thread has a valid GUI context `IsValidGuiThreadContext` and dealing with `gptiCurrent`<sup>43</sup>, it calls `xxxActivateKeyboardLayout` routine. If required in a flag provided as parameter, this routine orders the list of keyboard layouts (they are represented as a list that the user can switch to the next or the previous layout in the list, especially with hotkeys) with `ReorderKeyboardLayouts`. But `xxxActivateKeyboardLayout` is mainly dedicated to call `xxxInternalActivateKeyboardLayout` routine. This routine has the ability to call either `ApiSetEditionImmActivateThreadsLayout` or `ApiSetEditionImmActivateLayout` from `win32kbase.sys`. The first calls `EditionImmActivateThreadsLayout` from `win32kfull.sys` which is a wrapper to `xxxImmActivateThreadsLayout` that is supposed to send messages to windows via `xxxImmActivateLayout` thanks to a call to `xxxSendTransformableMessageTimeout`. Routine `ApiSetEditionImmActivateLayout` is a direct call to `xxxImmActivateLayout`.

Back to `xxxInternalActivateKeyboardLayout` routine, once the notification has been performed, if the calling thread is the foreground thread (`gptiForeground` object), there is a call to `xxxChangeForegroundKeyboardTable` followed by a call to `xxxWindowEvent` (to notify a `WM_INPUTLANGCHANGE` event if the window has the focus) and `ApiSetEditionNotifyShellLanguageHook`. Otherwise, there is just a call to `xxxChangeForegroundKeyboardTable` routine. Finally, the routine `xxxInternalActivateKeyboardLayout` finishes with `ApiSetEditionSendIMENotification` which calls `EditionSendIMENotification` from `win32kfull.sys`. This routine calls `xxxSendTransformableMessageTimeout`. However, the main action is performed within the `xxxChangeForegroundKeyboardTable` routine from `win32base.sys`.

The `xxxChangeForegroundKeyboardTable` routine takes two parameters: the old keyboard layout to update and the new one. It checks first if there is a global keyboard layout registered and that the new keyboard layout selected is different from the new one to use. This routine is used to update internal structures to change the keyboard layout from the provided parameters. In addition, a specific set of procedures is implemented to deal with Japanese keyboards which are quite specifics. At the end, global values are updated through a call to `SetGlobalKeyboardTableInfo` routine. These values are: `gpKbdTbl` (keyboard table which corresponds to the keyboard layout), `gpKL` (which is the current keyboard layout itself — referenced `tagKL` symbol from Microsoft Windows' debug symbols), `ghKbdTblBase` and `guKbdTblSize` to locate the keyboard table in memory and `gpKbdNlsTbl` where NLS stands for *National Language Support* [922]. The NLS allows applications to set the locale for the user, identify the language in which the user works and retrieve strings representing times, dates, and other information formatted correctly for the specified language and locale. More information about NLS and specific terms when working with NLS functionality can be retrieved in [923].

Note that `gpKbdTbl` is initialized in the entry point of `win32kbase.sys` with `KbdTablesFallback` value. It can be changed in `RemoveKeyboardLayoutFile` and `SetGlobalKeyboardTableInfo` routines. The last routine is called from `xxxLoadKeyboardLayoutEx` itself called from `xxxSafeLoadKeyboardLayoutEx` which is called by `NtUserLoadKeyboardLayoutEx`. This is the kernel interface for `LoadKeyboardLayout` function [907] in user mode. This function loads a new input keyboard layout into the system.

In the context of `xxxChangeForegroundKeyboardTable` routine, this one can call `xxxManageKeyboardModifiers` routine. This routine calls `xxxKeyEventEx` (which is an interface to `xxxUpdateGlobalsAndSendKeyEvent`). The last calls `UpdateAsyncKeyState`, `xxxWindowEvent`, `ApiSetEditionHandleRawInput`, and `ApiSetEditionHandleAndPostKeyEvent` among the most relevant routines called inside. These last routines have the ability to notify windows with messages, update the raw input thread and update internal structures representing the key pressed in `win32k`. More generally, the change of the keyboard layout for the foreground thread updates more than the

---

<sup>43</sup>This value is guessed by us to represent the current thread which has a capacity to deal with input.



layout itself. It generates a set of events notifying user-mode application that such a change occurred in addition to update (to clear keystrokes mostly) the current memory structure representing the state of keystrokes for the keyboard.

Apart the call to `xxxManageKeyboardModifiers` in `xxxChangeForegroundKeyboardTable` routine, the last deals with `gfkKanaToggle`, `gafAsyncKeyState`, and `gptiForeground` objects. From a general point of view, it helps to represent internally the last keystrokes (with `gafAsyncKeyState` object) from the foreground thread (`gptiForeground`) and it has the ability to deal with complex layout (such as *kana* from Japanese alphabet). Note that while the management of some of these specific alphabets has been officially standardized, the custom use from different software companies imposes sometimes a *de facto* standard that Microsoft has to follow (case of Korean jamo [924]), which explains the high complexity about the management of these specific alphabets.

Note that despite the long list of keyboard layouts already supported by Windows, it is possible to generate a custom keyboard layout with internal tools from Microsoft [925, 926]. This can be done by generating a Dll following some specifications [927] and using a dedicated compilation process with the Keyboard Layout Generator tool [928]. This Dll needs to export a function called `KbdLayerDescriptor` (and optionally a function called `KbdNlsLayerDescriptor` for NLS purposes [922]) which provides an access to a `KBDTABLES` structure which is partially documented [929]. Example hosted on MSDN<sup>44</sup> can be considered as a *de facto* documentation to create a keyboard layout Dll. Note that there are samples and some documentation online [930, 931], but it remains close to Windows' documentation and far to release true open-source projects.

The layout Dll is loaded by the window manager when needed. One of the examples is the logon but it can be performed at any time. If the keyboard layout is registered in "HKLM\SYSTEM\ControlSet001\Control\Keyboard Layouts" in the registry of Windows, the default set of the input locales is set in the "HKCU\Keyboard Layout\Preload" registry key. Contrary to HKLM (which stands for `HKEY_LOCAL_MACHINE` [932]) which requires administrator rights, HKCU (which stands for `HKEY_CURRENT_USER` [932]) does not require such rights since it can be modified by a regular user. This is done to load user's preference, which can be customized by the Regional and Language Options application in Control Panel [927]. That way, the window manager reads the HKCU registry and loads the keyboard layouts accordingly.

From a security point of view, we need to be an administrator to register a new layout in the system (writing in HKLM hive and Dll file should be stored in System32). However, when it already exists, it is easy to have a keyboard layout loaded automatically at start-up by inserting in *Preload* key some string values where the name is the loading order number and the content is the keyboard layout language identifiers.

---

<sup>44</sup>[https://github.com/microsoft/Windows-driver-samples/releases/download/81843/Keyboard\\_Layout\\_Samples.zip](https://github.com/microsoft/Windows-driver-samples/releases/download/81843/Keyboard_Layout_Samples.zip)

### 5.2.6 Sending keyboard input to an application

#### Key Point 4.44:

- ☞ It is possible to simulate a key pressed from the keyboard with a software (and without physical action on the hardware).
  - ☞ Function `SendInput` is designed for that purposes.
  - ☞ It drives the RIT the generate messages and all internal events as the key would have been pressed.
  - ☞ Events include LED lamps management, system hooks, updating internal keyboard state structures...
  - ☞ Better that simulating messages directly with `SendMessage` function.
- ☞ Insertion of keystroke simulation is performed through virtual key code (otherwise a conversion is performed via `VKFromVSC`).
- ☞ The keystroke simulation involves a lot of the internal structures of the RIT presented in section 5.1.4.
  - ☞ The RIT is in charge to display specific events linked to simulated keys.
  - ☞ A lot are dealing with *National Language Support* (NLS) and *accessibility feature* (such as *sonar mouse*).
  - ☞ A special management is taken for the numeric pad which can produce action keys when the NUM LOCK key is pressed.
- ☞ Generally speaking, it is possible to make applications believe that a simulated key comes from the keyboard device.
  - ☞ Because it comes from the RIT which manages any simulated key as if it would come from the device.

To be complete on the subject of message management, it is possible to create and inject messages to the message queue of the focused window. This message manipulation is performed at software level and it does not require any hardware interaction. In a way, this can be seen as *event simulation*. Instinctively, one may be tempted to reproduce the chain of messages received when pressing a key on the keyboard. This operation could be performed thanks to `SendMessage` [933] or `PostMessage` [934] functions [935]. Indeed, these two functions have the ability to insert a message in the message queue associated with the thread that created a specified window. But if the result seems to be efficient, this does not constitute the correct way to proceed [936].

Undeniably, keyboard input is a complex subject, especially when dealing with non English keyboard. Languages with accent marks have dead keys, far-east languages have a variety of *Input Method Editors*<sup>45</sup> and scripts languages (for instance Japanese with Hiragana and Katakana) are another world. Enter a character is more than just reporting that a pressing key event occurred. Another reason why not using `SendMessage` function is that some internal keyboard structures (especially in win32k) will not be updated if such a message forging procedure is used. This means that a message signaling a keystroke is well received by an application, but the state of the keyboard does not reflect the same reality. By using this message simulation technology, we make sure that some sensors do not have the same information as others. This is an issue when facing complex input procedures, especially with video games. The lasts monitor different sources of input which, in the end, would may not all describe the same reality.

Using the `SendInput` function [940] is the correct way to proceed. This function synthesizes keystrokes, mouse

<sup>45</sup>An *Input Method Editor* (IME) is a service which is used by applications in order to allow easy text entry using a standard keyboard for East Asian languages such as Chinese, Japanese, Korean, and other languages with complex characters [937, 938]. This is part of *Input Method Manager* (IMM) technology [939].

motions, and button clicks since it was designed for injecting input into Windows. The purpose of `SendInput` is to insert a set of events via `INPUT` structures [941] serially into the keyboard or mouse input stream. These events are not interspersed with other keyboard or mouse input events previously inserted either by the user. This function does not reset the keyboard's current state which corresponds to internal structures in `win32k`. It means that any keys already pressed when the function is called might interfere with the events that this function generates.

Internally, `SendInput` function is exported by `user32.dll` which is an *import forward* [942, 943] from `win32u.dll`. In the last Dll, this is the function `NtUserSendInput` which performs a syscall to give the hand to `NtUserSendInput` in `win32kfull.sys`. The pseudo-code of this routine is given in Figure 4.98. After checking the size of the parameters provided and if pointers are correctly initialized, the user-mode buffer holding the list of input events to insert is copied in an allocated kernel-mode memory buffer after having been probed for security reasons [944, 945, 946]. When user-mode buffer management has been correctly performed, the kernel-buffer holding information from the user-mode buffer is given to `xxxSendInput` routine.

```

1 | int64 __fastcall NtUserSendInput(UINT cInputs, LPINPUT pInputs, int cbSize)
2 | {
27 |     cbSize_1 = cbSize; // The size, in bytes, of an INPUT structure.
28 | // If cbSize is not the size of an INPUT structure,
29 | // the function fails.
30 |     pInputs_1 = pInputs; // An array of INPUT structures. Each structure represents
31 | // an event to be inserted into the keyboard or mouse input
32 | // stream.
33 |     cInputs_1 = cInputs; // The number of structures in the pInputs array.
34 |     if...
35 |     InputTraceLogging::ThreadLockedPerfRegion::ThreadLockedPerfRegion(
36 |         (InputTraceLogging::ThreadLockedPerfRegion *)&u26,
37 |         L"SendInput",
38 |         0i64);
39 |     EnterCrit(0i64, 1i64);
40 |     Win32ThreadLock = 0i64;
41 |     u24 = 0i64;
42 |     u25 = 0i64;
43 |     if ( cbSize_1 != 40 )
44 |     {
45 |         if...
46 |         goto _leave_error;
47 |     }
48 |     if ( !( _DWORD )cInputs_1 )
49 |     {
50 |         if...
51 |     leave_error:
52 |         nbEventsInserted = 0;
53 |         UserSetLastError(ERROR_INVALID_PARAMETER);
54 |         goto __leave;
55 |     }
56 |     AlignmentWow64 = PsGetCurrentProcessWow64Process(); // If current process a 32-bit application?
57 |     ProbeForRead(pInputs_1, 40 * cInputs_1, AlignmentWow64 != 0 ? 1 : 4); // Check the input structure.
58 | // Allocate kernel memory to host the user-mode buffer.
59 |     knl_pInputs = (struct tagINPUT *)Win32AllocPoolWithQuota(40 * cInputs_1, 'issU');
60 |     knl_pInputs_1 = knl_pInputs;
61 |     knl_pInputs_2 = knl_pInputs;
62 |     if ( !knl_pInputs )
63 |         ExRaiseStatus(STATUS_NO_MEMORY);
64 |     memmove(knl_pInputs, pInputs_1, 40 * cInputs_1); // Copy from user-mode buffer to a kernel-mode one.
65 |     PushW32ThreadLock((__int64)knl_pInputs_1, &Win32ThreadLock, (__int64)Win32FreePool);
66 |     nbEventsInserted = xxxSendInput(cInputs_1, knl_pInputs_1); // Send inputs to the keyboard handler.
67 |     Win32kThread = W32GetThreadWin32Thread(*MK_FP(__GS__, 0x188i64));
68 |     *( _QWORD *) (Win32kThread + 16) = Win32ThreadLock;
69 |     if ( knl_pInputs_2 ) // Release memory.
70 |         Win32FreePool(knl_pInputs_2);
71 |     if...
72 |     leave:
73 |     UserSessionSwitchLeaveCrit();
74 |     InputTraceLogging::ThreadLockedPerfRegion::~ThreadLockedPerfRegion((InputTraceLogging::ThreadLockedPerfRegion *)&u26);
75 |     return nbEventsInserted;
76 | }

```

Figure 4.98: Pseudo-code of the `NtUserSendInput` routine in `win32kfull.sys`.

The `xxxSendInput` routine acts on keyboard and mouse. For mouse, it uses `xxxMouseEventDirect`, `xxxSynchronizeDWMWindowChanges` and `xxxWaitForDITMouseInjectionFlush`. If the first routine obviously deals with the mouse's events stack, the last two routines are linked to *Desktop Window Manager* (aka *DWM*) technology [947]. Introduced in Windows Vista, it fundamentally changed the way applications display pixels on the screen. This technology is housed by `dwm.exe` Windows service which is responsible to centralize all drawing events

and to enable visual effects. Technically, when desktop composition is enabled, individual windows no longer draw directly to the screen or primary display device as they did in previous versions of Windows. Instead, their actions are redirected to DWM which manages off-screen surfaces in video memory in order to be grouped and displayed in a more efficient way on the display. This is that technology which is behind the miniature windows we have when moving the cursor on icons in the explorer bar. The DWM is involved a lot in win32k.

Interaction with the keyboard is performed with `xxxInternalKeyEventDirect` routine. That one checks first if `grpdeskRitInput` is coming from `csrss.exe` process or if `RtlAreAllAccessesGranted` routine (which is used to check to if all desired accesses are granted, in our case access write is checked) returns true. In this case, `IsGpqForegroundAccessibleCurrent` routine is called to check if `gpqForeground` exists or at least the input to send matters for a thread. If it does not matter, operation is stopped at that point (since no thread would take care about it). If the operation continues, `dwFlags` member from the `INPUT` structure [941] provided is checked to see if the input to insert is by scan code (flag `KEYEVENTF_SCANCODE`<sup>46</sup>) or by virtual key code. If it is by scan code value, there is a call to `VKFromVSC` routine to convert it from scan code to virtual key code.

The translation from the scan code to the virtual key value in `VKFromVSC` routine is driven thanks to the *keyboard table layout* [948]. This table is retrieved from the current foreground thread (represented by `gpti-Foreground`) or from the global value `gpKbdTbl`. The rest of the operations represent linear transformations and readings in the keyboard layout tables. It is possible to add specific flags in the virtual key code thanks to `GetModifierBits` called with `Modifiers_VK_STANDARD` and `gafRawKeyState` objects (for instance key pressed, ALT, and so on).

When a virtual key code is provided, there are transforms depending on the virtual key provided. For instance, `VK_SHIFT`, `VK_CONTROL`, or `VK_MENU` are transformed to be seen either as a left or right shift, control or menu key. If the message sent uses the `dwFlags` set with `KEYEVENTF_KEYUP`, the most significant bit of the virtual key code is set to one. In addition, there are some transforms when the virtual key is coming from the Numeric Pad. Actually, the goal is to prepare the call to `xxxProcessKeyEvent` routine from `win32kbase.sys` with an undocumented `KEYBOARD_VIRTUAL_DEVICE_INFO` structure. This routines calls soon `UpdateRawKeyState` which updates the last entry of `gafRawKeyState` object with the virtual key code provided before calling `ApiSetEditionUpdateModifiersForHotkey` which updates `gfsRawModifiersForHotKey` global value in `win32kfull.sys`. Once `UpdateRawKeyState` routine has been called by `xxxProcessKeyEvent` routine, this one calls `CInputGlobals::UpdateInputGlobals` (from `win32kbase.sys`). This routine is part of the raw input thread activity. If no time-stamp has been provided in initial `INPUT` structure, the time-stamp is generated automatically from `CInputGlobals::GetLastInputTime` routine.

Once it has been performed, there is a call to `ApiSetEditionHandleSonarKeyEvent` routine. This one is trying to interface with `EditionHandleSonarKeyEvent` routine from `win32kfull.sys`. This routine aims to call `zzzStartSonar` if and only if the virtual key is `VK_CONTROL` and the global value `gbLastVkForSonar` is equal to `VK_CONTROL` too. Actually, this operation corresponds to the Mouse Sonar accessibility feature [949] (Figure 4.99). This functionality briefly shows several concentric circles around the mouse pointer when the user presses and releases the CTRL key. It enables a user to locate the mouse pointer on a screen that is cluttered or with resolution set to high, on a poor quality monitor, or for users with impaired vision.

```

27 |     else
28 |     {
29 |         if ( (_BYTE)vk == VK_CONTROL && gbLastVkForSonar == VK_CONTROL && *(_DWORD *)gpdwCPUserPreferencesMask & 0x4000 )
30 |             zzzStartSonar(a1, vk, a3);
31 |         result = (_int64)gpdwCPUserPreferencesMask;
32 |         if ( *(_DWORD *)gpdwCPUserPreferencesMask & 0x4000 )
33 |         {
34 |             if ( gbLastVkForSonar )
35 |                 gbLastVkForSonar = 0;
36 |         }
37 |     }

```

Figure 4.99: Extract from the pseudo-code of the `EditionHandleSonarKeyEvent` routine in `win32kfull.sys`.

The sonar operation is confirmed by the pseudo code of `zzzStartSonar` routine. That one initializes a specific

<sup>46</sup>When this one is set, `wScan` member in `INPUT` identifies the key and `wVk` member is ignored.

structure to define a map where to draw on the screen. We can see that the initial radius for the sonar is equal to 100 pixels. Using `CreateFadelInternal` routine, it creates a sprite around the current mouse cursor to draw the different circles surrounding the cursor's icon. The initial sonar is drawing thanks to `DrawSonar` while the animation is given by `zzzStartFade` and `zzzAnimateFade`. We note the use of ellipse (`NtGdiEllipse`) and pens (`GreSelectPen`) or brushes (`GreCreateSolidBrush` and `GreSelectBrush`) to draw concentric circles on the screen (Figure 4.100).

```

1|HDC __fastcall zzzStartSonar(__int64 a1, __int64 a2, __int64 a3)
2|{
3|    __int64 v3; // rcx@1
4|    unsigned __int32 v4; // eax@3
5|    HDC hDc; // rax@3
6|    __int64 v6; // rdx@4
7|    __int64 v7; // rcx@4
8|    __int64 v8; // r8@4
9|    __int64 v9; // r9@4
10|    char v10; // [rsp+38h] [rbp-28h]@4
11|    struct tagRECT WhereToDraw; // [rsp+38h] [rbp-20h]@3
12|
13|    v3 = *((_QWORD *)gpsi);
14|    gptSonarCenter = *((_QWORD *)gpsi + 4960164);
15|    if ( *((_DWORD *)gFade + 12) )
16|        StopFade(v3, a2, a3);
17|    giSonarRadius = 100;
18|    WhereToDraw.left = gptSonarCenter - 100;
19|    WhereToDraw.right = gptSonarCenter + 100;
20|    WhereToDraw.top = HIDWORD(gptSonarCenter) - 100;
21|    WhereToDraw.bottom = HIDWORD(gptSonarCenter) + 100;
22|    v4 = W32GetCurrentThreadDpiAwarenessContext(HIDWORD(gptSonarCenter), a2, a3);
23|    hDc = CreateFadelInternal(0164, &WhereToDraw, 1000, 192, v4);
24|    if ( hDc )
25|    {
26|        DrawSonar(hDc);
27|        UserAtomicCheck::UserAtomicCheck((UserAtomicCheck *)v10);
28|        zzzStartFade(v7, v6, v8, v9);
29|        zzzAnimateFade();
30|        UserAtomicCheck::~UserAtomicCheck((UserAtomicCheck *)v10);
31|        hDc = (HDC)1;
32|    }
33|    return hDc;
34|}

```

```

1|void __fastcall DrawSonar(HDC hDc)
2|{
3|    HDC hDc_1; // rdi@1
4|    HBRUSH MagentaBrsh; // rax@1
5|    HBRUSH MagentaBrsh_1; // rsi@1
6|    __int64 Pen; // rax@2
7|    __int64 Pen_1; // rbp@2
8|    __int64 Prev_Pen; // r15@3
9|    __int64 GreyBrsh; // rax@3
10|    __int64 GreyBrsh_1; // r14@3
11|    __int64 Prev_Brsh; // rbx@4
12|
13|    hDc_1 = hDc;
14|    MagentaBrsh = (HBRUSH)GreCreateSolidBrush(0xFF00FF164);
15|    MagentaBrsh_1 = MagentaBrsh;
16|    if ( MagentaBrsh )
17|    {
18|        FillRect(hDc_1, &rc, MagentaBrsh);
19|        Pen = GreCreatePen(0164, 0164, 0xFFFFF164, 0164);
20|        Pen_1 = Pen;
21|        if ( Pen )
22|        {
23|            Prev_Pen = GreSelectPen(hDc_1, Pen);
24|            GreyBrsh = GreCreateSolidBrush(0x808080164);
25|            GreyBrsh_1 = GreyBrsh;
26|            if ( GreyBrsh )
27|            {
28|                Prev_Brsh = GreSelectBrush(hDc_1, GreyBrsh);
29|                NtGdiEllipse(hDc_1);
30|                GreSelectBrush(hDc_1, MagentaBrsh_1);
31|                NtGdiEllipse(hDc_1);
32|                GreSelectBrush(hDc_1, Prev_Brsh);
33|                GreDeleteObject(GreyBrsh_1);
34|            }
35|            GreSelectPen(hDc_1, Prev_Pen);
36|            GreDeleteObject(Pen_1);
37|        }
38|        GreDeleteObject(MagentaBrsh_1);
39|    }
40|}

```

Figure 4.100: Pseudo-codes of `zzzStartSonar` and `DrawSonar` routines in `win32kfull.sys`.

More than the curiosity of knowing how the sonar mechanism is drawing on the screen, it is the observation of the management of the inserted input events that is relevant. Indeed, by simulating the press of control key with `SendInput` function when the mouse sonar is activated, it invokes the mouse sonar procedure the same way the real press of the control key would have done. This is one of the main difference than using regular `SendMessage` function [935]. This one cannot reproduce consequences from the system configuration dealing with specific keys.

Back to the `xxxProcessKeyEvent` routine, after `ApiSetEditionHandleSonarKeyEvent` call, there are multiple calls to specific routines able to deal with the specificities of some keyboards. These routines are `KEOEMProcs`, `xxxKELocaleProcs`, and `xxxKENLSProcs`. The second in the list is just about performing specific checks on global values `gdwKeyboardAttributes` and `gpKbdTbl` and optionally calling `xxxAltGr` or `xxxShiftLock` routines. These routines are used to manage `AltGr` and `ShiftLock` or `CapsLock` to have keyboard layout-specific behavior. Both usually resulting in the use of `xxxKeyEventEx` routine to notify the use of such keys and actions resulting from their use. Routines `KEOEMProcs` and `xxxKENLSProcs` are more about to execute a list of pointer of routines. The first is dealing with `aKEProcOEM` list holding `xxxICO_00` and `xxxNumpadCursor` routines (`win32kbase.sys`).

The `xxxNumpadCursor` routine manages the *numeric keypad* keys (whose virtual key codes are stored in the `ausNumPadCvt` table) when this one is used as directional arrows (with the `xxxKeyEventEx` routine to simulate them). The translation from numeric keypad keystrokes to specific events (including directional arrows) when `NUM LOCK` key is set is performed thanks to `ausNumPadCvt` table. This one is provided in Figure 4.101 as a set of couple of bytes, representing each a translation with virtual key code. For each couple of byte, the first byte corresponds to the virtual key code numeric key pad value and the second byte the equivalent virtual key code value when `NUM LOCK` is set [516]. For instance, `0x62` corresponds to `VK_NUMPAD2` where a down arrow is usually printed too. And if we check the corresponding value in the table in Figure 4.101, we have `0x28` which corresponds to `VK_DOWN`, the down arrow key. Of course, this procedure to convert keystrokes

coming from numeric keypad is reused for regular keystrokes with the keyboard device.



Figure 4.101: Illustration about the translation performed in xxxNumpadCursor routine with ausNumPadCvt table from win32kbase.sys.

Coming back to xxxProcessKeyEvent routine analysis, we have xxxKENLSProcs routine that is dealing with gpKbdNlsTbl and a set of three callbacks recorded in aNLSVKFProc list. The first is StubDispFillPath which does nothing. The second is KbdNlsFuncTypeNormal which calls GenerateNlsVkKey routine which uses one of the fifteen routines registered by aNLSKEProc (Figure 4.102). Some of these routines do nothing or just return zero or one, others are doing more stuff specific to some NLS context call.

```

000001c01d4e20 dq offset ?StubDispFillPath@GVAHPEAU_SURF0BJ_0PEAU_PATH0BJ_0PEAU_CLIP0BJ_0PEAU_BRUSH0BJ_0PEAU_POINTL@KKGZ ; StubDispFillPath(_SURF0BJ *,_PATH0BJ *,_CLIP0BJ *,_BRUSH0BJ *,_POINTL *,ulong,ulong)
000001c01d4e28 dq offset ext_ns_win_moderncore_win32k_base_ntuser_11_4_0_HasHidTable
000001c01d4e38 dq offset ?StubDispFillPath@GVAHPEAU_SURF0BJ_0PEAU_PATH0BJ_0PEAU_CLIP0BJ_0PEAU_BRUSH0BJ_0PEAU_POINTL@KKGZ ; StubDispFillPath(_SURF0BJ *,_PATH0BJ *,_CLIP0BJ *,_BRUSH0BJ *,_POINTL *,ulong,ulong)
000001c01d4e40 dq offset ?NLSSendParam@GVAHPEAUtagKE0_KKGZ ; NLSSendParam(tagKE *,unsigned __int64,ulong)
000001c01d4e48 dq offset ?NLSKanaHidToggleProc@GVAHPEAUtagKE0_KKGZ ; NLSKanaHidToggleProc(tagKE *,unsigned __int64,ulong)
000001c01d4e50 dq offset ?NLSAlphanumericHidProc@GVAHPEAUtagKE0_KKGZ ; NLSAlphanumericHidProc(tagKE *,unsigned __int64,ulong)
000001c01d4e58 dq offset ?NLSKatakanaHidProc@GVAHPEAUtagKE0_KKGZ ; NLSKatakanaHidProc(tagKE *,unsigned __int64,ulong)
000001c01d4e60 dq offset ?NLSShcsDbcsToggleProc@GVAHPEAUtagKE0_KKGZ ; NLSShcsDbcsToggleProc(tagKE *,unsigned __int64,ulong)
000001c01d4e68 dq offset ?NLSRomanToggleProc@GVAHPEAUtagKE0_KKGZ ; NLSRomanToggleProc(tagKE *,unsigned __int64,ulong)
000001c01d4e70 dq offset ?NLSCodeInputToggleProc@GVAHPEAUtagKE0_KKGZ ; NLSCodeInputToggleProc(tagKE *,unsigned __int64,ulong)
000001c01d4e78 dq offset ?NLSHeIp0rEndProc@GVAHPEAUtagKE0_KKGZ ; NLSHeIp0rEndProc(tagKE *,unsigned __int64,ulong)
000001c01d4e80 dq offset ?NLSShmedrClearProc@GVAHPEAUtagKE0_KKGZ ; NLSShmedrClearProc(tagKE *,unsigned __int64,ulong)
000001c01d4e88 dq offset ?NLSKanaHidProc@GVAHPEAUtagKE0_KKGZ ; NLSKanaHidProc(tagKE *,unsigned __int64,ulong)
000001c01d4e90 dq offset ?NLSKanaEventProc@GVAHPEAUtagKE0_KKGZ ; NLSKanaEventProc(tagKE *,unsigned __int64,ulong)
000001c01d4e98 dq offset ?NLSConv0rNonConvProc@GVAHPEAUtagKE0_KKGZ ; NLSConv0rNonConvProc(tagKE *,unsigned __int64,ulong)
    
```

Figure 4.102: List of routines held in aNLSVKFProc value from win32kbase.sys.

The third routine from aNLSVKFProc is KbdNlsFuncTypeAlt. That one uses GenerateNlsVkAltKey and GenerateNlsVkKey. Both routines are using the list of routines from aNLSKEProc (Figure 4.102). Technically, this is where specificities of NLS are handling depending on what has been configured in the system options and which actions are engaged by the user.



Finally, in `xxxProcessEvent` routine, there is a final call to `xxxKeyEventEx` which is may be one of the most important one. This is that one which would set the event for the rest of the system. This one is based on `GetKeyEventInputSource` to retrieve the current foreground thread, `CKeyboardProcessor::HandleLeftRightVKs`, `ApiSetEditionKeyEventLLHook` to deal with low level hook in the system and `xxxUpdateGlobalsAndSendKeyEvent` or `CKeyboardProcessor::ForwardInputToKeyboardOverrider` to forward the event to the rest of the system. The first is a big routine using `UpdateAsyncKeyState` to update the list of the last keystrokes, dealing with the numeric pad through `LowLevelHexNumpad`, adjusting LEDs on keyboard device with `UpdateKeyLights` (via `CKeyboardSensor::UpdateKeyboardLEDs`), executing specific actions linked to hotkeys with `ApiSetEditionDoHotKeys`, interacting with the raw input thread in `ApiSetEditionHandleRawInput` and sending the message to the rest of the system by `ApiSetEditionHandleAndPostKeyEvent` which uses `InputExtensibilityCallout::CoreMsgSendMessage` routine. The second is much more simple since `CKeyboardProcessor::ForwardInputToKeyboardOverrider` crafts a message with `CKeyboardProcessor::CreateKeyboardInputMessage` and send it with `InputExtensibilityCallout::CoreMsgSendMessage`.

In conclusion, it is easy to understand the advantage of using a function like `SendInput` rather than `SendMessage` to simulate keystrokes. Not only is the virtual simulation of a keystroke (or mouse action) is easier to perform, but it also fits perfectly within the framework of input management. It also makes it possible to trigger a whole bunch of actions that are specific to keyboard shortcuts or hardware interactions. The latter would not be possible otherwise by using `SendMessage` function. It is also a way for us to show the chain of actions that is setup during the management of inputs. Indeed, the simulation of inputs is a place that concentrates them without undesirable confusion coming from the complexity of the raw input thread architecture.



### 5.2.7 From scan codes to virtual key codes

#### Key Point 4.45:

- ☞ How is the translation from scan code to virtual key code and character done?
  - ☞ The translation can be done in either direction.
  - ☞ At application level, there are three main functions: `MapVirtualKey(Ex)`, `VkKeyScan(Ex)`, and `ToUnicode(Ex)/ToAsciiEx`.
  - ☞ In practice, most of the work is performed thanks to `InternalMapVirtualKeyEx` routine in `win32kbase.sys`.
- ☞ The translation process is complex, involving keyboard layouts and a whole series of translation tables.
  - ☞ We explain most of them as best as we can understand them.
- ☞ Conversion from virtual key code to character can produce both ASCII (`ToAsciiEx`) or Unicode (`ToUnicodeEx`) characters.
- ☞ It is possible to add some modifications in the translation of scan codes to virtual key codes.
  - ☞ Starting from Windows 2000, *Scan Code Mapper* is a special entry in registry used to define custom conversions.
  - ☞ Used to easily fix or correct firmware errors but also to disabling or exchanging the behavior of some keys.
  - ☞ Applied to the whole system and it requires to reboot the machine to be taken into account.
  - ☞ But it cannot be used to prevent features from Windows (such as screenshot keystroke) — API is still usable.
- ☞ Technically, it is the responsibility of the applications to perform the conversions they need if they are not using automatic translations provided by regular messages channel.
- ☞ As a reminder, without an explicit call, the translation is automatically performed by the *Raw Input Thread* in the input messages delivered to an application.

Because each keyboard can have different scan codes, there is a translation between scan codes and virtual key codes. This translation is performed in the context of the raw input thread when that one is about to broadcast `WM_KEYDOWN` messages and more directly as explained in section 5.1.4.7. The translation is not always the same from one thread to another, from one window to another and so on. This is due to the fact that the virtual key codes and scan mapping is specified in the DLL specific to the hardware keyboard and it is active when the keyboard is in focus. It also changes when the language, or localization of the operating system, or keyboard layout change [950].

Since the translation has already been explained from kernel-mode point of view (section 5.1.4.7), we can try to explain how this translation is performed from user-mode point of view. For an application, there are three possibilities to perform translation. Using `MapVirtualKey(Ex)` [906], `VkKeyScan(Ex)` [951], or `ToUnicode(Ex)` [952] functions. The first is used to maps a virtual key code into a scan code or character value, or it translates a scan code into a virtual-key code based on an input locale identifier (a keyboard layout). The second translates a character to the corresponding virtual-key code and shift state (although based on an input locale identifier). The third translates the specified virtual-key code and keyboard state to the corresponding Unicode character or characters if several are generated by the virtual key code. The different phases of translation are given in the Figure 4.103.

Internally, these functions do not use the same interface to proceed. Starting with `MapVirtualKeyEx` functions, this one is exported by `user32.dll` but it is a direct call to `ZwUserMapVirtualKeyEx` in `win32u.dll`. `ZwUserMapVir-`

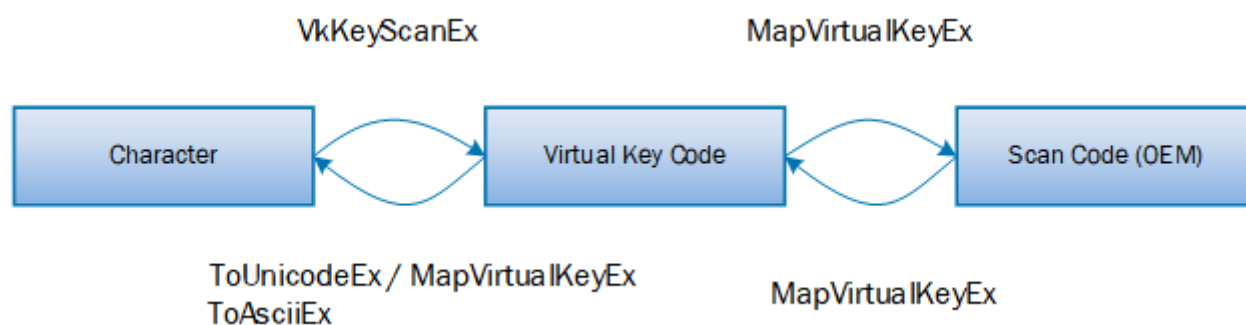


Figure 4.103: Different phases of translation with the corresponding API to go from one phase to another.

`MapVirtualKeyEx` function from `win32u.dll` performs a syscall to call `NtUserMapVirtualKeyEx` from `win32kbase.sys`. This routine can act only if it is not in the context of `csrss` or `dwm` services. If the caller to this routine is a user-mode application, that one will convert the handle to a pointer with `HKLtoPKL` routine from the `Win32thread` undocumented structure. We can bet that "KL" in `HKLtoPKL` stands for *keyboard layout* object. Then, it is going to call `InternalMapVirtualKeyEx` with the keyboard layout and parameters given during the call to `MapVirtualKeyEx` function (Figure 4.104). The last parameter provided is an undocumented sub-member from the keyboard layout internal structure retrieved previously. The first member (offset `+0x30`) corresponds to a *keyboard layout file* which is used internally to store the list of keyboard layouts, including the current one. Note that this structure holds the DLL name from where this keyboard data has been loaded (offset `+0x38` — "KBDFR.DLL" for French keyboard in Figure 4.105). The second access (offset `+0x20`) in the keyboard layout file is to retrieve a pointer to the keyboard layout data.

```

87 | Win32Thread_3 = (_QWORD *)W32GetThreadWin32Thread((PKTHREAD)*MK_FP(_GS_, 0x188i64));
88 | if ( IsUserModeCaller )
89 |     Pk1 = HKLtoPKL(Win32Thread_3, dwHkl_1);
90 | else
91 |     Pk1 = Win32Thread_3[0x36];
92 | if ( Pk1 )
93 |     status = InternalMapVirtualKeyEx(uCode_1, uMapType_1, *(PVOID *)*( _QWORD *) (Pk1 + 0x30) + 0x20i64);
94 | UserSessionSwitchLeaveCrit();
95 | return status;

```

Figure 4.104: End of the pseudo-code of `win32kbaseNtUserMapVirtualKeyEx` routine from `win32kbase.sys`.

```

1: kd> db ffffffff05`006429e0
ffffffffff05`006429e0  45 00 01 00 00 00 00 00-04 00 00 00 00 00 00 00  E.....
ffffffffff05`006429f0  00 00 00 00 00 00 00 00-00 d0 63 00 05 ff ff ff  .....C.....
ffffffffff05`00642a00  00 d0 63 00 05 ff ff ff-98 10 00 00 00 00 00 00  ..C.....
ffffffffff05`00642a10  00 00 00 00 00 00 00 00-4b 00 42 00 44 00 46 00  .....K.B.D.F.
ffffffffff05`00642a20  52 00 2e 00 44 00 4c 00-4c 00 00 00 00 00 00  R...D.L.L.....
ffffffffff05`00642a30  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00  .....
ffffffffff05`00642a40  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00  .....
ffffffffff05`00642a50  00 00 00 00 00 00 00 00-01 00 00 00 00 00 00  .....

```

Figure 4.105: Content of the *keyboard layout file* internal structure retrieve for the call to `InternalMapVirtualKeyEx` routine.

The last structure provided to `InternalMapVirtualKeyEx` routine is quite interesting despite being totally undocumented. Dumped with Windbg debugger, it is possible to retrieve internal information as given in Figure 4.107 about the current keyboard layer on the system (a french keyboard in our case). This structure (framed in black on the left of the Figure 4.107) is composed of a set of several pointers. Each of these pointers references a table for different purposes. The first table (circled 1 on top left of Figure 4.107) holds a list of Unicode strings describing different functionalities provided by the current keyboard. Without taking care of

regular alphanumeric keys, we have the list of specific keys (control, numeric pad, windows key with left and right description and so on). Note that the text of keys corresponds to the current keyboard layout language which is french in our case.

The second table is given on top of Figure 4.107 green circled. That one holds different sub-lists given in the purple rectangle at right. The first three sub-lists seem to represent internal conversion tables. These tables could be used to translate scan codes or virtual key codes. The next sub-list describes the layout of the keys on the current keyboard. Indeed, working with an "AZERTY" keyboard (briefly the french layout), we have the Unicode representation of each single alphabet-key lower and upper case. The last sub-table is used to translate virtual key code from the numeric pad into displayable characters (technically it is equivalent to substrate 0x30 to entry values since "0 key" is equal to 0x30 in the Microsoft's virtual key code list [516]).

The third member of the keyboard layout tables is a list of codes which is probably used to translate from one code to another on a specific part of the keyboard. The fourth table referenced is a translation from scan codes to Unicode string describing the scan code associated. The first 8 bytes values give the virtual scan code<sup>47</sup> while the following 8 bytes represent an address given the Unicode string content. We can perfectly check that we are dealing with scan codes by recording the scan codes sent for each keystroke. The values provided from the keyboard corresponds to the key pressed (and the unicode string describing the key). Using the numeric pad, we have the following scan codes linked to the virtual key codes given in table 4.16. Note that the numeric pad code values are not given in alphanumeric order but with the logic in which the keys are arranged on a keyboard (in a square of three keys — Figure 4.106). It is the same logic which is used to order the strings in the fourth table.

```

BYTE aVkNumpad[] = {
    VK_NUMPAD7,  VK_NUMPAD8,  VK_NUMPAD9,  0xFF, // 0x47 0x48 0x49 (0x4A)
    VK_NUMPAD4,  VK_NUMPAD5,  VK_NUMPAD6,  0xFF, // 0x4B 0x4C 0x4D (0x4E)
    VK_NUMPAD1,  VK_NUMPAD2,  VK_NUMPAD3,           // 0x4F 0x50 0x51
    VK_NUMPAD0,  VK_DECIMAL,  0,           // 0x50 0x51
};

```

Figure 4.106: Representation of the scan code content for the numeric pad.

Virtual Key Code	Scan code	Description
0x60	0x52	VK_NUMPAD0
0x61	0x4f	VK_NUMPAD1
0x62	0x50	VK_NUMPAD2
0x63	0x51	VK_NUMPAD3
0x64	0x4b	VK_NUMPAD4
0x65	0x4c	VK_NUMPAD5
0x66	0x4d	VK_NUMPAD6
0x67	0x47	VK_NUMPAD7
0x68	0x48	VK_NUMPAD8
0x69	0x49	VK_NUMPAD9
0x6e	0x53	VK_DECIMAL
0x0d	0x1c	VK_RETURN
0x6b	0x4e	VK_ADD
0x6d	0x4a	VK_SUBTRACT
0x6a	0x37	VK_MULTIPLY
0x6f	0x35	VK_DIVIDE

Table 4.16: Numeric pad codes translations.

<sup>47</sup>In practice, not all the 8 bytes are used since scan codes can be encoded on a single byte. But for memory alignment reasons, a bunch of 8 bytes are used on a 64-bit CPU architecture.

The fifth table is globally the same than the fourth one. This one translates other specific keys from a french keyboard layout. The sixth table is an extension to provide additional strings to describe specific french accents. These accents are specific to Latin languages and not present in English language. The seventh table represents the virtual key code arrangement of the different keys used by the keyboard. This one is useful to get the virtual key code of a pressed key since we know the position of each key on the keyboard, allowing an association between the scan code for a key at a given position and the corresponding virtual key code. The eighth member of the structure is a constant (0x7f in our case). The ninth table is still a translation table. And the tenth table is finally a list of Unicode strings describing specific keys (not alphanumeric ones) on a keyboard in the french language. We might bet that all these different tables represent different context of translation (from scan codes to virtual key codes for instance or to give a human readable description of a keyboard layout for the system) but also that such an architecture using many tables is used to cover backward compatibility where old keyboards did not have the same number of keys<sup>48</sup> that we have today. It could include specific scan codes used as shortcuts to trigger a special functionality which is now available in one key press on modern keyboard [953].

Routine `InternalMapVirtualKeyEx` acts according to its second parameter, that is to say the translation direction. According to Figure 4.103, `MapVirtualKeyEx` function is able to translate from scan code to virtual key code and from virtual key code to displayable character. The different possibilities of conversion are given in table 4.17.

Value	Symbol	Description
0	MAPVK_VK_TO_VSC	From virtual key code to scan code.
1	MAPVK_VSC_TO_VK	From scan code to virtual key code.
2	MAPVK_VK_TO_CHAR	From virtual key code to character.
3	MAPVK_VSC_TO_VK_EX	Extended version of MAPVK_VSC_TO_VK.
4	MAPVK_VK_TO_VSC_EX	Extended version of MAPVK_VK_TO_VSC.

Table 4.17: Value used by `MapVirtualKeyEx` function to perform translation.

The first operation performed by `InternalMapVirtualKeyEx` routine is to check for which translation type it has been called. In the case where the translation would be from a virtual key code to a scan code (0 or 4), the routine checks first if the code provided (`uCode`) is between or equal `VK_SHIFT` and `VK_MENU` values. In this case, `uCode` is transformed such as  $uCode = 2 * uCode + 0x80$  to convert ambiguous shift, control and alt keys to left-hand keys (documented in [906] to be the default behavior when it is not possible to know if the key is coming from the left or the right on the keyboard). Then `InternalMapVirtualKeyEx` routine retrieves the member at offset +0x38 in the keyboard layout table provided by `NtUserMapVirtualKeyEx` routine. This one corresponds to 0x7f constant value in our case. Then the routine uses the pointer stored as the seventh member in the keyboard layout tables structure. This pointer corresponds to the seventh table in Figure 4.107. This table is read word per word<sup>49</sup> until there is a match between the code provided as routine's parameter and a code stored in the table (where the content size of the table is the 0x7f value previously read), the routine loops. When a match occurs, the routine returns the index value where the match occurs. This is the way the translation from virtual key code to scan code works (Figure 4.108). In the case of translation with `MAPVK_VK_TO_VSC_EX` flag, the flag 0xE000 is added to the returned result.

<sup>48</sup>For instance, on PC/XT keyboard layout, many of the keys we are familiar with today simply did not exist back then, and one key (PrtSc) existed in a very different form [953]. Windows operating system must ensure the support for keys beyond the basic 84-key PC/XT keyboard.

<sup>49</sup>Two bytes per two bytes. More information about Windows data type can be found in [954].



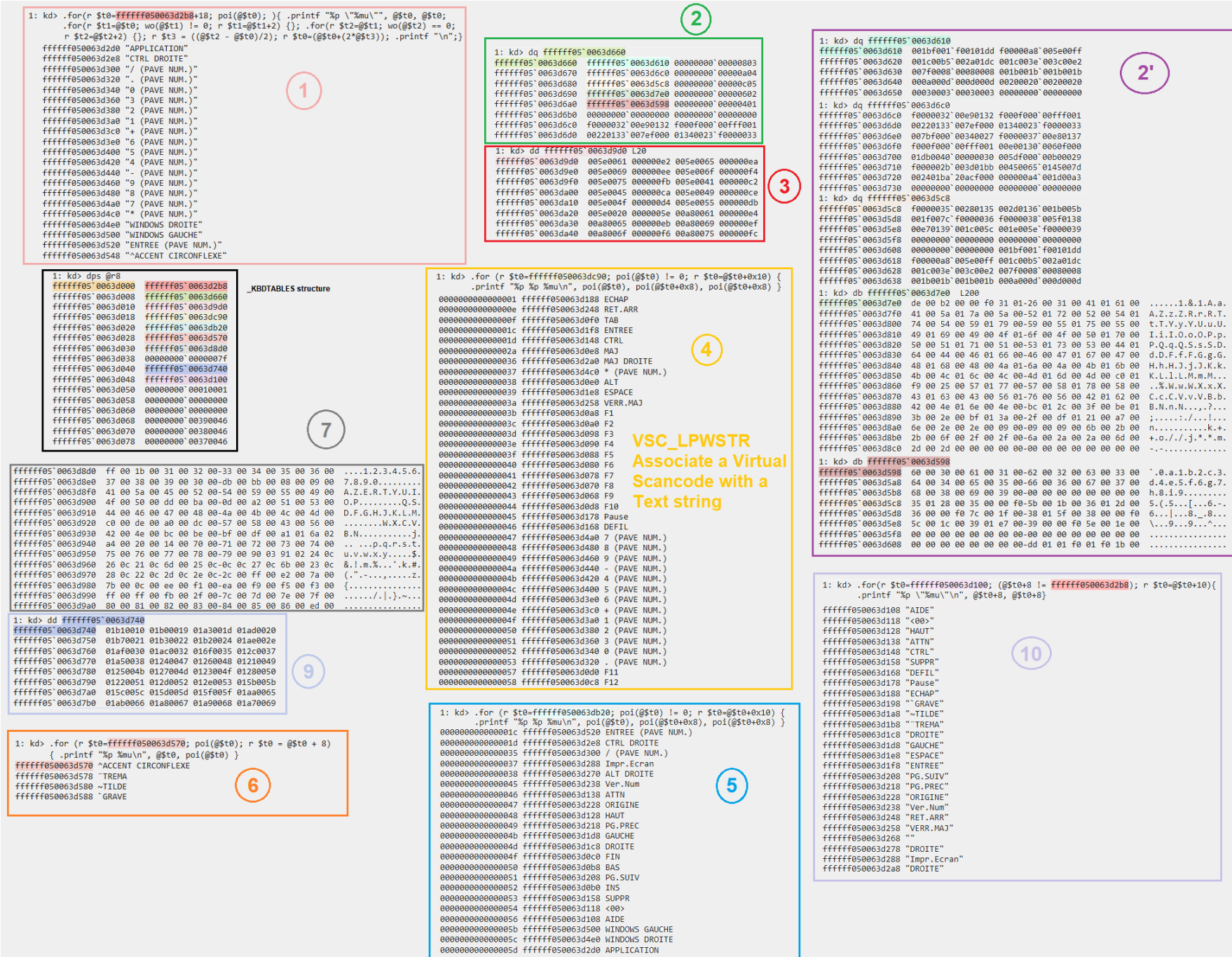


Figure 4.107: Dump of the tables layout structure provided to InternalMapVirtualKeyEx routine.

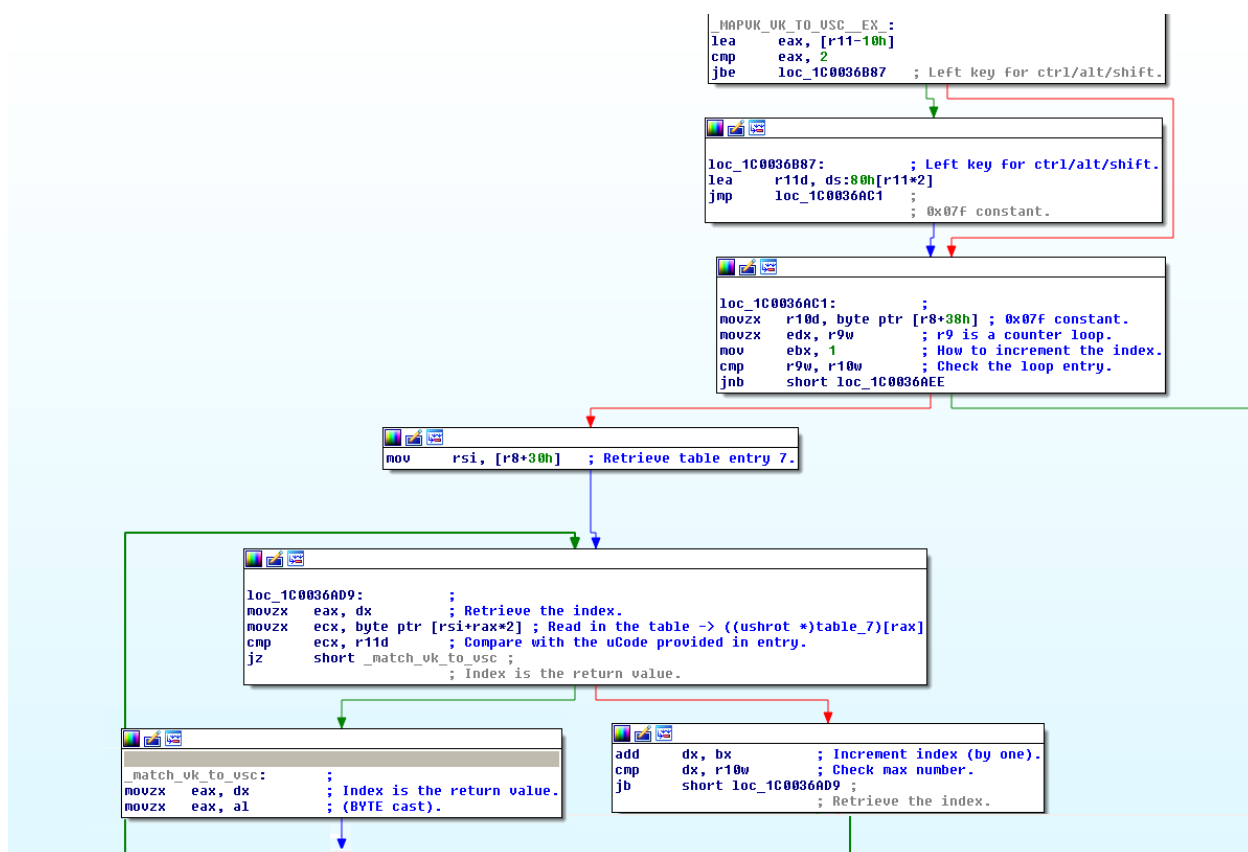


Figure 4.108: Assembly code from a portion of `InternalMapVirtualKeyEx` routine in the case where the translation is performed with `MAPVK_VK_TO_VSC` flag.

In the case where the code would not be found in the seventh table, only in the case of where the flag `MAPVK_VK_TO_VSC_EX` would be set, the eighth table would be used. The search procedure is identical in this case with the one used in the seventh table. If there is a match in the eighth table, `0xE100` is added to the return index value. If the search does not match in both the seventh and the eighth table, in such a case, the search keeps going in the list of scan codes registered for the numeric pad on keyboard. This table is universal and directly stored in a global value `avkNumpad`. The search procedure (Figure 4.109) is similar to the one used with the seventh and the eighth tables. The difference lies in the returned value. Offset is computed from the matched code to the base address of `avkNumpad` and `0x47` constant is added. It makes sense since the first entry in the numeric pad is `VK_NUMPAD7` and its scan code value is `0x47`. If there is no match at all in no table, the return value is zero.

We are now going to consider a conversion from virtual key to character. There are technically three possibilities to proceed. With dedicated functions such as `ToUnicodeEx` [952], `ToAsciiEx` [955] (which is the ascii version of the previous function) or with `MapVirtualKeyEx` [906] functions. Continuing our description with `MapVirtualKeyEx` function. That one still calls `InternalMapVirtualKeyEx` routine in kernel mode. Converting from virtual key code to displayable character is possible by the use of `MAPVK_VK_TO_CHAR` flag. The first operation in this case is to check if the provided virtual key code (in `uCode` value) is between 'A' value and 'Z' value. In such a case, the `uCode` value is directly returned since there is a direct map between ASCII alphabet and virtual key codes representing uppercase alphabet.

If the provided code is not an upper case alphabet character, the second table in the keyboard layout tables is used (if this table is not present, error `ERROR_INVALID_PARAMETER` is given to the calling application). According to Figure 4.107, that one contains a list of sub-tables. Each value in the sub-table is tested with the provided virtual key code until there is no more values to test in the sub-table (the last value in the sub-table is zero). Test is performed as a byte check. If the byte is defined in the sub-table, that one is cast to

```

142     CurrentVkNumPad = ::aUkNumpad[0];
143     aUkNumpad = ::aUkNumpad;
144     if ( ::aUkNumpad[0] )
145     {
146         while ( CurrentVkNumPad != uCode_1 )
147         {
148             CurrentVkNumPad = ++aUkNumpad;
149             if ( !*aUkNumpad )
150                 return 0;
151         }
152         return (_DWORD)aUkNumpad - (unsigned __int64)::aUkNumpad + 0x47;
153     }
154     return 0;

```

Figure 4.109: Extract of the pseudo-code from `InternalMapVirtualKeyEx` routine called with `MAPVK_VK_TO_VSC` flag as a final search in the numeric pad.

fit a 32-bit value in order to be compared to the virtual code provided in `uCode`. If there is no match, the next byte is retried by adding `0x09` to the current base address of the sub-table and looping with the current byte to extract in the sub-table before comparison check. When a table has been fully tested, another one is tested until there is not more table to test. It is hard to define the profile of all tables since there are layouts which are keyboard-dependent. This procedure is detailed in Figure 4.110.

Once a match has been found in one of the table, the equivalent character to display is read by extracting the value following the code matched in the sub-table. Once it has been done, the code extracted is checked with the `0xF001` value. In case of match, it means we are dealing with a dead character. That way, the next entry in the sub-table contains the expected value. The most significant bit of this value is set to one to indicates the returned value comes from a dead key (diacritic) [906]. Dead key occurs when dealing with some non-English keyboards that contain character keys that are not expected to produce characters by themselves [9]. Instead, they are used to add a diacritic to the character produced by the subsequent keystroke. These keys are called dead keys. The circumflex key on a French keyboard is an example of a dead key. To enter the character consisting of an "ô" ("o" with a circumflex), it is expected to type the circumflex key followed by the "o" key. When it happens, from an application point of view, the `TranslateMessage` function generates a `WM_DEADCHAR` message [956] when it processes the `WM_KEYDOWN` message from a dead key. In the message, it is possible to retrieve the character code of the diacritic for the dead key (`wParam` parameter). But it is generally ignored by an application. Instead, it processes the following `WM_CHAR` message generated by the subsequent keystroke which directly holds the character code of the letter with the diacritic.

After checking `0xF001` as a special value, `0xF000` is checked as another special value. In case of match, it means there is no value and the return value is zero. If the returned value does not match one of these special values, the translated one from a sub-table is returned.

Function `ToUnicodeEx` is based on a different stack of internal routines. That one uses the `NtUserToUnicodeEx` routine from `win32kbase.sys`. This routine performs checks on different parameters provided by the user-mode call before calling `xxxToUnicodeEx` routine. The latter performs various transformations on the data passed as an argument before calling `xxxInternalToUnicode`. This is in this routine that the translation is performed. The way to proceed is much complete and modern than the approach used with `InternalMapVirtualKeyEx` routine. This one can use `InternalVkKeyScanEx` routine in addition to have the ability to compose with complex alphabets. The strategy used by `ToAsciiEx` function is to call `ToUnicodeEx` function and then translate from Unicode to ASCII the return character thanks to `GetLocaleInfoW` [957] and `WideCharToMultiByte` functions [958].

The reverse operation to translate a character to a virtual corresponding virtual-key code and shift state is performed thanks to `VkKeyScanEx` function [951]. This translation is performed by using the input language and physical keyboard layout identified by the input locale identifier. It means it has the ability to manage local specificities. For instance, the right-hand ALT key is used on some keyboards (including the French keyboard



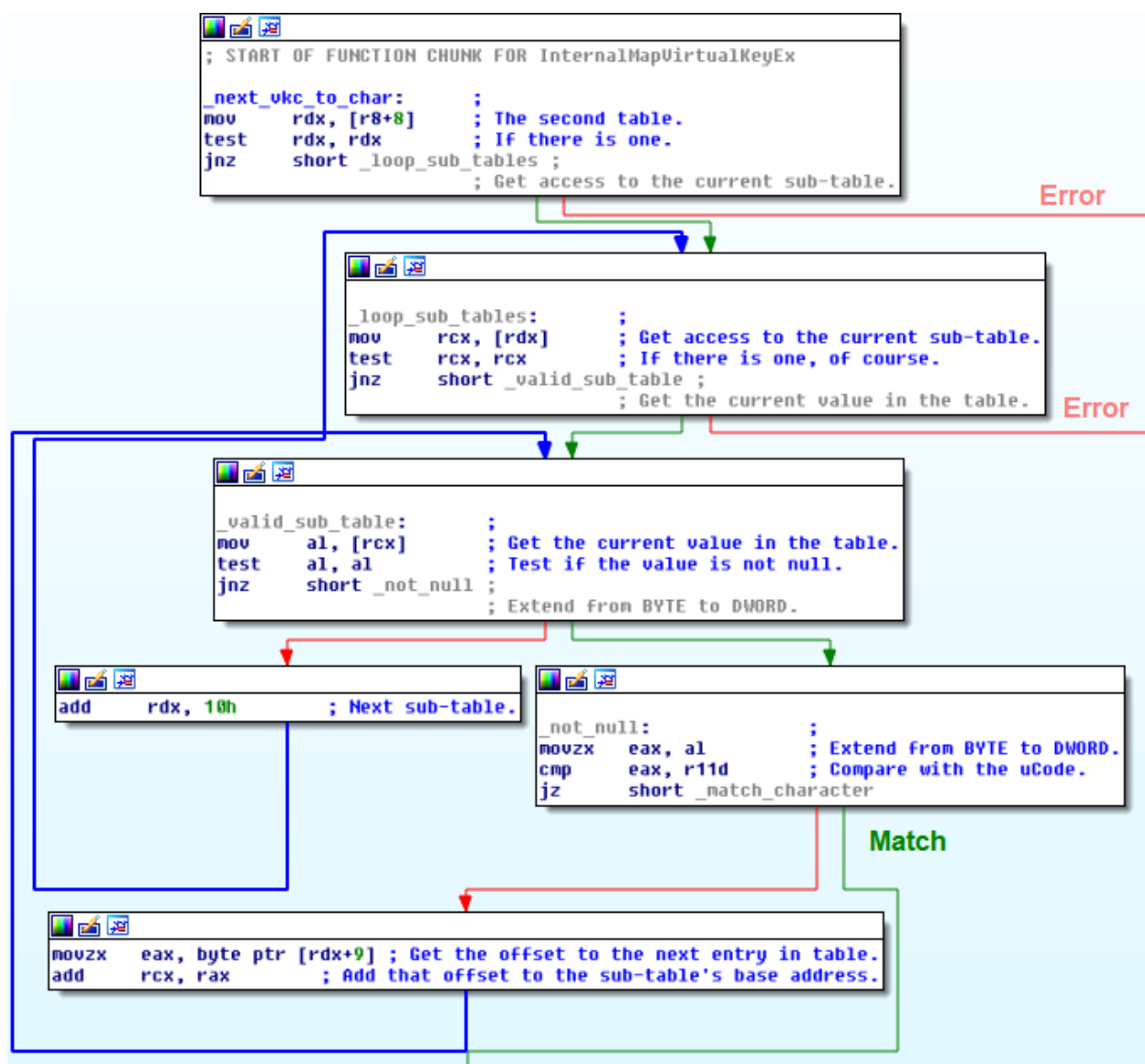


Figure 4.110: Assembly code from a portion of `InternalMapVirtualKeyEx` routine in the case where the translation is performed with `MAPVK_VK_TO_CHAR` flag.

layout) as a shift key. In this case, the shift state is represented by the value 6, because the right-hand ALT key is converted internally into CTRL + ALT. If the function succeeds, the low-order byte of the return value contains the virtual-key code and the high-order byte contains the shift state, which is a combination of flag bits (one for shift, control, alt and other for a key state combined). Internally `VkKeyScanEx` function calls `NtUserVkKeyScanEx` routine from `win32kbase.sys` with a third parameter set to one (to tells that it is a caller mode notification).

In a similar way than `NtUserMapVirtualKeyEx` routine, the `NtUserVkKeyScanEx` routine is going to retrieve the current keyboard layout (avoiding for optimization reasons to deal with `csrss` or `dwm` processes) before calling with it `InternalVkKeyScanEx` routine. This routine reuses the same logic described with `InternalMapVirtualKeyEx` routine. The translation is performed through tables or sub-tables in the keyboard layout. Note that if there is no tables in the keyboard layout is provided, tables stored in the keyboard layout `gspklBaseLayout` global value are used. This value is set in `xxxLoadKeyboardLayoutEx` routine. Note that there are two versions of `VkKeyScanEx` function. One for ASCII input and another for Unicode input. The difference is that the

ASCII version `VkKeyScanExA` translate its input to Unicode with `GetLocaleInfoW` and `MultiByteToWideChar` [959] functions. This is finally equivalent to a direct call to `VkKeyScanExW`.

The last operation to cover is the translation from the scan code to the virtual key code. This one is performed thanks to `MapVirtualKeyEx` function. The call to `MapVirtualKeyEx` function must include either `MAPVK_VSC_TO_VK` or `MAPVK_VSC_TO_VK_EX` flag. It first retrieves the value at offset `+0x38` in the keyboard layout tables structure which gives the maximum size of the seventh table (`0x7f` in our case). If the provided value is below that number, it means it will be read from the seventh table, the same used for translating from virtual key code to scan code. If the value is above, the eighth and the ninth tables can be used for reinforcement to perform the translation. We guess these two tables correspond to extended scan codes `0xE000` and `0xE100` sets. The search in these tables is performed with the `uCode` value until there is a match. The translation is given by reading the following values until a potential match happens. In the case of the provided value would be below the size of the seventh table, the translation is direct by using `uCode` as an index to get access to the translated code in the seventh table. Whatever is the translation, in both codes, we should have a value. This value is then checked specifically in the case where the call has been performed in the context of `MAPVK_VSC_TO_VK` and the translated virtual key code value is between `VK_LSHIFT` and `VK_RMENU`. In this case, there is a conversion between left or right shift, control or alt keys to ambiguous keys (neither left nor right). Otherwise if the translated value is equal to `0xFF`, this one is translated to be 0. If it is not, the translated virtual key code is directly returned.

Note that it is possible to change the behavior of a given keyboard layout [879]. Indeed, if a device produces an incorrect scan code for a certain key, the wrong virtual key message will be sent. One could fix this issue by writing a filter driver that analyzes the scan codes generated by firmware and modifies the incorrect scan code on-the-fly to one understood by the system. But this solution requires to write a filter driver which is far from being obvious and prone to error generation. Especially for such a task. This is why starting from Windows 2000 and Windows XP, a new *Scan Code Mapper* is present to provide an easy method that allows translation of scan codes into specific virtual key codes. This operation is performed through the registry of Windows in "HKLM\SYSTEM\CurrentControlSet\Control\Keyboard Layout" key. To update the translation from scan code to virtual key code, a `Scancode Map` value must be added inside the key. This value is a `REG_BINARY` (little Endian format) and it uses a specific documented binary format to allow translation. For short, there is a header which is not so relevant and a list of mappings preceded by the number of mapping stored in the value. The mapping is one `DWORD` value in length which is divided into two `WORD` length fields. Each word field stores the scan code for a key to be mapped. The first word is the scan code to be replaced and the second is the replacement scan code. Note that the value zero can be used as a replacement scan code to disable a key on the keyboard.

The *Scan Code Mapper* has several advantages and disadvantages. About advantages, historically, it has been used to easily fix or correct firmware errors. It is also about changing the mapping of frequently used keys or disabling those which are not often used (for example, right CTRL key) or exchanging the behavior of some keys. Indeed, key locations can be altered easily to customize the user's experience on a given device. But this customization is far from being without consequences. First, it requires a system reboot to activate it once the mapping is stored in the registry. This mapping is active at a system level and it is applied to all users. It means that the mapping engaged in the registry cannot be set to work differently depending on the current user. It cannot be set to be specific to a keyboard layout or to a given device. It applies to all keyboards connected to the system. In addition, it is not perfect to manage specific *functionalities* in the system. For instance, to disable the screen capture functionality, one software might be tempted to disable to the *Print Screen* (PrtSc) key on keyboard [960]. This is usually not a good idea since third-party software can directly simulate keyboard input thanks to `SendInput` function and directly capture an image of the screen [961]. It makes more sense to use specific functions to complicate the screen capture procedure by using `SetWindowDisplayAffinity` [962] function. But, this is just an obstacle, not a security measure. If somebody is determined to get pixels from the screen, this step is only going to slow them down a little [963].

This finalizes explanations about conversion between scan codes, virtual key codes and displayable characters. This is an important point because there is here all the abstraction between what the keyboard hardware connected to the machine provides as information (and in particular the USB HID interface makes it possible

to identify the type of connected hardware and then the layout used by the system). It is also a way to change the layout between the hardware used and what is understood by the system or applications. As a result, a key pressed on a given keyboard can have different meanings for an application, depending on the keyboard layout selected. It also explains how characters entered on a keyboard (with accents, modifiers such as control, shift or alt) can be represented on the screen. Technically, it is the responsibility of the applications receiving the messages to perform the conversions they need (scan codes, virtual key codes or characters) or they can use the automated channels (`WM_KEYDOWN` to get both scan code and virtual key code of a key pressed) with the message loop or `TranslateMessage` function to get `WM_CHAR` messages to retrieve characters. Behind the scene, it is the functions and routines presented here that handle the translation work from these automated channels. For instance, `TranslateMessage` function, after checking initial parameters from the `MSG` structure [836], it calls `NtUserTranslateMessage` routine from `win32kfull.sys`. This routine uses `xxxTranslateMessage` routine which itself uses `xxxInternalToUnicode`, the same used by `ToUnicodeEx` function, and `PostMessageExtended` routine to translate in Unicode and dispatch the new `WM_CHAR` message.

---

## 5.3 Other means to access keyboard

### Key Point 4.46:

- ☞ While the message system remains the backbone of keystroke transmission, there are other ways to retrieve them.
  - ✍ If these methods can appear as "out of message subsystem", they are nevertheless all dependent (not to say driven) by the *raw input thread*.
  - ✍ This does not question the central position of the *raw input thread*.
- ☞ Some of the methods presented in this subsection are popular with malware.
  - ✍ We try to explain why by presenting the advantages and disadvantages of each.

As explained in section 5.1, the raw input thread is a central point in the broadcasting process of keystrokes but it is not the only point. In this section, we propose to cover the other means available to interact with the keyboard for user-mode applications [9] and how all of them are handled by the kernel. Most of them are based on different points already covered, which explains why less details are provided here.

### 5.3.1 Keyboard state

#### Key Point 4.47:

- ☞ Internally within RIT, there are different structures that represent the state of the keyboard keys (pressed and released).
  - ✍ It can be interesting to know whenever a key is pressed if another one is also pressed (shift for uppercase management, multiple keys combinations within shortcuts context, and so on).
  - ✍ Access can be synchronized with the reception of a message or completely asynchronous.
  - ✍ In the case of synchronous access, it is necessary to have access to a message from the keyboard and therefore to have the *focus* for the application.
  - ✍ In the case of asynchronous access, the freedom is much greater.
  - ✍ Possibilities to listen to the whole keyboard stealthily but restrictions for the current desktop only.

#### 5.3.1.1 Direct access to keyboard keys state

In addition to proceeding with keyboard messages, an application may need to determine the status of a given key besides the one that generated the current message. For instance, this could happen in the case where the application would need to check the status of a specific shortcut. Notwithstanding the registration of hot key controls [964], an application might check the status of a specific key when another has been pressed. Taking an example, to handle "CTRL+C" shortcut, when receiving the message when the "C" key has been pressed, the application have to check the status of the control key. One solution would be to register the previous state key message received but it could require a complex implementation in some cases. Instead of, an application can determine the status of a virtual key by using either `GetKeyState` [965] or `GetAsyncKeyState` [704] functions. The fist function is used in the context of a keyboard-input message. This function retrieves the state of a key when the input message was generated. More directly, the output of `GetKeyState` changes according to input messages pumped from the calling thread's message queue. The status does not reflect the interrupt-level state associated with the hardware. To get access to the status of a key regardless of whether a corresponding keyboard message has been retrieved from the message queue, the function `GetAsyncKeyState` is required. That one returns the state (up or down) of a given key at the time the function is called.

Function `GetKeyState` returns the virtual key state which means that it reports the state of the keyboard based on the messages retrieved from the calling thread's input queue. This is not the same as the physical keyboard state [966] returned by `GetAsyncKeyState` function. Since `GetKeyState` is used in a message context, when a key is pressed, `GetKeyState` will not report a change until there is a call to `PeekMessage` or `GetMessage` functions to pump the message list from the input queue. In practice, if an application wants to make the distinction in a keys combination, it is going to use `GetKeyState` function. Indeed, it is possible to know if a given key was down when the user pressed another one. It is different to know if the key is down this very instant, a result provided by `GetAsyncKeyState` function. In addition, if the application using `GetKeyState` function loses the keyboard focus, then the function will not see the input that the user typed into another application, since that input was not sent to its thread input queue. Another difference with `GetAsyncKeyState` function which is not attached to a message input queue. Indeed, this function is able to provide the status of a given key at the moment it is pressed.

Technically, the use of `GetKeyState` and `GetAsyncKeyState` functions to read the status of a key requires to select the virtual key code of that key in parameter. To get access to the complete list of virtual key codes (composed of 256 elements), we can use the `GetKeyboardState` function [967]. This one takes as parameter an array of 256 bytes to be filled by the system. Status of elements in the list is linked to the thread's keyboard messages management from its message queue.

### 5.3.1.2 GetKeyboardState function

#### Key Point 4.48:

- ☞ `GetKeyboardState` function is used to retrieve the full state from all keys of the keyboard.
  - ✎ It only provides a limited view of the keyboard internal state from the current desktop.
  - ✎ If the requiring thread is not the current *foreground thread*, it has access to only few keys' state.
  - ✎ The keyboard status changes as a thread retrieves keyboard messages from its message queue.
  - ✎ This is why this function is considered as a *synchronous* method.

The function `GetKeyboardState` [967] takes as a single parameter an array of 256 bytes used to receive the status data for each virtual key. For each entry in the array, if one or more bits are set in a value, it means that the key referenced as the index in the array is down or toggled.

Internally, `GetKeyboardState` function is exported from `user32.dll` and `win32u.dll`. This one is interfaced in the kernel by `NtUserGetKeyboardState` routine in `win32kfull.sys` driver. The routine first checks if the provided buffer address is a user-mode address and that one is writable for security reasons<sup>50</sup>. Then the Win32 Thread structure is extracted from the current thread. This one is compared to `grpdeskRitInput` value in order to check if our thread is the current foreground desktop thread.

In the case where the requesting thread is the current foreground desktop thread, information will be provided about keys. Otherwise, information provided will be limited. This restriction is applied for security reasons, to avoid a thread to get access to information belonging to another desktop.

At the end of `NtUserGetKeyboardState` routine comes a loop iterated from 0 to 255 in order to cover the whole array provided. This one covers the list of all possible virtual key codes (which represents 256 different values [516]). Technically, a list of virtual key codes is stored in the Win32 Thread structure. That one is *compact*ed to save space in memory. A subset of bits is used to describe the state of each key. This is why the routine uses bit shifts and bit masks operations to access data (Figure 4.111).

---

<sup>50</sup>The procedure used here does not call directly `ProbeForWrite` routine [968] but operations performed are similar and it could be the result of a compilation optimization.

---

The description of a key is given by a set of three states. The first is that the key is released, which means a state equals to zero. If the key is pressed, the state is one. And if the key is a toggle key, which happens when a pressed key is considered definitively pressed until the user press it again — for example CAPS LOCK — the state is two. These states are translated in the array provided by the user as binary values. According to the documentation of `GetKeyboardState` function [967] and what we observe in Figure 4.111, the most significant bit of the byte representing the virtual key code in the provided buffer is set to one if the key is pressed and the least significant bit is set to one if the key is toggled.

```

32 | Index = 0;
33 | Array_3 = Array_1;
34 | while ( Index < 256 )
35 | {
36 |     *Array_3 = 0;
37 |     if ( bDeskRitInputCheck || (LOBYTE(i_1) = i, (unsigned int)IsKeyStateCached(i_1)) )
38 |     {
39 |         LastBitsFromI = i & 3;
40 |         ShiftedIndex = (unsigned __int64)(unsigned __int8)i >> 2;
41 |         ReadInPti = *(_BYTE *) (ShiftedIndex + InPti + 0xEC);
42 |         if ( (unsigned __int8)(1 << 2 * LastBitsFromI) & ReadInPti )
43 |         {
44 |             *Array_3 |= 0x80u;           // If the high-order bit is 1, the key is down; otherwise, it is up.
45 |             ReadInPti = *(_BYTE *) (ShiftedIndex + InPti + 0xEC);
46 |         }
47 |         i_1 = (unsigned int)(2 * LastBitsFromI + 1);
48 |         if ( (unsigned __int8)(1 << (2 * LastBitsFromI + 1)) & ReadInPti )
49 |             *Array_3 |= 1u;           // If the key is a toggle key, for example CAPS LOCK, then the
50 |                                     // low-order bit is 1 when the key is toggled and is 0 if
51 |                                     // the key is untoggled.
52 |     }
53 |     Index = ++i;
54 |     ++Array_3;
55 | }

```

Figure 4.111: Extract from the pseudo-code of `NtUserGetKeyboardState` routine in `win32kfull.sys` to report the current content of the keyboard for an input thread.

Note that `NtUserGetKeyboardState` routine is using `IsKeyStateCached` routine (from `win32kbase.sys`). This last one is called when the request is performed in a different context from the one stored in `grpdeskRitInput`, which could mean that we are not acting in the context of the current foreground thread. More directly, we are accessing information belonging to another foreground thread, which could rise security issues. In such way, some keys are still accessible but not all (for security reasons). The pseudo-code of `IsKeyStateCached` routine is given in Figure 4.112. This one returns one if the provided virtual key code in parameter corresponds to a cached key and zero otherwise. All keys listed in `KeyStateCached` global value (the content is given in Figure 4.112) or keys which are below `VK_SPACE` key code (i.e. mouse and control keys) or all keys which are not between `VK_LWIN` and `VK_RMENU` or not below `0x9F` (this value is after the virtual key codes of the numeric pad) or not above `VK_RWIN`. To make it clear, there is an embargo concerning all keys which can be used by user for writing purposes in order to preserve the confidentiality of what is typed. Other keys (which are not usable for direct writing purposes) can be accessed (states keys).

### 5.3.1.3 GetKeyState function

#### Key Point 4.49:

- ☞ `GetKeyState` function is used in a received message context to know if another targeted key has been pressed.
  - 👉 Function which used in a synchronous context.
  - 👉 Close to `GetKeyboardState` in its logic but only for a single virtual key code.
  - 👉 It uses internally `NtUserGetKeyState` routine.
  - 👉 Hard to manage especially in the context where the foreground thread would change during message reception.

```

1 signed __int64 __fastcall IsKeyStateCached(unsigned __int8 vkCode)
2 {
3     unsigned int Index; // edx@1
4     char *KeyStateCached; // rax@2
5
6     Index = 0;
7     if ( vkCode >= (unsigned __int8)VK_SPACE )
8     {
9         KeyStateCached = &::KeyStateCached;
10        while ( *KeyStateCached != vkCode )
11        {
12            ++Index;
13            ++KeyStateCached;
14            if ( Index >= 14 )
15            {
16                if ( vkCode < (unsigned __int8)VK_LWIN
17                    || vkCode > (unsigned __int8)VK_RMENU
18                    || vkCode <= 0x9Fu && vkCode > (unsigned __int8)VK_RWIN )
19                {
20                    return 0i64;
21                }
22                return 1i64;
23            }
24        }
25    }
26    return 1i64;
27 }

```

KeyStateCached	db	UK_CAPITAL
	db	UK_KANA
	db	UK_NUMLOCK
	db	UK_HOME
	db	UK_SCROLL
	db	UK_DEM_ATTN
	db	UK_DEM_FINISH
	db	UK_DEM_COPY
	db	UK_DEM_AUTO
	db	UK_DEM_ENLW
	db	UK_DEM_BACKTAB
	db	UK_ATTN
	db	UK_PLAY
	db	UK_ZOOM
	db	0
	db	0

Figure 4.112: Pseudo-code of IsKeyStateCached routine in win32base.sys.

The case of `GetKeyState` [965] follows a logic close to `GetKeyboardState` function, with optimization. First, it takes in parameter a single argument corresponding to the virtual key code to be checked. In practice, an application calls `GetKeyState` in the context of a keyboard-input message reception. This function retrieves the state of the key when the input message was generated in order to know if there is a key combination. That way, `GetKeyState` function is synchronous. Indeed, this function should proceed messages in a context of message queue, which means that it should react quickly to be relevant. This is why `user32.dll` — which exports `GetKeyState` function — tries to act directly.

Function `GetKeyState` first checks in the current *Process Environment Block* (PEB) [216] the undocumented *Win32ThreadInfo* structure which should be equivalent to the *Win32 Thread* structure in kernel-mode. From that point, the function checks if the thread is active and it is a foreground thread. If not, the function returns zero. Indeed, it

Otherwise, for optimization purposes, the function checks if the requested virtual code is above `VK_SPACE`. If it is not, it directly retrieves the virtual key codes cache in *Win32ClientInfo* and it tests the status of the key based on the virtual key code provided (the cache is a compressed version, which explains the bit shifts operations). The same way that with `GetKeyboardState` function, if the key is toggled, the least significant bit is set and if the key is pressed, the most significant bit is set. Note that in Figure 4.113, the value `0xFF80` is used to set the most significant bit for a key. This is reminiscent of Windows 3.1 which also set the most significant bit with this function but due to a cast concern on the return value, there was an extension of the sign since `GetKeyState` return a short value (two-bytes) and the concerned bit is on the first byte. For backwards compatibility reasons, this originality has been kept.

If the key is not below `VK_SPACE` (and the list of possibilities does not look to be not the most used keys), the function `GetKeyState` calls `NtUserGetKeyState` from `win32u.dll` which interfaces the same routine name in `win32kfull.sys`. This routine is more complex since it is based on performance constrains. This one first checks if the current thread is cross session attached (with `IsThreadCrossSessionAttached` routine). It helps to retrieve the *Win32Thread* structure holding the internal cache of virtual key codes. Then, the routine checks if this thread needs a key state update. Indeed, `GetKeyState` function is usually in the context of a message loop, looping on hardware events and dealing with messages with `PeekMessage` function. But it may not be proceed that way and messages could be lost or internal cache not aware of specific hardware events, including keystrokes. It could happen when an application processes keyboard input but the focus is changing at the same time. For instance, "Windows+M" reduces all windows on the screen, changing foreground thread the same time it



```

1 SHORT __stdcall GetKeyState(int nVirtKey)
2 {
3     unsigned int VirtKey; // ebx@1
4     SHORT result; // ax@4
5     int CacheOfVirtualKey; // er8@6
6     SHORT retval; // r9@6
7
8     VirtKey = nVirtKey;
9     if ( !*( _DWORD *)(*MK_FP(__GS__, 0x30i64) + 0x78i64) && !NtUserGetThreadState(14i64) )// Win32ThreadInfo
10        return 0;
11     if ( VirtKey >= UK_SPACE || *( _DWORD *)(*MK_FP(__GS__, 0x30i64) + 0x870i64) != *( _DWORD *) (gpsi + 0x1B48) )// Win32ClientInfo
12        return NtUserGetKeyState(VirtKey);
13     CacheOfVirtualKey = *(unsigned __int8 *)(((unsigned __int64)(unsigned __int8)VirtKey >> 2)// Used as index.
14        + *MK_FP(__GS__, 0x30i64)
15        + 0x874);// From Win32ClientInfo, probably the cache of UKCodes.
16     retval = _bittest(&CacheOfVirtualKey, (unsigned __int8)(2 * (VirtKey & 3) + 1));
17     result = retval | 0xFF80;
18     if ( !_bittest(&CacheOfVirtualKey, (unsigned __int8)(2 * (VirtKey & 3))) )
19        result = retval;
20     return result;
21 }

```

Figure 4.113: Pseudo-code of GetKeyState function in user32.dll.

receives input messages. In this case, as soon as transition events change queues happens, the queues of all different input threads are marked with a specific flag in *Win32Thread* structure. This first check allows a call to *PostUpdateKeyStateEvent* routine to perform a copy of the asynchronous key state (*gafAsyncKeyState* global value in *win32kbase.sys*). This value is composed of bits representing the keys that have changed since the last update. It is composed of 64 bytes in raw and taking into account that the encoding is performed on two bits per key, a byte can hold 4 keys, which allows a representation of 256 keys ( $64 \times 4 = 256$ ). Then *PostUpdateKeyStateEvent* routine calls *PostEventMessageEx* to post an event which updates the local thread's virtual key-state table cache, ensuring the thread's key-state is always up-to-date. This architecture ensures that all queues are input-synchronized with key transitions, no matter where such transitions occur.

The rest of the *NtUserGetKeyState* routine's procedure checks if the required virtual key is cached (with *IsKeyStateCached* routine) or if *grpdeskRitInput* corresponds to the current raw input thread. In both case, the access to the key state is guaranteed (otherwise the routine returns 0). A check is performed on *gpqForeground* to be sure that there is a foreground thread, internal members inside its undocumented structure are also checked (including rights that are guaranteed to the calling thread to access keyboard input with *CheckAccess* routine) and finally a call to *IsKeyboardDelegationEnabledForThread* routine must return zero. This last check is linked to specific hooks in the system which are not documented.

All these checks finish with the virtual key requested to be below 0x100 (one more than the maximum limit of the virtual key codes — if the code is invalid, error *ERROR\_INVALID\_PARAMETER* is set for the calling thread). This last test seen as true, the procedure used to retrieve the state of the virtual key code provided is not very different to the procedure used by *NtUserGetKeyboardState* routine. But instead of looping on all possible virtual key codes, the provided one is directly used as an index to retrieve the state of the selected key. This state is translated on extremities bits from the returned byte, the same way that *NtUserGetKeyboardState* routine does.

#### 5.3.1.4 GetAsyncKeyState function

##### Key Point 4.50:

- ☞ GetAsyncKeyState function is probably the most popular technique to retrieve keyboard entry.
  - ✍ This is an asynchronous function, which means it does not depend from current thread's message queue.
  - ✍ Neither subject to *keyboard focus* nor concerned by *foreground thread* property.
  - ✍ Simple to use (in a loop since it checks one virtual key code at time) and efficient result (*best seller* for malware).
  - ✍ But not free from drawbacks: only for current desktop, relatively slow and it could miss some keystrokes.
- ☞ GetAsyncKeyState is based on internal RIT structures (mainly `gafAsyncKeyState`) that represent the current state of the keyboard.

One famous keyboard function to describe is `GetAsyncKeyState` [704]. That one is may be the most widely used since it is easy to use since it does not require to take into account the notion of system messages or keyboard focus. At the opposite to `GetKeyState` and `GetKeyboardState` functions, this one is not synchronized with a message input queue but with the thread's virtual key-state table cache. But from an efficiency point of view, the `GetKeyState` function is very fast when called while `GetAsyncKeyState` is not, despite similar optimization used in both cases.

Technically, `GetAsyncKeyState` function [704] is implemented in `user32.dll`. This one takes a single parameter: the virtual key code of the key to check and its return value is similar to what is returned by `GetKeyState` function. Since virtual key code concerns mouse buttons too, the function checks first if the caller desires to access mouse buttons. In this case, asynchronous key state checks on the state of the physical mouse buttons, not on the logical mouse buttons that the physical buttons are mapped to. In case of the two mouse's buttons would have been swapped, the function will return the state of the original physical mouse button, regardless of whether it has been swapped before (caller can takes care of swapping after by using `GetSystemMetrics` [969] function with `SM_SWAPBUTTON` parameter). Then `GetAsyncKeyState` function checks if the required virtual key code is one of the common keys (below `VK_SPACE`). In this case, it tries to retrieve it from its cache in user-mode. Note that this time, at the opposite to `GetKeyState` and `GetKeyboardState` functions, the most significant bit informing that a key is pressed is set with `0x8000` and not with `0xFF80`. If the requested key is not below `VK_SPACE`, the function calls `NtUserGetAsyncKeyState` function. It is interfaced by the kernel in `win32kbase.sys`.

The first part of `NtUserGetAsyncKeyState` routine is to perform all the required checks to avoid other threads from processes to spy on other desktops. In addition, it checks if the current foreground thread (`gptiForeground`) belongs to another process and if this thread allows the caller to get access to input data by the hook or with the *journal record*<sup>51</sup> (with `RtlAreAnyAccessesGranted` on the current `Win32Thread` structure). In addition, it manages one event monitor `gpAsyncKeyEventMonitor` to call `CAsyncKeyEventMonitor::OnKeyStateRequested` in case of. If one of these checks fail, the caller receives zero and an `ERROR_ACCESS_DENIED` code is set. Otherwise, and if `IsKeyboardDelegationEnabledForThread` returns false, `GetAsyncKeyState` routine (from `win32kbase.sys`) is called with the virtual key state to retrieve.

Routine `GetAsyncKeyState` is quite simple. After checking that the requested virtual key code belongs to the virtual key code space (below `0x100`), it retrieves information documented in `GetAsyncKeyState` function [704]. This one returns two types of information. On the first hand, the least significant bit of the return value indicates whether the key has been pressed since the last query. But this information is not really accurate since the multitasking nature of Windows can preempt the current application for another one which can call `GetAsyncKeyState` function. That way, the second application would receive the "recently pressed" bit instead

<sup>51</sup>More information about *journal record* in section 5.3.2.

of the first one. The behavior of the least significant bit of the return value is retained strictly for compatibility with 16-bit Windows applications (which are non-preemptive) and should not be relied upon. On the other hand, the most significant bit is set when the requested key is down. This behavior is described in the `GetAsyncKeyState` routine's pseudo-code given in Figure 4.114.

```

1 __int16 __fastcall GetAsyncKeyState(unsigned int nVirtKey)
2 {
3     unsigned __int64 VirtKey; // r10@2
4     unsigned __int64 UpperVirtKey; // rcx@2
5     unsigned int LowerVirtKey; // er9@2
6     signed __int16 retval; // r11@2
7     int RecentDownKey; // eax@2
8     __int16 result; // ax@4
9
10    if ( nVirtKey >= 0x100 )
11    {
12        UserSetLastError(ERROR_INVALID_PARAMETER);
13        result = 0;
14    }
15    else
16    {
17        VirtKey = (unsigned __int8)nVirtKey;
18        UpperVirtKey = (unsigned __int64)(unsigned __int8)nVirtKey >> 3;
19        LowerVirtKey = VirtKey & 7;
20        retval = 0;
21        RecentDownKey = *(unsigned __int8 *)(UpperVirtKey + gafAsyncKeyStateRecentDown);
22        if ( !_bittest(&RecentDownKey, LowerVirtKey) )
23        {
24            retval = 1;
25            *(_BYTE *)(UpperVirtKey + gafAsyncKeyStateRecentDown) = RecentDownKey & ~(1 << LowerVirtKey);
26        }
27        result = retval | 0x8000;
28        if ( !(*(_BYTE *)&gafAsyncKeyState + (VirtKey >> 2)) & (unsigned __int8)(1 << 2 * (VirtKey & 3)) )
29            result = retval;
30    }
31    return result;
32 }

```

Figure 4.114: Pseudo-code of `GetAsyncKeyState` function in `win32kbase.sys`.

The manipulation of index to read `gafAsyncKeyStateRecentDown` table is different from the one used in `gafAsyncKeyState`. It is because the first table keeps the information whether the key has been recently pressed or not. But at the difference of `gafAsyncKeyState` where the encoding is performed on two bits (to store three states — nothing, down and toggle) here it is only needed to log is the key has been pressed recently or not. And it only needs only one bit per virtual key (down or not). It means that one byte can pack the states of 8 virtual keys. This explains the split of the virtual key codes in two parts (the lower three bits on one side, the remaining ones on the other side) and the bit test to check the state of the recently down virtual key code. Note that, if the *recent down bit* was set for a given virtual key code, this one is cleared in the global value `gafAsyncKeyStateRecentDown` right after, confirming the possible lacks in case of preemption by Windows in the documentation [704]<sup>52</sup>. The manipulation of `gafAsyncKeyState` is similar to the one observed in `GetKeyState` and `GetKeyboardState` functions. This one sets the upper bit of the return value if the selected virtual key is set (toggle information is discarded, probably to avoid compatibility problems).

Note that `gafAsyncKeyState` value is updated during the procedure of the raw input thread in many points. Just to cite few of them, we have `xxxKeyEventEx` in `win32kbase.sys` which is a key point of the access to `gafAsyncKeyState` table. This routine is used in `CKeyboardProcessor::ForwardInputToKeyboardOverrider` when it calls `CKeyboardProcessor::CreateKeyboardInputMessage` to create input message for the keyboard or to manage direct input from the keyboard. It is updated for instance in `UpdateAsyncKeyState` routine which is part of different routines, including `xxxKeyEventEx`. In addition, it is involved in `xxxChangeForegroundKeyboardTable` routine used by `xxxInternalActivateKeyboardLayout` to load (`NtUserLoadKeyboardLayoutEx` routine uses `xxxLoadKeyboardLayoutEx`), to activate (`NtUserActivateKeyboardLayout` routine uses `xxxActivateKeyboardLayout`) or to unload a keyboard layout (`NtUserUnloadKeyboardLayout` routine uses `xxxInternalUnloadKeyboardLayout`). The value

<sup>52</sup>It is impressive to observe here a code that maintains backward compatibility with 16-bit systems (before Windows 3.1 or Windows 95) even in the latest version of Windows 10. This feature is literally a living fossil...

`gafAsyncKeyState` is stored in `win32kbase.sys` but exported by the last and imported in `win32kfull.sys` driver. In this driver, it is manipulated by `UpdatePerUserKeyboardIndicators` routine which is involved by `NtUserUpdatePerUserSystemParameters` routine. This one is used in `IdleTimerProc` routine (through `xxxHungAppDemon`) in the `RawInputThread` routine via `gnRITdemonTimerId` timer. Also, it is used in `ProcessUpdateKeyStateEvent` called by `PostUpdateKeyStateEvent` in `PostInputMessage` or by `NtUserGetMessage`, `NtUserPeekMessage`, or `NtUserPostMessage`. It is also involved in or `xxxScanSysQueue` routine (from `win32kfull.sys`) which is managing internally all different types of messages.

At the end, `NtUserGetAsyncKeyState` routine clears or updates the cache of the current calling thread, depending on the result of `IsKeyboardDelegationEnabledForThread` routine (if the keyboard is delegated, the procedure clears the keyboard cache in user-mode). It is the last operation before returning the value expected by the caller. It remains that `GetAsyncKeyState` function will directly return the value provided by the kernel.

#### 5.3.1.5 SetKeyboardState function

##### Key Point 4.51:

☞ In addition to accessing the internal state of the keyboard, it is possible to manipulate it thanks to `SetKeyboardState` function.

Before to conclude, we have to mention `SetKeyboardState` function [970] (from `user32.dll`) to update the asynchronous key state table directly. This function takes in parameter an array of 256 bytes representing the content of the table to set (the same way `GetKeyboardState` function retrieves information). Internally, it is interfaced by `NtUserSetKeyboardState` in `win32kfull.sys`. After checking access to the provided array (with a procedure close to `ProbeForRead` [971]) and the access to the current thread key state table is guaranteed (in `gptiCurrent`). The main operation is performed in `SetKeyboardState` routine (from `win32kfull.sys`). This routine takes the address of the current thread cache table (from `gptiCurrent`) and updates it through a loop acting on the 256 entries. It manages both key down and toggle states. At the end of the loop, the routine increments a member in `gpsi` structure, probably the number of times the key cache table has been updated. Note that `gafAsyncKeyState` table is not updated here. Why? Because the `SetKeyboardState` function alters the input state of the calling thread and not the global input state of the system, as documented [970]. But this has not always been the case [972], even if setting specific toggled keys (such as NUM LOCK, CAPS LOCK, SCROLL LOCK or the Japanese KANA indicator lights on the keyboard) was not really efficient. It is better to use `SendInput` function to set or clear keys. Indeed, it simulates keystrokes a better way than `SetKeyboardState` function would do since the last only acts on asynchronous keyboard cache table of the calling input thread.

One interesting point is the relationship between simulated input (such as `SendInput`) and internal key-state functions (such as `GetAsyncKeyState`). One might wonder what happens when one calls the `SendInput` function followed instantly afterwards by the `GetAsyncKeyState` function to checks whether the input key is reported as down [831]. The `SendInput` function stimulates the whole chain of keyboard input. It means the function puts input packets into the system hardware input queue where the raw input thread picks them up. Even if the raw input thread is configured to run with a high priority, the code generating the input may continue to run on a different CPU core<sup>53</sup> than the one where the raw input thread is running. And as explained before, the raw input thread has a lot of things to do (dequeue different events, manage hooks which can involve third-party codes, broadcast messages to the rest of the system, manage display if needed, and so on) and it is only when they are done that it is about to update the `gafAsyncKeyState` table. This is that table which is checked by `GetAsyncKeyState` function. And it could happen that `GetAsyncKeyState` function does not see the simulated key sets in the table if the call happens too soon.

#### 5.3.1.6 Conclusion about keyboard keys state

Finally, it appears that the direct access procedure to the keyboard virtual key codes state table is an efficient procedure for retrieving data entered by the keyboard. There is a cache procedure used for optimization

<sup>53</sup>Considering we are working on a multi-core machine, of course.

purpose, especially for the most used keys (virtual key codes below `VK_SPACE`). If `GetKeyState` and `GetKeyboardState` functions are a bit complex to manipulate since they are part of the current thread input message queue management, they remain useful for special shortcut management mostly. But they are dependent to the application's keyboard focus.

If the application is not directly interacting with the user, that one has no (or so close) key status. This is not the case of `GetAsyncKeyState` which depends neither from the application's focus nor from the state of the foreground thread. Instead, it directly accesses the heart of the system containing the current status of the keyboard. This makes this function very popular among Windows application developers. Nevertheless, it often requires to loop through all possible values of the virtual key codes, posing some problems of optimization and CPU resource consumption. Moreover, depending on no synchronization, this function is at the goodwill of the Windows scheduler to give it the hand often enough so that it does not miss (too many) keystrokes. If it often appears simpler than other functions, it is nevertheless without constraints or consequences.

There is no user-mode API allowing a direct total copy of the entire `gafAsyncKeyState` buffer into the kernel. In a way, this would not be as useful as it may look like. Indeed, one would either have to strongly synchronize the competitive access to this variable (it is already synchronized in `NtUserGetAsyncKeyState` in a way) or to accept to lose information if a key is pressed while the copy is in progress. Ultimately, it should be kept in mind that this set of features is ultimately limited to a very specific subset of keyboard key management by the raw input thread. It is not enough to deal with `gafAsyncKeyState` in kernel-mode or to call `SetKeyboardState` to manage the keyboard.

## 5.3.2 Hook procedures

### 5.3.2.1 Introduction to hook procedure

#### Key Point 4.52:

- ☞ The hooks mechanism is part of the message system.
  - ✍ It allows to filter certain types of messages for an application.
  - ✍ It provides an access to specific messages from other applications.
- ☞ Interception of messages is performed through a *hook procedure*.
  - ✍ The *hook procedure* is notified for each event received.
  - ✍ It can log, modify or discard the event.
- ☞ There are different *types of hooks*.
  - ✍ Each type of hook provides an access to a specific part of the message-handling mechanism.
  - ✍ The system maintains a separate *hook chain* for each *type of hook*.
  - ✍ A *hook chain* is list of function pointers defining *hook procedures* registered by applications.
  - ✍ *Hook procedure* is notified in the context of the calling thread or in the context of any application, depending on its *scope*.
- ☞ The *scope* of the hook defines the level where the hook applies.
  - ✍ Depending on the *hook type*, the *scope* can be *global* or *local*.
  - ✍ *Global hook* monitors messages for all threads in the same desktop.
  - ✍ *Local hook* (or a *thread-specific hook*) monitors messages for a single thread only.
- ☞ Hooks tend to slow down the system because of extra work required for each message by the system.

Another way to retrieve the keyboard keys is to set up a *hook* mechanism. Often evoked previously, it seems important to explain the concept of *hook* [2]. Technically, a hook is mechanism by which an application can intercept events, such as messages, mouse actions, and keystrokes. It is part of the system message-handling mechanism [812]. The function registered and used to intercept a particular type of event is called a *hook procedure*. This hook procedure is notified for each event received and where the hook procedure can log, modify or discard the event. Hooks are useful but they should only be used when necessary and removed when there are no more needed. Indeed, they tend to slow down the system because they increase the amount of processing the system must perform for each message.

There are different types of hooks an application can register. Each type of hook provides an access to a specific part of the message-handling mechanism. For instance, an application can be interested by filtering messages retrieved from other applications message queues [973]. Internally, the system maintains a separate *hook chain* for each type of hook. A *hook chain* is a list of function pointers defining hook procedures and registered by applications. When a message occurs that is associated with a particular type of hook, the system passes the message to each hook procedure referenced in the hook chain, one after the other.

Technically, a hook procedure can perform different things depending on the type of hook involved. For some types of hooks, it is only possible to monitor messages while others allow a modification of messages or to stop their progress through the chain. In the last case, it would prevent them from reaching the next hook procedure or the destination window.

In practice, hook can be applied on different level and it has a *scope*. The *scope of a hook* depends on the hook type. Indeed, it could be applied on a targeted application or on the whole system. This is why we have



*local* and *global* hooks. A *global hook* monitors messages for all threads in the same desktop as the calling thread. A *local hook* (or a *thread-specific hook*) monitors messages for only an individual thread. Some hooks can be set only with global scope while others can also be set for only a specific thread. Note that global hooks include local ones. Table 4.18 gives the details of each type of hook and the scope associated to each.

Hook type	Scope of the hook	Reference	Description
WH_CALLWNDPROC	Thread or global	[974]	Monitor messages sent to window procedures. Called before passing the message to the receiving window procedure.
WH_CALLWNDPROCRET	Thread or global	[975]	Monitor messages sent to window procedures. Called after the window procedure has processed the message.
WH_CBT	Thread or global	[976, 977]	Called before activating, creating, destroying, minimizing, maximizing, moving, or sizing a window; before completing a system command; before removing a mouse or keyboard event from the system message queue; before setting the input focus; or before synchronizing with the system message queue.
WH_DEBUG	Thread or global	[978]	Called before calling hook procedures associated with any other hook in the system.
WH_FOREGROUNDIDLE	Thread or global	[979]	Called when the application's foreground thread is about to become idle. It is useful for low priority tasks during times when foreground thread is idle.
WH_GETMESSAGE	Thread or global	[973]	It enables an application to monitor messages about to be returned by the <code>GetMessage</code> or <code>PeekMessage</code> function. Can be used to monitor mouse and keyboard input.
WH_JOURNALRECORD	Global only	[980]	It enables a hook procedure to monitor and record input events. Useful to record a sequence of mouse and keyboard events to play back later by using <code>WH_JOURNALPLAYBACK</code> .
WH_JOURNALPLAYBACK	Global only	[981]	Used to play back a series of mouse and keyboard events recorded earlier by using <code>WH_JOURNALRECORD</code> . It is possible to insert messages into the system message queue. Regular mouse and keyboard input is disabled as long as this hook is installed. It returns a time-out value to tell the system how many milliseconds to wait before processing the current message from the playback hook.
WH_KEYBOARD	Thread or global	[982]	Used to monitor keyboard input posted to a message queue. It monitors message traffic for <code>WM_KEYDOWN</code> and <code>WM_KEYUP</code> messages about to be returned by <code>GetMessage</code> or <code>PeekMessage</code> functions.
WH_KEYBOARD_LL	Global only	[983]	Enables an application to monitor keyboard input events about to be posted in a thread input queue.
WH_MOUSE	Thread or global	[984]	Used to monitor keyboard input posted to a message queue. It monitors message traffic for mouse messages ( <code>WH_MOUSE</code> ) about to be returned by <code>GetMessage</code> or <code>PeekMessage</code> functions.
WH_MOUSE_LL	Global only	[985]	Enables an application to monitor mouse input events about to be posted in a thread input queue.
WH_MSGFILTER	Thread or global	[986]	It monitors messages passed to a menu, scroll bar, message box, or dialog box created by the application. It allows to filter messages during modal loops that is equivalent to the filtering done in the main message loop.
WH_SHELL	Thread or global	[987]	Used to receive important notifications. When the shell application is about to be activated and when a top-level window is created or destroyed
WH_SYSMSGFILTER	Global only	[988]	Same as <code>WH_MSGFILTER</code> but monitors messages for each application.

Table 4.18: List of hooks types with their scope associated (from [1, 2]).

Of course, thread scope is different than global scope for handling hook procedure. A global hook procedure can be called in the context of any application in the same desktop<sup>54</sup> as the calling thread. Since applications have their own memory space, the procedure must be written in a separate Dll module to be shared between all different applications. More directly, the Dll holding the global hook procedure is about to be injected in each process concerned by the type of hook selected. This is a regular Dll injection [989, 990, 991] technique which is used in this case by the kernel to insert the code in the memory space of concerned processes. Note that there is an issue concerning 32-bit applications hooking 64-bit applications<sup>55</sup>. Indeed, a 32-bit Dll cannot be injected into a 64-bit process, and a 64-bit Dll cannot be injected into a 32-bit process. The only way to

<sup>54</sup>Notifying a global hook in the context of another desktop would be a security hole. Indeed, it would mean that an application could access messages from another application from another desktop. A desktop which belongs to another user, probably dealing with different access rights. This is to avoid any elevation of privileges that *global* hooks are limited to the view of their current desktop.

<sup>55</sup>The same issue applied for 16-bits applications on 32-bits systems in former times. Note that current Windows versions running on a 32-bits CPU still keeps a retro-compatibility for such case.



impact both architectures is to have a 32-bit application and, at the same time, an application<sup>56</sup> compiled for 64-bits the same time. Both are registering for the same hook types and both are using their own Dll, compiled in 32-bits and in 64-bits to inject all applications running in the two different subsystems. Note that in the case of global hook, it is mandatory to keep pumping messages in the hooking application to avoid blocking normal functioning of the system.

At the opposite, a thread-specific hook procedure is called only in the context of the associated thread [992]. In this case, since the hook is dealing with threads in the same working space, it can be written in either the current application or in a dedicated Dll, this makes no difference. More generally, if an application installs a hook procedure for a thread working in a different application, the procedure must be in a Dll. But it must be noted that global hooks should be restricted to special-purpose applications and more specifically for debugging purposes. Indeed, such hooks impact badly the performances of the system. In addition, it is prone to conflict with other applications that implement the same type of global hook. Finally, a hook procedure remains a code that acts in a process that originally did not foresee it. This is a direct source of instability that is added to applications.

This is the case for hooks that are able to operate locally on a thread and globally on the system. Using a Dll solves the question but it is possible to handle a global hook from a dedicated thread in an application. This is usually how WH\_KEYBOARD\_LL keyboard hook procedure is implemented. In such a case, a dedicated thread is created to handle and pump messages coming from all different applications hooked. Here we act as a central point before redistributing messages to all applications. This is a very specific way to deal with global hooks and it is available only for a small subset of hooks.

### 5.3.2.2 Registration of a hook procedure

#### Key Point 4.53:

- ☞ Hook procedures are registered with the `SetWindowsHookEx` function.
  - ✍ Hook procedure is registered generally in a dedicated Dll for *global hooks*, anywhere for *local hooks*.
  - ✍ With a *global hook*, this implies a Dll injection in all targeted processes (to execute the hook procedure in the context of the hooked thread).
  - ✍ At the end of the hook procedure, `CallNextHookEx` must be called to notify other hooks in the *hook chain*.
- ☞ To remove a hook, we use `UnhookWindowsHookEx` function.
  - ✍ For *global hook*, if the hook is neutralized, the Dll is not released despite the call to `UnhookWindowsHookEx` function.
- ☞ Hook procedures are not allowed on threads belonging to a *protected process*.
  - ✍ From Windows Vista, to support Digital Rights Management, some processes are protected from other processes.
  - ✍ It is not a perfect security, this one can be bypassed by any administrator.

To register a hook procedure [993], there is the `SetWindowsHookEx` function [1]. This function installs the provided callback into the chain associated with the hook type selected in parameter. The hook procedure can be associated with all threads in the same desktop as the calling thread or with a particular thread. In the first case, the thread ID provided is equal to zero.

For security purposes, hook procedures are prevented on threads belonging to a *protected process* [994]. Indeed, from Windows Vista, Microsoft introduced a new type of process to enhance support for Digital Rights

<sup>56</sup>Note that the 32-bit and 64-bit Dlls must have different names, according to official documentation [1].

Management [995]. The main difference between a regular process and a protected process is the level of access that other processes in the system can obtain to protected processes. Note that this security can be bypassed [996]. Such security concerns also anti-malware services [997] from Windows 8.1.

In the case where we are dealing with a global hook, a separate Dll from the application must be present. That one must be accessible to the installing application before installing the hook. In other words, the application is responsible to load in its own memory space the Dll hosting the hook procedure with `LoadLibrary` [753]. Once the library is loaded, it is possible to retrieve a pointer to the hook procedure thanks to `GetProcAddress` function [754]. These two operations are required since `SetWindowsHookEx` function takes the base address of the Dll and the hook procedure as parameter.

Usually, a hook procedure can be registered with the following minimal Code 4.31. This code register a global hook (`WH_KEYBOARD_LL`) from a hook procedure ("*HookProcName*") stored in a Dll ("*Dllname.dll*").

```
HOOKPROC hkprcKbdLIHdler;
static HINSTANCE hModule;
static HHOOK hHook;

hModule = LoadLibrary(_T("Dllname.dll"));
if(hModule == NULL) { __leave; }
hkprcKbdLIHdler = (HOOKPROC)GetProcAddress(hModule, "HookProcName");
if(hkprcKbdLIHdler == NULL) { __leave; }

hHook = SetWindowsHookEx(WH_KEYBOARD_LL, hkprcKbdLIHdler, hModule, 0);
```

Code 4.31: Registration of a hook procedure for keyboard low level notifications.

Once this hook is no more needed, it must be removed thanks to `UnhookWindowsHookEx` function [998]. This one takes the handle to the hook procedure returned by `SetWindowsHookEx` function. Note that, for specific types hook (`WH_JOURNALPLAYBACK` and `WH_JOURNALRECORD`), the control of the hook removal procedure can be performed directly by the hook procedure itself. It happens when `VK_CANCEL` virtual key code<sup>57</sup> is received by the hook procedure recorded for `WH_JOURNALRECORD` type. According to the documentation for this type of hook [980], this virtual key code should be interpreted by the application as a signal that the user wishes to stop journal recording. The application should respond by ending the recording sequence and removing its hook procedure. Removal is important. It prevents a journaling application from locking up the system by hanging inside a hook procedure.

Note that there is a much more robust way to disable both `WH_JOURNALPLAYBACK` and `WH_JOURNALRECORD` hook types. Indeed, key combinations `CTRL+ESC` and `CTRL+ALT+DEL` cause the system to stop all journaling activities (record or playback), remove all journaling hooks, and post a `WM_CANCELJOURNAL` message [999] to the journaling application. This is to prevent the system to be totally stunk by a single application since as long as the journal playback hook procedure is installed, regular mouse and keyboard input is disabled. Another specificity of these hook type is unlike most other global hook procedures, these ones are always called in the context of the thread that set the hook. They do not need to be implemented in a Dll.

When `UnhookWindowsHookEx` function is used in the context of a global hook, if it neutralizes the procedure hook, it does not free the Dll containing the hook procedure. This is because the global hook stored in a Dll has been loaded by force through an implicit `LoadLibrary` by the system on all hooked processes. Because a call to the `FreeLibrary` function [1000] cannot be made for another process, there is no direct way to free the Dll. Of course, we cannot expect any injected process to directly call in for us since the last one did not expect to be hooked by a third-party application. Eventually, the system frees the Dll after all processes explicitly linked to the Dll have terminated. A solution could be to inject manually the Dll in every targeted processes with a call to `CreateRemoteThread` function [746]. That way, from the Dll, it might be possible to hook the targeted process and its thread as a local hook. However, this solution raises more problems than it fixes, starting with doing a "everything but light" operation in the `DllMain` entry point [1001], which is not recommended at all

<sup>57</sup>Which is implemented as the `CTRL+BREAK` key combination on most keyboards.

since it impacts badly the stability and it could generate deadlocks [1002, 1003, 1004].

The hook procedure is a well-defined HOOKPROC callback [975]. The `nCode` parameter is a hook code that the hook procedure uses to determine the action to perform. The value of the hook code depends on the type of the hook. Each hook type has its own characteristic set. The values of the `wParam` and `lParam` parameters depend on the hook code, but they typically contain information about a message that was sent or posted.

```
LRESULT CALLBACK HookProc(int nCode, WPARAM wParam, LPARAM lParam) {  
    // process event  
    ...  
    return CallNextHookEx(NULL, nCode, wParam, lParam);  
}
```

Code 4.32: Minimal code to handle a hook procedure (from [2]).

Technically, `SetWindowsHookEx` function always installs a hook procedure at the beginning of a hook chain [2]. When an event occurs and a hook procedure is registered to handle it, the system calls the procedure at the beginning of the hook chain associated with the hook. That is to say, the last registered hook procedure is notified first. When it is allowed by the hook type, a hook procedure in the chain has the ability to choose to pass the event to the next procedure or not. This chain hook is maintained through `CallNextHookEx` function [1005] called by the hook procedure.

Note that when hook procedure can only monitor messages, the system passes messages to each hook procedure [2], regardless of whether a particular procedure calls `CallNextHookEx`. Whatever the case may be, calling `CallNextHookEx` is not really an option and it is highly recommended. Indeed, by stopping messages distribution, other applications that have installed hooks will not receive hook notifications and they may behave incorrectly as a result. In addition, regular applications might expect to receive a message and behave incorrectly if they do not receive it, with dramatic consequences [1006]. Avoiding calling `CallNextHookEx` function should be reserved to absolutely needed situations, if such exist.

### 5.3.2.3 Internals of `SetWindowsHookEx` function

#### Key Point 4.54:

- ☞ A call to `SetWindowsHookEx` function to register *hooks* involves both user-mode and kernel-mode components.
  - ☞ Even if the function does not take any string as parameter, it makes the difference between two types of strings (Ansi and Unicode).
  - ☞ This is historical reasons and also for the *hook procedure* that can potentially receive such strings.
  - ☞ In practice, this is about registering a structure in the *hook chain list* for a given *hook type*.
  - ☞ It is not the address of the *hook procedure* that is saved but an offset to that one (to bypass ASLR).

Registering a hook procedure is an operation for which we propose to explain some internal details. Function `SetWindowsHookEx` is exported from `user32.dll` under the two names `SetWindowsHookExA` and `SetWindowsHookExW` for ANSI and Unicode versions. Usually, such distinction makes sense when the function deals with strings which can be represented with ANSI or Unicode encoding. But this distinction between the two versions does not make sense since here. Indeed, the `SetWindowsHookEx` function takes neither input strings nor it returns any. Instead, it takes a `HINSTANCE` of the file (Dll or executable) where the hook procedure belongs. Explanation comes from former times [1007] when Windows was 16-bits. At that time, there was no need to inject anything anywhere since all 16-bit Windows applications ran in the same address space. At that time, the hook setup was direct since it only required to instance the hook procedure. But on Windows

32-bits system, every process has its own memory space and its own handle table. Changing the `SetWindowsHookEx` function to take an input string would have not fixed the problem. Indeed, the window manager would have to do a `GetModuleHandle` [1008] anyway to recover the instance handle from where the hook procedure code belongs. And it would have broken the source compatibility with older Windows versions. This is why both `SetWindowsHookExA` and `SetWindowsHookExW` functions calls `SetWindowsHookExAW` function which uses `GetModuleFileNameW` function [1009] to convert the `HINSTANCE` to a string holding the path of the module hosting the hook procedure. The difference between ANSI and Unicode version lies in the last parameter passed to `SetWindowsHookExAW` function and which is relevant for kernel to know how to notify the hook procedure.

Once the module path name has been retrieved, there are two possibilities to record a hook. If the hook type is on thread local scope (in practice `WH_GETMESSAGE`, `WH_CALLWNDPROC` or `WH_MOUSE`) the targeted thread is the current one, the function uses `CLocalHookManager::AddHook` function. This function takes the hook type and the callback pointer. It first retrieves (or allocates if it did not exist already) the hook collection with `CLocalHookManager::GetLocalCollection` and then get access to the chain hook associated with the required hook type with `CLocalHookManager::GetHookChainHeadOfType` function. Then a hook structure is crafted thanks to `CHookFactory::CreateHook` which, for short, associates to the hook two sets of *vftables* [1010] (list of functions pointers), one generic (`CBaseLocalHook<7>::vftable` or `CBaseLocalHook<4>::vftable`) and one specific (`CCallWndProcHook::vftable`, `CGetMessageHook::vftable`, or `CMouseHook::vftable`). Then the registration is finalized for the current thread with one of the record procedure function stored in these tables.

In the case where it would not be possible to record the hook procedure to the current thread, `SetWindowsHookExAW` calls `_SetWindowsHookEx` function. That one translates the path routine into an `UNICODE_STRING` before calling `NtUserSetWindowsHookEx` which is interfaced by the kernel in `win32kfull.sys`. This one performs few checks such as the presence of a thread info structure for the targeted thread (with `PtiFromThreadId` routine) or if the `HINSTANCE` is coming from the current executable. At the end, `zzzSetWindowsHookEx` routine is called.

The first part of `zzzSetWindowsHookEx` routine is a list of checks. Checking if the hook type is valid, checking if the callback pointer is valid, checking if there is a thread info linked here and if the application is trying to set a local hook that is global-only, checking if the hook does to target a thread outside its desktop or an application from another user where rights would not be guaranteed, checking if the hook requires a Dll to access a different process, checking is the hook can be applied to an input message thread, checking if the targeted process is not a protected one, checking the hooks rights required (`WH_JOURNALPLAYBACK` and `WH_JOURNALRECORD` requires specific rights respectively `DESKTOP_JOURNALPLAYBACK` and `DESKTOP_JOURNALRECORD` [1011] while other hooks need `DESKTOP_HOOKCONTROL`) and finally checking if the current session's desktop is interactive (not session 0, for instance).

Once all the checks have been performed, the routine calls `HMAAllocObject` to allocate a new kernel hook structure. If a Dll is required for this hook, `GetHmodTableIndex` is called to register the library in order to load it into all the relevant processes. This part is based on *atom tables* [1012]. The library loaded has an internal counter in its atom table allowing to know how many dependencies it has. To track the list of dependencies, `AddHmodDependency` routine is used to increment the number of dependencies attached to that library.

If we are dealing with a targeted thread to hook (a local hook), internal flags are set to the thread's structure and in the kernel hook structure previously allocated. Access to distant thread memory is assured with `KeAttachProcess` [1013] and `KeDetachProcess` [1014] routines. If we are dealing with a global hook, some flags and members are initialized in the kernel hook structure. Then for both hook scopes, a flag is set in the kernel hook structure to know if the hook function expect ANSI or Unicode text as input. This is the reason why the ANSI or Unicode type information is kept by the "SetWindowsHookEx" function API. We note that there is a manipulation on the callback pointer address. Indeed, instead of storing the address of the hook procedure, this is the offset from the base address of the module which is holding it which is kept. Indeed, a Dll can be loaded at a different address in the memory space of different processes (due to ASLR [1015] among other reasons). This offset will then be reused in the hooked process to compute the linear address of the callback. And finally, the kernel hook structure is linked to the structure holding the previous hook procedures for this hook type in the kernel.

Then it comes a specificity for journal hook type with a call to `zzzJournalAttach` for synchronization purposes mainly. The `zzzSetWindowsHookEx` routine checks also if in the case of a global hook, that one is not running at a two low priority. Indeed, in such a case, the system could be paralyzed soon. To avoid that issue, the routine calls `KeSetPriorityThread` [1016] with a priority equals to 14 (`LOW_REALTIME_PRIORITY-2`) which is quite high. It ensures that the global hook procedure will respond quite efficiently<sup>58</sup> from a performance point of view. In the case where the hook procedure would be an update a journal hook type, the routine *jiggle* the mouse with `GenerateMouseMove` routine (from `win32kfull.sys`) so that the first event is always a mouse move in order to properly initialize the cursor position. At the end, after telemetry routines calls, the routine returns the address of the allocated kernel hook structure.

#### 5.3.2.4 Notification mechanism from kernel

##### Key Point 4.55:

- ☞ Kernel-mode components about hook procedure is a complex system of notification from kernel-mode to user-mode.
  - ☞ If the notification is made from the kernel, the hook is executed in the user-mode context.
  - ☞ Hooks procedures are notified part of the usual message management by the kernel (for instance in `xxxScanSysQueue` routine).
  - ☞ From applications' point of view, hooks notifications are managed not by type of hook but by type of message holds by the hook handler.

Hook notification is key mechanism. It is notably performed via `xxxCallHook`, `xxxCallHook2` and `xxxHkCallHook` routines from `win32kbase.sys`. These routines are quite complex and the full description of these ones is beyond the scope of this document. Indeed, to proceed, it would require to cover many internal aspects of Windows since there are different types of hooks dealing with different parts of the operating system. But for the sake of simplicity, we can say that `xxxCallHook` is using `xxxPointerCallHook` and `xxxCallHook2` to notify hook procedures. Routine `xxxCallHook2` is the true heart of the hook procedure. This is the *manager* of hooks, a real scheduler for hook procedures. It is about to manage and launch different hook procedures recorded in the hook chain (which is a simple list internally). Of course, it does this after checking if the hook has not already been unhooked before. It uses `xxxInterSendMsgEx` to notify hooked threads in different processes or `xxxLoadHmodIndex` routine to load a Dll holding a hook procedure in a targeted application.

For each hook procedure to call in `xxxCallHook2` routine, the call is performed via `xxxHkCallHook` routine. This one is also a complex routine which acts according to the hook type. It uses a set of `fnHkXxx` routines where "Xxx" is a different suffix for each type of hook. This set of routines is not only managed by hook types but rather by the type of message holds by the hook. Indeed, hooks deal with synchronously sent messages that point to message's structures that need to be correctly bundle to call the hook procedure in the right format. For instance we have `fnHkINLPMSG`, `fnHkOPTINLPMESSAGE`, `fnHkINLPCBTCREATESTRUCT`, `fnHkINLP-MOUSEHOOKSTRUCTEX`, `fnHkINDWORD`, `fnHkINLPKBDLLHOOKSTRUCT`, and so on... All these routines are generally designed in the same way. They start by checking different global values (`gdwInAtomicOperation` and `gdwExtralInstrumentations`) and to set diverse synchronization mechanisms. Then, they call `KeUserModeCallback` (from `ntoskrnl.exe`) to call the user mode callback registered as a hook procedure. This routine from the kernel initialize a stack (`MmCreateKernelStack`) before notifying the user-mode callback (that is to say the hook procedure in our case) via `KiCallUserMode` and `ExCallCallBack`. At the end of `KeUserModeCallback` routine, the stack is removed with `MmDeleteKernelStack`. Finally, the routine checks if the message manipulated by the hook procedure satisfies security concepts before releasing synchronization mechanisms and returning.

In addition to use `fnHkXxx` routines, `xxxHkCallHook` routine can use `gapfnScSendMessage` callbacks table previously presented (Figure 4.89). Callbacks are selected in the table thanks to an index previously computed

<sup>58</sup>Of course, if the hook procedure uses infinite loop or never responds, such optimization is meaningless. Developers are able to make their own life miserable with so many ways [1017]. But this solution is a good one since it could avoid some issues.

to be used in `MessageTable` table.

Note that a great consumer of hooks is `xxxScanSysQueue` routine which is used by `xxxRealInternalGetMessage` called by `NtUserGetMessage` routine (kernel interface for `GetMessage` function [816]). Routine `xxxScanSysQueue` is used to manage the system hardware message queue and to broadcast messages. It determines what window will be notified with the input message used, among other things<sup>59</sup>. This is a perfect place from where different hooks notification can be performed. Indeed, it is central point to scan system message queue and call specific hooks. Most of hooks are managed with `xxxCallHook` routine, but some are specific such as `xxxCallCtfHook`, `xxxCallMouseHook`, `xxxCallTSFNotifyHook` (part of `xxxProcessTSFEvent` called from `xxxProcessEventMessage`), `xxxGetNextSysMsg` (from `xxxGetNextSysMsg`), and `xxxCallJournalRecordHook` (from `xxxSkipSysMsgEx`).

At the end, from an application point of view, notification can be performed in different context. In the case of a local hook, the notification always happen in the context of the hooked thread. In a way, it could be seen as a similar result of a `AttachThreadInput` function [874], still with the synchronization operations to perform but within the targeted thread and not from another one. In the case of a global thread, it depends how the hook procedure is supposed to be implemented. For some hook, notification happens in the context of an arbitrary thread (usually thanks to an APC procedure [1018]) in the targeted process or in the context of a local thread in the hooking application, usually the one which registered the hook procedure.

---

<sup>59</sup>For instance, this is where the double click is managed with `gbClientDoubleClickSupport` global object.

---



### 5.3.3 Keyboard hook cases

#### Key Point 4.56:

- ☞ There are two main types of keyboard hooks: `WH_KEYBOARD` and `WH_KEYBOARD_LL`.
- ☞ About `WH_KEYBOARD_LL`:
  - ☞ It is *global hook* but notified in the context of the thread that installed the hook (no need of Dll).
  - ☞ Notification of the hook procedure is performed via messages.
  - ☞
- ☞ `WH_KEYBOARD` is both a *local* and *global* hook.
  - ☞ Depending of its registration, it can be attached to a targeted thread or set globally to all possible targeted threads in the system.
  - ☞ Usually, notifications are performed in the context of an arbitrary thread (APC) hosted in a dedicated Dll.
- ☞ `WH_KEYBOARD_LL` is used to monitor events about to be posted in a thread input queue while `WH_KEYBOARD` is notified when messages are already posted.
  - ☞ Consequently, `WH_KEYBOARD_LL` is at a lower notification altitude in the device call stack than `WH_KEYBOARD`.
  - ☞ When a key is pressed, `WH_KEYBOARD_LL` is more likely to be notified before the other hooks.
  - ☞ A modification in `WH_KEYBOARD_LL` will result in a repercussion in `WH_KEYBOARD` (the other way is not true).

From a practical point of view, there are two ways to set up a hook on the keyboard. One at level `WH_KEYBOARD` and one other at level `WH_KEYBOARD_LL`. There are differences between the two levels. A fist concerns `WH_KEYBOARD_LL` which is a global hook while `WH_KEYBOARD` is both a thread and a global one. Even if `WH_KEYBOARD_LL` is a global hook, this one does not need any Dll to be functional. It is notified in the context of the thread that installed the hook. Notification is performed via messages, this is why that thread must have a message loop [983].

The `WH_KEYBOARD` hook type is a bit more complex since its documentation is a bit confuse. This hook can be global or local. More precisely, it can be attached to a targeted thread or set globally to all possible targeted threads in the system (in such a case, the thread ID parameter in `SetWindowsHookEx` function is zero). According to the documentation for this hook [982], notification can be performed in the context of the thread that installed the hook by messages (if the installing thread has a message loop). But in practice, it is more likely to be performed in the context of an arbitrary thread. Naturally, it does not make sense to hook messages that the requesting application naturally obtains via its own message loop. In practice, this hook is more often used to capture messages from a third-party application, which requires to use a dedicated Dll holding the hook procedure.


A second difference lies in the *altitude* used to notify the hook procedure. Especially, `WH_KEYBOARD_LL` is used to monitor events about to be posted in a thread input queue. In the case of `WH_KEYBOARD`, it aims to monitor message traffic for `WM_KEYDOWN` and `WM_KEYUP` messages about to be returned by the `GetMessage` or `PeekMessage` functions. Consequently, `WH_KEYBOARD_LL` is at a lower notification altitude in the device call stack than `WH_KEYBOARD`, this is why when a key is pressed, it is more likely to be notified before the other hooks. Thus, a modification (edit or delete) in `WH_KEYBOARD_LL` will result in a repercussion in `WH_KEYBOARD` (the other way is not true).



Finally, a third difference is in the ease of use of one compared to the other. Since `WH_KEYBOARD` requires to use a Dll, it is necessary to compile a version for 32-bit and 64-bit subsystems. And there is a Dll injection which is everything but stealth. On the other hand, `WH_KEYBOARD_LL` does not necessitate a Dll at all since notifications are made through the classic message system. This explains its popularity.

### 5.3.3.1 Implementation of `WH_KEYBOARD` hook

#### Resume 30:

 This subsection details how to implement `WH_KEYBOARD` keyboard hooks.

Implementation and use of `WH_KEYBOARD` hook type is not complex. It is composed of two components, one is the executable file responsible to setup the hook procedure and the second is the Dll file holding the hook procedure. The first is given in Code 4.33 with only relevant lines (error-checking has been removed for the sake of simplicity).

```
int main(int argc, char *argv[]) {
    HINSTANCE hMod = NULL;
    PVOID pfHook = NULL;
    HHOOK hkb = NULL;

    hMod = LoadLibrary("Dllname.dll");
    pfHook = GetProcAddress(hMod, "KeyboardProc");
    hkb = SetWindowsHookEx(WH_KEYBOARD, (HOOKPROC)pfHook, hMod, 0); // Global, for all threads.

    // Wait procedure.
    // (...)

    UnhookWindowsHookEx(hkb);
    return 0;
}
```

Code 4.33: Entry point of the executable file to setup the hook procedure for `WH_KEYBOARD`.

The Dll holding the hook procedure is of course composed of a `DllMain` [1001] which does almost nothing and an exported function for the hook procedure. This one is called `KeyboardProc` in Code 4.34. The procedure is written to keep the content of every key pressed thanks to a `LogProcedure` function. The last is used to record keystrokes in a text file, for instance.

```
LRESULT __declspec(dllexport) __stdcall KeyboardProc(int nCode, WPARAM wParam, LPARAM lParam){
    // Key is down (30th bit in lParam) and HC_ACTION to be sure it is about a keystroke message
    if (((DWORD)lParam & 0x40000000) && (HC_ACTION == nCode)){
        // wParam holds the virtual key code of the pressed key.
        // lParam holds the scan code and other relevant information.
        LogProcedure(wParam, lParam);
    }

    LRESULT RetVal = CallNextHookEx(NULL, nCode, wParam, lParam);
    return RetVal;
}
```

Code 4.34: Hook procedure for `WH_KEYBOARD` in a dedicated Dll.

### 5.3.3.2 Implementation of WH\_KEYBOARD\_LL hook

#### Resume 31:

☞ This subsection details how to implement WH\_KEYBOARD\_LL keyboard hooks.

As opposed to WH\_KEYBOARD, WH\_KEYBOARD\_LL hook does not require a Dll. It means that everything can be implemented in a single executable file. Nevertheless, it remains that the procedure must be implemented in two separate functions. The first installs the hook procedure, the second is the hook procedure itself. The first is given in Code 4.35 and it is quite similar to the one given in Code 4.33 notwithstanding we can directly access the address of the procedure to record (LowLevelKeyboardProc in our case). Note that, for operational purposes, it is better to use a dedicated thread to arm the procedure. This is why we call CreateThread function [259] to create a dedicated thread to setup the hook and them to proceed messages. This two different steps are given in Code 4.35.

```
// Global value to keep track of the hook.
HHOOK hHook = NULL;

VOID SetupHookLowLevelInDedicatedThread(VOID) {
    HANDLE hThread = NULL;

    // Create and launch the thread.
    hThread = CreateThread(NULL, 0, InternalSetWindowsHookKeyboard, NULL, 0, NULL);

    // Wait until an event happens.
    WaitForSingleObject(hThread, INFINITE);

    // Remove the hook after use.
    UnhookWindowsHookEx(hHook);
}

DWORD InternalSetWindowsHookKeyboard(VOID) {
    MSG message = { 0 };

    // Set the hook procedure.
    hHook = SetWindowsHookEx(WH_KEYBOARD_LL, LowLevelKeyboardProc, GetModuleHandle(NULL), 0);

    // Message loop management.
    while (GetMessage(&message, NULL, 0, 0) != 0) {
        TranslateMessage(&message);
        DispatchMessage(&message);
    }

    return 1;
}
```

Code 4.35: Installation of WH\_KEYBOARD\_LL hook procedure in a dedicated thread.

The hook procedure is in the same logic than the one given in Code 4.34. This one is given in Code 4.36. Note that lParam parameter is a KBDLLHOOKSTRUCT structure [1019]. This one host the virtual key code, the scan code, the timestamp of the message event and a flag value which describes different properties about the key (especially if the key has been injected or if it comes from a real keystroke). Parameter wParam is the identifier of the keyboard message (WM\_KEYDOWN, WM\_KEYUP, WM\_SYSKEYDOWN, or WM\_SYSKEYUP).

```
LRESULT CALLBACK LowLevelKeyboardProc(int nCode, WPARAM wParam, LPARAM lParam) {

    PKBDLLHOOKSTRUCT kbDllHook = (PKBDLLHOOKSTRUCT)lParam;

    // Do not process message.
    if (nCode < 0) {
        return CallNextHookEx(hHook, nCode, wParam, lParam);
    }
}
```

```
// wParam holds the type of key event received.  
// kbDllHook holds virtual key code and scan code.  
MessageFormatted = LogProcedure(kbDllHook, wParam);  
  
return CallNextHookEx(hHook, nCode, wParam, lParam);  
}
```

Code 4.36: Hook procedure for WH\_KEYBOARD\_LL.

## 5.4 Miscellaneous about keyboard

This section discusses various information about other way to retrieve information from the keyboard. There is no central purpose, but it aims to allow the reader to answer potential questions that may arise, with regard to actuality or the interaction with keyboard by itself.

### 5.4.1 Dedicated library to access keyboard

#### Key Point 4.57:

- ☞ In practice, there are libraries that manage the keyboard directly. To do this, two approaches are possible.
  - ☞ Either it bypasses (or ignores) the Windows message system to manage the keyboard directly at kernel level (with a user-mode interface) — DirectX.
  - ☞ Or it is only a wrapper of the Windows API (an overlay) hiding the complexity with a nice interface — Qt, SDL, OpenCV, Tk, Gtk, script languages...

It is quite possible to find libraries of codes that manage the access to the machine's keyboard. All *Graphical User Interfaces* (GUI) are part of the set of libraries which handle the keyboard (Qt, SDL, OpenCV, Tk, Gtk, etc.) with maybe DirectX [1020] as the most famous one. From that point, two strategies are possible. Either we create a specific software stack to capture the keyboard, or we rely on the existing Windows API as an extra layer. To be very direct, it is generally the second solution that is preferred, for simplicity reasons. One of the few cases (not to say maybe the only one) that implements its own keyboard management is DirectX.

### 5.4.1.1 DirectX library

#### Key Point 4.58:

- ☞ DirectX is a graphics library used mainly by video games. It offers the possibility to manage the keyboard too.
  - ✍ DirectX bypasses the *raw input thread* to deal directly with the keyboard driver.
  - ✍ This implies a loss of all the interactions and facilities offered by the RIT (perfect for video games' purposes).
- ☞ In practice, DirectX's keyboard management can be seen as a parallel channel to the one provided by Windows to convey the keystrokes.
  - ✍ There is a total disconnect with the Windows API.
  - ✍ There is a bypass so that other applications (using Windows API) can be deprived of the keyboard keys.
- ☞ DirectX requires to follow a specific protocol to use it.
  - ✍ The initialization allows us to define how we interact with the keyboard.
  - ✍ The keyboard must be *acquired* to be read (which locks the configuration). It can be released thereafter.
  - ✍ There are two types of possible read procedures: *indirect* and *buffered*.
- ☞ DirectX uses a specific code to represent physical keys from the keyboard device (and called *dik code* by us) different from virtual key code.
  - ✍ In practice, dik code corresponds more or less to the internal Windows scan code set (Key-Point 4.25).

This last one is mostly used in video games industry and it offers powerful graphical display API since 1995. It is perfectly possible to manage the keyboard with it [1021]. It is usually regrouped under the `Microsoft.DirectX.DirectInput` namespace [1022]. Procedure is documented by Microsoft [1021, 1023] who provides in addition sample codes [1024]. In case of, there are some third party code samples online [1025, 1026, 1027] which provide a step by step explanation.

#### Initialization of DirectX keyboard

The first thing to do to use keyboard with DirectX is to use `DirectInput8Create` function [1028] to create a pointer to an instance of the `IDirectInput8` interface [1029]. This instance will allow us to access all the devices connected to the client computer of the application. Technically speaking, this function is going to load the `dinput8d.dll`<sup>60</sup> if this one has not already been loaded, before calling `DirectInputCreateHelper` function. To select the keyboard in particular, we need to use the `IDirectInput8::CreateDevice` method [1030] from the `IDirectInput8` interface. With this method, we provide a dedicated GUID called `GUID_SysKeyboard` representing the keyboard [1031]. In case of success, the method called provides us a `DIRECTINPUTDEVICE8` structure used as an instance from `IDirectInputDevice8` interface [1032] to access keyboard specific methods.

To get access to keyboard's data, it is required to define the way data will be retrieved. On the first hand, we have to define the format with whom data will be accessed by DirectX. To proceed, we use `IDirectInputDevice8::SetDataFormat` method [1033] with a predefined data format structure called `c_dfDIKeyboard` [1034]. On the other hand, we have to define *cooperative level* [1035] which determines how the input is shared with other

<sup>60</sup>In case of, the procedure is about to retrieve the Dll by looking for the registry key "HKEY\_CLASSES\_ROOT\CLSID\25E609E4-B259-11CF-BFC7-444553540000\InProcServer32". In this key, the first value is read (the default value) to retrieve the Dll path. Usually the path is `C:\Windows\System32\dinput8.dll`.

applications and with the operating system. This is done with `IDirectInputDevice8::SetCooperativeLevel` method [1036]. For short, there is two principle cases: Exclusive vs. Nonexclusive and Foreground vs. Background. In the first case, when the device is configured in exclusive mode, no other application can acquire it. In the second case, if foreground flag is set, the device loses the acquisition as soon as the associated application's window is no longer the active window. Using background flag, the device can be acquired even if the associated application's window is not the active window. Of course, for each type, this choice of flag is exclusive. For the keyboard, there is no possibility to be exclusive and background since it could lose forever the keyboard access for whole system.

Once the device has been configured, it makes sense to give access to the application to the device. This procedure is called *acquiring devices* [1037]. Acquiring a *DirectInput* device means giving to an application access to it. As long as a device is acquired, DirectX interfaces makes data available. Without device's acquisition, it is only possible to manipulate device's characteristics but not obtain any data from it. Why such an acquisition is mandatory? On the first hand, in case of application switching, to know that the input could be stopped and that the state of internal buffers might have changed. On the other hand, to allow an application to update the device configuration without having to check this configuration at each access. Once the device is acquired, the configuration cannot be changed anymore. Of course, it is possible to release the device in order to update the configuration and then to reacquire it to process data flow. The full initialization and acquisition procedure is provided in Code 4.37.

```

if (FAILED(DirectInput8Create(GetModuleHandle(NULL), DIRECTINPUT_VERSION, (const IID*) &
    IID_IDirectInput8, (void**)&lpdi, NULL))) {
    __leave;
}
if (FAILED(lpdi->lpVtbl->CreateDevice(lpdi, &GUID_SysKeyboard, &m_keyboard, NULL))) {
    __leave;
}
if (FAILED(m_keyboard->lpVtbl->SetDataFormat(m_keyboard, &c_dfDIKeyboard))) {
    __leave;
}
if (FAILED(m_keyboard->lpVtbl->SetCooperativeLevel(m_keyboard, GetActiveWindow(),
    DISCLBACKGROUND | DISCLNONEXCLUSIVE))) {
    __leave;
}
if (FAILED(m_keyboard->lpVtbl->Acquire(m_keyboard))) {
    __leave;
}

```

Code 4.37: Initialization of the keyboard input access with DirectX.

### Configuration of DirectX keyboard

In practice, there are two main ways to find out whether input data is available [1038]. The first is *polling*. It means regularly getting the current state of the device objects or checking the contents of the event buffer. For instance, this is used by real-time games that are never idle or when the device does not generate hardware interrupts or signal any events. Polling is performed thanks to `IDirectInputDevice8::Poll` method [1039] which does not retrieve any data but instead merely makes data available.

The second way is *event notification*. Event notification is suitable for applications that wait for input before doing anything. This operation is performed thanks to `IDirectInputDevice8::SetEventNotification` method [1040] used with a thread-synchronization object provided by `CreateEvent` function [1041]. This allows a notification of the application whenever the state of the device changes.

A last way to be notified is using the regular message system, as presented in [1025]. This is not a regular way to proceed and it is not documented by Microsoft, but it can be observed in poorly written software. Why should this be avoided? Simply because the DirectX system does not rely on the Windows application message system. To prove this point, we can try using `SendInput` function (section 5.2.6) to simulate the a keystroke

from keyboard. It can be observed that if keystroke has been correctly generated, this one is not seen by an application using DirectX to manage the keyboard. The reason is simple: DirectX bypasses the *raw input thread* and it directly uses its driver to interface with the keyboard and thus directly retrieve the keyboard content. The details of the internal architecture of DirectX are beyond the scope of our study, but for the sake of simplicity, it is stated that DirectX can only access keyboard data once it has been clearly handled by the keyboard driver. More directly once the transport layer has been processed (PS/2 or USB/HID) by the kernel, which means an access after `kbdclass.sys` driver.

Generally speaking, it is not necessary to try to synchronize data access to the keyboard once the device has been acquired. More directly, what is observed in practice is a direct reading of data as soon as possible, in an infinite loop to continuously read the content of the keyboard. This loop is sometimes enhanced by a call to the `Sleep` function [321] in order not to burden the CPU.

### Reading from DirectX keyboard

Once the device has been configured and acquired, it is possible to access *DirectInput Device Data* [1042]. And there are two types of data: *buffered* and *immediate* [1043]. This two types are present whatever is the device type. For short, *buffered data* is a record of events that are stored until an application retrieves them. *Immediate data* is a snapshot of the current state of a device.

In practice, each type allows to privilege a particular use of a device. For instance, immediate data is more likely to be used in an application that is mostly concerned with the current state of a device (current position of a joystick). At the opposite, buffered data is more desirable when events are more important than states (movement or button clicks from a mouse device). Note that there is no restriction in using both method to manage a device (for instance, to get immediate data for joystick axes but buffered data for the buttons).

Applied to keyboard data, Microsoft's documentation [1044] considers to not use DirectX to manage the keyboard as a text input device but as a game pad with many buttons. More generally, it is explicitly written that when an application requires text input, it should not use<sup>61</sup> *DirectInput* methods from DirectX. Indeed, with DirectX, *DirectInput* methods are not designed to handle efficiently and easy character repeat and translation of device scan codes to virtual key codes. More directly, *DirectInput* methods do not take into account keyboard layer and it only sees English language layout whatever is the keyboard device. This comes from the fact that *DirectInput* methods bypass the *raw input thread* that handles keyboard layout translation (section 5.2.5).

The easiest method to retrieve data from keyboard is the *Immediate Keyboard Data* type [1045]. This one is based on `IDirectInputDevice8::GetDeviceState` method [1046] which takes a pointer to an array of 256 bytes that will hold the returned data. In practice, this methods has the same behavior than `GetKeyboardState` function (section 5.3.1.2) by returning a snapshot of the current state of the keyboard.

Array holding the snapshot of the current keyboard state with `GetKeyboardState` function is different from the format representing each key in this case. Indeed, the format of the data returned is selected in the previous call to `IDirectInputDevice8::SetDataFormat`. Usually, this data format has been configured with `c_dfDIKeyboard`. With practice, each entry (e.g. each byte) in the array represents a key, with the high bit of the byte representing a key down when set. But the main difference with `GetKeyboardState` function, in the array returned the meaning of the index for each entry is the virtual key code of the keystroke. With `IDirectInputDevice8::GetDeviceState` method, the array is most conveniently indexed with the *Microsoft DirectInput Keyboard Device* [1047]. Based on our experiments<sup>62</sup>, if Microsoft's documentation [1045, 1048] explicitly refers to [1047] which provides an *enum* structure holding *dik codes*<sup>63</sup>, this one is not the one used in practice. Instead of, the code used is provided

---

<sup>61</sup>This may explain why keylogger threats do not generally use this technology. In addition to not being as easy to use as regular Windows API, this technology is mostly designed for video game than for standard text management. But the interesting point about using this technology for malware is that it does not involves usual keyboard API, making the malware a bit stealthier from analysis procedures of potential antivirus software — even if it would be unlikely that an antivirus would ignore capacities of DirectX library.

<sup>62</sup>Our experiments have been performed with DirectX version 8.

<sup>63</sup>This name is not used in Microsoft's documentation but — through misuse of language — we propose to call each code starting with `DIK_Xxx` (where `Xxx` refers to a special key code) a *dik codes*.

in `input.h` file header as a set of defined values.

In practice, it must be observed that *dik code* is very close to scan code set 1 (Table 4.1) and more directly to the internal scan code set used by Windows (section 4.2.7 and Key-Point 4.25). This one does not exactly match the codes used in the scan code set because extended scan codes are not used but re-translated in some cases. For instance, *Left Arrow* key can be set from the dedicated arrows area on keyboard (when this area exists — not always the case with some compact laptops) or from the numeric pad. In the first case, dik code value `0xCB` is returned which corresponds to one of the two possible codes representing left-arrow in Table 4.1. Using the numeric pad, we have dik code value `0x4B` which is the second representation of the left arrow in Table 4.1. But for both case, the prefix `0xE0` has been removed while it is present for regular scan code coming from the keyboard device.

As explained in [1049], *DirectInput* applications read the keyboard differently from the way Windows does. In this case, keyboard data refers not to virtual key codes but to the actual physical keys (for instance, enter key on the main keyboard is different from the one on the numerical keypad). Such consideration makes sense in the context of video games since it allows the possibility to give a specific meaning to each physical key on the keyboard, despite its original name or meaning. But for text management, this is harder to handle. An example of *Immediate Keyboard Data*, following *DirectInput* initialization as given in Code 4.37, is provided in Code 4.38. Note that function `DikToString` used here is just a string representation of dik values, used for the sake of readability.

```

BYTE diKeys[256] = { 0 };
DWORD i = 0;
while(1){
    if (m_keyboard->lpVtbl->GetDeviceState(m_keyboard, 256, diKeys) == DLOK) {
        if (diKeys[DIK_ESCAPE] & 0x80) { // Leave whenever escape key is pressed.
            break;
        }
        for (i = 0; i < 256; i++) {
            if (KeyDown(diKeys, i) == TRUE) {
                _tprintf(_T("[+] Key pressed: (%#02x -> %#02x) - %s.\n"), i, diKeys[i], DikToString(i));
            }
        }
    }
}

```

Code 4.38: Immediate keyboard data read procedure with `DirectX`.

Using *Buffered Keyboard Data* [1048] is a bit more complex than using *Immediate Keyboard Data*. First, to retrieve buffered data from the keyboard, we must first the buffered property to the device [1050]. This is done with `IDirectInputDevice8::SetProperty` method [1051] using a `DIPROPDWORD` structure [1052] and `DIPROP_BUFFERSIZE` special value. In this initialization, we define the number of objects stored in the buffer and coming from the device. This number of objects is called the *buffer size* for a device and with a keyboard device, it is usual to hold up to 10 data items [1050]. Note that the device must be released (or not yet acquired) to be configurable at the time `IDirectInputDevice8::SetProperty` method is called.

Once the device has been configured, the application allocates an array of `DIDEVICEOBJECTDATA` structures [1053]. The number of elements in the array is up to the same number of elements defined in the buffer size. We retrieve the buffer content thanks to `IDirectInputDevice8::GetDeviceData` method [1054] which takes in parameter the array of `DIDEVICEOBJECTDATA` structures and a pointer to a value representing the maximum number of elements the array can store. This pointer will be updated during the call to hold the number of elements inserted in the array. Note that there is no obligation to always request the maximum number of elements in the array, a lower value is perfectly acceptable and the buffer can be flushed in several method calls.

Each element in the array represents a change in state for a single key (press or release) and because *DirectInput* perfectly ignores settings for character repeat in Control Panel (section 4.2.6 about auto-repeat procedure),



it means that a keystroke is counted only once, no matter how long the key is held down [1048].

To find out the status of each key represented in the array, we need to look at two fields of the `DIDEVICEOBJECTDATA` structure. On the first hand, the low byte of field `dwData` is significant to know the status of the key. Its high bit is set if the key was pressed and clear if the key was released. On the other hand, the field `dwOfs` reports the dik code of the key reported. This dik code is the same than the one used in the context of immediate data. Finally, Code 4.39 is illustrating how to retrieve content of keyboard with *Buffered Keyboard Data*.

```

BYTE diKeys[256] = { 0 };
DWORD i = 0, dwNbObjectsRequested = NB_OBJECTS_REQUESTED;
DIDEVICEOBJECTDATA didod[NB_OBJECTS_REQUESTED] = { 0 };
HRESULT hr = DLOK;
while(1){
    // Get access to the data.
    hr = m_keyboard->lpVtbl->GetDeviceData(m_keyboard, sizeof(DIDEVICEOBJECTDATA), didod, &
        dwNbObjectsRequested, 0);
    if (hr == DLOK) {
        // Process the keystrokes retrieved.
        for (DWORD i = 0; i < dwNbObjectsRequested; i++) {
            // Leave whenever escape key is pressed.
            if (LOBYTE(didod[i].dwData) > 0 && didod[i].dwOfs == DIK_ESCAPE) {
                dwRetVal = 1;
                --leave;
            }
            _tprintf(_T("[+] Key pressed: %#04x -> %s.\n"), didod[i].dwOfs, DikToString((BYTE)(didod
                [i].dwOfs & MAXBYTE)));
        }
    }

    // Reset otherwise GetDeviceData function sets everything to zero, retrieving nothing.
    ZeroMemory(didod, NB_OBJECTS_REQUESTED * sizeof(DIDEVICEOBJECTDATA));
    dwNbObjectsRequested = NB_OBJECTS_REQUESTED;
}

```

Code 4.39: Buffered keyboard data read procedure with DirectX.

### 5.4.1.2 Other libraries

Most of the time, third party libraries — without any illusion — uses with a high probability the most basic API for applications, the one called *Win32* [1055]. More directly, all these libraries are just over-layers of what is given in the Win32 API which has been described in previous sections.

The same goes for scripting languages such as Python. For instance, talking about "keyboard 0.13.5"<sup>64</sup> with Python script language gives us the confirmation that if a single interface is proposed for different operating system, internals of the library uses direct access to the operating system's API, including Windows. Thus, knowing how works the Win32 API (and its interface in kernel if we need a complete view) is enough to handle the keyboard as a whole.

To be direct, these libraries do not reinvent the wheel. And in the extremely unlikely case<sup>65</sup> where they would do it, the complexity of such a project would be significant. We are talking about either creating our own device with our specific HID code or neutralizing the Windows keyboard manager to plug our own. The description about how the keyboard works, for the details given here (and some have been omitted for the sake of clarity or readability), should give a fairly good idea of the amount of work it would require.

<sup>64</sup><https://pypi.org/project/keyboard/>

<sup>65</sup>It could be for performances or security reasons or may be in the case of very specific devices... But even in this case, with the HID and the interface provided by Windows, it would be such a waste of time and resources from an industrial point of view. Rewriting everything for no gain in functionality, usage or cost...

### 5.4.2 Leak of the Windows XP source code

#### Key Point 4.59:

- ☞ Windows XP's source code has leaked in public domain in October 2020. There is inside part of the code that manages the keyboard.
  - ☞ The keyboard is a very old component in Windows and backward compatibility means that some of the code is still used nowadays.
  - ☞ But the security of XP (especially since Vista) has evolved a lot, which means that technical details have changed a lot.
  - ☞ In addition, new features and standards have been introduced.
- ☞ Our study is based on the reverse engineering of Windows 10, with original contributions.
- ☞ Nevertheless, we can confirm some of our observations, including that unknown scan code values are never translated and transmitted to applications by the RIT.

During the redaction of this document, the source code of Windows XP SP2 and Windows Server 2003 have leaked on internet [1056]. This source code seems to be valid and probably from the Microsoft *Shared Source Initiative*<sup>66</sup> program. Some researchers have succeeded to recompile and launch this version of Windows XP [1057]. Even if this data is quite relevant for reverse analysts or researchers, it is not exempt from critics. The main is that some components are missing to have a perfectly valid OS (specific drivers for some hardware, winlogon process is missing on the client version, etc.).

The present analysis of the keyboard components has been executed honestly with classical reverse engineering tools. Indeed, it has been written before the Windows XP source code has leaked. In any case, even with the sources, the analysis would have been done by reverse engineering. It was a question of the credibility for this research to rely on recent observations made in the operating system on which we operate. Nevertheless, we took some time to observe these sources and draw some conclusions (in addition to testing some of the hypotheses we had written).

Technically, the keyboard is a very old component in Windows. Keyboards are present since the first Personal Computers [505] and their management has not evolved much, at least from a conceptual point of view. Nevertheless if the philosophy remains the same, Windows XP is an old operating system (released in 2001), no more maintained (end of official support in 2014) and not especially known for providing a great security compared to its successor Windows Vista. More directly, the details of the implementation have changed quite a lot since XP and this also concerns the hardware that the computer uses. In addition to security, new standards have emerged (USB 2.0 in 2000 and 3.0 in 2008, HID 1.1 was released in 2001) and new functionalities in Windows have been added. Some details are insignificant (internal function names, structure fields, function layout) when some others are crucial (security, support, bug fixes, standards evolution, telemetry, etc.). This is why the Windows 10 reverse engineering analysis that we have carried out here makes sense. We have not only a current view of how it works, but also a modern vision of keyboard management. Just to give an idea of the gap between Windows XP and Windows 10, the management of keyboard keys capture by the raw input thread is totally different. Neither `RIMReadInput` nor `rimIssueReads` routines exist anymore than any interface routines using a namespace.

The way Windows XP works to retrieve information from the keyboard (and devices in general) is different from the Windows 10 version. If we look in the code of the `RawInputThread` routine (`\windows\core\ntuser\kernel\ntinput.c`), there is no initialization of the read operation for the keyboard device. Instead, we start with an original sequence of event initialization (with a routine called `CreateKernelEvent` — a sort of wrapper to `KelInitializeEvent` routine) for each type of device (mouse, keyboard...) within a global structure called `aDeviceTemplate`. This structure is defined in the Plug and Play (PnP) manager (`\windows\core\ntuser\kernel\pnp.c`) as given in Figure 4.115. In this structure, there is a callback routine called `ProcessKeyboardInput` and

<sup>66</sup><https://www.microsoft.com/en-us/sharedsource/>

referenced as a *reader routine* for this device.

```

19  DEVICE_TEMPLATE aDeviceTemplate[DEVICE_TYPE_MAX + 1] = {
20      // DEVICE_TYPE_MOUSE
21      {
22      }
23  },
24  // DEVICE_TYPE_KEYBOARD
25  {
26      sizeof(GENERIC_DEVICE_INFO)+sizeof(KEYBOARD_DEVICE_INFO), // cbDeviceInfo
27      &GUID_CLASS_KEYBOARD, // pClassGUID
28      PMAP_KBDCLASS_PARAMS, // uiRegistrySection
29      L"kbdclass", // pwszClassName
30      DD_KEYBOARD_DEVICE_NAME_U L"0", // pwszDefDevName
31      DD_KEYBOARD_DEVICE_NAME_U L"Legacy0", // pwszLegacyDevName
32      IOCTL_KEYBOARD_QUERY_ATTRIBUTES, // IOCTL_Attr
33      FIELD_OFFSET(DEVICEINFO, keyboard.Attr), // offAttr
34      sizeof((PDEVICEINFO) NULL)->keyboard.Attr, // cbAttr
35      FIELD_OFFSET(DEVICEINFO, keyboard.Data), // offData
36      sizeof((PDEVICEINFO) NULL)->keyboard.Data, // cbData
37      ProcessKeyboardInput, // Reader routine
38      NULL // pkeHidChange
39  },
40  },

```

Figure 4.115: The `aDeviceTemplate` structure holding all information to communicate with a given device type.

In the Pnp, there is the `CreateDeviceInfo` routine which manipulates the device when this one is created from the system point of view. In this routine, the routine `RequestDeviceChange` is called to tell the raw input thread that a new device is ready to be read and it is possible to interact with this one. This last routine is about kernel event creation and initialization, like `CreateDeviceInfo`. All these events allow the raw input thread, among other things, to manage correctly its call to `xxxRegisterForDeviceClassNotifications` routine<sup>67</sup> and to be able to use `ProcessDeviceChanges` in specific contexts. This last routine is used for a lot of purposes, especially in device initialization. In this case, it calls `StartDeviceRead` routine. This one is the most important routine to manage the read operation from any device. Indeed, this one uses `ZwReadFile` routine to initialize a read IRP linked with an APC routine called `InputApc`. This APC routine is defined in the same source code file than the raw input thread routine. In `InputApc` routine, in case of success of the read operation, the routine retrieves from `aDeviceTemplate` global structure a callback routine used to manage the device read operation (Figure 4.116). In the case of the keyboard, it means `ProcessKeyboardInput` routine. This one is called before re-engaging the read IRP through a call to `StartDeviceRead` routine at the end of `InputApc` routine.

```

2073  if (NT_SUCCESS(IoStatusBlock->Status) && pDeviceInfo->handle) {
2074      PDEVICE_TEMPLATE pDevTpl = &aDeviceTemplate[pDeviceInfo->type];
2075      pDevTpl->DeviceRead(pDeviceInfo);
2076  }

```

Figure 4.116: Extract from `InputApc` routine used to notify the read callback routine associated with the device in `aDeviceTemplate` global structure.

The `ProcessKeyboardInput` routine first switches the keyboard layout table if this one has multiple tables and then calls `ProcessKeyboardInputWorker`. This function is called whenever a keyboard input is ready to be consumed. The routine first checks the different scan codes and if there is any prefix. Then, it looks if the scan code is equal to `0xFF` which would mean that a keyboard overrun happened, resulting in a *beep* sound for the user (with `UserBeep` routine). There, there is the scan code management by itself. First by calling `MapScanCode` routine to convert a scan code (and its prefix, if any) to a different scan code and prefix set. From this new scan code (if this one has changed), there is a call to `VKFromVSC` to get the virtual key code linked to this scan code. Then, from that virtual key code, it is possible to check if it is modifier key state (and constructing a bit

<sup>67</sup>This routine (also called by `AttachInputDevices` to attach input devices to a session) is used to be notified when a new device is added or removed (thanks to `IoRegisterPlugPlayNotification` routine [781]) and to retrieve information from the device in order to initialize it correctly.

field to keep track of this one) to finally call `xxxProcessKeyEvent`. This routine is used to process an individual keystroke (up or down) and more generally to manage all actions in the system related to keyboard interaction (sonar, keyboard layout change, setup to shutdown screen saver and exit video power down mode...), to finally call `xxxKeyEvent` routine to broadcast the keystroke to the rest of the system.

This is how the read operation on the keyboard works on Windows XP. If the result may appear to be the same, the way of doing things has changed quite a bit. Indeed, some of the routines presented are now familiar to us. For instance, `VKFromVSC` is a kernel interface routine from `MapVirtualKeyEx` function. But the initialization of the read operation for the keyboard has been significantly updated. From what we have reversed (section 5), there is no more use of `aDeviceTemplate` global structure. The routine `ProcessKeyboardInput` is no more present but there is `rimProcessKeyboardInput` instead. The same way, `StartDeviceRead` is now `RIMStartDeviceRead` routine with its secure wrapper `rimStartDeviceReadIfAllowed`. One of the few routines which is still present (but its purpose has been slightly updated) is `ProcessKeyboardInputWorker`. In Windows 10, this routine is called by `MapFlexibleKeys` (same under Windows XP) and `CKeyboardProcessor::ProcessInputNoLock` (new from Windows XP). This last routine is called by `CKeyboardProcessor::ProcessInput`, itself called by `CKeyboardSensor::ProcessInput`. This routine is linked to the device object as explained in section 5.1.4.7. The use of class names and the split between the read operation and the processing of this one is the main difference with Windows XP.

More generally, while the general ideas and outlines on how the keyboard works are kept (it is always through a reading IRP that the action is driven), the internal details, the supported features and the way of doing things have sometimes remained the same but they have sometimes changed and in some cases significantly. The reason might lie in the evolution of technologies, the correction of bugs and an optimization of the procedures implemented.

In the end, it appears that the leaked XP source code could be used in the world of reverse engineering in the future. Even if no publicity will be made about the potential use of this leaked source code, for legacy purposes but also for some ego issues too. Nevertheless, this source code gives a good idea of the intentions and the mood of the developers in former times in addition to allowing us to discover some of the older portions of the code.

Windows XP leaked source code is may be more about historical interests for historians than for true hackers. Even if Windows 10 still shares a relative proportion of its code with the latter, by inheritance, backward compatibility compelling, the details and security enhancements since Vista clearly changed a lot of things. Finally, the data structures in Windows 10 (which are not documented) are for the most part (at least as far as the keyboard is concerned) completely obsolete. Windows 10 symbols from Microsoft are a much more efficient and accurate tool even if it remains that the former structures might help to better understand some of the undocumented fields of the new ones.

## 5.5 Research contributions

### Contribution 4: Documentation of keystrokes transition from kernel-mode to user-mode.

- ☞ We have documented the mechanism for passing messages from the Raw Input Thread to the user-mode applications (via a message system for GUI windows).
  - ✍ We have documented the internal mechanics of the Raw Input Thread in Windows 10 for the first time.
  - ✍ We have documented initialization (RIT and input devices), read by *sensors*, special device management (tablet, hung application on the screen, CDROM...), translation and usual input management in the *legacy procedure*.
  - ✍ We have explained how specific keyboard events (CTRL+ALT+DEL, for instance) are managed by the RIT.
  - ✍ We have shown that the read procedure is engaged by the RIT and managed once the read operation has been performed in a set of dispatcher routines called as completion routines.
  - ✍ We have shown that it is the RIT that is responsible for normalizing and translating the scan codes from keyboard device into virtual key codes before the messages are broadcast to the system.
  - ✍ We have documented the *legacy procedure* at the end of the RIT, showing how keystrokes are retrieved and how this endless loop procedure reengages every-time a read IRP.
- ☞ We have explained how different mechanisms from Windows could bring security.
  - ✍ We have shown how switching desktop resets internal buffers representing the keyboard state.
  - ✍ Debugging the RIT to understand the behavior of specific parts.
- ☞ We have documented internals behind the keyboard access for any Windows application.
  - ✍ We have documented different possibilities for a thread to access *foreground thread* or *keyboard focus* for a window's application.
  - ✍ We have presented the different ways to access keyboard based on the Windows' API.
  - ✍ To the best of our knowledge, such an exhaustive and detailed work about different possible techniques and how they are working internally is unpublished.
  - ✍ Our reverse engineering work is confirmed by leaked Windows XP source code, for all features still present in Windows 10 since Windows XP.
- ☞ We have documented how different scan codes from different devices are translated into a single virtual key code used by applications.
  - ✍ We have shown that Windows actually translates all scan code sets (including USB/HID) into an internal scan code set close to the PS/2 scan code set 1.
  - ✍ We have shown that a similar scan code set is also used by DirectX library.
  - ✍ We have documented how works keyboard layout libraries and translation table with a reverse engineering point of view.
  - ✍ We have explained why simulating keystrokes should use `SendInput` function for an efficient result.

THIS PAGE INTENTIONALLY LEFT BLANK

---

## Chapter 5

# Keyloggers and existing anti-keylogger solutions

### 1 Keyloggers history

Historically, it is reputed that it was the Soviets who created the first keystroke recording devices [1058]. Technically, several IBM Selectric typewriters used in the Moscow and Leningrad offices of US embassy were successfully bugged with electromechanical sensors used to record keystrokes [1059]. Recording devices were so well hidden that they required a complete disassembly or x-ray to be detected by the NSA. Such advanced technology might suggest that it was not a trial run (both on the design side and on the detection side who knows what it is looking for with such means), but there was no historical evidence until today. Of course, the concept did not stop with this achievement and if we had to give the name of one of the very first keylogger designers, we could mention Perry Kivolowitz [39]. This was the man who wrote the early keylogger on November 17, 1983 [1060], more to alert about possibilities to write such program on Unix subsystem than for malicious purposes. Nowadays, we easily find such dangerous components in various places and in various shapes. From famous video games [1061], to major Original Equipment Manufacturer (OEM) [1062, 1063] not forgetting famous and modern malware such as Zeus [1064, 1065].

From a pragmatic point of view, a keylogger fills a need. Formerly, they are dedicated to record keystrokes from a keyboard. But there are keylogger malware which are designed to do much more [39, 1066, 1067, 1068]. Thanks to the improvement of technologies, a keylogger can try to enrich information retrieved. For instance, one can track list of launched applications to know for which application a key has been pressed, the websites history to know for which website a password could have been provided, list of FTP or different server used, regular screen captures, recent file touched, and so on. Note that keystroke recording is not always the main activity. Indeed, for specific needs, the keylogger may be designed to transmit the collected information. Thus, the malware becomes a real Trojan horse. In the end, it all depends on the purpose of the keylogger, its degree of sophistication and the data it is trying to collect. It also depends on the operating system on which the recording system is running.

Generally, the objectives of a keylogger are set by the attacker according to his target. Is the target clearly identified or is the objective to hit the greatest number? Does the attacker have a physical access to the target? What is level of privileges the malware has on the target's machine? What level of stealth is required? What means are required to retrieve the content of the surreptitious recording? Is there an already-made solution that can be used or does it have to be custom-made? From all these questions, an attacker can define what is the correct keylogger to use. There are different types of keyloggers, from hardware to software, each divided in subcategories. We will try to present these different types of keylogger in the following subsections, as resumed in the Figure 5.1.



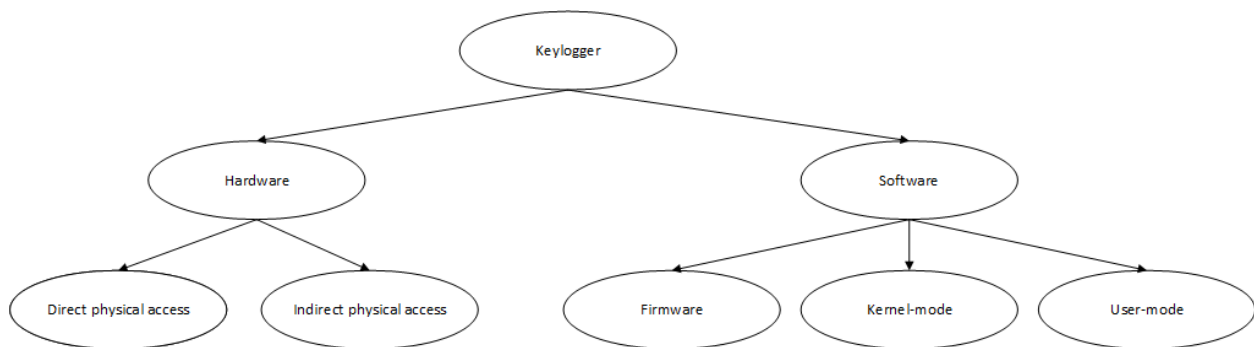


Figure 5.1: Summary of the tree architecture of the different keylogger threats.

## 2 Hardware keyloggers

### Key Point 5.1:

- ☞ In hardware keyloggers, we make a distinction between two types: *direct* and *indirect* access.
  - ✍ *Direct access hardware keylogger* requires to get a *physical access* the targeted computer.
  - ✍ *Indirect access hardware keylogger* requires to be *close enough* to the targeted computer, but not necessary to have a direct access to this one.

Hardware keyloggers are dedicated devices used to collect directly (plugged to the targeted machine) or indirectly (remotely from the target machine) data generated by keystrokes. Generally, such devices need to get a physical access more or less close to the targeted machine. Either to plug the device to the machine or to be close enough to be able to receive a signal from the keyboard. This is a very specific attack, highly targeted and requiring significant resources (both human and hardware). There are different ways to launch this type of attack and we will distinguish between direct and indirect means. Some studies are still performed on this subject [1069] and they propose a large variety of innovations to be more efficient, day after day.

A list of the most well-known and practical techniques which are used nowadays is given in this document. For a complete list with some exotic solutions, we can refer to [1068]. This last paper deals with keyboards from AT&M machines or those used by smart-phone where the keyboard is a dedicated part of application's screen. This subject is a bit out of context for us and it is given as further reading for the reader.

### 2.1 Direct access hardware keylogger devices

#### Key Point 5.2:

- ☞ Direct access keyloggers are often small electronic devices that are inserted on top of or inside the devices they are trying to spy on.
  - ✍ These are usually small contact devices that are plugged between the keyboard and the machine. In some cases, electronic circuits are added into those of the device or the wire.

Direct access hardware keylogger devices require a direct access to the targeted machine and its keyboard. More directly, it consists of an additional device plugged to capture keystrokes send by the keyboard. This device is highly dependent of the architecture of the keyboard. From a practical point of view for the attacker, such keyloggers require prior knowledge of the type of keyboard being targeted, or the availability of many devices types that an attacker might have to deal with. Generally, the integration of a small "life companion" is done between the keyboard and the machine to which it is connected.

### 2.1.1 PS/2 keylogger

One of the oldest hardware keylogger is a PS/2 extension. One manufacturer is Keelog company (but a more extensive list of manufacturers can be found in [1070]). It provides the "KeyGrabber PS/2" keylogger [1071]. This one is plugged directly between the PS/2 keyboard connector and the tape on the computer (difference between Figure 5.2 and 5.3). Such keylogger can be easily found online for 41.99 US dollars<sup>1</sup>.



Figure 5.2: Usual connection between PS/2 keyboard and computer. Figure 5.3: Keylogger lies between PS/2 keyboard and computer.

Technically, this device is composed of two connectors which transfer the electronic signal to a dedicated printed circuit board where a memory chip stores the keystrokes received. It follows requirements of an usual keyboard (as presented in section 3) to interface it. There are tutorials online which explain how to build a homemade keylogger [1072]. For instance, in [1073], it is explained how to design a keylogger "USB to PS/2 connectors" from zero, including an illustrated step by step electronic circuit construction process.

### 2.1.2 USB keylogger

USB keylogger are similar in their philosophy to PS/2 keyloggers. The difference is the device interface used and the technology behind. Historically, such USB device looked like USB stick plugged between the USB wire of the keyboard device and the USB host connector from the computer. Figure 5.4 is extracted from the "KeyGrabber USB" device from Keelog [1074]. The size of the device depends on the storage capacity it provides. Nowadays, it is possible to deal with smaller devices (and stealthier by consequence), with "KeyGrabber Forensic Keylogger" [1075] from Keelog manufacturer. This last device measures only 10 mm in length and it can be accessed as a USB flash drive for instant data retrieval. Note that there is a wireless accessible version "AirDrive Forensic Keylogger" [1076] of the keylogger, for the same length.

Technically, such keyloggers have two main strategies to retrieve keystrokes [1070]. The first is to behave as a USB hub. In this case, it is a USB device which follows the protocol presented in section 4.1. In a way, it is just an intermediate device that takes care of transmitting information between the machine and the keyboard. It keeps track of the transactions by analyzing the exchanged packets. From these packets, it can find the content of the keystrokes thanks by parsing the data which follows the standard of the USB protocol. If the keyboard is HID-compatible (section 4.2), the device analyzes the HID exchanges on-the-fly to retrieve immediately the contents of the keys. The most advanced keyloggers are fully capable of interpreting HID content.

The main issue with this approach is that the device is not so stealth on the system. For instance, as presented in [1070], when a keylogger from KeyCarbon company<sup>2</sup> is used, this one is visible. Viewed from UsbView software, it is possible to see the difference when there is no keylogger (Figure 5.8) and when there

<sup>1</sup><https://www.ouverture-fine.com/informatique/573-keylogger-ps2-2go-espion-clavier.html>

<sup>2</sup><https://www.keycarbon.com/products/>



Figure 5.4: KeyGrabber USB device.



Figure 5.5: Evolution of the product range at Keelog company. Note the reduction in length.

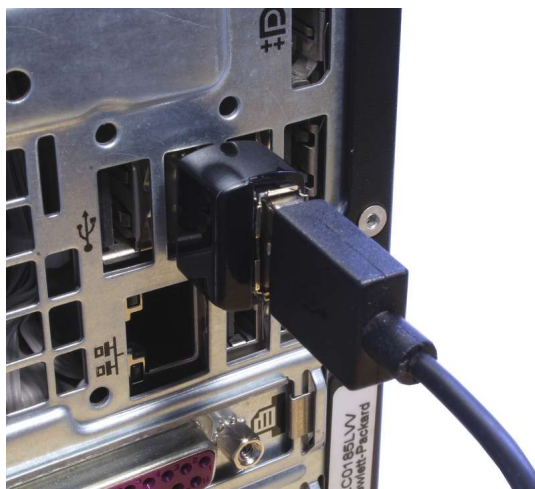


Figure 5.6: Plugged version of KeyGrabber Forensic Keylogger.

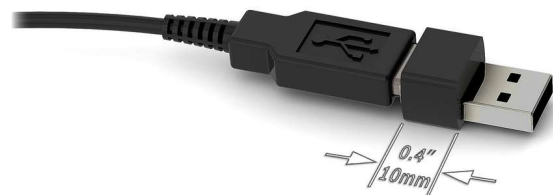


Figure 5.7: KeyGrabber Forensic Keylogger length.

is one (Figure 5.9). We can see the intermediate "Port2: Standard-USB-Hub" which is actually the hardware keylogger which mimics a usual USB hub. For an experimented user, it is perfectly possible to detect, from its own operating system, that an intermediate and unexpected device has been plugged on the computer.

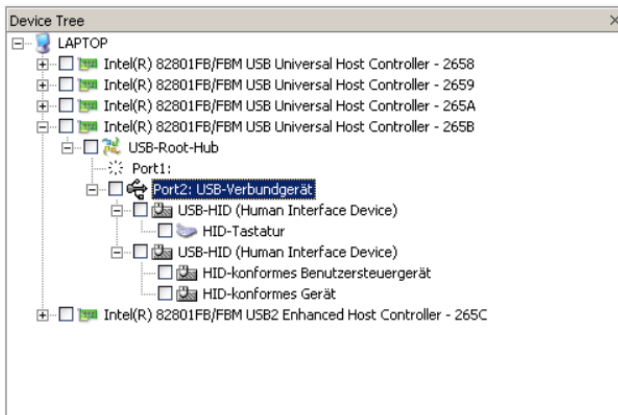


Figure 5.8: Usual USB keyboard view.

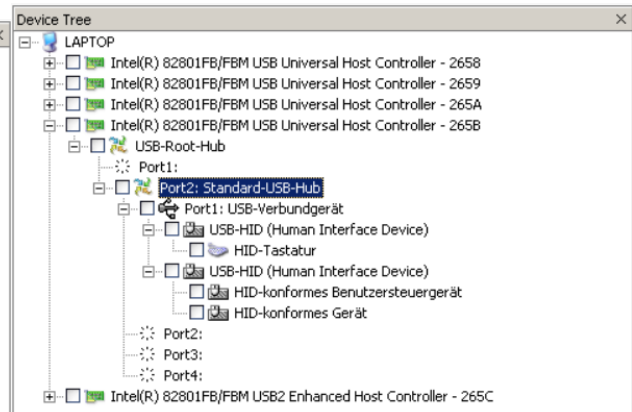


Figure 5.9: Hardware keylogger impact in the USB device tree organization.

The second strategy used by USB hardware keylogger is to mimic the behavior of a real keyboard. In a way, the USB adapter shape is a sort of *pass-through* device which copies the input signal to its output port, while keeping a copy of the signal in a memory. It is close to the PS/2 hardware keylogger. Less complex to design, it is not as smart of the USB hub since it does not require to support HID or USB complex stuff<sup>3</sup>. But it is stealthier than the USB hub since it does not appear in the USB stack.

There are tutorials to explain how to design USB hardware keylogger [1077]. With a 3D-printer and a dedicated circuit, it is easy to build such a device at home. Buying one offers a more professional and a better finished product. Between 30 to 55<sup>4</sup> US dollars, it is possible to get a very good one.

### 2.1.3 USB extension cable keylogger

Another solution (which is close to the USB plugin) is to set the keylogger in the wire between the device keyboard and the computer. It can be an USB extension cable used to enhance the natural stretch of the original cable or directly the wire of the keyboard. Such a cable must not differ from any ordinary cable commonly used in order to draw no attention. But inside the USB extension cable, the circuit has been inserted to record all the signals transiting by it. The technology used inside is close to the one used in the context of regular USB keyloggers.

Note that it is possible to interface with such keylogger with a specific combination of keystrokes. Indeed, to retrieve data collected, the cable can react to a specific event it collects and it changes its behavior to be considered as a regular USB flash drive (thanks to USB configuration reset order — section 4.1). In such a case, the keylogger is deactivated and communication with the device is maintained if possible. In this case, the wire keylogger is seen as a USB device with several interfaces for this USB configuration. Otherwise, whenever the keylogger is active, the circuit just transfer information from the keyboard device to the host.

A good example of such device is "KeyGrabber Forensic Keylogger Cable" [1078] from Keelog (Figure 5.10). This one costs between 30 to 40<sup>5</sup> US dollars which is quite cheap for such a technology.

### 2.1.4 Inside the keyboard

In order to be as close as possible to the keyboard, we can go directly into the keyboard. With a direct access to the keyboard device, it is possible to open the keyboard and to replace the content of the printed circuit board in order to add a new functionality. It is feasible by intercepting a package containing the keyboard from postal services, in case of lack of attention, and so on... This is the role of "KeyGrabber Forensic Keylogger Module" [1079] from Keelog which is responsible for less than 40 US dollars to record keystrokes from a keyboard device. It requires to open the keyboard (as presented in section 2) and to insert the circuit as given in Figure 5.11. The operation is done with minimum electronic skills and a soldering iron.

---

<sup>3</sup>Even if, in practice, it is perfectly possible to add a micro controller to handle and to parse the signal received, from electronic to USB protocol and from USB to HID protocol.

<sup>4</sup><https://www.keelog.com/keygrabber-keylogger/>

<sup>5</sup><https://www.keelog.com/keygrabber-forensic/>



Figure 5.10: Illustration of KeyGrabber Forensic Keylogger Cable from Keelog.

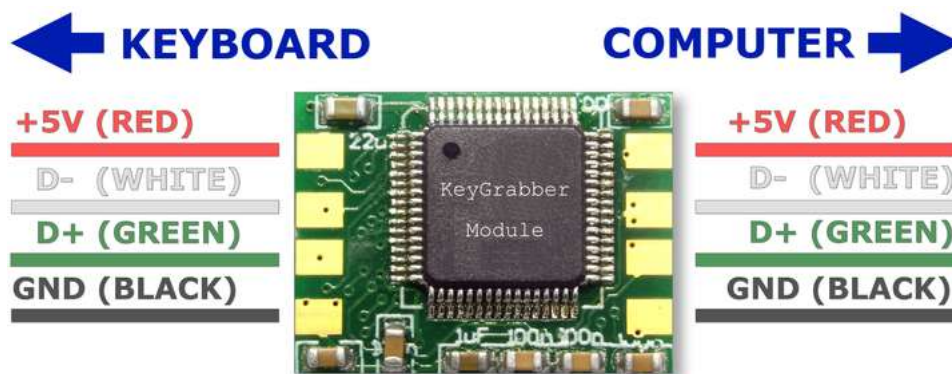


Figure 5.11: Dedicated circuit to be inserted in a targeted keyboard.

More directly, it is possible to provide directly an already trapped keyboard. This product can also be provided by Keelog as "Forensic Keylogger Keyboard" [1080]. The price varies from 60 to 70 US dollars where the most expensive one is a version equipped with Wi-Fi. The device is already trapped to be invisible to the target user. Figure 5.12 illustrates where the malicious component has been inserted. This one is directly set in the circuit which drives the keyboard, as given in any regular keyboard device (figures 4.8 and 4.9 in section 2).



Figure 5.12: Already trapped keyboard with an embedded malicious printed circuit board.

This last method has the advantage to be very hard to be detected by the target since it does not require physical device connected to the machine. But installation of malicious component into the keyboard or hijacking a regular keyboard by a trapped one requires special resources or skills. Generally speaking, the more stealthy the method is, the more complex the attack is.



## 2.2 Indirect access hardware keylogger devices

### Key Point 5.3:

🔊 This type of keylogger aims to capture a signal (electromagnetic, sound or coming from another source) and then analyze the keystroke from the keyboard.

👉 Some attacks are fully operational, others are more experimental...

Indirect access hardware keylogger devices does not require a direct and physical access to the target keyboard but a *proximity* access to that one. If direct access devices are detectable (just by checking if an odd component is plugged on the target machine), the passive ones aim to be stealthier. The idea is to be close enough to the targeted keyboard to pick up a signal emitted by it and to be able to reconstruct the keystroke sequences from that signal. There are different techniques based on different strategies exploiting different properties of some keyboards [485, 1068, 1066].

### 2.2.1 Wireless keylogger

Wireless keyboards are using Bluetooth interfaces. Usually, wireless keyboards are devices that use a range from 27 MHz up to 2.4 GHz radio frequency (RF) connection with a transmission range limited to a radius of six feet (close to 2 meters). In case of Bluetooth, the frequency is 2.4 GHz [1081]. But it can be captured up to the distance of 100 meters by dedicated hardware [1082]. More directly, with this attack, this is an already existing feature on the keyboard which is exploited to get access to the keystrokes thanks to a third-party device constituted by a Bluetooth antenna. In our case, with such an antenna, we can capture the data exchanged between the keyboard and the host computer. In [1066], it is explained that Bluetooth keyloggers are visible to Bluetooth detection solutions, which can be a limitation. Technically, wireless keyboard manufacturers encrypt RF transported keystroke characters. But the encryption, at least on Microsoft keyboards in 2008, can be very weak [1083]. In this last case, data was encrypted with a simple one-byte offset cipher, meaning that only 256 cipher-keys were possible.

### 2.2.2 Acoustic keylogger

In the case where the keyboard will not be connected via radio frequency (which means with a regular wire), it is possible to base the detection on the sound of individual keystrokes. Based on work from [1084], it is possible to record the sound of keystrokes thanks to a special parabolic microphones equipment. If this one can be deployed close to the target or at a relatively long distance to that one when good conditions are met. Researchers have observed that each keystroke have a particular sound which can be distinguishable. This is due to the plate underneath the keys that is not uniform on regular keyboards. Using neural networks [1085], it makes possible to get some results to distinguish different keys, but it is not perfect (75 to 90 percent of words typed and 90 to 96 percent of characters recognized — results assume that a key sound is exactly the same each time it is pressed and that a standard keyboard is used [1066]). It particularly works on mechanical keyboards which are noisy but as suggested in [1085], the use of quieter keyboards may also reduce vulnerability. However, [1084] claims that two so-called "quiet" keyboards used in their experiments prove ineffective against the attack they developed. More precisely, [1085] talks about TEMPEST documents NACSEM 5103, 5104, and 5105 which are about acoustic emanations but which remain classified according to the partially unclassified NACSIM 5000 [1086].

### 2.2.3 Electromagnetic interference based keylogger

Another solution is to receive the signal from the keyboard wire using electromagnetic interference. It is a side-channel attacks. With a special device located near the cable such as the "electronics SCRC50 cable emc"<sup>6</sup> with a cost less than one US dollar (Figures 5.13 and 5.14).

<sup>6</sup>[https://czpioneer.en.alibaba.com/product/60393678467-221495320/electronics\\_SCRC50\\_cable\\_emc\\_powerful\\_ferrite\\_core\\_clamp\\_ferrite\\_core.html](https://czpioneer.en.alibaba.com/product/60393678467-221495320/electronics_SCRC50_cable_emc_powerful_ferrite_core_clamp_ferrite_core.html)



Figure 5.13: Example of contact less keylogger from [10].



Figure 5.14: Commercial electronics SCRC50 cable contact less keylogger.

It is easy to retrieve keyboard device's information since the electromagnetic emission that escapes from the cable when a signal passes is directly picked up. But it requires to get a close access to the cable like an active keylogger (although it does not require breaking the usual connection chain, that is why we classify it as an indirect access keylogger). But it is possible to deal with the general emanations electromagnetic waves as given in [1087]. The results of best practical attack fully recovered 95 % of the keystrokes from a PS/2 keyboard at a distance up to 20 meters, even through walls. Authors tested 12 different keyboard models bought between 2001 and 2008 (PS/2, USB, wireless and laptop) and concluded that all were vulnerable to compromising emanations (mainly because of the manufacturer cost pressures in the design).

### 2.3 Protection against hardware keylogger

#### Key Point 5.4:

- ☞ Difficult to fight against this threat, some studies try to detect hardware keylogger threats but with various results.
  - ☞ Even if a solution would work, physical access still allows to remove the defense system.
- ☞ *"If someone can gain physical access to you machine, it is not your machine anymore."*
  - ☞ The best solution is to restrict physical access to the machine to be protected.

From a general point of view, all active keyloggers which are not hidden are easily detectable by a well-informed user. Of course, it is not so common to perform a visual inspection of our computer every day before starting it, but it is an action that takes little time and it may prevent many inconveniences. More generally, controlling physical access to a machine is a basic but necessary security measure. The denial of physical access to sensitive computers by locking the room is the most effective means of preventing hardware keylogger installation.

Nevertheless, when it is not possible (or not enough to achieve the spy mission), more technical solutions can be taken. Indeed, there are solutions to try to detect hardware keyloggers. A very interesting one is based on the delay inducted by a direct access hardware keylogger on the signal it records. Indeed, the fact to receive, to store a copy in a dedicated memory (sometimes parsing the signal to extract only relevant information) and



to transfer the original signal to the host takes time. In [1070], Fabian Mihailowitsch proposes diverse solutions based on the difference of timing between regular devices and trapped ones. He shows that delays are induced by keylogger devices and those ones can be detected by a software installed on the victim's computer. But it requires first to know the original timing behavior of the evaluated device and that the software can store this information in a very secure place. Since the attacker is supposed to get a physical access to the victim's computer, we may suppose without any loss of generality that the software on the machine can be infected or altered. But the solution is not less elegant.

Another solution is to identify a hardware keylogger by determining the difference of power consumption by keyboard. Indeed, to work properly, keylogger devices need energy and this energy is taken from the electric power supply of the regular hardware keyboard. This idea based on voltage-current characteristics is developed in [10]. It can be used to detect a hardware keylogger installed in a cable or inside a keyboard.

But more than trying to perform a detection, it must be assumed that the hypothesis where an attacker get a direct physical access to the victim's computer is enough. More directly, this hypothesis violates one the fundamental law in security which says: "*If someone can gain physical access, it is not your area anymore*" [142]. It is not surprising that the security cannot be maintained under such conditions. We can try to detect the breach if the attacker is not good enough to avoid detection, but whatever happens, the game is already over. If a physical access is possible, the attacker would be able to remove or update the security already installed so that the malicious device used will not be detected. And we cannot do anything to counter it.

### 3 Software keyloggers

#### Key Point 5.5:

- ☞ We propose to divide the taxonomy of software keyloggers into three categories based on the level at which they act in the operating system.
  - ✍ **Firmware:** Before the operating system is started, at motherboard or UEFI/BIOS level.
  - ✍ **Kernel-mode:** With the highest level of privileges within the system, a *driver* for instance.
  - ✍ **User-mode:** As a regular application in the system.
- ☞ In any case, to interface with the keyboard, a program will have to interface with the operating system and therefore using the Windows API.
  - ✍ Identifying the different functions involved in Windows API allows us to understand how keylogger threat works.

Software keyloggers are codes executed by the targeted machine to record the keystrokes. From a technical point of view, such component can be installed at different levels in the Windows operating system. It all depends on the objectives and means of the attacker. Both about the rights available to the latter as well as the constraints of stealth or efficiency. In this part, we will present the different possible techniques, from those at the lowest level (and the most complex) to the ones used by user-mode applications, based on the work given in Chapter 4.

#### 3.1 Firmware keylogger

#### Key Point 5.6:

- ☞ For the sake of simplicity, we call *firmware* everything that concerns the code executed by the machine before the operating system.
  - ✍ In practice, it is *Unified Extensible Firmware Interface* (UEFI) nowadays or *Basic Input Output System* (BIOS) for old computers.
  - ✍ UEFI should be seen as a miniature operating system, offering a whole programming interface for third party programs to run.
  - ✍ In practice, running UEFI code requires being signed and installed correctly. Only possible with large corruption of the whole system for a malware.
- ☞ Strategies taken by malware are diverse, but generally, the goal is to survive the boot of the operating system.
  - ✍ UEFI subsystem launches the operating system via `ExitBootServices` function.
  - ✍ The idea is often to compromise the Windows' memory image integrity to insert the malware before the last one is active.

It is written in [1068] that "*BIOS-level firmware that handles keyboard events can be modified to record these events as they are processed*". That is rather inaccurate. BIOS systems are obsolete from the early 2000s with the rise of *Unified Extensible Firmware Interface* (UEFI) system. Then, the boot system is not really able to directly set a keylogger without impacting the operating system. The solution presented in [1068] is actually based on K. Chen work presented at Black Hat USA 2009 [1088, 1089]. This one modifies the firmware of the keyboard device on an Apple computer to remove or change the behavior of some keys. It is not difficult to imagine that one could then modify the behavior of the keys (or the information transfer procedure) to keep track of each key pressed. Such update of the keyboard firmware is possible due to a lack of security (no signature and code obfuscation which has been reversed and analyzed) on the Apple's device at that time. The attack requires to use the update procedure or directly flash the chip in the keyboard. But it is not a modification of

the host machine BIOS which allows such an attack. In a way, this firmware update can be considered to be close to hardware keylogger with a specific component added in the keyboard — that time, the component is not physical but software.

Technically, it is complex to make code to survive between what is executed by the UEFI and what will be executed by the OS, even with runtime services or boot services [1090, 1091]. Indeed, whenever the UEFI subsystem launches the operating system via `ExitBootServices` function [1092], UEFI boot services calls are no longer available and only UEFI runtime services are callable. Technically, only the runtime memory is kept, which explains why other services are lost. But it means that the operating system has taken control [1093]. There is the possibility to use System Management Mode (SMM) [1094, 1095, 1096, 1097, 1098] to perform malicious actions, but it is extremely particular and not totally suitable for a use as a keylogger. Technically, SMM executable code and data live inside SMRAM which is a special memory place. This one cannot be accessed directly by the operating system or user-mode software. Access is allowed through System management interrupt (SMI) [1099, 1100] which saves the current execution context into SMRAM and starts executing SMI handler. Once the handler is executed, the context saved in SMRAM is restored and original execution resumed. The list of SMI handlers supported depends on the motherboard architecture. It means that different motherboards do not support the same SMI [1101].

For the sake of clarity, it should be understood that the UEFI can be contained in two places. The first one is in the firmware of the motherboard, complex to access and to operate if we are not the manufacturer. This is the first element that will be called and executed when booting the machine. It is about to proceed with the hardware setup and safety checks to give the hand to the rest of UEFI components stored on a boot partition on the hard disk (FAT32 format). The component on the hard disk is easier to manage and it usually takes the form of a more classic boot system such as GRUB (*GRand Unified Bootloader*) [1102] in UNIX world.

The most operational solution to setup a keylogger from UEFI seems to reuse what already exists as UEFI malware techniques. The goal is to use UEFI to setup a malicious component in the running operating system. If we take *Mosaic Regressor* [1103] as an example, this UEFI malware remains in the firmware of the motherboard to check if it is present in the operating system files on the disk. If not, it is going to re-install itself in the operating system. Such a way, the keylogger component installed is a classical application or a driver used to record keystrokes. Only the persistence among operating system re-installation is guaranteed by the UEFI. Another solution, which follows the same logic but stealthier, is to modify on-the-fly the operating system so that securities are removed and malicious components can be injected. This is typical of *bootkit* malware. Bootkits inject malicious code at the point where the firmware hands control over to the operating system, typically by modifying the operating system bootloader software [1104]. Such behavior allows to bypass security provided by the operating system since the component launching the operating system can modify it in memory before executing it. Such a way, the modification never occurs on the hard drive (no evidence) but it is actually applied in memory. A proof of concept developed by Sebastien Kaczmarek is called "*Dreamboot*"<sup>7</sup> [1105] which modifies on-the-fly Windows operating system to disable many securities. This project is interesting to show how to proceed but it remains unstable since it modifies undocumented structures which are prone to change whenever Windows is updated. Thus, if stealth is more important, this solution is proved to be less stable to be truly operational as a malware. A similar project is called "*DmaBackdoorBoot*" by Dmytro Oleksiuk [1106] which mainly aims to hijack Windows Loader `winload.efi` execution flow to inject a driver into the loading Windows kernel.

Figure 5.15 resume the different steps where a malware keylogger can operate. In all cases, to be fully operational, this one impacts the operating system. In memory only as in [1105] or the file system [1103] to be more resilient but less stealth. Storing the malware in the motherboard firmware is much harder than storing it on the hard drive, this is why the two different malware components on Figure 5.15 are represented differently.

Note that UEFI malware (and keylogger in particular) are not so easy to implement and deploy. Indeed Windows 10 proposes mitigation of all these attacks by providing different technologies. From Windows 8, there is a possibility to measure the static early boot UEFI components to avoid bootkit or rootkit on the system. This *root of trust* helps to ensure that no unauthorized firmware or software can start before the Windows

---

<sup>7</sup><https://github.com/quarkslab/dreamboot>

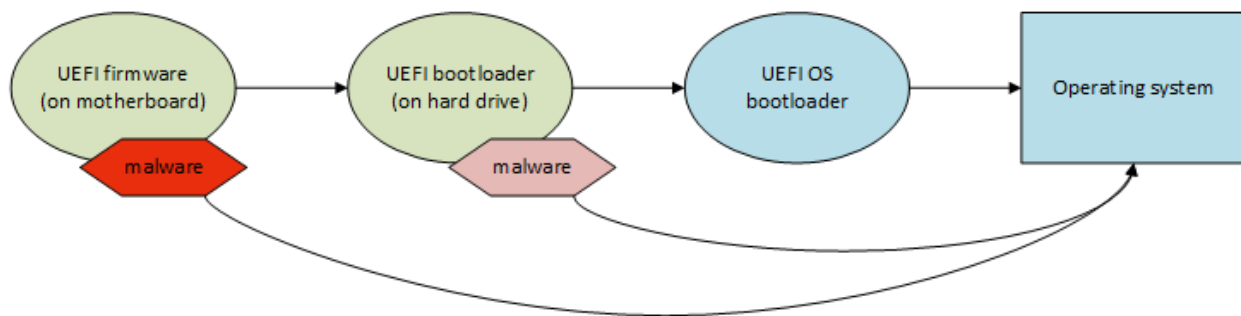


Figure 5.15: Various possibilities of infection from the UEFI.

bootloader. This technology is called the *Static Root of Trust for Measurement* (SRTM) based on the Trusted Platform Module (TPM) specified in the National Institute of Standard and Technology special publication 800155 [1107]. That way, it is hard to setup a bootkit in the UEFI preceding the launch of Windows. But the SRTM is not perfect [1108] and dealing with a lot of models of UEFI BIOS and linked versions. To manage it, the SRTM usually uses a list of known components trusted by the system. If a single bit changes in one of these components, the full chain of trust is compromise... This makes the management of the SRTM system hard to maintain. This is why *Dynamic Root of Trust* (DRTM) [1109, 1110] has been introduced in the context of the *Windows Defender System Guard Secure Launch* [1111, 1112]. According to the documentation [1111], in this mode, the system freely boots into untrusted code initially. But shortly after, the system is checked for trust by taking control of all CPUs and forcing them down a well-known and measured code path. This has the benefit of allowing untrusted early UEFI code to boot the system, but then being able to securely transit into a trusted and measured state. This simplifies the management of SRTM. Indeed, there is no longer any need to keep lists because security is no longer dependent on the underlying BIOS hardware. Note there are possibilities to enhance the security about the use of SMM [1113].

More directly, this means that malicious techniques presented here may no longer work or may be unstable on modern CPU or on Windows 10. This is why it may be a better idea to deal with improved technology like kernel development, which provides almost the same level of rights for less complexity.

### 3.2 Kernel-mode keylogger

#### Key Point 5.7:

- ☞ Kernel-mode keyloggers are usually malicious drivers whose purpose is to record everything that comes from the keyboard.
  - ☞ In practice, we are dealing with different methods to interface content of keyboard in one of the different layer presented in section 5.
  - ☞ Some methods are obsolete (SSTD or ISR hooking) with modern versions of Windows.
  - ☞ Today's methods aim to write drivers that follow general rules but whose purpose is malicious.
  - ☞ Note that since Windows XP 64-bit, all drivers must be signed to be executed, which strongly limits the possibilities for attackers.

An interesting point for keyloggers is the possibility to live within the operating system kernel. From the privileged access to information, the stealth offered as well as the difficulty to fight the threat, it is a place of choice for this type of malware. They are often cited in literature [485, 39, 1068] but few papers directly talk with technical details about them [1114, 41]. The most technical and the most complete remains [1115] from Emre TINAZTEPE, which stands out as an exhaustive presentation of what it is possible to do with keyloggers. It is a presentation of a workshop explaining how to develop keyloggers. But all these papers or presentations are quite outdated ([1115] is designed on Windows 7 32-bits machine in 2014, [1114] is from 2011). More directly, this means that the techniques presented may no longer work or may be unstable. We will present here different

techniques where some are stable while others are not anymore. Note that kernel-mode malware usually relies on general techniques available for usual driver. Hence, it is possible to get a good overview of what is possible in kernel-mode with the keyboard or the mouse devices with the Microsoft documentation [1116]. Most malicious solutions use all or part of the technology presented in the official documentation.

To summarize the range of possible threats, the table below lists what will be presented in the next paragraphs.

Name	Description	Efficiency
SSTD hooking	Direct hook set in the kernel	Totally obsolete
Ctrl2Caps	Old WDM filter driver	Still efficient
ISR hooking	Hook ISR for PS/2 keyboard	Only relevant for PS/2 keyboard
KbFilter	New WDF filter driver	Efficient

### 3.2.1 SSTD hooking - obsolete technique

#### Key Point 5.8:

- ☞ A SSTD hooking attack aims to modify the integrity of the Windows kernel image in memory.
  - ☞ The goal is to hijack some specific routines to take the control of the flow of keystrokes.
  - ☞ Since Windows Vista, this is no longer possible, thanks to *PatchGuard* and *HyperGuard* with Windows 10.
  - ☞ Such attacks on former version of Windows required to be administrator.
  - ☞ But being administrator is enough to bypass any security despite *PatchGuard* and *HyperGuard*.

If one is looking for information about keyloggers, it is common to find presentations that evoke various hacks to proceed [1117]. Usually, it is about dubious hooks within the operating system. These hooks are technically different from those presented in section 5.3.2 even if the general philosophy of the procedure remains the same. In this case, hook procedure is about what we can find in library such as *Detours*<sup>8</sup> [419]. The idea is to allow on-the-fly modification of a function (respectively a routine) to reroute any call to this one to another segment of code. The goal is to execute a function before (or/and after if needed) a targeted one. To proceed, technically, there are two main possibilities. By address when we update the address of an exported function with the address of another function. This address can be stored in the import table (IAT) of an executable file (kernel and user-mode) or in the *System Service Dispatch Table* (SSDT) used by the system (kernel-mode). The second possibility is to change the first op-codes of the hooked function to be hijacked in order to execute a jump to a function we want to execute before (this replacement function is called the *hook function*). This technique is usually implemented in a *trampoline hook* as described in papers such as [1118, 1119].

This way of working is no longer possible today on a modern Windows operating system. Specifically, since Windows XP 64-bit, a protection called *PatchGuard* [1120] has been implemented to prevent malicious programs from attacking critical kernel components. This mechanism prevents the SSDT from being modified on-the-fly (to hook routines exported by the kernel) and guarantees the integrity of the executable images used by the kernel in memory (to avoid hook trampoline techniques). Even better, starting with Windows 10, there is *HyperGuard* [1121, 1122] which is (among other things) a stronger hypervisor version of *PatchGuard*.

More directly, all techniques aimed at modifying the kernel to insert executable code in it or to modify relevant structures are prohibited and they should be banned. Thus, all kernel-based hook techniques using these processes are now destined to fail on 64-bit Windows sub-systems. Indeed, for reasons of backward compatibility, only 64-bit systems are subject to this protection. 32-bit subsystems are not impacted — which is why the demos from [1115] workshop are made on virtual machines with 32-bit Windows version. It should

<sup>8</sup><https://github.com/microsoft/Detours>

also be noted that such type of kernel-keyloggers were especially fashionable at the time of Windows XP. Why? May be because at that time, even if writing a driver was not a piece of cake, the complexity of the systems was lower. But mainly because the drivers did not need to be signed to be executed at that time. Indeed, from Windows Vista and latter 64-bit versions [1123, 1124], Windows device installation uses digital signatures to verify the integrity of driver packages and to verify the identity of the vendor (software publisher) who provides the driver packages. It makes sense that malware authors do not want to leave their complete identity (which is checked) at Microsoft office or third-party vendors certified by Microsoft. This may explain the declining popularity of this type of threat, but also the technical studies on the functioning of the Windows keyboard, which generally stop at Windows XP ([1114, 731]).

In practice, there are few vulnerabilities in the windows kernel which would have allowed to bypass the driver signing security. One publicly known is the CVE-2020-0601 [1125] disclosed by the US National Security Agency [1126]. This type of vulnerability is extremely rare. It allows to bypass one of the flagship security of Windows. But it is not less that before exploiting the latter, it is necessary to be able to write on the targeted machine and to register a driver (thus to be an administrator). Let us be honest, being an administrator on a machine for an attacker is more than enough to lose any notion of defense and computer security for the victim [1127, 1128, 1129, 1130]. Being able to execute code in kernel-mode is just an additional facility, certainly not a sufficient condition for the proper execution of an attack. This puts such flaws into perspective: they are quickly corrected and reported by organizations that have not always been known to be very cooperative when dealing with software vulnerabilities, but they also already require a high level of privileges to be fully operational.

This is because hooking techniques are now forbidden in the kernel that we will not cover them. They are irrelevant nowadays. And more generally, the set of techniques presented in [1115] (which nevertheless remains the most exhaustive and technical state of the art and perfectly interesting at the time it was presented) is now obsolete. Indeed, this work uses various hooks in Windows routines, following the whole keyboard chain as presented in section 5 to retrieve at different levels the content of the keyboard keys. All these hooks are now impossible and if some routines still exist since Windows 7, there are changes starting with Windows 10 and new routines could be considered when others should be abandoned. In addition, the study also aimed to retrieve some data directly from memory using undocumented structures. Needless to say that these structures are no longer the same on the latest versions of Windows 10, confirming the danger of basing project design on this type of practice.

---

### 3.2.2 Ctrl2Caps & KbFiler drivers based keyloggers

#### Key Point 5.9:

- ☞ Writing a driver is a complex task that requires precise skills from the developer (especially to have a reliable and efficient result).
- ☞ But it is not very complicated to design a keyboard driver since there are codes online.
  - ☞ Malware authors use these codes slightly customized to carry out their dirty work.
  - ☞ Of course, the result in terms of quality is sometimes very random but often sufficient to allow the malicious action.
  - ☞ We present two open-source examples that are generally used as a basis by malware: *Ctrl2Caps* & *KbFiler*.
  - ☞ We present two other methods: obsolete *ISR-hooking* and an original one based on *keyboard layouts* never seen in the wild.
- ☞ Handling keyboard from kernel-land requires to interface with `kbdclass.sys` driver, which is the entry point of all technology specific drivers (PS/2 or USB/HID).
  - ☞ An old-style way with IOCLT handler routines (*Ctrl2Caps* driver) or with ISR hooking (for PS/2 devices only).
  - ☞ A modern way with `KeyboardClassServiceCallback` callback routine.

For the few functioning codes online, we have examples based on "*kbfiler*"<sup>9</sup> as referenced in [1131]. The latter is a direct legacy of the *Ctrl2Caps* driver [1132] developed by Mark Russinovich himself. *Ctrl2cap* is a kernel-mode device driver that filters the system's keyboard class driver in order to convert caps-lock key into control key. Such functionality meets the needs of Mark who, coming from the UNIX world to work under Windows NT, have experienced control key located where the caps-lock key was on his standard PC keyboards. This is to retrieve the former layout<sup>10</sup> of his previous keyboard that Mark decided to develop this project.

Of course, Mark Russinovich's project is not a keylogger, no more than the *kbfiler* project given as an example by Microsoft. But these are generally the sources of inspiration that have shaped the keylogger drivers. From an evolutionary point of view, these are the common ancestors of many keylogger projects. This is why rather than presenting reverse engineering of couple of malware that would be very empirical (and bit tedious to draw generalities), we prefer to rely on the well-documented and technical base code used by keylogger authors. Not only does this present the threat, but it is in line with what was presented in section 4 and what will be presented in the section 6 when explaining our countermeasures.

<sup>9</sup><https://github.com/microsoft/Windows-driver-samples/tree/master/input/kbfiltr>

<sup>10</sup>Note that nowadays, on modern version of Windows, it should be better to use *Scan Code Mapper* [879] as presented in section 5.2.7. But at the time Mark wrote this driver, this technology was not present. Therefore, it is reasonable to think that it was to generalize the needs that Mark met (and may be other Windows' users had) that this technology — more easily accessible than the development of a driver — appeared.



### 3.2.2.1 Ctrl2Caps driver

#### Key Point 5.10:

- ☞ Ctrl2Caps is an old WDM driver which has been written by Mark Russinovich to swap two keystrokes on keyboard.
  - 👉 In a way, this was the ancestor of the *Scan Code Mapper* feature (Key-Point 4.45).
  - 👉 Based on IRP handling (IRP\_MJ\_READ), the driver reads keystrokes and modify some of them on-the-fly.
  - 👉 Ctrl2Caps is registered as an *UpperFilters* driver in the registry of Windows to be automatically inserted into the keyboard drivers device stack.

The source code of Ctrl2Caps driver is available on GitHub<sup>11</sup>. This is a WDM driver written in a simple and effective style. The code is designed to run on different version of Windows, including before Windows XP. The `DriverEntry` routine registers handlers for different dispatch routines [888], especially `IRP_MJ_READ` and `IRP_MJ_PNP` before registering its device handler `Ctrl2capAddDevice` routine.

Prior to Windows XP, it was required to create a dedicated device with `IoCreateDevice`, linked to `"\Device\KeyboardClass0"` thanks to `IoAttachDevice` and only `IRP_MJ_READ` needed to be handled. After Windows XP, this operation of linkage is performed in the add device handler (`Ctrl2capAddDevice` routine). This last creates a device object with `IoCreateDevice` and attaches it to the device stack thanks to `IoAttachDeviceToDeviceStack`. In both case, the central operation is performed in the `IRP_MJ_READ` routine handler: `Ctrl2capDispatchRead`. This one is called each time a read request is armed by the system. But the relevant information, the key's content is only provided once the key has been pressed, that is to say when the read IRP request is completed. This is why `Ctrl2capDispatchRead` calls `IoSetCompletionRoutine` to be notified (thanks to the registered routine `Ctrl2capReadComplete`) once the IRP read request will be completed. This last routine retrieves the information from the associated IRP system buffer [1133] which is an array of `KEYBOARD_INPUT_DATA` structures [737]. Each structure holds a keystroke to be managed by the system. This is where Mark Russinovich is checking if a key is the cap locks key in order to translate it to left control key. Note that we are dealing with scan codes at that point and Mark's driver is not aware of other scan code sets that the first one standard.

The Ctrl2Caps driver itself does not do anything to bind itself to the keyboard stack (except considering the pre-Windows XP version, which would be a bit risky). In fact, this type of driver only makes sense if we take into account the way it is installed in the system. Registry of Windows is a central place for device drivers [1134]. Generally, this operation is performed via the `.inf` file associated with the driver [1135]. But it seems that at former times, Mark preferred to write a single application to perform the same task. Technically, he registers the driver in `"HKLM\System\CurrentControlSet\Services\Ctrl2cap"` [1136, 1137, 1138] and he registers a keyboard filter driver under `"HKLM\System\CurrentControlSet\Control\Class\{4D36E96B-E325-11CE-BFC1-08002BE10318}"` [568] to reference a filter driver for keyboard object [1139, 1140, 1141, 1142]. In the last registry key, Ctrl2Caps registers the filter thanks to a value `UpperFilters` [1143]. This notion of `UpperFilters` and `LowerFilters` has already been discussed in sections 3.3 and 4.2.4.3, in particular with the Figure 4.38. This method of registry manipulation does not provide the flexibility to specify *exactly* at which position to register a particular filter, but this is now possible since Windows 10 version 1903 [1144]. For more information and descriptions about the device-specific registry entries, the reader can refer to [1145].

Conceptually, Ctrl2Caps driver is not a keylogger. But it interacts with keystroke content at low level in the driver device stack. This is why it has been forked and modified so that it can be used as a keylogger. Its simple structure allows almost anyone with minimal knowledge to rebuild a logging version. The main change lies in the `Ctrl2capReadComplete` routine. Instead of checking the scan code of a keystroke to exchange it with another one, the operation of writing in a file (or a memory buffer written later in a file) is enough to deploy a keylogger.

<sup>11</sup><https://github.com/baohaojun/system-config/tree/master/gcode/Ctrl2Cap>

### 3.2.2.2 ISR hooking PS/2 driver

#### Key Point 5.11:

- ☞ As explained in section 3.3 (Key-Point 4.5), PS/2 keyboard's driver uses *interrupt service routine* (ISR) to retrieve keystrokes from the device.
  - ✍ It is possible to write a driver to register interrupts in addition to those present for the keyboard.
  - ✍ Registration can be done generically with `IoConnectInterruptEx` or through a dedicated IOCTL code sent to drivers in the keyboard device stack.
  - ✍ The `INTERNAL_I8042_HOOK_KEYBOARD` IOCTL provides a dedicated structure useful to provide a ISR hook callback to manage the keyboard.
  - ✍ This technique is present in a more or less official way in the Microsoft *KbFilter* driver.
- ☞ This way of doing things is not really popular since it is only for PS/2 keyboards and not really easy to use.

A slightly more advanced WDM filtering technology is about to use dedicated callbacks for PS/2 handling purposes [1146]. Among the different callbacks, we can find different callback routines for keyboards (same are present for mouse). These callbacks are exported and referenced through a special IOCTL code [1147] sent with `INTERNAL_I8042_HOOK_KEYBOARD` structure [1148] which carries all the available callbacks for a PS/2 upper filter driver [1149]. Such registration is performed by the driver at initialization phase<sup>12</sup> and reserved to upper filters only since lower filters would be out of the range of the `I8042prt` driver (section 3.3). Among the callbacks supported, there is `PI8042_KEYBOARD_INITIALIZATION_ROUTINE` [1150] which is notified when a keyboard device is setup by `I8042prt` driver. At that place, it is possible to directly drive the read and write operations for the keyboard device [1151, 1152]. But this is not the regular way to proceed since it is easier to let the system drive the read and write operations to be notified only when they occur. Instead, the initialization IOCTL is used to record an `PI8042_KEYBOARD_ISR` [1153] callback routine that changes the behavior of the original ISR used to manage interruptions from keystrokes in PS/2 architecture. An example of such routine is given in Microsoft's documentation through the name of `KbFilter_IsrHook` [1154] in *KbFilter*<sup>13</sup> project.

This technology, however more flexible and — in a way more — efficient than the one used in `Ctrl2Caps` driver, nevertheless has two disadvantages today. The first is that it only targets PS/2 type keyboards, depriving us today of a very large share of USB (or assimilated) keyboards. The second is that the system is not as flexible as the one provided by `Ctrl2Caps` driver. Indeed, `Ctrl2Caps`'s simplicity of architecture is only based on reading IRPs that finally makes it still relevant today, as long as the installation process works<sup>14</sup>. In fact, PS/2 ISR filtering technology has never been very popular. On the one hand because it arrived a bit late (`Ctrl2Caps` was already doing it very well and USB keyboards rose) and also because its documentation is far from being impressive or easy to learn.

Nevertheless, this technology can be used to create keyloggers. Technically, only two callbacks are required: `PI8042_KEYBOARD_INITIALIZATION_ROUTINE` and `PI8042_KEYBOARD_ISR`. The first is used in order to register and to handle the different keyboards to take into account. The second is used to retrieve information about the keystroke responsible for the interruption handle by the ISR. In this last case, it is not possible to directly record the keystroke content into a file. Indeed, this callback routine is called in the context of the original `I8042prt` keyboard ISR which means that it runs in kernel mode at the same IRQL. This one can be different from a passive level which is the only one which guarantees a safe access to the hard drive.

<sup>12</sup>Theoretically, such operations could be performed latter in the driver's code. But generally, it makes sense to initialize the filter process as soon as the driver starts.

<sup>13</sup>At the line number 692 in <https://github.com/microsoft/Windows-driver-samples/blob/master/input/kbfiltr/sys/kbfiltr.c>

<sup>14</sup>`Ctrl2Caps` only targets "KeyboardClass0" device, which corresponds to the first detected keyboard by the system. That could be a limit in case of multiple keyboards used by a computer, but the driver can be adapted to support many devices.

### 3.2.2.3 KbFiler driver

#### Key Point 5.12:

- ☞ KbFiler driver is a WDF driver that proposes to present almost all the ways to interact with the keyboard from kernel-land.
  - ☞ WDF stands for *Windows Driver Framework* which is an evolution of *Windows Driver Model* (WDM).
- ☞ It is possible to register a driver callback routine to be notified each time a key is pressed or released.
  - ☞ In practice, `IOCTL_INTERNAL_KEYBOARD_CONNECT` request is sent for each keyboard device plugged in the system with a `CONNECT_DATA` structure.
  - ☞ It is possible to update `CONNECT_DATA` structure with a callback routine from the driver to interface with `kbdclass.sys` driver (Key-Points 4.5 and 4.26).
  - ☞ The callback can read, modify or delete keystroke received from the keyboard.
  - ☞ To maintain the original behavior, our callback routine will have to call at the end the routine originally provided in `CONNECT_DATA` (usually `KeyboardClassServiceCallback` if there is no other filter).
- ☞ To be inserted in the keyboard driver device call stack, KbFiler driver is registered as an *UpperFilter* driver in the registry.

As explained before, *KbFilter* is a project which is in the right continuity of the `Ctrl2Caps` drivers but also a compilation of the different keyboard management technologies. Technically, it is not a good thing to use it directly. It is better to see it as a "tutorial" about what can be done and how it can be done.

This one is written with WDF (Windows Driver Framework) [1155, 1156]. Windows Driver Frameworks (WDF) is a wrapper around Microsoft Windows Driver Model (WDM) interfaces [1157]. It is a framework written in C++ and it is open source<sup>15</sup>. Also, the framework simplifies many WDM concepts and hides others completely so that it can simplify the developer's life. Technically, WDF defines a single driver model that is supported by two frameworks: Kernel-Mode Driver Framework (KMDF) and User-Mode Driver Framework (UMDF) [1158]. It means that there is a framework for each purpose (kernel or user modes) but which are very familiar (especially from UMDF 2 supported since Windows 8.1). It allows a higher flexibility in writing code. In addition, if a driver requires access to some kernel-mode resources or features, it is possible to split the driver into two parts. One in kernel-mode and the other in user-mode. This approach offers the benefits from developing and running drivers in user mode. UMDF drivers run in a driver host process, which runs with the security credentials of a `LocalService` account, although the host process itself is not a Windows service. Thus, user-mode drivers are as secure as any other user-mode services, with impersonation technology [1159] which can be used in order to run in the security context of the notification from the kernel-mode. This splitting makes it possible to move part of the code that was traditionally in the kernel to user-mode, which results in software which are more stable, easier to develop and to debug.

But more than this UMDF particularity, it should be noted that the WDF framework offers a unified implementation at the kernel-mode level. The goal is to cover complex aspect from WDM. For instance, WDF creates a framework device object for each WDM device object. That way, using a WDF allows a framework-based development for drivers to access device objects instead of WDM device objects, through a dedicated API (i.e. the framework). In addition, WDF drivers typically do not directly access IRPs. The framework converts the WDM IRPs that represent read, write, and device I/O control operations to framework request objects. More differences are given in [1160]. WDF technology has been introduced in Windows 2000 and from that time, Microsoft encourages to port WDM drivers to the WDF model [1161].

<sup>15</sup><https://github.com/Microsoft/Windows-Driver-Frameworks>

Going back to KbFilter project, because it is written with WDF, it differs from the traditional Windows's driver documentation. Nevertheless, it is not hard coming from WDM technology to understand the WDF one. As any driver, we have a driver entry point whose name is still `DriverEntry`. There is an event handler for device insertion with `KbFilter_EvtDeviceAdd` routine in KbFilter. This routine is responsible to register a dispatch routine for internal device control requests. This dispatch routine is `KbFilter_EvtIoInternalDeviceControl` which handles `IOCTL_INTERNAL_KEYBOARD_CONNECT` and `IOCTL_INTERNAL_I8042_HOOK_KEYBOARD` [1147]. The last is perfectly optional and it only matters if we need to deal specifically with PS/2 keyboards. But the `IOCTL_INTERNAL_KEYBOARD_CONNECT` request [705] is much more relevant since it connects the `Kbdclass` service to the keyboard device.

This is the responsibility of `Kbdclass` driver to send this request down to the keyboard device stack before it opens the keyboard device by itself. According to its documentation, the keyboard filter notified with this IOCTL should respond to the request by saving a copy of the `CONNECT_DATA` structure [706] that is provided to the filter driver by `Kbdclass`. This operation happens before possibly modifying the connect data structure and passing this IOCTL request down to the device stack. It is precisely by modifying the `CONNECT_DATA` structure that it is possible to set up the filtering procedure. The idea is to supplement the operation of `KeyboardClassServiceCallback` [529] (sections 3.3 and 4.2.7), that is to say the `Kbdclass` class service's callback routine. The behavior with `IOCTL_INTERNAL_KEYBOARD_CONNECT` has been fully documented by us in section 4.2.7. Providing another callback routine in addition to the standard one in `Kbdclass` driver allows third-party drivers to perform filtering operations.

The filter service callback (either `KeyboardClassServiceCallback` or a third-party one) can filter the input data that is transferred from the device input buffer to the class data queue. Filtering operation allows to read, insert, modify or delete the input provided by the keyboard device. The callback notification is called at `IRQL_DISPATCH_LEVEL` (consequence to be called by the ISR dispatch completion routine of the function driver). The chaining of the various filters is left to the control of the filters themselves. More directly, the objective of saving the content of the `CONNECT_DATA` structure is precisely to retrieve the routine pointer referencing the original (or the previous) filter. This makes possible to call from the current filter driver the filters previously installed by another driver and (not to say *mainly*) the system's one: `KeyboardClassServiceCallback`. Not calling the `KeyboardClassServiceCallback` is allowed in case of it would be required to delete the received key. But generally, this is not a very good idea in particular to systematically reject the call to this system callback. This literally inhibits the keyboard which results in paralysis of the system. Example of such filtering callback is provided in `Kbdfilter` project with `KbFilter_ServiceCallback` routine [1162] which is ultra-minimalist since it only calls the original callback provided through the `CONNECT_DATA` structure.

Installation of `Kbdfilter` is an important point since the link with the keyboard is performed at that point. At the opposite to `Ctrl2Caps` driver, `Kbdfilter` uses a `.inf` file<sup>16</sup> [1135]. This one is quite classical for a driver. There are two main lines responsible to register the filter. The first is in the header of the `.inf` file with the `ClassGUID` registered for keyboard class driver [568] and a registry key crafted to register an upper filter (Code 5.1). With this last line, `HKR` corresponds to a relative root to the driver's service key [881].

```
[ kbfiltr .HW.AddReg ]
HKR, , "UpperFilters", 0x00010000, "kbfiltr"
```

Code 5.1: Extract from the `.inf` file in `kbfiltr` project.

In a direct way, it is possible to find codes that use modern versions of `KbFilter` forked in `keylogger` project. A good example is "`kmdf-keylogger`" from Ada Piekarska [1163]. This one is literally a copy and paste of `KbFilter` enhanced with a few features to give it its malicious behavior (even if the project is published for educational purposes). The main feature is to store in a buffer all the keystrokes notified in the filter callback. It is not possible to write directly into a file from this callback because its `IRQL` is too high. To log in a file, a buffer is

<sup>16</sup>In the official Microsoft's github project where it belongs, this file uses an `.inx` extension, maybe to avoid security issues with some antivirus software. <https://github.com/microsoft/Windows-driver-samples/blob/master/input/kbfiltr/sys/kbfiltr.inx>

completed, notification after notification, while a worker thread [1164] mechanism is used to regularly flush the buffer into a file at a passive IRQL level. Apart from the rather peculiar management of the file containing the pressed keys, there is no other difference.

Since it could be hard to estimate the cost of a software keylogger using drivers, it is possible to find *job opening* on the web. Found on the web, the offer given in Figure 5.16<sup>17</sup> proposes a remuneration between 1.500 and 5.000 US dollars to realize a complete software (driver and data extraction). There is no shortage of applicants for this type of proposal. Whatever may have been the reality of this offer, the fact remains that the proposal is coherent and the remuneration relatively in line with the type of competence sought.

### 3.2.3 Keyboard layout malware

As a preliminary remark, we would like to point out that to the best of our knowledge, this type of malware has never been observed, either in malware or in academic or other publications. As presented in Chapter 4, section 5.2.5 (Key-Point 4.43, it is possible to create our own library to implement your own *keyboard layout*.

Most of the documentation to implement such a library comes from *Keyboard Layout Samples* project [927]. This one shows how to implement a layout Dll executed in the kernel context. A keyboard layout Dll consists of a set of tables. One of the tables converts scan codes to virtual key codes, while the other table provides the conversion rule from the virtual key codes to characters. In practice, it corresponds to the helper tables used by MapVirtualKey(Ex) [906] (Chapter 4, section 5.2.7, Key-Point 4.45) and more precisely to conversions as represented in figure 4.103. Note that some key combinations generate specific characters (specific accents with vowels in french language, for instance) while modifier keys, such as the SHIFT key or the CTRL key, alter the character generation, but do not generate the characters.

Technically speaking, all the conversions are technically performed through a set of tables, called *conversion tables*. The conversion table for the character generation has multiple columns, where the number of columns could vary from layout to layout, mostly to take into account all possible combinations of the modifier keys. Each column represents the modifier status to apply on the generated character.

More directly, there is no real code executed directly from the layout Dll. Instead of, a set of predefined structures, documented by the project itself, which are called by the kernel to perform translation. For the sake of simplicity, it is about converting a scan code to a virtual key code (or a character) thanks to a read in a given structure to find the equivalent conversion.

A keylogger cannot use directly these conversion tables to log scan codes or virtual key codes provided to the layout Dll. Indeed, there is no direct code executed here (in any case, as far as what the keyboard layout sample project [927] explains it since it is in fact one of the only official sources of information. Nevertheless, the Dll is not passive and at least one of its functions is called. Indeed, instead of exporting tables as regular values [1165], the window manager requires to call one or more entry points from layout Dll to retrieve the table address and the information about the layouts. In the examples provided [927], it is a simple return of address without any additional action being performed. Nevertheless this *modus operandi* must have an explanation, probably with backward compatibility we guess, at a time it would had been necessary to initialize a table specifically (maybe depending on the present device hardware) before exporting it this way.

It is thus perhaps possible to imagine running malicious code on of these entry points to interface with the kernel and install a keylogger. Of course, tampering with the memory image of the Windows kernel is no longer possible (section 3.2.1), but continuously reading some structures in memory (after reverse engineering them on-the-fly or based on Windows debug symbols) could be an interesting approach.

More reverse engineering would be desirable to arrive at a project capable of interfacing properly with the core. In particular, perhaps look at whether it is possible to manipulate (or even better to provide) a callback routine for *National Language Support* (NLS) context... But more generally, this layout Dll should be seen as a code executed in kernel mode, which could open the possibility of mapping the image of a keylogger driver

---

<sup>17</sup><https://www.freelancer.com/projects/c-c/keylogger-win-kernel-mode-driver/?ngsw-bypass=&w=f>



Keylogger Win2K/XP kernel-mode driver
BUDGET \$1500-5000 USD

Freelancer > Jobs > C Programming > Keylogger Win2K/XP kernel-mode driver

We need to have developed a high-performance Win2K/XP kernel-mode driver which runs silently at the lowest level of Windows 2000/XP operating systems.

- All keystrokes must be recorded, including the alt-ctrl-del trusted logon and keystrokes into a DOS box or even a Java chat room.
- Must be able to email the log file via SMTP.
- Must be able to HTTP POST the log file.
- Have to has a send out file log scheduler.
- Must be a Win2K/XP kernel-mode driver (.sys)

Developer must code the application in C++.

**Skills:** C Programming

**See more:** kernel mode keylogger, kernel keylogger, keylogger driver, keylogger symbian, kernel keylogger driver, driver keylogger, keylogger kernel, kernel based keylogger, driver based keylogger, kernel mode, symbian keylogger, keylogger code logs via mail, keylogger kernel driver, driver kernel keylogger, windows kernel mode keylogger, win2k keylogger, kernel mode driver keylogger, designing kernel key logger, write keylogger driver, kernel mode driver socket

**About the Employer:**

0.0 ★★★★★ (0 reviews) Buenos Aires, Argentina

Project ID: #114505

**Looking to make some money?** PROJECT IN PROGRESS

**Your email address**

Apply for similar jobs

✓ Set your budget and timeframe

✓ Outline your proposal

✓ Get paid for your work

✓ It's free to sign up and bid on jobs

**Awarded to:**

**mnirahd** 🇺🇸

Well, i think we can negotiate over the number of days required to complete the job.

**\$1500 USD** in 21 days

0.0 ★★★★★ (0 Reviews)

1.9 \$ ██████████

**14 freelancers are bidding on average \$2650 for this job**

**technoparkcorp** 🇺🇸

Looking forward to hearing from you in PMB. Thanks

**\$2000 USD** in 30 days

5.0 ★★★★★ (1 Review)

4.8 \$ ██████████

Post a project like this

**Other jobs from this employer**

- ▶ Remote Desktop Java Application (\$300-1500 USD)
- ▶ aMSN winks implementation (\$300-1500 USD)

< Previous Job Next Job >

**Similar jobs**

- ▶ Read and Write data from OMron PLC CJ2M (\$10-100 USD)
- ▶ multi recharge software (₹12500-37500 INR)
- ▶ OpenSource project rebuild-QGIS (C++|Python|Qt) (\$30-250 USD)
- ▶ c script not from scratch base. (\$10-30 USD)
- ▶ C Programming, Math works and Microcontroller (₹400-750 INR / hour)
- ▶ C++ String Algorithm Expert – 2 (\$30-250 USD)
- ▶ Cryptocurrency multi wallet api (€750-1500 EUR)
- ▶ PWm Switch (₹12500-37500 INR)
- ▶ Finding C++ String Algorithm Expert (\$30-250 USD)
- ▶ I need 'Bucket sort' algorithm to be parallelized in both OpenMP, MPI, Cuda in C language. (\$10-30 USD)
- ▶ Tradingview indicator to python or c# (\$10-30 USD)
- ▶ Write Function in C Language (\$10-30 USD)
- ▶ Roobet Crash Predictor Program (\$10-30 USD)
- ▶ option chain excel (₹1500-12500 INR)
- ▶ C# (Paths and Pipes Game) (\$30-250 NZD)
- ▶ tumblr site (\$30-250 USD)
- ▶ incomplete esp32 spwm inverter project (€30-250 EUR)
- ▶ software development experts .. (\$30-250 NZD)
- ▶ Convert Think Script to Pine Script on TradingView (\$10-30 USD)
- ▶ Debug a solution error (₹750-1250 INR / hour)

Figure 5.16: Job proposal for a keylogger development.

in memory and then linking it to the driver device call stack of the keyboard, in order to be notified naturally. This would be a *fileless attack* that would be formidable and unprecedented.

### 3.3 User-mode keylogger

#### Key Point 5.13:

- ☞ User-mode keylogger can be differentiated according to the system API they use to retrieve keyboard keys.
  - 👉 Technically speaking, user-mode keylogger manages the keyboard like any other legitimate application.
  - 👉 The difference is in the final aim of the data captured by the keyboard.

User-mode keyloggers are, by far, the most popular ones. Why? Because they are simple to implement, there are many tutorials [1166] online, it is not very complex to find examples about them on github [1167, 1168]<sup>18</sup>, and their deployment does not involve any particular issue. The ease of implementation can be explained by two main reasons. The first is the abundance of example codes on the web. Copying and pasting or adapting from one programming language to another is not a complex operation. The second reason is the cause of the first one: reading keyboard input is a perfectly legitimate and a normal activity for any program offering a minimum of interaction with the user. The real problem is what is done with the data captured from the keyboard. This is where keyloggers finally come in.

Technically, we can mobilize all the elements exposed in Chapter 4, section 5. It describes all the means available under Windows to obtain the content of the keyboard device. Among the different means, there is a difference between the general means that do not require access to the keyboard focus and the others that do. We are going to propose a taxonomy between the different user-mode keylogger based on their different access means.

#### 3.3.1 Keyloggers without keyboard focus

#### Key Point 5.14:

- ☞ The most effective keyloggers are independent of keyboard focus. Several methods are possible, they are listed here in order of fame.
  - 👉 Using `GetAsyncKeyState` is the most used technique, since it is simple and highly efficient.
  - 👉 Using `SetWindowsHookEx` is common but a bit more complex to use. Malware can use in an undifferentiated way `WH_KEYBOARD` and `WH_KEYBOARD_LL`.
  - 👉 Using raw input access method is another efficient way for malware to get access to keyboard. Less common than the two others, this requires the creation of a specific GUI window which can be hidden.
  - 👉 Using `DirectX` is a not common method, almost never seen. Harder to manage, less accurate than others, however it proposes a stealthier approach by using a famous video games library.

Keyloggers using the API that do not require keyboard focus are by far the most operational mean for malware developers. The lack of keyboard focus brings two important advantages. On the one hand, it is not necessary for the keylogger to appear in the foreground of the GUI windows displayed to the user. This is a very welcome stealth. On the other hand, it allows malware to passively listen to all the other applications that receive focus from the keyboard at some point. In a way, to use a well-known verb, we are eavesdropping here. In practice, there are several ways to perform such an operation, presented in the following paragraphs.

<sup>18</sup>To find more results, one can use: <https://github.com/topics/keylogger>.



### 3.3.1.1 Keylogger based on `GetAsyncKeyState` function

In general, access to the keyboard keys is performed using the `GetAsyncKeyState` function. This one has already been presented in Chapter 4, section 5.3.1.4 (Key-Point 4.50). Usually, the operation is performed within a loop that tests the 256 possibilities related to virtual key codes. From what is generally observed, if the function `GetAsyncKeyState` returns a value other than zero, it means that the key is pressed. In this way, it is possible to log thereafter the index value tested during the call to `GetAsyncKeyState` function. The operation is repeated indefinitely to continue capturing keys over time. It may happen that the use of the `Sleep` function [321] temporarily suspends the activity to release CPU workload. With, the CPU activity would be very high for a given process and such an activity would be suspicious to user's eyes not to mention any antivirus. This approach reduces the CPU consumption but it allows a potential miss of some keystrokes which could be pressed and released during the sleep period. Code 5.2 is a possible illustration about what it is possible to do.

```
UCHAR i = 0;
USHORT out = 0;
while(1){
    do {
        i++;
        out = GetAsyncKeyState(i);
        // Sleep(10); // Optional.
    } while (out == 0);

    // Log the keystroke and reloop.
}
```

Code 5.2: Usual case to use `GetAsyncKeyState` for keylogger purposes.

### 3.3.1.2 Keylogger based on `SetWindowsHookEx` function

Another technique lies in the use of `SetWindowsHookEx` already presented in Chapter 4, section 5.3.2 (Key-Point 4.52, 4.53 and 4.56). The goal is to register a callback function into the message hook chain of Windows. There are two types of keyboard hooks: `WH_KEYBOARD` and `WH_KEYBOARD_LL`. Both have been illustrated in section 5.3.3. Hence, using them in a context of a keylogger is not a complex task. Indeed, it is just about updating the hook procedure in codes 4.34 and 4.36 to write the keystroke information collected in a file.

The version using the low-level keyboard hook is the most popular (Key-Point 31). In addition to provide an information which has not be altered by the raw input thread or other hooks (since it is provided before to be injected in the message loop), this one does not need a Dll. Indeed, it does not need a Dll injection in its global form, making it somehow more stealthy but also easier to develop. Indeed, a simple loop message in a dedicated thread is enough, there are many examples of malware using this design [1168].

### 3.3.1.3 Keylogger based on raw input access method

Another example of global access to the keyboard is the use of the *raw input* technique [878] as presented in Chapter 4, section 5.2.4 (Key-Point 4.42). This one is based on the use of `RegisterRawInputDevices` function. This method requires to create a dedicated GUI window which can be hidden thanks to `ShowWindow` [818] with its last parameter set to zero. That way, calling `RegisterRawInputDevices` function with Usage and Usage Page corresponding to HID keyboard definition [712] is enough to see the window's procedure callback be notified with `WM_INPUT` messages [892]. From that point, `GetRawInputData` function is used to retrieve the `RAWINPUT` structure holding keystroke's information. To be a keylogger, an application keeps the content of the `RAWINPUT` structure.

### 3.3.1.4 Keylogger based on `DirectX`

The last global method presented is far from being common in malware's environment. To the best of our knowledge, there is no real and operational malware based on `DirectX` keylogger (at least publicly known

or as part of a relatively successful malware). However, it is only possible to find a few examples of codes online which describe what is possible [1026, 1169]. But we must keep a certain critical view on what is proposed. One first example is given in [1169] and it involves *Direct Input* [1170] from Microsoft DirectX library.

This method to access keyboard has been introduced in Chapter 4, section 5.4.1.1 (Key-Point 4.58). Requiring the DirectX SDK to compile the application using this method, it produces executable files linked to the `dinput8.dll`, like a video game. From Code 5.3 (extracted from [1169]) `DirectInput8Create` function creates a *DirectInput* object which is linked to the selected version number of DirectX (8.x, 7.0 or 9.0). The set the data format we want to retrieve is then set with `IDirectInputDevice8::SetDataFormat` before calling `IDirectInputDevice8::SetCooperativeLevel` with `DISCL_NONEXCLUSIVE` and `DISCL_BACKGROUND` flags to be sure that our keyboard capture is global and will not be lost if the application loses the keyboard focus.

```
HRESULT hr;
hr = DirectInput8Create(g_hModule, DIRECTINPUT_VERSION, IID_IDirectInput8, (void **)&m_din,
    NULL);
hr = m_din->CreateDevice(GUID_SysKeyboard, &m_dinkbd, NULL);
hr = m_dinkbd->SetDataFormat(&c_dfDIKeyboard);
hr = m_dinkbd->SetCooperativeLevel(m_hWnd, DISCL_NONEXCLUSIVE | DISCLBACKGROUND);
```

Code 5.3: Register a DirectX keyboard access (from [1169]).

To retrieve the keyboard's content, following what is provided in [1169], we can use Code 5.4. In this case, this is the immediate datatype which is implemented with `IDirectInputDevice8::GetDeviceState` method which is similar in its philosophy to `GetKeyboardState` function. A short illustration of this procedure is given in Code 5.4 extracted from [1169].

```
BYTE keystate[256] = { 0 };
lpCDDirectInputKeylog->m_dinkbd->Acquire();
lpCDDirectInputKeylog->m_dinkbd->GetDeviceState(256, keystate);
```

Code 5.4: Read keyboard content with DirectX (from [1169]).

Of course and as explained in Chapter 4, section 5.4.1.1 (Key-Point 4.58), whatever is the keyboard data acquisition procedure, that one must be performed in a loop to be efficient (Codes 4.38 and 4.39). In a video game (which is the usual use of DirectX technology), this action is performed at almost each frame generation to know in *real time* the status of the keyboard. More information about how to deal with DirectX keyboard input in [1027].

Interestingly, [1169] uses `MapVirtualKey` function [906] to translate the keyboard scan codes into virtual key codes. They propose the Code 5.5 with variable `i` which stands for the scan code value seen as pressed from `IDirectInputDevice8::GetDeviceState` method.

```
UINT virKey = MapVirtualKeyA(i, MAPVK_VSC_TO_VK_EX);
```

Code 5.5: Translation of keyboard code retrieved with DirectX (from [1169]).

As explained in Chapter 4, section 5.4.1.1 (Key-Point 4.58), *DirectInput* from DirectX uses its own code (called by us *dik code*) to represent physical keys from the device but it corresponds more or less to the internal scan code used by Windows (Key-Point 4.25). This translation is not perfect (for instance *Left-Arrow* key is not correctly translated when coming from the arrow pad area (0xCB DIK\_LEFT value in dik code) but correctly taken into account when providing from the numeric pad (0x4b dik value DIK\_NUMPAD4 which corresponds to virtual key code value 0x25, that is to say VK\_LEFT). On the one hand, this approach could lead to the loss of some keys which would then be incorrectly translated with Code 5.5. On the other hand, since *DirectInput* of DirectX does not take into account the keyboard layout [1049] because it ignores the *raw input thread* (Key-Point 4.37) but `MapVirtualKey` function does, it allows a kind of matching to get the real meaning of the user's keystrokes. But still, it does not allow to perform a direct distinction between uppercase and lowercase

characters.

In the case of [1026] example, translation is performed manually for each key code. This allows a better precision but the management of the source code is not really efficient. In addition, [1026] bases its key retrieval procedure on the windows message system... It is a non-sense since *DirectInput* is not correlated to the message system managed by the *raw input thread*. And even worse, rather than relying on messages that would signal the activity of a device such as the keyboard, the author proposes to use a timer to regularly read the contents of the keyboard in *immediate keyboard data* (designed much more for real-time data acquisition while buffered keyboard data would fit better to the proposed architecture [1043]). This is the risk of improvising DirectX's *DirectInput* technology, which is not designed for textual keyboard management.

In the end, it appears that malware using DirectX technology are rather rare, if not only being at the proof of concept stage. The reason is that this technology, if it has the merit of avoiding the usual keyboard management APIs (and thus trying to potentially escape detection by betting on the fact that a defense system will pay less attention to this type of implementation — if such system with such a lack exists), it is not made to retrieve what is entered textually by the user. And therein lies the main flaw. DirectX is made for video games, it is more important for this library to have a large number of different keys than the precise function associated with each key (since it is the video game that gives meaning to each of the keys). Nevertheless, it remains relevant to present this technology because it is a possible means to serve malware purposes.

### 3.3.2 Keyloggers with keyboard focus

#### Key Point 5.15:

- ☞ To be minimally effective, a keylogger must be able to access the keyboard without having the keyboard focus.
- ☞ This is not the case with keyloggers that need keyboard focus.
- ☞ In practice, they literally have to tamper with the focus system, which creates a whole bunch of visible annoyances for the user (GUI windows disappear, keystrokes are missing in front-ground software and so on).

This is the type of keylogger which is the less used. Because such keyloggers are complex to manipulate, keyloggers based on `GetKeyboardState` and `GetKeyState` are not famous. Indeed, these two functions require to get access to the keyboard focus to be efficient, as explained in section 5.3.1. They can be used with `AttachThreadInput` function but it rises technical issues to manage [875]. The use of `GetKeyboardState` is more common when associated with `GetAsyncKeyState` and `ToUnicodeEx` function. Indeed, this last one helps the conversion to Unicode character by taking into account other key-state which potentially would have been pressed. Example of code is given in [1171]. Note that the use of `GetKeyboardState` with `ToUnicodeEx` is not necessarily a good idea. The `GetKeyboardState` function gives the keyboard's point of view for the current application when the code retrieved by `GetAsyncKeyState` may come from the state of a key generated by another application (with a result of `GetKeyboardState` that would be different from the one seen from the keylogger application). It might results incoherent or false results when calling `ToUnicodeEx` function since the array of keystrokes returned by `GetKeyboardState` is not necessarily the one used in the application's context where `GetAsyncKeyState` reports the status of the key.

### 3.3.3 Direct hooking in a targeted process

#### Key Point 5.16:

- ☞ An efficient way to target the keyboard input of a particular process is to retrieve this input directly from it.
  - ☞ We focus on the functions that interface with the keyboard and we hijack their execution flow (via a function hooking technique [419]).
  - ☞ This is a targeted method but hardly stealthier than using `SetWindowsHookEx` because there is nothing legitimate about this type of operation.

This last technique is meant to be a convoluted way to access the keyboard. The idea is no longer to use the official keyboard API to capture its content, but to go directly into the memory space of a process to retrieve the keyboard keys it receives. The goal is to hijack the targeted application. The idea is to perform a regular Dll injection [1172] with `CreateRemoteThread` function [746] for instance. From there, it is possible to access the memory space to retrieve the functions manipulating the windows' messages (`SendMessage` [816], `TranslateMessage` [838] and so on).

This technique is neither very original nor very discreet since it would require a Dll or, at least, that the attacking application directly modifies the memory of the target application to inject malicious opcodes or read memory. Such operation uses the regular API used by a debugger [201, 200], that is to say `OpenProcess` [1173] function to get access to the targeted process, optionally `VirtualAllocEx` [1174] to allocate memory for housing opcodes in this one, and finally `ReadProcessMemory` [1175] and `WriteProcessMemory` [1176] to read memory or update opcodes.

This method, which is more or less discreet, imitates the behavior of a regular debugger and it requires the same right (which is not always cheap) [1177]. Stealth gained on the one hand (by not using keyboard access API) requires specific (and meaningful) actions on the other hand (debugger actions or Dll injection). Afterwards, it should be recognized that from an antivirus point of view, it is complicated to detect this kind of threats that closely mimics legitimate debuggers without creating potentially damaging false positives. This does not mean that everything is allowed and some antivirus programs refuse this type of operation if it is not done by a well-known debugger.

Generally speaking, this method can be seen as the one using the `WH_KEYBOARD` hook. The difference is that instead of using the `SetWindowsHookEx` function, the procedure is implemented manually. It may be a little more discreet but it requires more skills (even if there are example codes and libraries that make life easier [1178]) and stability management if the targeted application is updated. In the end, the result is broadly the same. The limit is that this method only targets a single process, while `SetWindowsHookEx` touches more. The choice to use this method therefore depends on the objectives of the attacker.

### 3.3.4 Miscellaneous about software keylogger

Generally speaking, the fight against keyloggers has often been carried out using detection mechanisms. This is why there is research on how to make a keylogger undetectable [1179]. Generally, they do not try to update techniques to record keystrokes (since they are limited by the operating system which provides a limited number of API functions) but they try to enhance the obfuscation of the executable file holding the keylogger [1180]. But here, we are at the limits of what keyloggers can do to venture into reverse engineering protection (and more generally malware analysis).

## 4 Anti-keylogger solutions

### Resume 32:

- ☞ We propose a taxonomy of anti-keylogger prevention solutions in the following subsections.
- ☞ We make a distinction between academic solutions (published in the scientific literature) and industrial solutions.
- ☞ A state-of-the-art about *academic solutions* are divided into two main parts: *passive* and *active* solutions.
  - ☞ *Passive solutions* are about detecting keyloggers as a classical anti-virus software in order to avoid malware execution.
  - ☞ *Active solutions* aim to mitigate consequences of keyloggers without trying to detect them.
- ☞ A state-of-the-art about *industrial solutions* is done with public software aiming to prevent keylogger activity.
  - ☞ This state-of-the-art aims at making an inventory of the different strategies proposed by the solutions and not the technical details to know how they are implemented.
  - ☞ In addition to keep the coherence with academic studies (where one only reads the authors' statements), it is also an opportunity to bring a new working methodology different from reverse engineering.

Anti-keylogger solutions are really the heart of the subject we cover here. It is important to understand that if the threat is there, the research has focused on the subject in order to propose solutions. Among the proposed solutions, there are two types of solutions: *passive* and *active*. Passive solutions focus on detecting malicious programs, mostly by the side effects they generate. Active solutions aim at learning how to live with a keylogger in order to neutralize its damaging effects. More directly, it seeks to deceive or lock access to information for keyloggers. However, this type of protection must not result in a disruption of the user's experience with conventional programs.

In this section, we will present the academic and industrial solutions that exist (or have existed) on the market. We will try to identify their context of use, the problem to which they respond as well as their strengths and weaknesses when it is possible. The final goal is to be able to differentiate between all these products and to define how our solution, presented in section 6, differs from what exists or the characteristics which makes our solution better.

## 4.1 Passive solutions

### Key Point 5.17:

- ☞ The detection of keylogger threats (regardless of the method used) is very complex, not to say almost impossible.
  - ☞ Keyloggers capture keystrokes just like any other legitimate program.
  - ☞ They use the Windows API to get the keyboard data stream. And this is legitimate since it is perfectly documented.
  - ☞ The difference with a legitimate program lies in what is done with this data stream.
  - ☞ And it is very difficult to characterize the *intent* of a program.
- ☞ Two detection approaches are presented here.
  - ☞ Detection based on API functions used by a malware (as a traditional antivirus software).
  - ☞ Detection based on side effects inducted by the presence of a keylogger threat.

### 4.1.1 Detecting keylogger by API use

It is usual to find an academic article that explains how an antivirus does to detect keylogger type malware [485, 1181, 1182, 1183]. In general, it is assumed that the usual state of the art antivirus' detection is done through a knowledge base of the characteristics (through a collection of hash footprints) of the malicious files already known. This description is a bit simple since it does not take into account the complex reality of what a serious and modern antivirus software is today. It is easy to find serious studies on advanced keylogger detection methods [1184], not to mention the analysis of an antivirus itself, composed of drivers fully capable of intercepting the activity on the machine in real time.

One passive solution is to study one specific threat of keyloggers, based on functions hijacking and hooks techniques. Hook technique considered and used by malware is the one refereed, for instance, by Detour library [419], where a function is modified by a malware to detour its normal execution (as given in section 3.3.3). The method of detection proposed tries to detect if a hook is set by a malware. This method is a direct application of methods proposed in [1185, 1186]. The goal is to check if the functions manipulated by a legitimate process which access the keyboard are not hijacked by a malware. This is performed by checking the opcodes in the file hosting the analyzed program and its executed image in memory. In addition, address in the import address table (IAT) (where the link between Windows' API and the executable file is done) is investigated to know if there has been no unexpected modification.

If this technique may appear simple and only addresses a sub-part of the problem (only malware using this technique), the method is still very effective. Indeed, it targets the way the malware operates. Since it is a necessary checkpoint and the means of attack are known, it is quite possible to detect the threat that way with great success. But if we are looking false positive rate (especially in [1181]), this one may be important because any application using for legitimate reasons this technique (typically an antivirus) will be erroneously detected.

The other hook technique evoked in the context of keyloggers is the use of the `SetWindowsHookEx` function. For instance, [485] evokes two techniques of detection with [1187, 1188] to detect such keyboard hook. But these two detection techniques are based on regular hook detection and not on the detection of the use of `SetWindowsHookEx` function. We may suppose the cited papers have not been correctly understood by [485]. A confusion may have been performed between general hook technique (Detour [419]) and the function `SetWindowsHookEx` which has "hook" in its name in addition to be able to act on keyboard.

In [1189], there are different techniques proposed to perform hook detection based on the API used by a keylogger but the paper comes to the conclusion that any pure detection based on the API will inevitably generate false positives because legitimate software uses them too. And it includes detection based on the use of `SetWindowsHookEx` function, supported by [1190] which goes in the same direction. The last proposed to



disassemble all executable codes before running them, looking for `SetWindowsHookEx` function. Even if it could look smart, this solution is technically impracticable (there are so many ways to deceive disassembly with a self-modifying code) and naive (there are several ways and functions to use to design a keylogger, as shown previously). This may explain why this paper does not propose any detection and false positive rates.

Another approach called STARK is proposed by Tilo et al [1191]. The idea is to perform a scan of the system before booting it. Using a live USB boot-key, it is possible to start a small operating system able to scan the disk and engage regular detection techniques. This approach has the advantage of analyzing without having to suffer from any malicious action of any malware already running on the machine. Thus, we are not disturbed here by possible rootkit techniques. For the rest, the detection procedure is quite classical. Note that the project offers the ability to securely verify the authenticity of the PC before entering their password by using one-time boot prompts that are updated upon successful boot.

#### 4.1.2 Detection based on keylogger side effects

An original method of detection is not based on technical aspects of a keylogger but rather on the side effects induced by the last. Studies start from the observation that access to keyboard data is not in itself a malicious activity (so trying to detect such behavior is inevitably leading to a high rate of false positives) but it is in the way the collected data is used that it is possible to find a malicious behavior.

Method proposed in [39] aims to perform detection on the network activity generated by a keylogger in order to detect it. Indeed, keystrokes collected are intended to be used by an attacker, in one way or another, to exploit and gain more privileges — thus by exfiltrating these data. The network is obviously an excellent way to access this information, both in terms of the ease it offers and the flexibility provided. The idea is therefore to see if there is a correlation (in the mathematical sense of the term) between pressing the keys on the keyboard and network transmissions. If the approach proposed in the paper is intended to be experimental and designed as a proof of concept, it must be acknowledged that the results obtained are rather interesting. To ensure results and to reduce false-positive detection, a forensic in-memory analysis of the binary is done to look for specific keylogger traces and if any network activity is seen to be correlated with keyboard activity from the the targeted process. The idea<sup>19</sup> of this detection may be based on the fact that keyloggers are generally *cheap* programs. They are simple programs with a lot of example codes online (as shown) and it is easy to reuse them as they are. After all, keyloggers are not very complex program by design. Thus, malicious developers are tempted to add code as soon as a key is pressed to immediately forward the information. Certainly, it is simple and effective. But this is naive because the correlation is very easy to show.

This observation can easily be transposed to [1192] article. In this article, the approach favors a correlation between the keystrokes and the input/output (I/O) activity on the hard disk. The article is quite formal and it makes differences between several areas that make sense only to explain the concept and the methodology of their analysis. In fact, the authors will evaluate several malware by simulating keyboard activity using the `SendInput` function (Chapter 4, section 5.2.6, Key-Point 4.44). In real world, this is the real user using its own keyboard which is sufficient for detection — simulation is only used for laboratory evaluation purposes. Such evaluation has already been used for detection purposes [1193]. From there, hard drive activity is analyzed using *Windows Management Instrumentation* (WMI) [1194]. In particular, the performance I/O counters of each process are made available via the class `Win32_Process` [1195]. Thanks to this mechanism that simulates the keys of the keyboard (following statistical models to reproduce the behavior of a real user) and that recovers the keyboard activity, it is possible to reach disturbing results of efficiency on some keylogger software.

Here again, the malware architecture is the most relevant point. Just as some network keyloggers are tempted to forward keystrokes as quickly as possible, others store them in files as soon as possible. This is usually the standard architecture found on online codes. There are potentially several reasons for such an architecture: the codes are given as examples, they are not intended to be fully operational and there is no optimization issue since keystroke is not such a frequent event from the point of view of the CPU (which is nowadays clocked at high frequency, so much so that the `Sleep` function is sometimes necessary in some keyloggers). But the main reason

---

<sup>19</sup>This one is not explained that way but it can be deduced with our analysis and our practice of malware analysis.



could be simply because it does the job without any antivirus program able to detect such a malware's behavior.

The effectiveness of this proposed method of detection is essentially due to the fact that it is the keylogger's side activities that betray their true behavior and not the access to the keyboard itself. Not relying on a given technology used by the keylogger to perform a detection is a great strength because it allows a generic approach to the problem. But this solution is not perfect because it does not cover the whole diversity taken by keyloggers. In fact, it presupposes a "cheap" keylogger architecture, taken directly *out-of-the-box* or built without any real consideration of stealth or efficiency. The case of kernel mode keyloggers is an illustration of an advanced keyloggers that does not necessarily respect an architecture for immediate keystroke processing. This may be due to the IRQ level where keystroke interception is performed (and which does not allow access to hard disk device, which is usually reserved at a passive level) but also for optimization purposes. For instance, it could make sense to send a file over the network only once when the machine shuts down or to write only when several keys are already registered and general activity of the system is low (sections 3.2.2.2 and 3.2.2.3).

## 4.2 Active solutions

### Key Point 5.18:

- ☞ The solutions presented here aim to mitigate the possible malicious actions performed by a keylogger.
  - ☞ The idea is not to detect keyloggers but to make them inefficient.
  - ☞ In practice, this means jamming the received data from the device or ciphering data to make such a data not readable.

Active solutions are more about decoy or neutralize any keylogger by active means. In a way, the goal is not really about detecting a keylogger but more about to know how to live with it without any (or almost) risk.

### 4.2.1 Active defense against user-mode keyloggers

If finally keyloggers use the Windows API to achieve their goals, why not neutralizing this API to neutralize keyloggers? This is the idea exposed in [41]. In this article, the author exposes the internal mechanisms of the Windows keyboard functions (user-mode only). This is a reverse engineering of Windows 7 32-bits version and its API in order to see how it is possible to detect a process (usually from the Windows kernel with a driver or a debugger) used by the keyboard and potentially neutralizing access of the targeted process to the device.

Article [41] is short and quite technical. This one is mainly addressed to an audience that masters reverse engineering and Windows. But it is conceptually limited and not fully operational. Why? Because like our study<sup>20</sup> in Chapter 4, the data presented here is only a snapshot of Windows at a given time. The described structures often evolve according to Windows updates and this is why it may be hazardous to rely on them. Today, few of the methods presented in this article would be able to work without redoing the work of reverse engineering or even changing the method for a solution able to work without any debugger (which removes security as *PatchGuard*).

In addition, the techniques presented here detect keyboard usage by a process but not the malicious purpose of a keylogger. Such detection is much more complex than the one proposed here. It goes without saying that even if it presented relevant results, there would be a highly significant false positive rate if this method would be applied to legitimate applications. Such comparison between legitimate and malicious applications is not provided in the article, only the technical detection based on undocumented structures from Windows is presented. More directly, this solution is not only technically obsolete quickly (due to Windows updates), but it is conceptually not able to perform a decent detection job since it would detect any application using the keyboard.

---

<sup>20</sup>We must be under no illusion on this point, our study is fated to decline, at least in the details it gives of certain aspects of the keyboard implementation under Windows 10. But the general idea and approach of starting from PS/2 or USB/HID nevertheless helps to explain why and how some major principles will always be implemented as we have described. In addition, main routines and functions should be not subject to change in medium-term. This is an important difference with [1169, 41].

## 4.2.2 Decoy techniques for keyloggers

### Key Point 5.19:

- ☞ In practice, decoying a keylogger means *jamming* it in several research papers.
- ☞ Jamming the data flow from the keyboard can be done with a driver that inserts false keystrokes.
  - ☞ Jamming procedure is performed by flooding the keyboard device stack with fake random keystrokes.
  - ☞ The difficulty is not so much in the jamming procedure as in not negatively impacting the user experience.
  - ☞ Random keystrokes should not appear magically in the middle of a document displayed on the screen.
- ☞ Two approaches are possible: *local* and *global*.
  - ☞ The *local* one targets certain processes to protect them from keyloggers, whereas the *global* one affects the whole system.

Based on the observation that we cannot effectively detect keyloggers (without risking false positives), we may not try to detect them in order to remove them, but we may try to nullify their actions. Finally, a keylogger retrieves keystrokes to transcribe them into a file or send them directly to its designer. This stream of captured data makes sense as long as the data collected corresponds perfectly to what the user has typed in.

The idea is therefore to disrupt this captured stream by injecting false keystrokes so that they can be recorded by a keylogger without leaking any relevant information. Of course, this action must be performed without disturbing the system by itself. The idea of *jamming* systems in the fight against malware is not new [1196]. In a generic way, the procedure acts as if a second keyboard would have been inserted into the device stack to *flood* the input stream of the real device. This can be done through a given software that simulates a keyboard. The idea is the processes that listen to the keyboard will certainly have the keys entered by the user, but also those provided by the protection driver. The result will then be unreadable for anyone as long as it is not possible to make a clear distinction between the real keyboard and the simulated one. This is where the security lies. But it is not without restrictions. Hence, the main problem is not so much about the jamming of the keyloggers than the non-disturbance of the system by this unexpected injection of keystrokes. For instance, when using notepad software, it must be ensured that fake keystrokes should not be inserted randomly in any legitimate software, otherwise, fake characters will be displayed on the screen, which would result in significantly altering the user's experience...

Figure 5.17 is an illustration of the desired architecture. On the lowest layer, keyboards transfer information provided by the user. It is processed by the appropriate drivers for the keyboard technologies used (USB or PS/2 — section 4). From that point, it is the role of `kbdclass` to handle all keyboard information, whatever the technology of the keyboard device is. The goal is to insert a device driver into the device drivers stack in order to insert fake keystrokes before the `kbdclass` driver. That way, keyloggers which are logging after `kbdclass` (in most cases — section 3, Key-Point 5.5) are about to be jammed. The same way, user-mode keylogger will receive decoyed keystrokes stream. The most important point of this technology is to correctly route the stream so that legitimate applications are not themselves jammed.

Two papers [42, 11] describe this approach clearly and they illustrate two different strategies for implementing it. The first is about a targeted solution aiming to protect only specific applications. The second is more global in order to protect the whole system. Both solutions have pros and cons, which aims to be described in the following paragraphs.

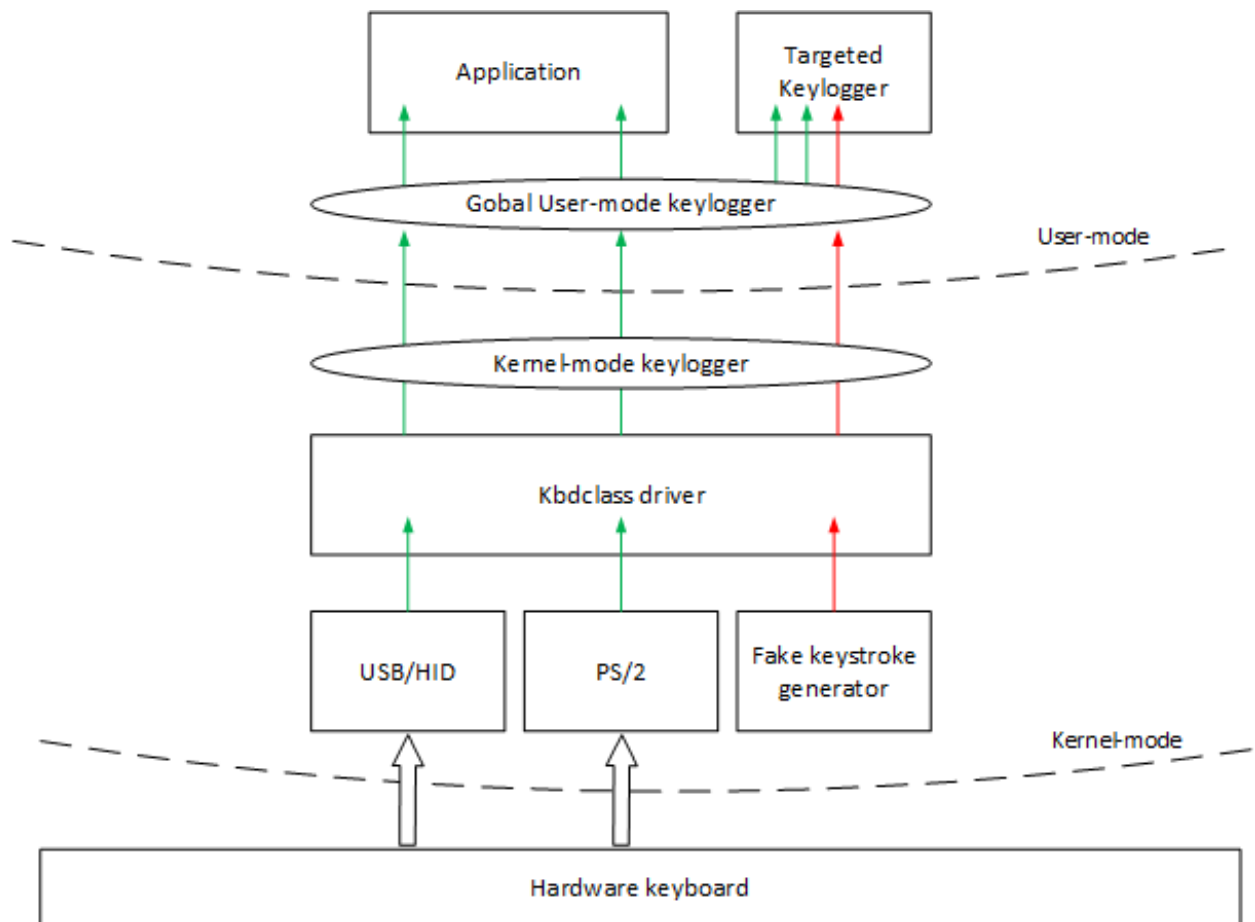


Figure 5.17: Illustration of a decoy architecture to fool keyloggers and preserving the whole system.

## 4.2.2.1 Targeted decoy

**Key Point 5.20:**

- ☞ Targeted technique that aims to simulate the keystrokes when a protected process has the keyboard focus.
  - ✍ The simulation is done with `SendInput` which impacts the whole system (and therefore any keyloggers).
  - ✍ With this additional stream of random data inserted in the original one, keylogger threat is jammed.
  - ✍ Decoy procedure is stopped once the protected application loses the focus to not impact user experience.
- ☞ Some drawbacks:
  - ✍ The use of Dll injection is not a very reliable solution. In addition, Dll injection from malware side could bypass the protection.
  - ✍ Malware based on DirectX library would ignore such a protection.

In [42], published in 2012, a project called "NoisyKey" is presented to transparently flood with dummy data the *event channel* (i.e. the message system used with GUI windows) used to deliver the user keystrokes to the intended application. Technically, authors designed a Dll (called `noisykey.dll`) to be injected (with regular injection Dll techniques such as [1172], for instance) in a targeted process which should be *keylogger protected*. This Dll has two purposes: injecting noise in the form of dummy keystrokes and removing them before they reach the protected application. To proceed, it manipulates from a dedicated thread in the protected application the internal flow of keystrokes handled by `csrss.exe` via its raw input thread (point (a) on Figure 5.18 extracted from [42]). This is done via `keybd_event` function [1197] which is an obsolete function now [727] (`SendInput` should now be preferred instead of `keybd_event` function). The injection of noise is driven by a complex statistical mechanism which — according to the authors — is essential to fool keyloggers as well as possible (it mimics real user activity and it avoids keyboard shortcuts which would impact the whole system — CTRL+ALT+DEL, for instance). From a technical point of view, this part improves the quality of the protection but it does not change its general principles. The other action of the Dll is to remove the noise injected in the raw input thread. From the authors' analysis, it is often through `user32.dll` that the interaction with the keyboard is managed and in particular with the `DispatchMessage` function. Thanks to the Detour framework [419], the Dll intercepts any call to this last function in order to remove the keystrokes previously injected (point (b) on Figure 5.18).

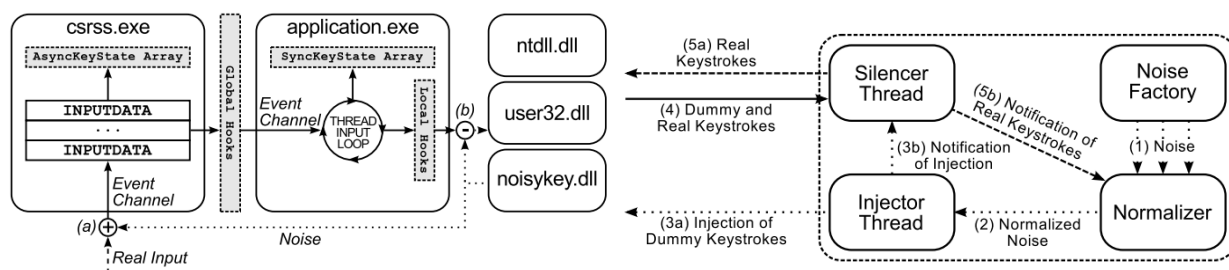


Figure 5.18: General architecture of NoisyKey project.

Figure 5.19: Architecture of `noisykey.dll`.

The purpose of the Dll (Figure 5.19) is simply to implement this architecture. There is a split between the various modules composing the Dll (noise generation with *Noise-factory* and *Normalizer*, noise injection with *Injector-Thread* and the noise filter with *Silencer-Thread*). One of the key-point of the decoy technique used to jam keyloggers is the way the injected noise is removed for legitimate applications. In [42], this is the *Silencer Thread* which interrupts the *Injector Thread* at each invocation to remove all the dummy data injected from

the last invocation.

Tolerating the presence of a user-space keylogger by injecting flood should not prevent other applications to work properly. Here, the security mechanism is only activated for a targeted application (which receives `noisykey.dll`). That way, it constantly checks that the protected application has the focus of the keyboard. When this one has the focus, the security is in place. When it loses it, the security stops to not impact other applications in the system. That way, the protection is only activated for the protected application when this one is able to receive information (via the keyboard focus) from the keyboard device.

This was tested on several keyloggers with some success in [42]. But this solution has some weak points. The first is that it relies on Dll injection and function hijacking mechanisms, which strongly looks like means implemented by malware. According to the authors, several antivirus software detect this solution as being malicious. Moreover, it is based on the hijacking of mechanisms that are not documented and therefore prone to change. In addition, as explained in Chapter 4, section 5.3 (Key-Point 4.46), keyboard management for a legitimate application does not rely solely on message management and the `DispatchMessage` function. Finally, the proposed solution is said to be *entirely unprivileged*. This is not exactly true. Protecting a privileged application would require performing a Dll injection with at least the same privileges of the privileged application. Not to mention the fact that a keylogger in kernel-mode can avoid or bypass this type of protection (by detecting the hook procedure in memory).

It can be noted that the solution protects against keylogger threats that would base their eavesdropping system on mechanisms that do not require keyboard focus (Key-Points 4.50 and 5.14). Indeed, the keystroke simulation system (which uses the ancestor of `SendInput` but whose principle remains the same) goes through the raw input thread (Key-Point 4.44). This means that not only the message system will be impacted by this decoy technique, but also all other systems underlying the raw input thread, which is a very good protection in the end. Nevertheless, it is necessary to see a limit with the injection of Dll. Indeed, if the protection system could do it, a malware could also do it. If a Dll injection would be performed by a malware, it allows a direct access to the keyboard, despite the defense in place. Note also that if the malware is based on the DirectX library (Key-Point 5.14), which ignores the simulation system based on `SendInput` (Key-Point 4.58), the solution will be ineffective.

Finally, this targeted decoy solution presents relevant points. The first comes from the fact that it acts within a targeted application framework. It is an important choice of the context in which the solution will be used, with strengths and limitations. On the one hand, such a design provides targeted protection only for the sensitive applications that the user wishes to protect. On the other hand, it only protects the necessary time where the sensitive application is in use, considerably reducing the possible nuisance and the impact in terms of performance. Of course, this presupposes that the user defines which applications are relevant. But it is still possible to generalize the principle for each application by systematically injecting the Dll into any process. *De facto*, each application would have its own jammed distribution channel. Nevertheless, one has to see the induced cost of such a solution (CPU performances, stability about Dll injection with some targeted software, potential security break provided by the Dll...) without mentioning the questionable necessity for certain applications (which do not use the keyboard or for nothing sensitive - video games, drawing software, etc.).

#### 4.2.2.2 Global decoy

##### Key Point 5.21:

- ☞ The global jamming solution is not viable, both with impact induced on the user experience, and the architecture of the latter as proposed in the literature.
  - ☞ In this subsection, we review an article [11] to show that without additional information coming from the authors, it would be possible to question some of their assertions.

### General idea as presented by Julian Rrushi & Co.

Another approach is to generalize the luring with an architecture which is embedded more deeply in the operating system. This is the approach taken by Julian Rrushi & Co. [11] in 2017. Contrary to [42] which proposes an operational approach to "live with" keyloggers, we are here in an intermediate approach which aims to detect keyloggers while decoying them. The idea is to develop here a driver that simulates the action of the keyboard whenever there is no more direct input activity from the user on the machine. The authors of the paper propose to carry out this insertion of virtual devices at the level of HID keyboard devices (Key-Point 4.26). Their goal is to appear as a standard USB keyboard in the Windows device tree. It improves the stealth of the solution since it is invisible to the user and indistinguishable from a real keyboard from malware eyes. In fact, the solution needs to let the keyloggers act to better detect them.

Authors use an open source project called *vmulti* [1198] to simulate HID input. This project is a HID filter-driver<sup>21</sup> using the Kernel Mode Driver Framework (KMDF) to emulate sort of actions performed by a HID device. The driver is driven by a user-mode application which retrieves the virtual device created by *vmulti*'s driver thanks to `SetupDiEnumDeviceInterfaces` function [1199]. *De facto*, it is the user-mode application that forges the HID requests it wants to see issued by the driver. It is fairly basic but it produces the desired effect: simulation of mouse or keystrokes. In their projects, authors call this driver *Kshadow*.

*Kshadow* is inserted as a filter driver positioned between the keyboard class driver (`kbdclass.sys`) and the keyboard HID client mapper driver (`kbdhid.sys`). Driver *Kshadow* creates a filter device object (FiDO) [1200, 1201]. This procedure is only about to attach the driver to the stack of the device that the driver filters, which means in this case the keyboard's device stack. The objective by registering with the I/O manager as a filter driver is to allow *Kshadow* to see all IRPs bound for the keyboard, whatever the request comes from physical or decoy driver. The decoy driver (responsible to produce fake keystrokes) is called *DK* in [11], which seems to be logically the driver inspired by the *vmulti* project. According to [11], all communications between *Kshadow* and *DK* driver take place through direct function calls, and do not involve the I/O manager. More directly, it seems that *DK* is imported thanks to the import address table (IAT) of *Kshadow* as a kernel library of routines. From the point of view of the operating system, *Kshadow* and *DK* are one and the same driver.

*Kshadow* is responsible to manage the period of real user keyboard activity and those of inactivity where the decoy procedure can be engaged. To know that there is not activity provided by the user, *Kshadow* measures the time elapsed between the press of two keys. If the latter is long enough, it sets up the decoying procedure. From a practical point of view, the authors use a process called *dprocess* to receive the decoy keys generated. That way, sending decoy keystrokes (in addition to the legitimate keystrokes of the user) to a dedicated process floods the keyloggers that record the illegitimate keystrokes. But it does not impact the system since the keystrokes are handled (even to be totally ignored or dropped) by *dprocess* and not by an unexpected process. This architecture used is clearly resumed in Figure 5.20 extracted from [11].

### Discussion about the aforementioned work

Notwithstanding the detection part based on the ability to watch escaped data generated by their driver from a keylogger (on network, for instance), the main idea of the authors is based on this two-drivers architecture. While the general idea is easily understandable (drivers generate noisy jamming on the whole system to fool keyloggers during inactivity periods), there are still some questions regarding what is announced in their paper and our observations from our study. On this point, we must be very clear and say that our analysis could only be based on the authors' paper since the project (to the best of our knowledge) is not open-source. With this source-code, it would be possible to answer clear questions and maybe to chance some observations we could write. More generally, in research area, the lack of auditability or reproducibility of results forces trust in the authors. But it is sometimes possible to analyze certain statements in the light of others' knowledge or with our own research.

---

<sup>21</sup>Technically speaking, *vmulti* is a KMDF filter driver on `mshidkmdf.sys` driver, a driver of Windows registered as a "Pass-through HID to KMDF Filter Driver". That is to say, it is not directly part of the HID device stack but a sort of access point for any KMDF driver which would need to interact with HID devices. In addition, *vmulti* is loaded part of the "PNP Filter" drivers load order group.

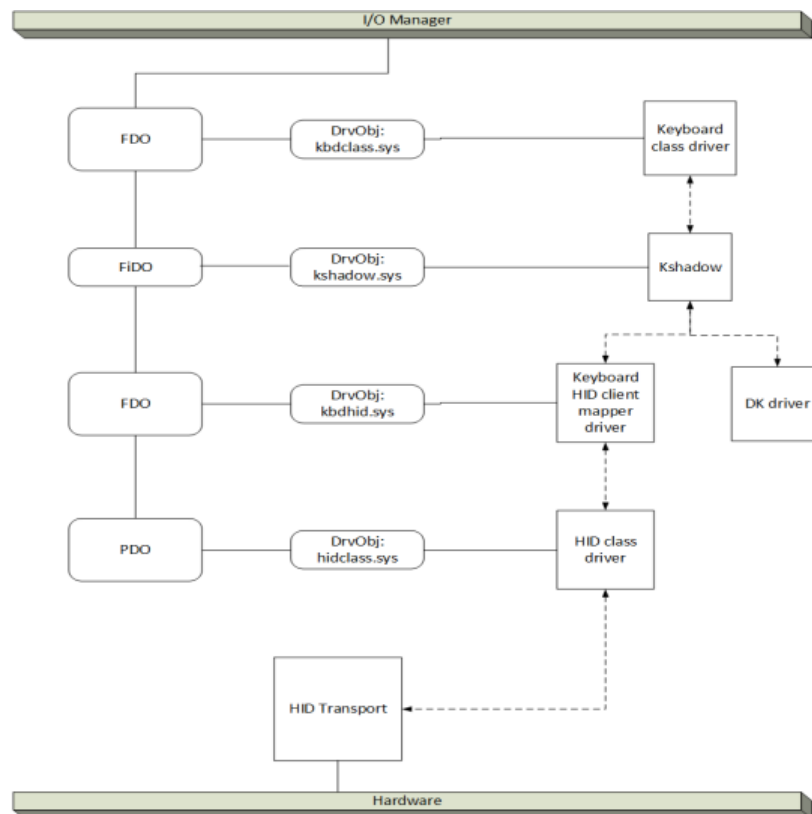


Figure 5.20: Integration of keyboard shadowing with the driver stack of an HID keyboard from [11].

Firstly, a question rises in the choice of vmutil for authors' project. Indeed, in practice, one would think that the authors would have used a *Virtual HID Framework* [20] to implement their virtual keyboard as a driver in HID context. This is the practical solution to achieve what they wanted to do especially the way they claim to do it. But instead, they preferred to use another technical solution based on an vmutil. Note that this solution is enough for doing what they want to do.

Then, authors of [11] announce that they have modified the vmulti solution to achieve their goals, without further technical details. Surprisingly, they insist a lot on the notion of IRP. Even if keyboard is managed through IRPs (Key-Point 4.34), with a driver using the KMDF framework [1157] and because of the vmulti architecture, they do not normally have to deal with IRP directly. This is the user-mode application linked to the vmulti's driver which initiates the request after having forged it by itself. Such requests are sent via HID API used as an original communication channel... And vmulti's driver does not directly handle IRP other than with the specific API of KMDF framework.

When reading the article, the way the keystrokes are sent from the kernel DK component to *dprocess* is quite confusing. This is where the paper takes some distances with the technical knowledge we have established about how the keyboard works under a modern version of Windows (Chapter 4). From the authors' paper, they clearly wrote about their solution:

*"The defender's objective is to proactively misdirect malware into intercepting keystrokes emulated by the decoy keyboard. An attack surface is created by sending emulated keystrokes to a decoy process (dprocess), which runs in user space."*

It means that a user-mode process (called *dprocess*) is responsible to receive fake keystrokes. In addition,



authors claim that Kshadow has access to the process ID of *dprocess* so that “*Kshadow knows whether an IRP originated in dprocess or in another process*”. That way, Kshadow can make the distinction to know when to jam keyloggers or not. Indeed, it knows which process requested access to the keyboard device. Moreover, the rest of the text is very enlightening to understand how the project work.

*“Furthermore, for each IRP, Kshadow retrieves the process ID of the thread that requested the I/O operation. This is how Kshadow knows whether an IRP originated in dprocess or in another process. Regardless of the source, Kshadow receives IRPs from the keyboard class driver. If an IRP originated in a process other than dprocess, Kshadow passes it down to the keyboard HID client mapper driver. If an IRP originated in dprocess, Kshadow simply passes it to DK driver, which populates it with the scancodes of emulated keystrokes.”*

But the operation of the IRP management controlling the keyboard is not decentralized for each process (Chapter 4, sections 5.1 and 5.2, Key-Point 4.28). This is the procedure of broadcast through the Windows’ messages system queue driven by the raw input thread (Chapter 4, section 5, Key-Point 25) from csrss.exe which is responsible to handle keyboard’s IRPs. It is easy to prove with a debugger that is csrss.exe which manage requests to the keyboard and not any running process, as given in Key-Point 4.36.

From the kernel point of view, managing IRP in the context of the calling process (such as *dprocess*) does not make any sense since it is always csrss.exe process’ raw input thread which is responsible to handle this operation. More directly, Kshadow has no way<sup>22</sup> to send keys directly to a targeted process via an IRP<sup>23</sup>. This section in [11] is clearly inaccurate. But it does not mean that their solution does not work. It just works differently from what they claim.

Indeed, conversely, if we take the architecture proposed by vmulti, where it is the user-mode process that initiates the request for the driver, it could make sense. In this case, we must consider that *dprocess* permanently generates a key generation request to be executed by the DK driver. Such operation could be technically implemented through an IOCTL request [614]. The fake keystroke can be generated by delaying the response to *dprocess*’s requests (by synchronizing the request between the process and the driver, thus with overlapping [1203]) or by looping endlessly around a generation request and only responding to them whenever Kshadow considers it is necessary (with a system of “go” and “stop”<sup>24</sup> orders, as explained in the paper). That way, during a period of inactivity, the driver DK responds to requests by generating HID commands issued by *dprocess* so that they can then be retrieved by the one at the end. That way, it matches authors’ statement about “*dprocess requests keystrokes, and the DK driver generates them*”. This looks to be the only possible solution to match authors’ requirements. Even if this technical correction could help to understand the architecture of the solution, that one has more or less undesirable consequences, as presented in the paper.

Indeed, the results are not very eloquent because they focus more on some observation of delays induces by drivers on system (due to keystroke simulation) than detection. In addition, there are observations of questionable behavior, for instance with the screen saver. Authors wrote:

---

<sup>22</sup>More precisely, no *documented* way to proceed. Even if there is `IoGetRequestorProcessId` routine [1202], this one is only relevant for IRP issued in the context of the caller (such as `IRP_MJ_CREATE`, for instance. Of course, we can always imagine ugly and undocumented procedures to retrieve what the author call the “*process ID of the thread that requested the I/O operation*”. It is stored in the IRP as shown in Chapter 4, section 5.1.1. But it has two drawbacks. On the first hand, it is not reliable to use undocumented fields in an undocumented structure and on the other hand, because, in this case, it will reference “csrss.exe” process, as given in this section 5.1.1...

<sup>23</sup>Not to mention that even if it would be possible for a driver to send an IRP containing a keystroke to a process waiting for it, this operation would short-circuit the raw input thread. More directly, intermediate buffers (used by `GetAsyncKeyState` function — Chapter 4, section 5.3.1.4, Key-Point 4.50) and input messages would simply not be transmitted. This would mean that keyloggers using these technologies (because there are no more malware able to direct read from keyboard) would simply not see the decoying keys generated by the combo of Kshadow and DK drivers. A paradox in itself!

<sup>24</sup>For the sake of completeness, we can mention that Kshadow *knows* it has to stop decoy procedure when it notices a completed IRP is coming from underneath, meaning the physical keyboard has become active. For short, its mechanism of activity detection is based on completion routines [689, 695, 690].

*“However we saw no anomalies with the use of the real keyboard when we returned to work on the computer. The only observable was that the screen saver and the power saving mode appeared to be affected by the operation of the decoy keyboard (...)”*

System is woken up whenever the keyboard keys are simulated by the decoy system. Authors assume that *dprocess* requests keystrokes and the DK driver generates them, Windows assumes that there is a user working on the computer. This is more or less true, but it is not necessarily due to *dprocess* activity.

Without explaining the detailed functioning of the screen-saver, the latter is implemented in both kernel-land and in a user-land process driven by the system [1204, 1205, 1206] (it is even possible to use our own screen saver process [1207]). In a more direct way, the disturbance of the screen saver could be the fact that the system processes are also impacted by the decoy keystrokes generated. Indeed, the generated keys are not intercepted and digested by *dprocess* only. This means that they are also received by the other processes. Thus, the process that manages the screen saver could be impacted by receiving an input message. In consequence, it could decide to cut the screen saver procedure. Note that if this is the kernel which manages the screen saver, it does not change anything. In this case, it is up to the raw input thread procedure to manage the keyboard. Indeed, the key generation is *in-fine* processed as if it would come from kbdhid.sys driver (Chapter 4, section 4.2.6) which itself notifies the kbdclass.sys driver in relation with the raw input thread. In the end, if the screen saver is impacted, it means that all processes are impacted. Of course, we might imagine that their solution does not impact other processes with fake keystrokes, but what about the benefits of their solution against malware processes, which would not be impacted neither?

This negative consequence could also have another side effect. Supposing they are broadcasting fake keystrokes as they claim, as soon as the user stops using the keyboard long enough, the keyboard looks to work automatically and randomly. It is not hard to imagine the surprise of a user who reviews (reading without using its keyboard) a document in a text editor and observes the content of the document automatically modified without having touched the keyboard keys. If this consequence did not appear to the authors of the paper, it may be because they may have forgotten important details about how to avoid this pitfall (and this is annoying from a scientific point of view), or the tests may have not been performed seriously enough. But this mystery might be solved in the explanation of the detection results they provide.

The authors claim to realize a detection thanks to their system which can be declined in two types (called *two orders* in the paper). According to [11]:

*“First order detection of a keylogger happens when the malware attempts to intercept keystrokes and is detected in the act. Second order detection refers to detection at a time that postdates the keystroke interception mounted by a keylogger”.*

In the last case, it can happen when the malware contacts its C&C or when it sends information about a VPN where user’s credential would have been logged. Technically, the authors expose the fact that they performed tests on 50 malware but they do not give the true detection rate, for both orders of detection. In an original way, they just say that all the malware have intercepted the simulated keys. Observations are reported directly via the malware control interface (GUI) or with a debugger and breakpoints in samples to check the content of the exchanges. This confirms the first order detection type. Concerning those of second order, it seems that they are sometimes “supposed” because the authors having operated on a test bed isolated from the network, they were not able to guarantee the proper behavior of some malware.

In addition, authors claim they analyzed architecture of well-known malware with the IDA Pro tool. They wrote that:

*“(...) some of the malware samples use keylogger modules that intercept keystrokes by probing their target keyboard. Those probes resulted in IRPs bound for the target keyboard.”*

Again, this is not how keyboard interface works under Windows, *a fortiori* keyloggers as we have presented them (Chapter 4, section 3, Key-Point 5.5). This dubious assertion by the authors deserves further investigation because it seems to corroborate conclusions that go more in the direction to justify what has been developed by them than in the direction of what has been observed and what is observable. Technically, they are using either message queue interface (Key-Point 5.2.2), hook procedures (Key-Point 5.3.2), or asynchronous keyboard state check (Key-Point 5.3.1.4). None of this procedure produces an IRP since they are all resulting from the raw input thread IRP management.

Whatever is the technical accuracy of the article and even supposing it could be correct, the article presents an interesting approach in the detection of keyloggers threat. The second-order evaluation based on the consequences of the keylogger (sending data through network) is quite relevant since it is in line with the work performed in [39]. But it remains that the proposed solution has a serious flaw.

Why would we use a technique like decoy and be interested in making the approach as transparent as possible to both normal users and attackers if it is not to use it on systems with real users? It is written in the conclusion this approach “*can operate on computers in production*”. Indeed, the decoy keys are inserted during periods of user’s inactivity, which does not impact user’s experience. That is to say, the detection system is designed to be used on real computer, processed by real users in real life. Detection should ensures the security while decoy system protects potential information retrieved by malware before detection...

But, unlike [42] which continuously jams as soon as the application is active by taking the focus on the keyboard, in [11], the jamming procedure is only activated when there is no activity. And therein lies the problem. Hence, this implies that there is no jamming of what is typed by the user. Instead, we have sequences of keystrokes typed in by the user, sometimes interlaced with random keystrokes during inactivity periods. To make a long story short, sensitive data (credentials, pin code, sensitive paragraphs and so on) are completely readable with this solution, as long as we are looking for coherent text in a portion of continue incoherent text sequence. An illustration of a log file from a keylogger with this protection solution would look like Figure 5.21.



Figure 5.21: Illustration of the sequences between user’s keystrokes and decoyed ones. It is not hard to find user’s relevant information inside log files for keylogger’s managers.

But there is even better. Assuming that authors are really able to target the process (*dprocess*) with which their driver sends keyboard keys (as they claim), it would mean that a malware can completely identify which process is receiving what. More directly, malware would be able to perform distinction between the keyboard’s data from a web browser and the data from what they call *dprocess*. And this is a good thing, because modern malware are already doing it! Of course, they are not using the process’s ID of the IRP handler. They use the name of the application (with `GetModuleFileNameEx` [1208]) in which they could have been injected thanks to `SetWindowsHookEx` [1] function for instance, or which process has the focus of the keyboard (for instance with `GetActiveWindow` [868]). Such examples of malware are easy to find [1209, 1210]. Afterwards, for sure, real keyloggers malware are usually based on the keyboard focus that corresponds to the technology deployed under Windows, rather than a dubious IRP story that the processes do not manipulate since it is a pure kernel-mode concept.

It may be necessary to look at the solution proposed in [11] with distance and with a critical eye. Not all technical information provided in the paper are accurate and some statements may tend to contradict each others. In the same way, the explanations for the results could have been improved to be more accurate, with detailed detection rate, false positive rate and so on. However, this solution cannot be deployed on real machines

where real users manipulate potentially sensitive data. The keyboard's data is transmitted in clear-text to the keyloggers and the decoy approach is not efficient enough.

Our review of the paper is maybe a reason why scientific publications should provide, whenever possible, the source-code of the project and the possibility to reproduce the experiments. In this case, it would have helped to validate or invalidate some of the observations we wrote. Without this, it is unfortunately not possible to rely on anything other than the published article. And in this case, it is possible to question some points presented in this article... More directly, one can question the real relevance of the research presented in this paper and the message it may convey... It is finally specific to scientific research to bring a different perspective on past studies and to doubt in order to better prove (or change) certain assertions...

More generally and in the absence of any new solution on the subject we would not be aware, it is not possible to validate keylogger interference at global level. Even if this solution could potentially solve the blind spot of local solution with DirectX (because the keystroke jamming is performed at HID level, lower than DirectX in the device kernel stack), it is not enough. The reason lies in the fact it is hard to remove the induced noise to protect the legitimate applications of the system from the generated pollution. The screen-saver in [11] is only the cruel illustration of this principle. One solution would be to target each application and to remove the fake keystrokes, the same way as in [42] with a targeted decoy solution. But it would result to use the technology of the targeted decoy, eliminating global decoy solution. A solution could be to give to each application a dedicated keyboard access independent to each others. But this is clearly not the architecture used under Windows operating system.

---

### 4.2.3 Dynamic layout technique

#### Key Point 5.22:

- ☞ The idea is to change the dynamically the keyboard layout (in the broadest sense) while user is typing.
- ☞ This single technical solution deals with two highly correlated problems. Both are dealing with short texts to protect such as passwords.
- ☞ The first is *over-the-shoulder* attack, meaning to protect the user when this one is entering its password, supposing an attacker could log what is happening on user's screen.
  - ☞ The solutions are often original and play on the visual and user experience to enter a password (pictures, sounds, captcha, and son on).
  - ☞ Only usable for short text (passwords) and always a balance between security and user experience.
  - ☞ Still an active field of research adjacent to our study, but without a definitive solution.
- ☞ The second is to protect text by updating on-the-fly random keyboard layouts to *encrypt* keystrokes.
  - ☞ The idea is to change the global keyboard layout of the system while preserving the one of the application to be protected.
  - ☞ Since the translation of scan codes into virtual key code is done at the kernel level, user-mode keyloggers are fooled.
  - ☞ But it is often possible to recover the original scan code, not to mention the default keyboard layout of the user's session (Key-Point 4.43).
  - ☞ Note that it is possible to do a frequency analysis of each character in the text to try to guess the original keyboard layout (and retrieving the original text protected).
- ☞ In any case, the random layout is generally not performed low enough in the system to be really efficient.

One of the most popular methods to counter keyloggers is to introduce a dynamic layout into the system. The states of the art on the subject regularly mention this type of solution as having effective results [43]. The idea is to introduce for all processes interacting with the keyboard random layouts instead of the traditional QWERTY or AZERTY [1211]. There are two main approaches. The first is an application-targeted version while the second is a global system one.

#### 4.2.3.1 Virtual onscreen keyboard and over-the-shoulder attack

Targeted version aims to route the real keystrokes to one application only. The idea is to deny to keyloggers access to their usual means of listening. This can take many forms. Original solutions [14] are based on graphical methods to capture a user's password through various visual mechanisms coupled to a device (usually the mouse). This can consist of displaying a virtual onscreen keyboard with a random layout for keys. Such keyboards can take different shapes, from classical ones in Figure 5.22 to modern ones in Figure 5.23 without forgetting original ones in Figure 5.24. Whatever is the device (phone [1212] or regular computer), the idea is always the same: providing a way to send a password for the user without using regular keyboard usage. It can evolves third-party devices (mouse, camera to track eyes movements [1213], voice and so on) to a different keyboard layouts displayed on the screen.

Of course, such protection can be bypassed by advanced keyloggers which are perfectly able to take screenshots of specific applications (in addition to cursor's coordinates) when they are about to require password (or keystrokes) [1066]. The use of UI debuggers can sometime help to bypass such protections. Since a countermeasure is available for some keyloggers, researchers have tried to counter this countermeasure. There are different



Figure 5.22: Example of random layout from [12].



Figure 5.23: Example of random layout from [13].

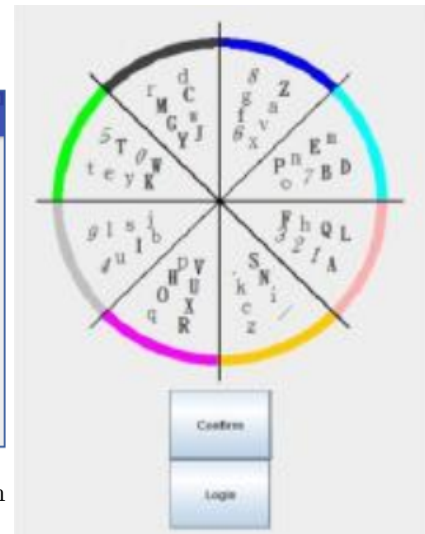


Figure 5.24: Example of random layout from [14].

solutions [1214, 1215] which have been developed to prevent such new threat. But to be totally honest, what the user sees and what the user types with the keyboard or interacts with a device can be intercepted by any software in the system. And as explained before in Chapter 4, section 5.2.7 about screenshots, it is possible to complicate the capture of screenshots with different techniques [1216, 963, 1217] but impossible to prevent it totally.

Some papers [15, 1218] claim that it is possible to design a solution able to bypass this limit. The solutions proposed are based on a virtual keyboard displayed where all keys are replaced by stars or unmeaning symbols. When the user sets the mouse pointer to a key, the text content of the key is updated with the real key character and removed as soon as the user removes the mouse pointer. That way, the users know how the random layout keyboard is and what to type on its own keyboard. Instance of such keyboard extracted from [15] is given in Figure 5.25.



Figure 5.25: Illustration of a random keyboard with hidden key's content from [15].

But this solution is far from being operational. Indeed, it ruins the user experience pretty soon. In addition, despite what the authors claim, it is totally possible to capture the shape of random layout for the keylogger with their solution. For short, conceptually the user needs to see displayed on the screen the content of the keys at least once (to know where they are mapped on the current instance of the random keyboard). From there, if the user can see keys displayed on the screen, the keylogger can do it too. Technically, it is enough to hook the message system of a window (Key-Points 4.40 and 4.52) to retrieve any display update messages for any GUI windows. More directly, it involves to deal with WM\_PAINT messages [1219] which are sent by UpdateWindow function [1220] when the window must be refreshed. This message happens every time a key is updated on the screen to be displayed to the user's eyes.



But even if it is possible to make password input more complex [1221, 13, 1222] (without ruining the user experience too much), it is not possible to prevent all attacks. Indeed, the security requires more and more an external device such as one-time password [1223, 1224] than the user's machine to deal with a password. Such device can be a smart-phone [1225]. It is possible to attack this device or simply involving the human factor with an accomplice linked to the attacker. The goal is to look *over-the-shoulder* of the victim to guess victim's password or the protection it uses. This is a research field in itself [12, 1226, 13] to design solution able to resist *over-the-shoulder* attacks while keeping a correct user experience when dealing with these solutions [1227].

If the evolution of threats have forwarded the use of randomized layout on-screen keyboards (this is often the case with websites of banks) or third-party authentication factors (sometimes required for online payment or gaming sites), it must be kept in mind that threats are highly adaptive to these security solutions. And the war seems to be already lost. Why? Because defensive solutions must deal with two major constraints. On the one hand, there is the conceptual advance driven by the imagination of the attackers and to which the defenders must respond. On the other hand, without restricting the user experience (who stops using the software if it is too restrictive or if it changes their habits too much), it is not always possible to go as far as one would like. More generally, the means of action installed in order to counter the unexpected logging procedure will inevitably clash with the means of action used to collect the password legitimately. More directly, keylogger threats need only to imitate legitimate means to intercept information between input interface of data and processing part to achieve their ends.

As a conclusion, an effective solution should not be impacted by being dependent on what is displayed or updated on the screen. On the one hand, because it can impact the user's experience. And no solution is good when it does not go with the user experience, so there is no reason to blame the user. On the other hand, because a malware running on the machine would be able to retrieve screenshots and record what happened. This is part of the criteria taken into account in the development of our solution.

#### 4.2.3.2 Random multiple layouts

In [1228], authors propose a new prevention technique based on the observation that each intercepted key must have been translated according to the current language-specific keyboard layout, selected by the user or application. The idea proposed is in line with the use of multiple layouts for applications. The goal is to update the keyboard layout for applications so that keystrokes look inconstant in order to fool keyloggers. As described in Chapter 4, section 5.2.5 (Key-Point 4.43), keyboard layout can be manipulated with the set of `LoadKeyboardLayout` and `ActivateKeyboardLayout` functions.

Technically, the authors propose that for each key pressed on the keyboard, the current keyboard layout is changed, and replaced randomly by one of the multiple predesigned layouts. Such layout can be one already registered in Windows or a specific keyboard layout as evoked in (Key-Point 4.43) when it is a custom one [929, 928]. According to authors, by this way, each keystroke received by user-land applications has been translated by the keyboard layout at kernel stage. That way, an application keylogger will log unreadable information because the keyboard layout is inconstant from keylogger's eyes. Of course, that way, all applications are impacted by this protection system. Thus, changing the keyboard layout affects both legitimate and illegitimate applications. To be operational, the solution integrates a way to translate the information to its original keyboard layout for legitimate applications. In this paper, the authors illustrate their results by presenting two graphical applications linked to the keyboard. The first is the legitimate application (which would be an email writer) which contains the characters readable in correct English, the other one, is a web browser running a keylogger which displays unreadable characters (probably due to lack of Unicode support). Thus, the keylogger has no access to the data whenever the legitimate application has access to it.

More directly, this solution acts on the virtual key code used by applications. Virtual key codes are dependent of the current keyboard layout while scan codes are device hardware dependent (even if generally, they follow convention described in Key-Points 4.2 and 4.6). In the paper, a distinction is made between the scan code which would be the prerogative of the kernel-mode Windows' drivers while the virtual key codes would be reserved to applications only. That way, scan codes would be translated into virtual key code to be subsequently used as virtual key code by applications. Technically and as shown in Chapter 4, section 5.2.7 (Key-Point 4.45),



the translation is effectively done at kernel level. Once the translation is done, authors explain that the keyboard driver generates a `WM_KEYDOWN` [848] windows message which contains this virtual-key code information. The proposed solution depends on that principle.

This last point is relevant since it betrays different approximations about how the Windows operating system works. Furthermore, it involves the inherent security of the solution provided. For example, [1229] explains that the solution proposed in [1228] is only valid for user-mode applications. That way, it cannot prevent a kernel-based keyboard filter driver to record keystrokes (scan code) when this one is recorded before it is translated via the keyboard layout. This is perfectly true, even if there is no need to use a driver to register the scan code... Naturally, the recovery of the virtual key code for an application is based on the `WM_KEYDOWN` message. However, according to the documentation for `WM_KEYDOWN` message [848], the original scan code is contained in the bit-field returned (similar to the structure given in Code 4.29). That is to say, in the context of the `WM_KEYDOWN` message, we receive the virtual key code in the `wParam` parameter and the scan code part of the `lParam` parameter.

Thus, it is totally possible for a keylogger to get the original key that was pressed by the user, just by reading the parameter provided in the `WM_KEYDOWN` message. A translation from scan code to virtual key code is easy with `MapVirtualKeyEx` function [906], which even takes an input locale identifier (keyboard layout) selected by the caller. That way, it is possible to translate the scan code into the appropriate virtual key code from the usual keyboard layout registered in the Windows' registry [1230]. It is truly astounding that authors in [1228] did not even completely read the Windows' documentation which their solution is based on. This is a pity because it totally falsifies the expected result. Of course, decent malware also log scan codes. This is precisely in order to be independent of the keyboard layout.

But suppose, for the sake of argument, that the scan codes may have been falsified in some way by the protection system. Would the proposed solution be valid for this reason? The answer is a cruel no. In fact, changing the keyboard layout does not change the frequency of each recorded key. The frequency of occurrence of a letter in a given language is globally known (e.g. with the Turkish language [1231]) and widely used for cryptography purposes [1232]. Thus, as long as the text typed by the user is long enough<sup>25</sup>, it is possible to establish a frequency diagram of each keystroke typed. It makes no difference that the keyboard layout is changed with every keystroke. It is sufficient to always translate from one random layout in use to a common one used for all languages (English for instance) in order to have a uniform data set. From there, we can try to translate data back into different possible layouts until we get the layout used by the user when we get consistent data (low entropy, words, phrases, and so on).

More directly, since a keylogger runs on the user's machine, there is nothing complex about finding the layout used via the `GetKeyboardLayout` function or by searching in the registry which is configured as default at system start-up [1230]. In [1233], authors explain that the if security depends on character *positions* and not on the specific types of character that are allowed in the password, it would not remove the vulnerability, although it might improve security. For short, it does not offer a wider variety of characters, security is just about to increase the permissible lengths of passwords to slow down a potential attack. The result would not change the initial problem which is to secure the password, whatever is its length.

In the end, what this paper teaches us is that a jamming action (here at the keyboard layout level) must be performed low enough in the system to prevent any possible action to retrieve the modified values. This is an important point because it guarantees the security of the user's system.

---

<sup>25</sup>Such technique does not work for short text, such as password. The length is not important enough and the password entropy is too high. It means that there is too much uncertainty with a text that is too short. But with a long enough text, such technique should work.

## 4.2.4 Hypervisor based solutions

### Key Point 5.23:

- ☞ The use of an *hypervisor* provides *precedence* over hardware interactions.
- ☞ The idea here is to offer protection by using hypervisor as a parallel communication channel (and inaccessible to malware).
  - ☞ But it is hard to interface with close source operating system (to know where to "re-inject" keys, where they are used, and son on).
  - ☞ Note that escorting keystrokes in application memory is possible only for kernel-mode (thanks to reverse engineering).
  - ☞ For user-mode applications, it would be necessary to know *a priori* which code/memory sections manipulate the keyboard data.
  - ☞ There is an induced delay in the processing time of the keys, but imperceptible for the user.
- ☞ Solution such as KGuard [44] protects only for specific cases (as network authentication).
- ☞ Windows 10 uses hypervisor with virtualization-based security to enhance system of the security.

### 4.2.4.1 General concepts and possibilities with Hypervisor

In [43], a relevant technique is cited under the name of *KGuard* [44]. This solution has the merit of being elegant because it is based on hypervisors. Hypervisor is another name of visualized base technology, as provided by Intel VT-X technology [431]. For the sake of simplicity (because this subject is very complex), there is a mechanism on some modern CPUs [432] that allows to realize a virtualization mechanism. The idea is to execute code<sup>26</sup> in an isolated environment, *independently* from the rest of the system from the eyes of the virtualized code. It is as if the code we are executing is alone in the world, without knowing that it is virtualized or that other tasks run along with it. The interactions between the virtualized code and the rest of the system (third-party software or the underlying hardware) is managed by a specific code popularly known as *hypervisor* and *Virtual-Machine Monitor* in Intel's documentation [16] (the virtualized code is called *Guest* in the same documentation). This is a kernel-mode driver executing the Intel VT-X or AMD-V technology, to only mention the most famous ones. Basically, there is no interaction between the virtualized code and normal codes (even if, in practice, it is possible thanks to the hypervisor).

More directly, it is possible to execute code in a virtualized environment. This requires ring-0 access since a driver is needed. During code execution, certain code actions (usually related to, but not limited to, hardware interactions) will generate a special event from the virtualized environment. Such an event is called a *vmexit*. This event suspends the regular execution of the virtualized code to run the hypervisor *vmexit* handler. Depending on the *vmexit* code, it is possible to react differently. The virtualized code is resumed via another special event called *vmentry*, giving back the hand from the hypervisor to the virtualized code, until the next *vmexit*. Figure 5.26 extracted from Intel's documentation summarizes this procedure [16].

The idea here is to intercept at the hypervisor level the interactions with the keyboard device. Why are we doing this at such a level? The reason is that the hypervisor acts independently of the virtualized operating system. More precisely, it has a certain *precedence* over certain actions or events, which allows it to act a bit ahead of the classical kernel. Thus, it cannot be affected by any drivers whose code is virtualized in the virtualized operating system. This could help to resist keyloggers that would evolve in kernel mode.

### 4.2.4.2 DriverGuard protection

<sup>26</sup>The executed code in a virtualized environment can be more or less complex. It can be a collection of opcodes to a full operating system. In this last case, we talk about "virtual machine".

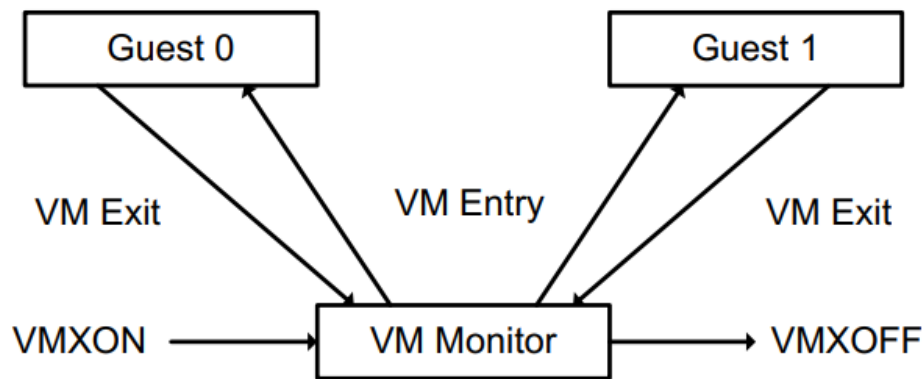


Figure 5.26: Interaction of a Virtual-Machine Monitor and Guests (from [16]).

Generally speaking, it may be attractive to intercept all the keys pressed on the keyboard with the hypervisor. Naively, one could expect that each interruption from the keyboard would generate an action at the hypervisor level. In this case, the last would then be able to act on the keyboard keys to modify them. But this solution has two main drawbacks. The first is about performances. Indeed, as explained in [43], such approach has too important impact on the user's experience. In addition, once the key has been protected, this protection must be removed for legitimate application only. However, at this point, the hypervisor does not have a direct control.

The solution is maybe not to act directly at the level of interruptions generated by the keyboard, but at the level of the memory accessed by the virtualized tasks that manipulates the keyboard. That way, as soon as a memory page related to input/output management (from keyboard) is manipulated by a thread (that is to say the content of structures containing keyboard information), the hypervisor takes the hand and it tracks the execution between a legitimate and an illegitimate application. This is what *DriverGuard* [1234] project does.

*DriverGuard* aims to provide the guarantee of confidentiality of the I/O data over the lifetime of the system. But it has to deal with the complexity of the I/O sub-system, since the hypervisor is not truly part of the kernel operating system. To do this, the project modifies some operating system drivers to be able to interface with the hypervisor. According to authors, *DriverGuard* as a fine-grained I/O protection mechanism, which enforces access control on the device interfaces and it encrypts the I/O data once it is moved into memory. To proceed, it has to distinguish security sensitive driver code sections (called *blocks* in the paper) that deal with the I/O data from the keyboard. Only these identified code blocks are granted to access device interface, and access decrypted I/O data. Other will deal with encrypted data. This is interesting because it secures the kernel chain by preventing any driver not clearly identified by *DriverGuard* (and the identification is done by precise code sections, difficult to bypass or alter since they can be protected to avoid unexpected write operations to detour them) from accessing the keyboard's content. For short, the idea is to *escort* the password in memory thanks to the hypervisor.

The main difficulty is that this solution is limited to the kernel only. More directly, it cannot be generalized to user-mode. Why? Because it is possible to identify the legitimate and common parts of the kernel that manipulate the keyboard. But it is not possible to identify *a priori* which are the code sections of legitimate applications to access the keyboard. Because this presupposes to know which is a keylogger and which is not.

In addition, the proposed solution adds an overhead on the general performance of keystroke management. Even if it is faster than generalized processing from the hypervisor, the fact remains that several hypervisor interactions are required to secure the entire keyboard processing chain. The global overload is about 160 % compared to regular keystrokes without *DriverGuard*. But this percentage must be compared to the real time inducted (i.e. 0.085 ms) which is still negligible as compared the speed of human keystrokes. This is not cheap but this does not affect the user's experience that much.

#### 4.2.4.3 KGuard protection

Project KGuard is a password protection system based web authentication using Firefox [44]. Authors claim their solution could be extended for other password authentication systems (e.g., SSH) which does not limit their solution to a specific case. In practice, according to authors, a good password protection solution should follow three criteria that guide the design of KGuard project:

- Offer the strongest security assurance with practicability perspective and not modification of the operating system ;
- Do not impact the easy-of-use of password authentication nor requiring user possession of extra devices ;
- Do not experience noticeable delay in an authentication session.

From a technical point of view, only a hypervisor can meet the requirements. Starting from a base of *Xen* hypervisor project [1235] where only a minimal set of features have been kept to reduce the attack surface exposed. The same way than *DriverGuard*, *KGuard* aims to avoid frequent interrupt caused by user keystrokes to prevent the time wasted by transiting from the protection mode and the regular mode. But the striking difference between the two projects is that the password is no longer directly provided to the operating system by the hypervisor. Why? Because it is not mandatory that passwords to be known for the local host, especially when they are about to be sent to a remote server through an SSL/TLS connection. For instance, in the case of a web-site authentication through HTTPS protocol. The basic idea of *KGuard*'s protection method is to intercept the password keystrokes and then inject them back to the SSL/TLS connection established by the web-browser securely. It means that all cryptographic operations will be performed by hypervisor itself. That way, the password provided by the user is always manipulated by the hypervisor in its own memory (inaccessible from the guest operating system). The operating system only deals with the ciphered shape of the password, as provided from the hypervisor.

Of course, the solution cannot be applied constantly for every keystroke since it only concerns passwords. Ideally, the protection is only activated by the user whenever needed. To proceed, an *on-demand* system toggled by pressing a prescribed key combination. Interception of all keystrokes is performed in the guest operating system for performance issues. As a regular keyboard handler process, that one checks for all keystrokes if the predefined key combination happens. In such a case, the process performs a hypercall which results in a *vmexit* notifying the hypervisor that all following keystrokes intercepted will be considered as a password. To not trigger it by mistake and for security purpose, the hypervisor securely displays (on the screen) a message informing that it is ready to receive the password. Note that the security is not directly concerns by keyboard simulation from the guest operating system since a message is displayed on the screen to inform the user that the security is turned on.

During protection, hypervisor retrieves the keyboard scan code before the guest. A solution to retrieve keystroke content should be to handle hardware interruption generated by keyboard when a key is pressed. But this solution is not free from constraints. First because some operating systems scan the keyboard input buffer without waiting for the interrupt to interpret it. That way, it could be possible to reuse previous key stored in the buffer while handling a hypervisor interruption from keyboard device. Then, identity of keyboard interrupt is not guaranteed. Interruption's number used for keyboard is not guaranteed to remain the same from one version of the operating system to another. In addition, such interruption can be shared potentially by several devices. If the interrupt provides the scan code, it does not help to know in which buffer it will be stored in the operating system.

The solution adopted in *KGuard* is similar to *DriverGuard* but on a smaller (and more focused) scale. Instead of dealing with intercepting the interruption, hypervisor intercepts the guest access on the keyboard input buffer used by the operating system. According to authors, this solution reduces the burden of the hypervisor since it is the operating system's responsibility to handle the device's input data. Technically, hypervisor memory page management technology is used to set up page-table based access control on both the keyboard I/O control region storing I/O commands and the keyboard input buffer storing the scan code (IOMMU [1236] is added as a security to avoid DMA to steal keyboard's content). In practice, at boot time, *KGuard* localizes in memory the region where the keyboard input buffer belongs. This buffer is then set as inaccessible by the

hypervisor for the operating system. That way, when the OS tries to access its keyboard buffer, it generates a page-fault since that one is marked as inaccessible. This is the hypervisor which handles this page fault to read from that buffer the current scan code in memory and to replace it by a dummy one irrelevant (such as '\*'). After, the hypervisor marks the page as read-write to let the operating system access it in a regular way (to retrieve the dummy scan code). The page is set back to inaccessible by the hypervisor latter when the operating system indicated the end of the I/O operation. That way, the mechanism is re-armed for the next keystroke. Note that it supposes that the password is provided keystroke per keystroke, with no combination of keystrokes. But such improvement could be implemented for an industrial version of *KGuard* project.

The rest of the paper discusses about handling SSL with the password provided and it is out of scope in our case. It should be noted, however, that several lessons can be learned from the proposed method. As authors claim, the proposed password protection mechanism relies on the security of the hypervisor and the user cooperation. Since the technology inherent to hypervisor provides a strong isolation between the hypervisor space and the guest space, it prevents the latter from accessing the password, exception from a ciphered form. At the implementation level, *KGuard* projects requires at least a driver (to notify the hypervisor via a hypercall table) and a process to detect the specific keystrokes combination to start the capture. The performance impact is about ten milliseconds induced for a connection on a classical website in HTTPS. All other things being equal, the time required to enter a password for the user is much longer. But it is therefore imperceptible from user's eyes.

While this solution may look attractive, it is limited for several reasons. Totally assumed by the authors, the first one is that the solution is limited to the case of passwords. We cannot imagine this kind of solution to protect a word processing software. Mainly because the content must typed by the user must be able to be displayed in front of the user's eyes. This would not be the case here because the characters of the password are systematically replaced by other dummy characters. In addition, the content is made to be accessible in clear text by the software in the operating system. However, here, it only makes sense if the data to be protected must be transmitted immediately in a ciphered form. More directly, the content of the protected password is never provided in a clear form to any application. These limitations are given by the authors.

But these are not the only limitations which apply to this solution. Indeed, there is the problem of managing the keyboard buffer used by the operating system with the hypervisor. The exact location of the latter is not documented in memory, since it is a generic way to describe different technical realities. Indeed, access control mechanism for the keyboard input buffer depends on the keyboard interface (PS/2 or USB). USB uses a read IRP armed for that purpose while PS/2 uses interruption mechanism (sections 3 and 4). Tests about *KGuard* have been performed on Windows 7 on a 64-bit CPU by authors. Windows 7's architecture is not too far from the one documented in this document. But whatever the keyboard technology used is, the exact position of the buffers involved in receiving the keyboard content is not documented. Either the *KGuard* project is able to reverse engineer the windows kernel on-the-fly to find expected buffers, or it uses hard-coded offsets in its own code. In both cases, these are dangerous solutions because they are a source of instability over time. The second solution seems to be the most viable taking into account that consequences of equivalent to ASLR mechanisms [1015, 1237] operating within the kernel are disabled. Such deactivation has the direct consequence of significantly weakening the security of the Windows' kernel.

#### 4.2.4.4 Windows hypervisor protection

##### Key Point 5.24:

- ☞ On modern version of Windows 10, *virtualization-based security* (VBS) uses a hypervisor to enhance the security.
  - ✍ In fact, there are "two kernels" running at the same time (VTL0 and VTL1). One is a secure version (VTL0) of the regular kernel (VTL1).
  - ✍ More directly, we no longer run Windows 10 directly but a virtual machine of Windows 10.
  - ✍ This is Hyper-V (Hypervisor from Microsoft) which is used to proceed.
- ☞ In practice, it is not possible (or simple) to run a hypervisor inside a hypervisor.
  - ✍ There are nevertheless solutions but they are often hypervisor emulations from a central hypervisor, with constraints on performances.
  - ✍ On Windows 10 today, there is no simple solution to keep the security provided by VBS and additionally use another hypervisor for a given security task.

Interestingly, if the authors claim to not desire to *modify*<sup>27</sup> the operating system on which their solution is deployed, they are forced to deploy a hypervisor before the guest operating system. This is not an issue to presuppose a hypervisor as a security solution requirement in 2012, when *KGuard* [44] has been released. But nowadays, with Windows 10, it is not so simple anymore. Technically, when Windows 10 is deployed with its maximum security, we are not directly in Windows 10 anymore but in a virtual machine holding Windows 10. Part of the reduction of attack surfaces policy [1238, 1239] from Microsoft, it is possible to use virtualization-based security (VBS) [1240]. For short, VBS uses hardware virtualization features to create and to isolate a secure region of memory from the normal operating system. This is a root mechanism from hypervisor. This security applies for *Application Guard* [1241, 1242] (to isolate enterprise-defined untrusted sites in order to protect employees browsing the Internet), or *Hypervisor-Protected Code Integrity* [1243, 1244] (to prevent unexpected code execution), or *Credential Guard* [1245] (to protect devices) to name a few. All of these technologies use Hardware-based isolation in Windows 10 [1246]. For the sake of simplicity, we can say that technically, there is a layer of hypervisor in Windows 10 used for security purposes. It allows to have *two kernels* running at the same time, one in a secure and isolated space (called VTL0) used for specific security operations (authentication, certificate signature management, and so on) and the other is the regular kernel (called VTL1) [1247, 1248] with some internal details given in [1249, 1250]. Figure 5.27 illustrates the new architecture of Windows 10 with VBS security activated. Generally, more information could be found in [17].

What is the direct consequence of this new security introduction? Simply those related to the presence of a hypervisor that is already running in the system. Microsoft doing things right, it uses its own hypervisor called Hyper-V [1251] on which it builds its security. Technically, Windows 10 could be seen as a Hyper-V virtual machine [1252] even if internal architecture of Hyper-V [1253, 1254] and VBS (especially when Windows 10 is booted with Secure Boot [1255]) is much more complex than this assertion. But it remains that Hyper-V architecture is involved in VBS, that is why it is running when Windows is started with VBS security.

For sake of brevity, Hyper-V is not a hypervisor that allows third-party hypervisor to be run as a guest easily [1256]. Moreover, this kind of architecture (VMs within VMs) creates a kind of schizophrenic introspection that is not very efficient in terms of performance (usually there is a hypervisor emulation performed from the lowest hypervisor, virtualization technology does not allow such nesting on most common CPUs). In fact, the *DriverGuard* and *KGuard* solutions are both based on Xen hypervisor. By consequence, they are no longer able to be implemented the way they have been designed (unless the most advanced Windows security mechanisms would be sacrificed, which is not acceptable).

A way to interact with Hyper-V is to use the Microsoft's user-mode API. This API is called Windows

<sup>27</sup>The fact remains that *KGuard* is not hypervisor-only since it requires at least one driver in the guest operating system.



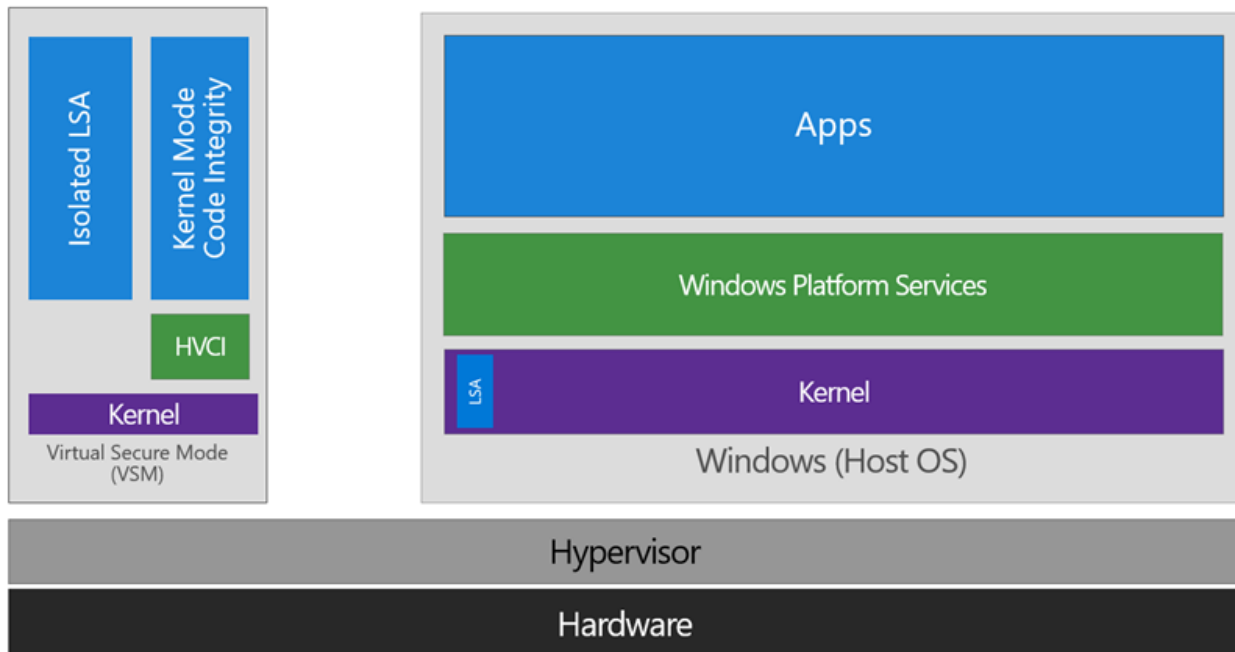


Figure 5.27: Windows 10's general architecture with VBS security activated (from [17]).

Hypervisor Platform API [1257, 1258], available from Windows April 2018 Update. This is a really new API and the documentation is not so extensive. But there are open projects using it, such as *Simpleator* [1259] from Alex Ionescu. This one enables platform support for virtual machines on Windows 10. It is part of *Windows Subsystem for Linux* (WSL2) [1260, 1261] which is a feature of Windows to use a true Linux operating system from a running Windows. Such technology could be used to adapt both anti-keylogger projects, even if there is no guarantee that it is possible, since it has neither be implemented nor tested.

#### 4.2.4.5 Conclusion

In the end, it appears that while hypervisor-based solutions are potentially efficient, they are still limited. On the one hand, they are limited on the subject of application on which they operate since they are able to manage only passwords. And to do so, they must be clever not to downgrade general performance of the machine too much while maintaining the user experience. Moreover, to interface with Windows, the difficulty of interacting with an undocumented OS is cruelly felt. The proposed solutions are only proofs of concept far from marketable and operational software. On the other hand, hypervisor technology is no longer easy to use under Windows 10. Because Windows 10 uses security mechanisms based on its own hypervisor, it is not really possible to interface a hypervisor before its own (without breaking some key system security), nor even after its own (except by going through a specific API which is not so vast). It is for these reasons that this technology is not especially a good idea for success, although theoretically attractive for the technical features offered by the hypervisor. Maybe if Windows implements it as part of its VBS package, this could be a solution.



### 4.3 Industrial solutions

#### Resume 33:

- ☞ We present in this subsection the analysis of the main software in the anti-keylogger domain that are on the market.
  - ✍ The analysis is mainly based on the statements provided by software vendors.
  - ✍ The goal is to analyze the different possible strategies to fight against keyloggers (and not to look for the veracity of vendors' statements).
  - ✍ When it is not possible to rely solely on the vendor's statements, we reserve the right to partially reverse engineer their product.
- ☞ The objective is to establish the different pros and cons for each strategy used by these software.
  - ✍ If necessary, we explain certain technical points or implementation choices and their consequences.

The following section aims to present the industrial solutions sold in the software market. We have to see the differences between academic publications on the one hand and software on the market on the other. These two worlds should not be opposed. Very often, the two have strong interactions and it is usual to see an industrial solution to adopt published concepts. Nevertheless, the approaches are not the same. While academic publications aim to explain how the proposed solutions work and how choices were made, the industrial world — in the case of anti-keylogger solutions — is generally built with closed-source software and marketed websites designed to sell their product without trying to explain it. Of course, it is possible to try to reverse-engineer these software to understand how they work, even if this is not always possible (part of the work could be performed on a remote server) or desirable (especially with regards to the end-user license agreement and copyrights).

The present analysis is therefore focused on what different anti-keylogger software declare in their official documentation (when they declare something) as well as on the means of action they claim to put in place. Even if it is not always possible to be perfectly accurate (due to the lack of reverse engineering on each software program), this section aims at identifying various strategies that may have been implemented. Note that the goal is not to describe these software perfectly — without reverse engineering, this would be impossible. But the objective is to illustrate different strategies to fight against keyloggers and thus to complete our analysis.

It is worth noting that only a small number of software programs exists on this subject [1262]. The reason is that antivirus software have historically acted as a defense against this type of threat. The tools for fighting against keyloggers are sometimes embedded in the core of antivirus software. Companies that would rely only on this software would not have a large business-to-customer market. Maybe more in business-to-business as a solution provider for antivirus, even if there is little documentation about it.

The present analysis is based on a methodology that intends to present each software, its means of action it claims and its advantages and disadvantages. The objective is to be able to identify a need that our solution could meet. For various reasons (including the fact that some software are no longer supported on modern versions of Windows or their documentation is available under non disclosure agreement), it has not always been possible to test all software. To keep the consistency of an approach with academic publications (which are also based on authors' statements), for the sake of fairness and for the sake of the exercise as well, the present analysis is only based on the available software documentation, when possible. Note that it is also another way of analyzing an IT solution and an opportunity to apply other methodologies than reverse engineering.

### 4.3.1 SpysShelter software

#### Key Point 5.25:

- ☞ An interesting solution that affects the sensitive point of malware: their access to the Windows API.
  - ✍ This is done with a system of Dll injection and hooks... Not always stable or evasion bullet-proof.
  - ✍ Security configuration is left to the user and some features seem experimental.
- ☞ As the security seems to be based only on API access, it could be tempting to deal with the keyboard focus to try to get (even temporary) access to the keyboard.

*SpyShelter Silent Anti Keylogger*<sup>28</sup> is a software which — according to its authors — aims at providing “a solid protection in real time against known and unknown “zero-day” spy and monitoring software”. Development is maintained by *Datpol* company in Poland. It is not only a solution which claims to fight against keyloggers but also against screen loggers, webcam loggers, and even advanced financial malware. It is written on the front page of company’s website that this product is nothing less than the “*World’s No.1 Anti-Keylogger Software*”. According to *SpyShelter’s “change-log”* history<sup>29</sup>, their solution has been published first in October 2009 and it seems that anti-keylogging features raised in 2013.

This software is still available and it is still maintained by the company, which is a good point. It is part of a wider solution which includes other protection modules, dedicated to various threats. Its operation is quite peculiar in the sense that it seems to act as a great *allowed rights firewall* for applications. More directly, the software analyzes each process running on the machine and limits the actions it can perform on the system. To control software actions, it uses a knowledge database more or less configured by default by the software vendor (for Windows software or virtual machines<sup>30</sup>). Such a design forces the user to configure the database, presupposing from the user a large knowledge about different software... In practice, even if we are a security and software specialist, such a configuration is very difficult to manage. Indeed, it is error-prone, based on operational trade-offs (software are very rarely designed to work with a downgraded API access) and it must take into account software updates. This is an indirect way to shift the security effort to the user and not to the software (which is used only as a tool). Figures 5.28 and 5.29 show the user interfaces involved in applications management. The left one is the global menu to manage all applications in order to prevent keylogger operations while the second is dedicated to manage access to any keyboard manipulated data (including clipboard) for a given application. This management allows the use of two types of protection against keyloggers: deny or restrict access to the keyboard and encryption of keystrokes.

The software runs on 32-bit Windows and potentially on 64-bit Windows (the FAQs between the French and English versions differ). The software is composed of a driver called *Keystroke Encryption driver* which can be activated or not during the installation of the complete software. It is a chargeable option of *SpyShelter* software and this driver is marked as *experimental* during installation procedure (Figure 5.30).

According to their documentation<sup>31</sup>, the driver *encrypts* all keystrokes in real time and sends them via a safe tunnel directly to the application on which the keyboard is focused. Keystrokes are automatically *decrypted* once they reach the active window. The company do not share detailed technical information about the encryption technique used. Nevertheless there is a demo video<sup>32</sup> which is enough to guess few technical details about the software.

Detection of the use of the keyboard by an application is performed from the user-mode API access. For short, it is the API described in Key-Points 29 and 4.46 assuming that user-mode keyloggers use the same API

<sup>28</sup><https://www.spysshelter.com/>

<sup>29</sup><https://www.spysshelter.com/blog/spysshelter-changelog/>

<sup>30</sup>[https://www.spysshelter.com/help/#\(\)processfilter](https://www.spysshelter.com/help/#()processfilter)

<sup>31</sup><https://www.spysshelter.com/help/>

<sup>32</sup><https://www.youtube.com/watch?v=xETHalUk Ug>

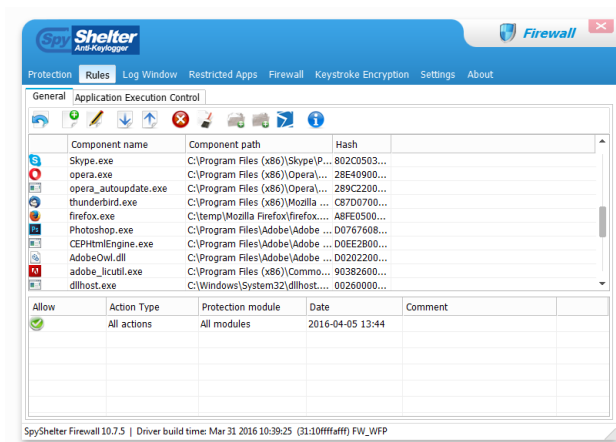


Figure 5.28: General SpyShelter anti keylogger rules manager.

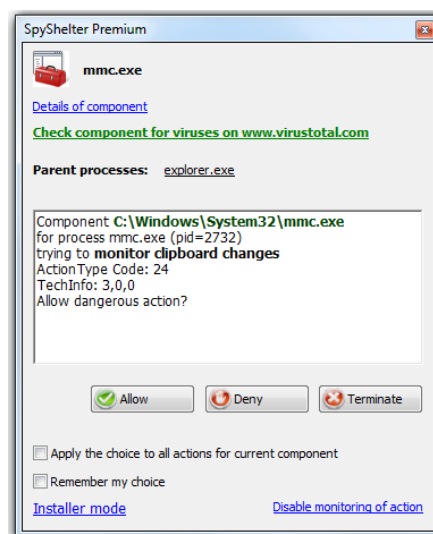


Figure 5.29: SpyShelter single application filter.

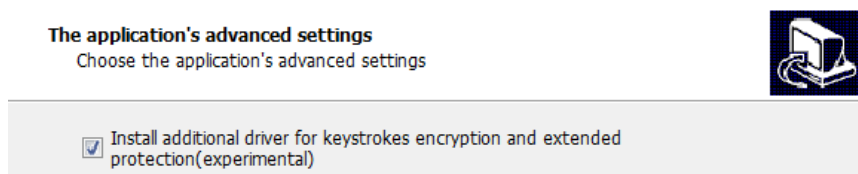


Figure 5.30: Setup procedure from SpyShelter proposes to install the Keystroke Encryption driver which is marked as an experimental feature.

as regular applications. The proof lies in the fact that in the demo video, after the test application has been denied to access to keyboard, it displays an error message referring to the impossibility to *hook* the keyboard. This should refer to the use of `SetWindowsHook` function and other functions which are not shown in the video. More generally, this conclusion can be corroborated by the fact that SpyShelter protects from a whole bunch of features that are not all driven from the kernel. We think in particular of the screenshots protection. In this case, we have to deal with application-specific and GUI display mechanisms. The simplest and the most efficient protection against this type of threat is to restrict access to the Windows' API for an application. This can very well be done with detour techniques [1178].

In a way that might be wrong, we can depict this system as a mix between a driver that encrypts access to the keyboard keys and a hook of the user-mode API in order to be able to retrieve original keyboard keys for the keyboard focused application. Generally speaking, it only requires to take a keyboard driver and encrypt the contents of the keys that pass through it. Contrary to a kernel-mode keylogger presented in Chapter 4, section 3.2 (Key-Point 5.7), the idea here is to modify the content of the scan codes from keystrokes. From there, in user-mode, each application is provided with an extra Dll executed a dedicated protection thread. The same way that [42] is removing the noise inducted by its protection system, this thread would decipher the keystroke content on-the-fly. It would only act if and only if the application has the keyboard focus.

This supposed pattern corresponds to the description of the software and it provides a valid technical solution to design it. Nevertheless, from these observations several mitigations can be made. The first is that security is not total since it is based on the access to the keyboard focus. With regards to the present information, it could be possible for malware to counter security by gaining access to the keyboard focus. This can be done through `SetForegroundWindow` [822] function or `SetFocus` [820] function. More generally, there are various methods to get this focus from the keyboard [1263, 1264] (Key-Point 4.41). Based on this observation, a malware may try

to execute itself through a process known to be authorized (for instance Windows Explorer<sup>33</sup>). This is done via a classic Dll injection [1172]. Moreover, the injected Dll, through a dedicated thread, retrieves (more or less) silently the access to the keyboard focus (creation of a hidden window that quickly takes the focus and release it, creation of a borderless window with a single pixel outside the screen frame, etc.). As the application whose injected window got the focus is considered to be legitimate, it receives the plain-text keys from the protection mechanism in SpyShelter. From there, it is enough to re-inject the keys via the Windows message system to the application that was originally the recipient of the keys. Note that, even if this attack is theoretically possible, it is far from being common<sup>34</sup>, since it is complex and it would result in complex problems of display and user-interaction...

This type of attack exploits the fact that the security solution is global. This means that security applies to all applications. Some are privileged, others affected, but generally all are concerned. From there, it is possible to make a lateral movement via a Dll injection (or using an exploit if there is one) on one of these applications (without it being displayed as being on the foreground) and to access the keyboard keys. This problem could perhaps be solved by more documentation, specifically on encryption mechanisms used, but this is not the case. Last but not least, the protection applies only to user-mode applications and it does not address kernel-mode keylogger threats. This is illustrated by the fact that virtual machine software<sup>35</sup>, which uses drivers to handle keyboard device, is particularly considered by SpyShelter.

Of course, it is possible to try to deny Dll injections into processes. According to developers, SpyShelter protects the files from injections<sup>36,37</sup>. This protection can be interesting, but it has an important cost. *De facto*, it prevents debuggers from working, potentially some antivirus and finally, it can make the system unstable because it might need this legitimate API, all else being equal. One more time, no technical details are available. But the devil is in the details. The protection can be efficient if and only if it is correctly implemented. And if it is performed through their Windows' API hook procedure, it is far from being enough to ensure the security.

Indeed, malware could bypass some techniques used for defense since they seem to be based on a Dll injection on regular applications. A malware could try to evade the hook techniques with different means [174, 1190]. That way, it could avoid to be blocked or manipulated by the protection software. This is just an illustration of [1216] which explains that it is almost impossible to prevent a software to perform a given task which is usually possible.

Nevertheless, this approach has the ability to counter many threats since it controls access to the most important element for this type of malware: the API of Windows. This would even allow to potentially master keyloggers based on DirectX (but no elements about are given by the vendor). Note that this assumes that the API is perfectly referenced by SpyShelter and that there are no omissions. Moreover, a too important targeting (or restrictions) can potentially make unstable legitimate software that need this API (since they call on it) and that did not necessarily foresee that it would not be accessible (since it is the hook of the injected Dll that comes to starve them about). It is therefore often a balance between what is tolerated and what is forbidden for security reasons, with potential failures when it is not handled correctly.

At the end, the fact remains that if this solution is highly effective, it would be possible to defeat it. There are not many details about how this software works. We have to rely on statements and demonstrations that nevertheless allow us to identify key points. Table 5.1 below summarizes the different characteristics that can be attributed to the SpyShelter software (2014) as it stands.

---

<sup>33</sup>This software is referenced as *safe* by SpyShelter in <https://www.spyshelter.com/help/#{}systemprocesses>.

<sup>34</sup>More directly, it has never been observed by us, it is an original proposal from us. But it does not mean that such a behavior could not have been implemented in a malware, since it corresponds to operational needs.

<sup>35</sup><https://www.spyshelter.com/help/>

<sup>36</sup><https://www.spyshelter.com/help/#{}systemprocesses>

<sup>37</sup><https://www.spyshelter.com/system-protection/>

---

Advantages	Disadvantages
Operational	Closed-source and far from being documented
Efficient when used with other protection modules	Global focus and protection strategy which can be abused
Still maintained	Requires from the user to know which application is legitimate or not
	Not a great user's experience with the firewall interface for each application
	Based on DLL injection — not ideal for stability and antivirus compatibility issues

Table 5.1: Evaluation of SpyShelter software as an anti-keylogger solution.

### 4.3.2 KeyScrambler software

#### Key Point 5.26:

- ☞ A solution that uses a ciphered parallel communication channel to the one of Windows (ie: the Raw Input Thread).
- ☞ This is an approach that is often used.
  - ☞ We show that if there is a parallel communication channel, ciphering is superfluous.
  - ☞ This parallel channel approach is more or less based on *security through obscurity* (if the channel is discovered or reversed, serious flaws could appear).
  - ☞ In practice, it only protects the administrator processes and even in this case, only asynchronous keyboard access (Key-Point 4.50) should be managed...
  - ☞ Malware threat running with administrator privilege is powerful enough to uninstall any security software all by itself (Key-Point 5.8).

*KeyScrambler* software has been developed by the US company *QFX Software Corporation* since august 2006. This software is humbly described<sup>38</sup> as “*the world’s leading anti-keylogging program*” and as “*the world’s most advanced keystroke encryption technology and protects Windows users’ inputs against known and unknown keyloggers*” by the authors. Pretended to be used by nothing less than 1,000,000 users around the world, this one is still regularly updated and maintained by the company.

Although closed source and completely undocumented, the software’s description about its use cases provides a great transparency about how it works. When we check the update details for each version<sup>39</sup>, it is common to observe a long list of software that are now supported by this protection solution. More directly, developers implement a specific interface of their security solution for each of the software on the planet that their customers are likely to use. It is staggering to imagine the size of the task which appears to be titanic. The solution is declined in different versions, from the free one to the premium one which is between 50 to 80 US dollars. The security provided looks to be the same, only the list of supported software is different from the different versions of the security solution (Figure 5.31).

The solution is based on a driver which interacts with the module of *KeyScrambler* used to protect a specific software. The solution pretends<sup>40</sup> to use both standard symmetric-key encryption (Blowfish 128-bit) and asymmetric-key encryption (RSA 1024-bit). The cipher module used is OpenSSL. In addition, a specific toolbar called *splash screen* is added on the screen for each application, displaying if the security is active and the ciphered content of the keyboard as a proof of efficiency (Figure 5.32). The location of the bar displayed on the screen can be configured. The software can be automatically set-up at boot-time or by a pre-registered combination of keys to be enable or disable. There is nothing more relevant in terms of technical information given on the company’s website.

The technical aspects of the solution can nevertheless be guessed from the constraints of the software and what is announced. The software includes a driver which must be close to `kbdclass.sys`. It is this driver that is

<sup>38</sup><https://www.qfxsoftware.com/about.htm>

<sup>39</sup><https://www.qfxsoftware.com/ks-windows/whats-new.htm>

<sup>40</sup><https://www.qfxsoftware.com/ks-windows/how-it-works.htm>

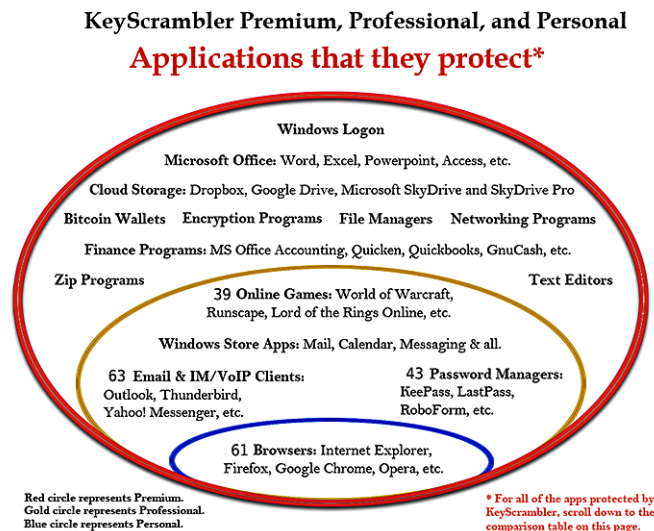


Figure 5.31: KeyScrambler is available in several versions. The difference belongs in the list of software to protect supported.

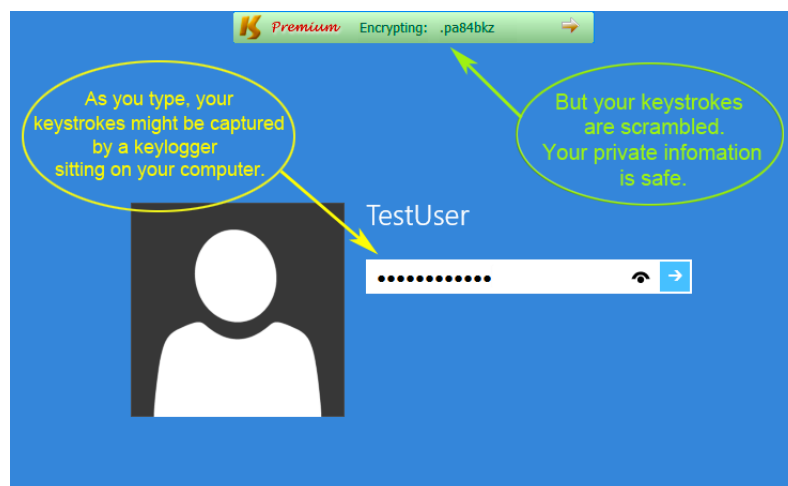


Figure 5.32: KeyScrambler uses a specific bar to display the content of ciphered keyboard data.

responsible for intercepting the keystrokes and protecting the contents of the scan codes using powerful ciphering algorithms. This one is linked to an instance of a user-mode process belonging to KeyScrambler, running with administrator privileges. There is a 32-bit and a 64-bit versions to handle both types of processes. This process manages the reception of keyboard keys through a dedicated channel (may be thanks to IOCTL codes). Such communication channel is dedicated between the process and the driver, such as the regular keystrokes pathway is bypassed. At best, the content of the original key is replaced by another randomly generated key.

User-mode KeyScrambler's process should also be responsible for the display of the control bar and probably the notification area [1265]. Once the process has been notified for handling a keystroke, the received content is transferred to a Dll that KeyScrambler previously injected in protected processes. This Dll seems to be injected thanks to the KeyScrambler process, which explains why two instances (32-bit and 64-bit versions) are running. The injection may be done by using regular Dll injection techniques [1172] or in a smarter way by SetWindowsHookEx [1] function to only target processes dealing with the keyboard.

The question of where the content of the keyboard is made clear remains open. This procedure can be car-



ried out in two different locations, depending on the objectives sought and the technical capabilities employed. The first solution is to perform the deciphering procedure in the KeyScrambler's process. This architecture of a centralized cipher key is by far the simplest and most reliable solution. Indeed, as in any cryptographic solution, the main problem is the management of cipher keys. Having only two agents facilitates operations. They can naturally exchange the key. In practice, when the KeyScrambler's process starts, it probably contacts the driver (via a predefined named *DeviceObject* [1266] previously set up by the driver) to retrieve the cipher key. Thus, the deciphering would be done in the process that would then transfer the clear data to its Dll to be taken into account by the protected process.

The other solution may be to perform the deciphering procedure in the Dll. This means that the central process of KeyScrambler is used as a simple proxy to pass driver's information to the protected software. Here, two strategies can be used. The first is to ensure that all processes share the same cipher key. This is equivalent to dealing with the key at the central process level, from a security point of view. The other strategy is to ensure that each protected process has its own cipher key. In this case, the operation is much more complex to manage. Why? Because the connection is no longer between two agents (the process and the drivers) but between multiple agents (the Dlls and the driver). More directly, each process must have its own cipher key meaning the driver has one cipher key per protected process. In addition, for security reasons<sup>41</sup>, the key exchange from the driver should not be done directly with the central process but directly from a thread injected in the protected process and coming from the Dll. The driver could be responsible for the ciphering key generation.

But, two complex problems arise. The first is to ensure the reliability of the operating system and protected applications. Indeed, any process<sup>42</sup> can contact<sup>43</sup> the driver to request a cipher key. Albeit different from the other cipher keys previously generated, it is possible to make a denial of service (by memory saturation in consequence of too many requests). The other problem is to know for the driver which cipher key to use, according to the application which has the focus. Indeed, there is no documented mechanism so that, at the kernel level, it is possible for a driver to know which process has the focus of the keyboard. There may be "homemade" solutions to do this (such as informing the driver from the Dll each time a dedicated thread from it detects the application has the focus in order to update the cipher key to use). In this last case, the procedure can be complex to implement in order to preserve the security (and avoid a malware usurpation of cipher keys).

In the Figure 5.33, we give a possible architecture used by KeyScrambler software. A step-by-step procedure is proposed to give a clear view of what could happen with this software.

1. A keystroke event is send to or from kbdclass.sys driver. It does not matter the exact location of the KeyScrambler's driver in the device stack. It is just about difficulty or redundant work with the kernel to do.
2. The content of the keystroke is transferred to KeyScrambler's driver.
3. The KeyScrambler's driver ciphers the scan code of the keystroke with the correct key.
4. The original scan code of the keystroke is updated with a fake, or random, or ciphered scan code. This one is used to lure eventual keyloggers in kernel-mode or user-mode.
5. The ciphered scan code is read by the KeyScrambler's central process from the driver.
6. The scan code (cipher or clear form) is transferred to the Dll injected in the protected process.

---

<sup>41</sup>If the exchange went through the central process, we would lose some of the interest of a decentralized infrastructure. Note that this can nevertheless work, even if the central process appears to be redundant with what a thread driven by the injected Dll could directly do.

<sup>42</sup>As a reminder, threads driven from an injected Dll are executed in the context of the process targeted by the Dll injection. That way, it is possible for a malware to contact the driver, the same way a protected process would do.

<sup>43</sup>Of course, some *secret handshake procedure* could have been implemented between the driver and its Dll to avoid such scenario to happen. But it should be kept in mind that on a single computer, whatever is the complexity of such handshake, that one can be bypassed [1267, 1268].

---



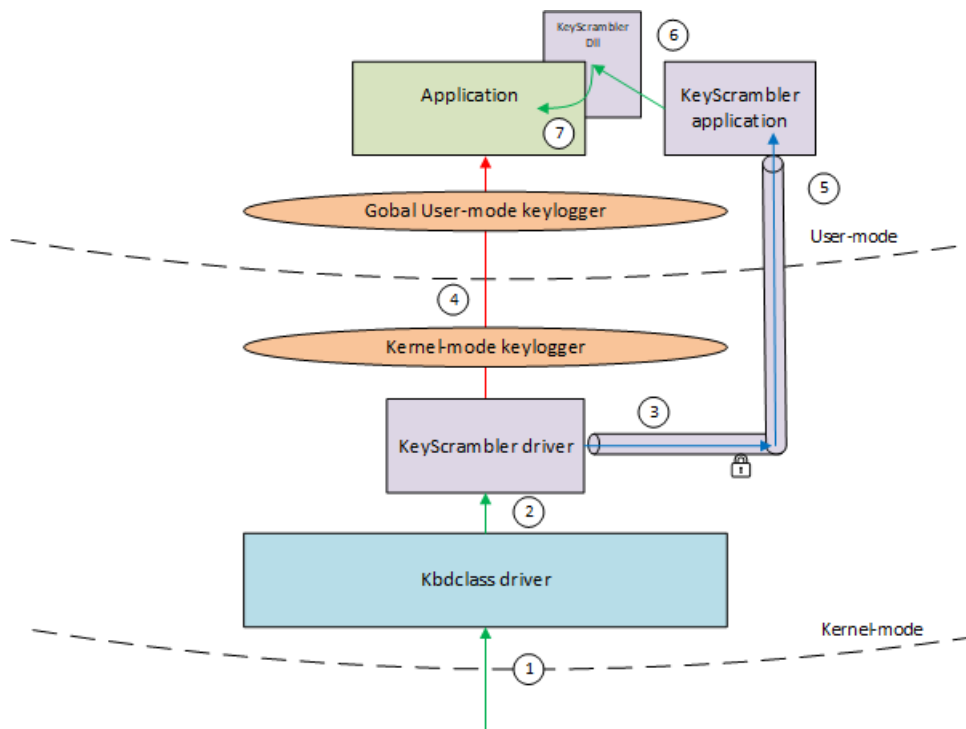


Figure 5.33: Illustration of the possible architecture used by KeyScrambler.

7. After a potential decipher procedure, the original scan code is transformed to virtual key code (to deal with asynchronous keyboard API — section 5.3.1) and finally a character<sup>44</sup> when dealing with window's input message queue.

From the architecture, it appears that this solution is relatively robust and effective. It offers strong security by diverting the classical path of a keyboard keystroke in the kernel. Instead of using the regular keystroke path, the solution provides a second one, a parallel dedicated and controlled path. But this solution is not perfect. First of all, it is kind of security through obscurity, because the project is not open-source and the internal details of operation are clearly not present. We can guess a lot with the *Frequently Asked Questions*<sup>45</sup>, but there are still a lot of shadows. The exact architecture, the role of the central process, the injected Dll and the cipher key(s) management are key points that are kept secret. Of course, this may be justified by commercial reasons. In addition, the defense system based on a finite list of supported software (increased over time) opens a lot of questions...

From a security point of view, the solution seems solid even if it has few weak points. The first is that cryptography is totally useless here. This may be surprising, but it could be explained if we understand the system as a whole. As a preliminary remark, it should be noted that if a different channel of communication for keystroke is used, then the usual logging functions used by keyloggers are irrelevant. But there is more. Indeed, the secret of the cipher key(s) is shared between a user-mode application and a driver. Assuming the defense system is implemented correctly, this means that the process is started with administrator rights. And it is required to have at least administrator rights to communicate with the driver (case of the centralized cipher key). Thus, if a malware has the same administrator rights, it can fully interact in the communication between the two agents. And this is where the security provided by cryptography is supposed to be effective. But it is not. Why? Because if a malware has administrator rights, the game is over from a long time ago

<sup>44</sup>This hypothesis comes from the fact that KeyScrambler can handle Korean or Japanese character for display, meaning we are dealing at a close level to the GUI used by the protected application. More information on: [https://www.qfxsoftware.com/uploads/images/Gallery/kswindows/Options\\_Advanced\\_for-gallery.png](https://www.qfxsoftware.com/uploads/images/Gallery/kswindows/Options_Advanced_for-gallery.png).

<sup>45</sup><https://www.qfxsoftware.com/support/faqs-windows.htm>

[1130, 1127, 1128, 1129].

For instance, an administrator can launch a malicious driver or a debugger and read the content of the kernel memory and thus retrieving the ciphering key(s). Keeping only user-mode solutions, a malware with administrator rights can also read into the memory of the central process that has a copy of the cipher key(s). In the case of decentralized key management, the operation is the same for any injected process hosting a cipher key. Let us note an additional facility that it is not necessary to be an administrator to read the memory of non-admin processes in the latter case. An administrator malware can also restart the ciphering driver and then pretend to be the central process. In this case, the cipher key(s) are provided directly to it. And finally, it can completely remove the driver and still display a bar imitating the one provided by KeyScrambler software, making the user believes that security is still there. To put in a nutshell, if the attacker has administrator rights, the game is over. Ciphering is only useful to avoid a user powerful enough to listen to exchanges between the driver and the central process. But if the solution is decently implemented, only an administrator can do that. Meaning the cryptography security is here simply superfluous and at best a marketing argument.

Since cryptography is superfluous in this case, this one can be withdrawn. Once the cryptography has been removed from the architecture, the solution appears to be a hijack of the traditional keyboard data stream to be sent to a dedicated third-party process instead of csrss.exe (Key-Point 5.1.2). Security is strong as long as the process is administrator and the driver is correctly implemented. But the keystrokes pressed must be transmitted to the software that is supposed to receive it. And therein lies the rub. Two possibilities: the recipient process is unprivileged or it is administrator. In the first case, the protected process can therefore be a victim of Dll injection [1172] (and this is obviously the case since the defense system relies on it — which reduces the possibilities of preventing it by KeyScrambler).

In the second case where the protected process is administrator, classic keylogger would probably not have been able to intercept the keys from an administrator process thanks to the Windows' security, except by using asynchronous methods discussed in Chapter 4, section 3.3.1, Key-Point 4.50. The question is to know if this asynchronous keystroke acquisition API is supported by the defense system. From a defensive point of view, it is necessarily supported since a lot of keyloggers rely on it (Key-Point 5.14). But from a user experience point of view, the question arises. Indeed, which keystroke is actually recorded? Does each injected Dll has its own local buffer containing the personal keyboard image for each application? Quid of shortcuts or keyboard interactions specific to each application and managed with this API? In a generic way, one might be tempted to answer that the driver returns a fake scan code and that it is this one that is taken into account by all applications except for the protected ones which, through a detour [419] mechanism, retrieves the correct scan codes. This may make sense, but it has two consequences. The first one is that the keyboard interactions of an application in the background are systematically inhibited — which could impact the user experience, but this should concern only few software. The second is that the driver is useless. Indeed, since the protection only concerns a finite number of applications looking for protecting themselves from asynchronous keylogger interceptions, it is possible to make it simpler. The first possibility is based on *switching* from user's *desktop* (Key-Points 4.31 and 4.33). The second one is based on keeping only the injection Dll system. We can reuse the solution from [42] about *NoisyKey* project and we can only deal with `GetAsyncKeyState` function [704]. We then obtain the same efficiency (we do not inject extra keys, we just change what the `GetAsyncKeyState` function returns) for a much lower footprint on the system.

Finally, we have to admit that if at the beginning the solution is impressive, after a critical analysis, there remains so few which is really relevant and impacting, from a security point of view. Only administrator processes are really protected. Notwithstanding they were already protected against non-administrator keylogger threats by Windows, attacks using asynchronous keyboard access methods are prevented. But at what cost? Because, as explained in Chapter 4, section 5, the management of the keyboard in the kernel is far from just being a passive transport layers of device information. All the activity of the raw input thread and the interactions it can generate on the whole system (message management, etc.) is totally disabled or corrupted here. Of course, we can bet that keyboard shortcuts such as CTRL+ALT+DEL are maintained, even if this is not obvious. But the fact remains that other shortcuts — less famous or specifically designed by some applications — can be impacted. Also, the software simulation (Key-Point 4.44) of keystrokes remains an open problem, due to a lack of documentation, since the solution short-circuits the raw input thread. Indeed, it seems there is nothing to

send them anything into the secure channel of the KeyScrambler’s driver.

And this is without mentioning the Dll injection that comes with a whole bunch of specific codes for a whole bunch of processes. Doing one interface per software suggests that each software is adapted in the Dll to provide an optimal user experience. Thus, on-the-fly and undocumented in-memory changes in these injected processes may be the norm with KeyScrambler. The stability of injections is always dubious [1269], in particular for adding some dubious features [1270]. Consequences are, on the first hand to inherit an unexpected Dll which introduces instability in the injected process (in case of update by the targeted software, crashes could come soon) and on the other hand to allow a Dll injected from an administrator process to run in the arbitrary context of any process — opening the way to a potential exploitation if it is not implemented correctly [1271].

In a way, such a solution reuses the principle developed by DirectX (Key-Point 5.4.1.1) when it uses its own communication channel for managing keyboard and bypassing the raw input thread procedure. This induces two observations. On the first hand, DirectX strategy is not designed to be secure and even if there is an exclusive access mode for the keyboard, keystrokes stream is still correctly handled by the raw input thread. More directly, it means that DirectX does not inhibit<sup>46</sup> the keyboard management made by the raw input thread but it just add another channel of communication. That way, keylogger threats are perfectly able to use DirectX technology (Key-Point 5.14) to get access to the keyboard, which could bypass KeyScrambler solution, depending where its driver is in the device stack. On the other hand, KeyScrambler manages the keyboard the same way than DirectX, by providing a parallel communication channel with the difference of ciphering the original communication channel. But as with DirectX, who knows how to interface with this library is able to retrieve keystrokes. The same could potentially apply with KeyScrambler.

At the end, the balance sheet of the software is not perfect. For a gain in security, certainly appreciable, but very targeted, there are possibilities of inducted instability with Dll injection, of vulnerabilities if it is badly implemented, with poor documentation and a shiny architecture more likely to flatter their customers’ ego than to really protect them, this software must be watched with a lot of care. The table 5.2 below summarizes the different characteristics that can be attributed to the KeyScrambler software (2006) as it stands.

Advantages	Disadvantages
Operational	Close-source and far from being documented
Still maintained	Possibilities to bypass or abuse the security in different context
A true protection against asynchronous interception for admin process	Use of unnecessary cryptography
	User’s experience is impacted by the design of the solution
	Based on Dll injection — not ideal for stability
	Close source

Table 5.2: Evaluation of KeyScrambler software as an anti-keylogger solution.

<sup>46</sup>It is perfectly possible to retrieve the content of the keyboard through `GetAsyncKeyState` function (Key-Point 4.50) despite `IDirectInputDevice8::SetCooperativeLevel` method (Key-Point 4.58) called with `DISCL_FOREGROUND` and `DISCL_EXCLUSIVE` flags.

### 4.3.3 Zemana AntiLogger

#### Key Point 5.27:

- ☞ Zemana antikeylogger solution is proposed as a *Software development Kit* (SDK) usable by third party vendors.
  - ☞ This a mix with a regular antivirus and antikeylogger technology which can be used as an add-on by third party software.
  - ☞ The solution bases its ciphering procedure on the keyboard focus thanks to `SetWinEventHook` function.
  - ☞ Due to the lack of sufficient (non-commercial) online documentation, we performed a limited reverse engineering (which contradicts some points of their documentation) of their solution.
  - ☞ Nevertheless, the technology of Zemana remains perfectible, as much on the quality of the code as on the proposed solution itself.

The solution proposed from Zemana Ltd. company founded in 2007, in Turkey, is called *Zemana AntiLogger*<sup>47</sup>. This solutions looks like an antivirus solution, protecting from diverse threats while preserving user's private life. In the different threats this antivirus solution detects, there is keylogger threat. But by looking further on the website, we find an anti-keylogger *Software development Kit* (SDK) proposed by the same company. Such product is a set of features already implemented and that a third-party company can use on its own product (in exchange for paying Zemana company to use it). There is one for regular antivirus sub-systems<sup>48</sup> and one to deal with keyloggers threats<sup>49</sup> directly.

There is a *technical documentation*<sup>50</sup> about *Zemana Keystrokes Encryption SDK*. This technical documentation is more a commercial documentation that promotes the product proposed by the company and its employment context, without finally explaining how it works. Internal name of this SDK is *KeyCrypt*. It is written that "*KeyCrypt is able to provide seamless system wide keystroke encryption*", and it "*protects every process and protection is automatically switched on*". The real purpose of the solution is a bit hard to define since it is written that keystroke "*encryption operates deep on the kernel level*" and that kernel-module "*serves as the security filter for the complete computer system by monitoring and analyzing all activities that are running on the PC*" as a traditional antivirus would do. But may be the most relevant information is that "*any given keystroke is transmitted only once. This means that only one application/process has the capacity to receive it*". Finally, the goal of this SDK is to be "*highly utilized worldwide and is simply and easily integrated into any software solution*". It has been developed by "*the best security team for 'OnlineFraud/IDTheft' in the world*" as "*the most stable and compatible keystroke protection solution on the market*" after having pointed out that the competitors' solutions are "*high risk of potential crashes*" and have "*negative impact on both user experience and the overall security posture of the products*".

We must be honest and recognize that it is not possible to conclude anything by reading this documentation. To be able to have more information, we would have to contract with the company Zemana, which is not in our objectives. Non disclosure agreement could be necessary, which would prevent us to explain how their product works. The only solution available may be to perform some reverse engineering on their products. It is not online and there is no link to the driver they are using. However it is possible to retrieve a Dll named `KeyCrypt64.dll`.

This Dll aims to be injected in a process to be *keylogger-protected*. This is proved by one exported function from the Dll called `InjectMe` and which corresponds to a Dll injection method [1172]. Most of the actions done by this Dll are performed from the `DllMain` entry point, which is not a good idea from a design point of view [1002, 1003, 1004]. This entry point does operations that are in direct opposition to what is documented by Microsoft about what must never be done (`LoadLibray`, `CreateThread`, call registry functions...) [1272]. This

<sup>47</sup><https://www.zemana.com/antillogger>

<sup>48</sup><https://www.zemana.com/sdk>

<sup>49</sup><https://www.zemana.com/partners/technology-partnership>

<sup>50</sup>[http://dl9.zemana.com/Website\\_Media/PartnerBrochures/Zemana%{}20KeyCrypt%{}20SDK.pdf](http://dl9.zemana.com/Website_Media/PartnerBrochures/Zemana%{}20KeyCrypt%{}20SDK.pdf)

clearly relativizes the praise given in the technical documentation about the high stability and quality of the code claimed... An observation that we are not the first to have about Zemana's code quality [1273].

The entry point acts only in the case where the Dll is loaded in a creation process context. One of the operation is to get access to the device named object (`\\.\{C0E0FDB1-0951-41D2-A3D9-0103BA422699}`) setup by the driver. Device IOCTL are sent to the driver via `DeviceIoControl` function [1274]. These codes are specific to the driver and its architecture, which makes it difficult to know for which code corresponds a specific feature. The path to the file's configuration of the security solution is defined in the Windows' registry. In the key `"HKML\SOFTWARE\KeyCryptSDK"` belongs `"AppDBPath"` value which stands for *application database path*. The header of the file starts with `"ADB"` string to be loaded in memory. Once the configuration has been loaded in memory, it is possible to know which process must be protected from whose is irrelevant.

There is a distinction between `logonui.exe` and `explorer.exe` processes and all the other ones which are protected. In the case where the protection is applied for both of them, a specific thread is created to handle the switch of keyboard focus. It firsts adds a set of messages (from `0x87D0` to `0x87D4`) to the User Interface Privilege Isolation (UIPI) message filter [1275] thanks to `ChangeWindowMessageFilter` function [1276]. For short, this mechanism ensures that, by default, a process cannot send a window message to another process with a higher integrity level. More directly, it means that if a lower integrity level application sends a message to a process with a higher integrity level, this sending will fail (except with some undocumented specific messages). When running with UAC, standard user privileged applications run with normal integrity level while administrator ones run with high integrity level. Such security makes sense to avoid application's manipulation by a low integrity application to a higher one. Technically, such restriction can be bypassed if a higher security application allows other specific messages to come in. This is the role of `ChangeWindowMessageFilter` function. The procedure implemented here aims at allowing *KeyCrypt-specific* messages to pass between different integrity levels (from `explorer.exe` to `logonui.exe`).

From that point, a disabled window is created with a window class called `"TKeyCryptIPCServerClass"` (Key-Point 4.40) and illustrated in Figure 5.34. The window handler associated to that class aims to deal with KeyCrypt-specific messages, in some case to update internal structures and communicates with the driver or simply to do nothing specific by calling `DefWindowProc` [845] function. This *internal* window and its handler are designed to be a pivot of the KeyCrypt messages sub-system manager.

One of the relevant point is linked to `explorer.exe` and `logonui.exe` processes. For both, the injected Dll manages their keyboard focus in a special way by creating a specific thread for them. This thread registers a special callback thanks to `SetWinEventHook` [1277] function. This callback is registered with `EVENT_OBJECT_FOCUS` [1278] event in order to be notified when an object has received the keyboard focus. The registered callbacks aims to check if the keyboard focus has changed from one process to another (since it can change from one object to another in the application). It is also used to communicate with the driver. More than managing the focus, the thread sets a timer which is recorded every second to perform various communications with the driver, especially about the current keyboard layout. This action is a bit similar to the one implemented for keyboard focus switching. This is a specific characteristic inherent in these two processes.

For each application (including `explorer.exe` and `logonui.exe`), there are two threads created by the Dll. The first is about to get access to two global named events (`{83D4BCA7-9AB9-4052-A0D9-8629D40CD089}` and `{A98D0DED-1846-4871-ACE6-48303AD0331C}`). If such events are set, a specific message is broadcast to the window belonging to the `TKeyCryptIPCServerClass` call in addition to send a KeyCrypt's specific IOCTL code to the driver. The second is about to list the modules (Dlls) present in the protected process. Every time a module present in the process is listed in the KeyCrypt's database file, a communication with the specific window and the driver is performed (using the same IOCTL code communication procedure than the one used with events managed by the first thread).

Finally, the last action aims to set up detour procedures on certain Windows' API functions. In a specific array, a set of six structures has been defined to select which functions must be hijacked (Figure 5.35). For each entry in this structure referencing a targeted function from a targeted Dll, this function is located in the protected process' memory to be modified on-the-fly. Modification aims to perform a trampoline hook [419] on

```

1HWND CreateWindowFromTKeyCryptIPCServerClass()
2{
3    HMODULE CurrentModule; // rax@3
4    WNDCLASSEXW Class; // [rsp+60h] [rbp-58h]@1
5
6    memset(&Class, 0, 0x50u);
7    Class.cbSize = 0x50;
8    Class.style = CS_DISABLED;
9    Class.lpFnWndProc = (WNDPROC)WindowProcedureHandlerTKeyCryptSrvClass;
10   *(_QWORD *)&Class.cbClsExtra = 0i64;
11   Class.hInstance = GetModuleHandleA(0i64);
12   Class.lpszClassName = L"TKeyCryptIPCServerClass";
13   if ( !RegisterClassExW(&Class) )
14       return 0i64;
15   CurrentModule = GetModuleHandleA(0i64);
16   return CreateWindowExW(
17       0,
18       L"TKeyCryptIPCServerClass",
19       0i64,
20       WS_OVERLAPPEDWINDOW,
21       0x80000000,
22       0x80000000,
23       0x80000000,
24       0x80000000,
25       0i64,
26       0i64,
27       CurrentModule,
28       0i64);
29}

```

Figure 5.34: Pseudo code of the function recording the KeyCrypt’s window messages handler.

the first opcodes of the function. Such procedure aims to call a function provided by the KeyCrypt’s Dll. Such opcodes modification is illustrated in Figure 5.36 where trampoline opcodes are stored in a local value. This function changes the rights of the targeted memory page where the original function belongs (with `VirtualProtect` [394] function) and it updates the first opcodes of the targeted function with the opcodes stored in its local values thanks to `memcpy` function call.

```

3180012240 ; PLIB_STRUCT ██████████ [6]
3180012240 dq offset aGetmessagea
3180012240 ; "GetMessageA"
3180012248 dq offset HookGetMessageA
3180012250 dq offset OriginalHookGetMessageA
3180012258 align 20h
3180012260 dq offset LibFileName ; "USER32.DLL"
3180012268 dq offset aGetmessagew ; "GetMessageW"
3180012270 dq offset HookGetMessageW
3180012278 dq offset OriginalHookGetMessageW
3180012280 dq 0
3180012288 dq offset LibFileName ; "USER32.DLL"
3180012290 dq offset aPeekmessagea ; "PeekMessageA"
3180012298 dq offset HookPeekMessageA
31800122A0 dq offset OriginalHookPeekMessageA
31800122A8 align 10h
31800122B0 dq offset LibFileName ; "USER32.DLL"
31800122B8 dq offset aPeekmessagew ; "PeekMessageW"
31800122C0 dq offset HookPeekMessageW
31800122C8 dq offset OriginalHookPeekMessageW
31800122D0 dq 0
31800122D8 dq offset LibFileName ; "USER32.DLL"
31800122E0 dq offset aIsdialogmess_0 ; "IsDialogMessageA"
31800122E8 dq offset HookIsDialogMessageA
31800122F0 dq offset OriginalIsDialogMessage
31800122F8 align 20h
3180012300 dq offset LibFileName ; "USER32.DLL"
3180012308 dq offset aIsdialogmessag ; "IsDialogMessageW"
3180012310 dq offset HookIsDialogMessageW
3180012318 dq offset OriginalIsDialogMessageW

```

```

21 opcodes_1 = 0x244AC7AAAAAAAAA68164; // push 0xFFFFFFFFaaaaaaaa
22 // mov DWORD PTR [rsp+0x4],0xbbbbbbbb
23 // ret
24 opcodes_2 = 0x00000004;
25 opcodes_3 = 0xc380u;
26 v19 = 0;
27 HookStruct = (PLIB_STRUCT)ToHookFunctionStruct;
28 v6 = 0;
29 TargetFunction_1 = TargetFunction;
30 f10idProtect = 0;
31 Size = (unsigned int)SizeOfOpCodesSections((int64)TargetFunction, 14);
32 BufferPage = VirtualAlloc(0i64, PAGE_EXECUTE_READWRITE, 0x1000u, PAGE_EXECUTE_READWRITE);
33 HookStruct->OriginalPointer = (QWORD)BufferPage;
34 if ( !memcpy(BufferPage) )
35     goto LABEL_10;
36 if ( !VirtualProtect(TargetFunction_1, (unsigned int)Size, PAGE_EXECUTE_READWRITE, &f10idProtect) )
37     goto LABEL_10; // Update opcodes to jump to the hook function.
38 memcpy((void *)HookStruct->OriginalPointer, TargetFunction_1, Size);

```

Figure 5.36: Part of the trampoline hook procedure used.

Figure 5.35: List of functions targeted for hook procedure.

The fact that the KeyCrypt solution uses hooking techniques is an interesting point because the opposite



is written in its technical documentation (Figure 5.37). Nevertheless, we have to admit that our reverse engineering is done on a slightly old Dll (2013 — version 1.6.6.247) and the architecture used nowadays could have changed. This could explain why the information in the documentation contradicts our observations. But in the absence of further information, this hypothesis can be neither confirmed nor denied.

## Zemana Keystroke Encryption SDK

- **Stable** - Uses exclusively supported Microsoft interfaces, no hooking techniques; Already operational on over five million PCs worldwide.
- **Flexible** - KeyCrypt can be pre-configured to provide system-wide or application-specific protection; No system reset is required for installation, updates or switching protection on and off.
- **Light** - Software Development Kit under 2MB; No user input required Doesn't slow down PC.
- **Heuristic Auto-Configuration Technology** - A series of self-check mechanisms test compatibility at the launch of every application.

Figure 5.37: Slides from the technical documentation of the *Zemana KeyCrypt SDK* project.

The hook procedure is more or less always the same for all hooked functions. The hook procedure is designed to call a *pre* and a *post* callback function. Figure 5.38 is extracted from the hook function used for `GetMessageW` function. It illustrates the general scheme of hook procedure in KeyCrypt project. The pre-callback, called before the original function, is empty in the version we have analyzed. The post-callback (pseudo code given in Figure 5.39) is dedicated to handle messages after it has been handled by the system. The management of messages is split between `WM_INPUT` and other inputs messages. In both case, the goal is to retrieve message's structure, to communicate with the driver to get original scan code, to perform conversion through virtual key code thanks to `MapVirtualKeyEx` [906] function and to modify the message structure with the correct information retrieved from the driver. Due to the lack of the driver, it is not possible for us to explain in details how the protection provided by the driver works.

To sum up, there are a lot of interactions between the driver and the Dll injected in the protected processes. Note that the injection of this Dll seems to be done in a systematic way since it is designed to interface with some Windows' operating system processes. It seems that keyboard keys are retrieved thanks to the driver which is directly notified by the Dll. We would then be in an architecture finally quite close to the one of KeyScrambler project (in a decentralized mode). But it is difficult to be able to assert this hypothesis because we do not have all evidences to prove our hypothesis. In any case, there seems to be no mention of cryptography within the Dll. This undermines the accuracy of information given in the SDK documentation which talks about "*keystroke encryption*".

Consequently, it appears that Zemana's solution is interesting, but that due to a lack of technical details, it is not really possible to document how it works. Nevertheless, it seems that the proposed solution is quite close to the strategy implemented by QFX Software Corporation. We note some interesting details like the use of specific messages (the role of the invisible window created in the explorer and logonui.exe remains unclear) or the management of notifications about keyboard focus switching event. Without being perfect, this solution nevertheless relies on hook and Dll injection mechanisms, which is not a guarantee of stability. In addition, it does not comply with the guidelines issued by Microsoft, particularly in terms of what can be done in the entry



```

1  __int64 __fastcall HookGetMessageV(LPMSG lpMsg, HWND hWnd, UINT wParam, LRESULT *pResult)
2  {
3      UINT wParamFilterMax; // ebx@1
4      UINT wParamFilterMin; // esi@1
5      HWND hWnd_1; // r12@1
6      LPMSG lpMsg_1; // rdi@1
7      unsigned int retvalue; // ebx@1
8
9      wParamFilterMax_1 = wParamFilterMax;
10     wParamFilterMin_1 = wParamFilterMin;
11     hWnd_1 = hWnd;
12     lpMsg_1 = lpMsg;
13     nothing(); // Pre callback (here: doing nothing).
14     retvalue = OriginalHookGetMessageV(lpMsg_1, hWnd_1, wParamFilterMin_1, wParamFilterMax_1);
15     EndManageMsg(&lpMsg_1->hWnd, 1u); // Post callback.
16     return retvalue;
17 }

```

Figure 5.38: Hook function in the Dll called by the original function.

```

1 void __fastcall EndManageMsg(HWND *hWnd, unsigned __int8 flag)
2 {
3     unsigned __int8 flagf; // r12@1
4     HWND *hWnd_1; // rbx@1
5     bool matchClassName; // di@1
6     unsigned int MessageType; // eax@5
7     CHAR ClassName; // [rsp+20h] [rbp-128h]@3
8
9     flagf = flag;
10    hWnd_1 = hWnd;
11    matchClassName = 0;
12    if ( !BAlreadyLaunchedInstance != 1 && hWnd )
13    {
14        if ( GetClassNameA(*hWnd, &ClassName, 260) > 0 )
15            matchClassName = memcmp(&ClassName, "#32768", 7ui64) == 0;
16        MessageType = *((_DWORD *)hWnd_1 + 2);
17        if ( MessageType > WM_INPUT )
18        {
19            if ( MessageType <= WM_SYSDEADCHAR && MessageType < WM_SYSCHAR )
20            {
21                switch ( MessageType )
22                {
23                    case WM_KEYFIRST:
24                    case WM_KEYUP:
25                    case WM_SYSKEYDOWN:
26                    case WM_SYSKEYUP:
27                        DealWithMessageKey((__int64)hWnd_1, matchClassName, flagf);
28                        break;
29                    default:
30                        return;
31                }
32            }
33        }
34        else if ( MessageType == WM_INPUT )
35        {
36            RawInputManagement((PRAWINPUT)hWnd_1);
37        }
38    }
39 }

```

Figure 5.39: Post-callback used after original hooked function has been called.

point of a Dll [1272]. This does not help to give credit to the authors of the SDK when reading a documentation where it is claimed that their solution is exempted of "code crash vulnerability". Except for very specific cases, Dll injection should be banned for software, especially from those which claim to provide security such a way.

Nevertheless, the idea of the SDK where developers can implement an application that would have its inputs protected by a given mechanism is interesting. Zemana's SDK aims to provide a turnkey solution for an antivirus-type security provider. That is to say, all we have to do is calling their SDK and protection against keyloggers is set up on the entire system. This means that in practice, they are doing Dll injections everywhere in order to communicate with their driver to retrieve the keystrokes. Finally, the table 5.3 below summarizes the different characteristics that can be attributed to the Zemana Keystrokes Encryption SDK (2013) as it stands.

Advantages	Disadvantages
Protection similar to KeyScrambler	
SDK which allows developers to use it smoothly	Very few technical documentation about the SDK
	Inaccurate if not misleading documentation
	Poor quality of the code that does not respect the standards
	Based on Dll injection — not ideal for stability
	Close source

Table 5.3: Evaluation of Zemana Keystrokes Encryption SDK as an anti-keylogger solution.

#### 4.3.4 LMT Anti Logger

##### Key Point 5.28:

☞ User-mode only solution should be avoided since they are too weak.

Le Minh Thanh, a Vietnamese developer, proposes a software which helps to protect malware, by detecting threats with a feature called *LMT Anti Logger*. The last pretends to provide enough security so that "keyloggers will be difficult to steal your data". This is the only documentation we have about this project. This is a

free software but not open-source. From what we can see in video<sup>51</sup> (Figure 5.40), it seems to get similar user’s experience than Zemana Keystrokes Encryption SDK and KeyScrambler projects. Far from being perfect (Figure 5.40 shows that the solution still needs a virtual keyboard displayed on the screen to handle specific characters), it is still under development and it does not aim to compete with regular antivirus companies on the market.

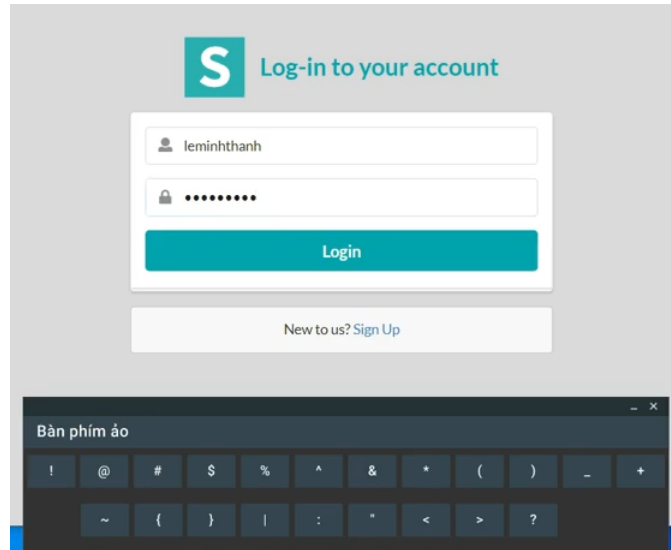


Figure 5.40: Illustration of LMT Anti Logger extracted from a youtube video.

Technically, the solution is based on driver coming from Callback Technologies Inc. company and called *CBFS Filter*<sup>52</sup>. This technology is based on mini-filter drivers [1279] to get real time notification about many activities on the system. This is regular antivirus technology but it has nothing to do with keyboard management. We might bet that the technology used in *LMT Anti Logger* solution is a only user-mode solution. Based on Dll injection [1172] and hooking techniques [419], this solution looks to update on-the-fly messages received by applications. This assumption could be enhanced by the change log file maintained by the developer where it is written that correction has been made on that part to avoid application crashes. Proving that there is a direct interaction with *protected* processes in user-mode. Technical details remain unclear, but this example is for us an illustration that anti-keylogger solution could be user-mode only.

In the case where this solution would be based on user-mode defense mechanism, as we suppose, it would not be hard to bypass this solution. Dll injection and hook procedures can be detected and eventually removed from the injected process itself [174, 1190]. In addition, it would not resist to a kernel-mode keylogger which would access to keystrokes a long time before the user-mode defense solution. Table 5.4 below summarizes the different characteristics that could be attributed to the Le Minh Thanh Anti Logger solution.

Advantages	Disadvantages
User-mode solution and not intrusive	Based on hooks and Dll injection
Still maintained	Easy to bypass (from user-mode)
	Not truly operational on some aspect of user’s experience
	Close-source

Table 5.4: Evaluation of LMT Anti Logger solution.

<sup>51</sup><https://www.youtube.com/watch?v=NXpWcjzf5zM>

<sup>52</sup><https://www.callback.com/cbfsfilter/>

### 4.3.5 GuardedID

#### Key Point 5.29:

- ☞ GuardedID software uses a parallel channel communication solution and replace original keystrokes by random numbers.
- ☞ GuardedID adds additional security modules: protection against driver insertion in the device call stack, display that security is active, kernel-mode drivers...
  - ☞ Far from being perfect (it is not possible to be perfect in this context), this enhances the global difficulty for an attacker to bypass the solution.
  - ☞ Protection against third party drivers can be performed by monitoring the registry in real time (CmRegisterCallbackEx) or executed program (PsSetLoadImageNotifyRoutineEx).
  - ☞ GuardedID leaves the administrator free to uninstall the software.
  - ☞ This is a voluntary limit and an imperative design choice to keep administrators in control of their own computer. Not doing so would be worse [1280].
- ☞ GuardedID is sometimes detected as malicious by antivirus vendors.
  - ☞ This might be explained by the large-scale Dll injection, a classic malware behavior.

*GuardedID* software is developed by *StrikeForce Technologies, Inc.* founded in 2001<sup>53</sup>. Developers claim that their solution eliminates vulnerabilities to data theft due to both known and unknown keylogger threats. It also provides extra protections, such as anti-clickjacking (to avoid malicious webpages using an iframe) [1281, 1282, 1283] and anti-screen capture technology, for multiple layers of protection from cyber-attacks. This software is only a paid software, close-sourced, sold 20 US dollars per year.

The technical documentation on their technology, far from being perfectly exhaustive, nevertheless describes the main mechanisms involved. The solution proposed is quite similar to *KeyScrambler* developed by QFX Software Corporation. Indeed, it is written<sup>54</sup> that "*GuardedID stops malicious keylogging programs by encrypting keystroke data and routing it directly to your internet browser or desktop through a secure pathway that's invisible to keyloggers*". It is added that the solution *bypasses the places keyloggers can reside* and it creates alternate pathway with a military-grade 256-bit encryption code. Technically, it creates an "Out-of-Band" channel bypassing the Windows messaging queue that keyloggers usually monitor to deliver ciphered version of keystrokes. To illustrate their solution, developers provide two figures. On the first hand, Figure 5.41 represents what the user and the keyloggers get respectively with the protection in place. On the other hand, Figure 5.42 gives an illustration of the architecture installed.

When we look at Figure 5.41, we can see that the keys sent to applications that are not accepted by the security software receive randomly generated data, like *KeyScrambler*. In this case, the keystrokes are just numbers. When we look at Figure 5.42, the resemblance is obvious with the architecture we suspected from the *KeyScrambler* software, as given in Figure 5.33. The resemblance is not so surprising. Both start from the same observation of the relative efficiency of traditional antivirus software's detection and the need to fight in a preventive (not to say proactive) way against keylogger malware. The same causes produce the same effects. That is why the two software are so similar. But if the resemblance is striking, we should be careful not to say that the two programs are identical.

There is a more detailed documentation<sup>55</sup> [1284] that explains some of the features of the software. Technically, the *GuardedID* software does more than its twin software. It is equipped with some additional technologies.

<sup>53</sup><https://www.strikeforcecpg.com/about/>

<sup>54</sup><https://www.strikeforcecpg.com/guardedid/>

<sup>55</sup>But perhaps from an older version of the software since screenshots are coming from Windows XP. But the logic with what is sold today and what is written in the document may remain the same.

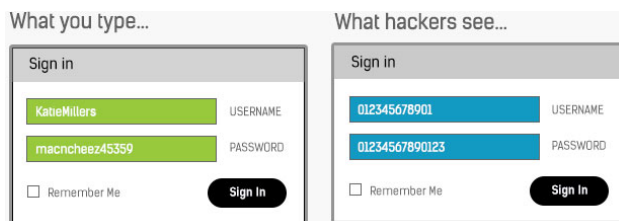


Figure 5.41: What keyloggers see.

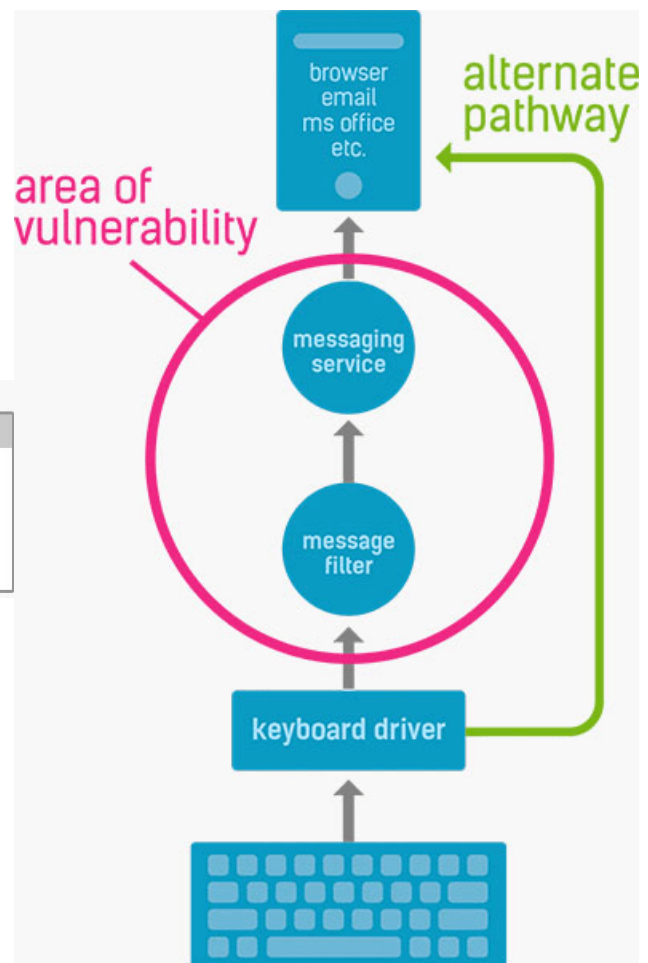


Figure 5.42: Architecture used by GuardedID.

GuardedID has a built in self-monitoring capability. The goal is to prevent the solution “*from being bypassed by another software*”. More directly, the solution constantly monitors the keyboard device drivers stack to detect untrusted drivers which could be keyloggers. The notion of *untrusted* driver is not defined. Perhaps it concerns unsigned drivers since this documentation<sup>56,57,58</sup> talks about drivers’ signature. But unsigned drivers would not be able to run on 64-bit Windows [1123], which makes this protection obsolete. Perhaps it may be an internal inclusion list allowing only a small subset of predefined drivers in the system. Whatever it is, an “Unknown Driver Warning”<sup>59</sup> dialog is displayed to the user when GuardedID detects such undesirable driver. Illustration of the different level or protection is given in Figure 5.43 from [1284].

To implement monitoring capability, the software could use two different approaches. On the first hand, this security mechanism can be based on monitoring the Windows’ registry [1285] from the solution’s driver. The API used would be based on CmRegisterCallbackEx [1286] routine to register a callback routine notified when an action is performed on the Windows’ registry. The callback routine would then be responsible to check if the action is about to register a service representing a driver attached to the keyboard. This is probably the most simple and efficient way to proceed. On the other hand, the solution is to use PsSetLoadImageNotifyRoutineEx [1287] routine to register a PLOAD\_IMAGE\_NOTIFY\_ROUTINE callback routine [1288] notified every-time an executable image is loaded (or mapped) into memory. That way, it is possible to be notified for every section of codes about to be loaded from the disk in order to be executed. From that point, it is possible to know the

<sup>56</sup><https://support.strikeforcecp.com/kb/article/33-driver-warning-i8042prt-or-kbdhid-on-windows-xp/>

<sup>57</sup><https://support.strikeforcecp.com/kb/article/35-driver-warning-unsigned-driver-valid-source/>

<sup>58</sup><https://support.strikeforcecp.com/kb/article/26-warning-untrusted-driver-detected/>

<sup>59</sup><https://support.strikeforcecp.com/kb/article/36-driver-warning-for-an-unknown-driver-and-unknown-source/>

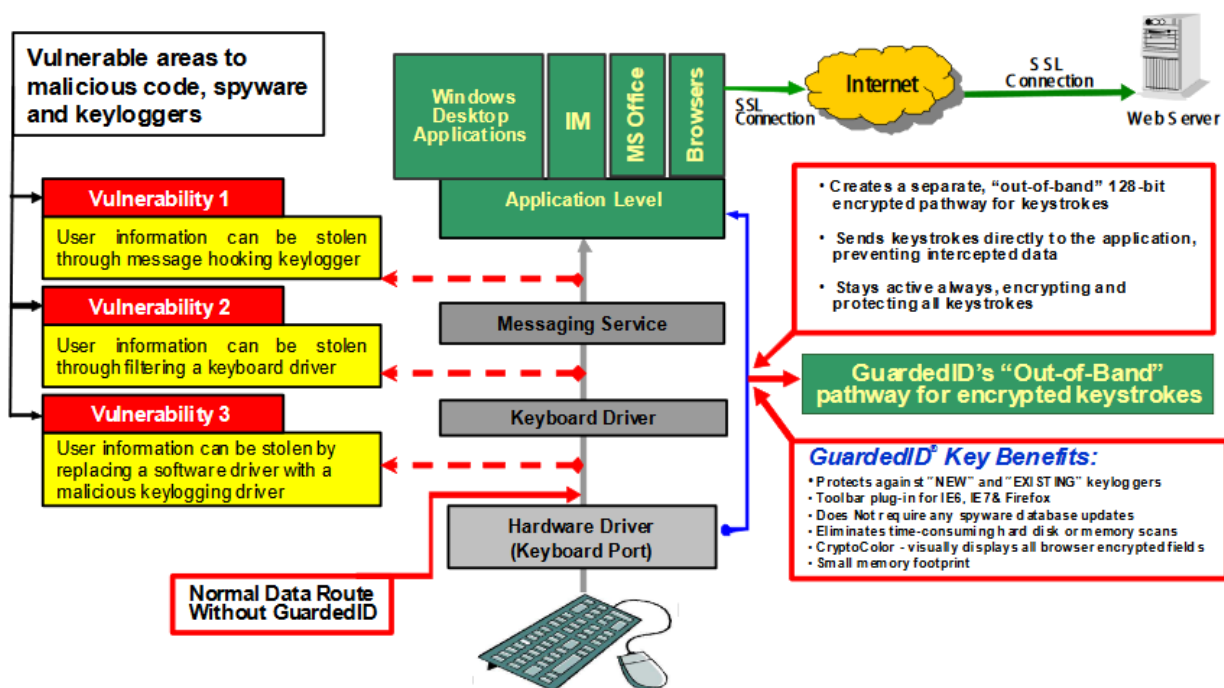


Figure 5.43: How GuardedID works with its own self-monitoring capability.

full path name of the file loaded in memory and with this information, to search in the registry if the targeted executable is a driver linked with the keyboard.

In GuardedID software, a module called *CryptoColor*<sup>60</sup> is used as toolbar kit in web browsers to signal the software is active. That way, the user knows that the security is enable and the user's input is secured when it is using the web browser. According to authors [1284], "this provides strong visual feedback to the user that they are operating in a secure environment and their keystrokes are secure".

The *CryptoColor* mechanism has a more general counterpart called *CryptoState*<sup>61,62</sup>. This system uses the notification area to signal if the defense software is enable or not. Note that this system can work even if *CryptoColor* is not present<sup>63</sup>. Indeed, the security can be ensured despite *CryptoColor* which is only present for display purposes. *CryptoState* is another module used in case of error or if the activation key must be updated. In addition, it is possible, for the user, to disable<sup>64</sup> the protection solution. This can be useful when the software may not correctly handle specific characters<sup>65</sup> or when it may slow down the system in some circumstances<sup>66</sup>.

A last module, called *Anti-Clickjacking* is present to defeat hidden frame that contains a clickable and invisible button that can be used to abuse user's click and to cause an undesirable action, such as, downloading malware, transferring money, buying something, etc. This module is a bit out of our scope, but the reader can refer to the documentation [1284] for more information.

If this solution appears as a similar but improved version of *KeyScrambler*, it still has some drawbacks. Of course, there are all the related remarks on its twin product (*Key-Point 5.26*). The ineffectiveness of the cryptography used here is exactly the same. And there is no explanation about the cipher key-management.

<sup>60</sup><https://lr04.guardedid.com/pdoc/01/1/en/v4.100/extensions.html>

<sup>61</sup><https://lr04.guardedid.com/pdoc/01/1/en/v4.100/interface.html>

<sup>62</sup><https://support.strikeforcecpg.com/kb/article/17-how-do-i-know-guardedid-is-working/>

<sup>63</sup><https://support.strikeforcecpg.com/kb/article/22-how-do-i-know-guardedid-is-protecting-browsers/>

<sup>64</sup><https://support.strikeforcecpg.com/kb/article/20-turn-off-guardedid/>

<sup>65</sup><https://support.strikeforcecpg.com/kb/article/51-wrong-character/>

<sup>66</sup><https://support.strikeforcecpg.com/kb/article/14-firefox-with-vista-sites-are-slow/>

Technically, there is no real mention of any central process used in this architecture (even if GIDD.exe process is part of the protection software<sup>67</sup>) neither there is a mention of any generalized Dll injection system. But the fact remains that the cryptography part in the solution is useless for non-administrator processes<sup>68</sup> (Key-Point 5.26), whatever the chosen architecture is — centralized or decentralized. Anyway, the keyloggers receive predefined keys that have nothing to do with what the user has typed. These fake keystrokes are the numbers from 0 to 9, in ascending order (Figure 5.41). Notwithstanding to hack the memory of any protected processes (which is enough to bypass the security in any case), keyloggers only see these fake keystrokes, which effectively confirms that cryptography is useless in this case too since the ciphered keystrokes are not broadcast.

In addition to these potential issues, the software has quality issues that are documented in its help section. First of all, there are many antivirus software that detect<sup>69,70,71,72</sup> it as a malicious program (although it is signed — and therefore clearly identified — by the company). The reason for this detection can be multiple. Maybe a packer is used to modify executable binaries to make them more resistant to reverse engineering. This mechanism is also used by some malware to escape analysis. This type of detection would be a static detection and it should normally be properly taken into account by the different antivirus editors, since the developers are able to document the problem and probably to contact the antivirus editors about it. But the reason may be that antivirus programs detect a large-scale Dll injection on the whole system, which is a rather classic malware behavior [1289]. In this case, it is understandable that antivirus vendors do not fix their software to allow a third-party program that, on the one hand, imitates malware and, on the other hand, uses a mechanism that induces instability in the system, weakening their customers' machines. Let us note that from the point of view of the GuardedID developers, it is the user's responsibility to correct this point by configuring the antivirus and not up to them to change any dubious methods they use to make security. Such a requirement from user reflects the mindset of the software's designers.

Last point about this solution is that if it handles correctly regular keyboards (PS/2 and USB ones)<sup>73</sup>, it does not handle<sup>74</sup> keystrokes generated in a remote desktop context [823]. It is not very important not to take into account this situation, which is not so common after all. In case of, it would require significant development to secure the network on which this technology is based. In practice, security solutions should use a an upper filter driver with the kbdclass.sys driver. Indeed, kbdclass.sys is able to handle any type of keyboard devices, meaning the transport layer (PS/2, USB, network...) is managed at a lower level by the operating system. More directly, the hard work is performed by Windows to interface the different types of keyboards thanks to its driver call stack. Security software should only process the output of system's keyboards after kbdclass.sys. That way, it behaves like classic kernel mode-drivers (Key-Point 5.9).

Despite its drawbacks, the solution remains interesting in at least two aspects. The first is that it provides a self-protection system that checks if the solution is not being disabled or bypassed. Good news, this mechanism can also be used to discover keyloggers. This is a really good idea because it can help to fight against malware running with administrator rights or using a driver. Of course, it does not ensure that the security is perfect. An administrator can always try to disable the driver. We can try to definitively prevent driver unloading, but we risk to ruin the user experience. Indeed, in such case, the user would no longer be able to shut down the protection driver if it has a problem (not to mention the work to update the driver while it is running, which could be a complex operation). Consequences of such behavior are the same as *unkillable* processes [1290]. That is to say, it is going to be a mess since it implies other nefarious consequences [1291] and it engages an arms race with the user [1280]. It really matters to wonder why such feature would be needed [1292]. In the end, there is often a balance to be found here. In the case of GuardedID, it has been chosen to detect threats that would lies in the device stack, but to leave administrators (and potentially malware running with administrator rights) to disable the solution. From our point of view, it seems to be an acceptable balance for an operational software.

---

<sup>67</sup><https://support.strikeforcecp.com/kb/article/105-how-do-i-fix-the-gid-agent-service-is-not-running-error/>

<sup>68</sup>An administrator process could read the memory of a protected process to retrieve the cipher keys, as a debugger would do.

<sup>69</sup><https://support.strikeforcecp.com/kb/article/80-malwarebytes/>

<sup>70</sup><https://support.strikeforcecp.com/kb/article/125-is-guardedid-compatible-with-bitdefender/>

<sup>71</sup><https://support.strikeforcecp.com/kb/article/132-guardedid-compatibility-with-avg-avast/>

<sup>72</sup><https://support.strikeforcecp.com/kb/article/114-when-i-try-to-download-guardedid-i-get-a-message-that-th>

<sup>73</sup><https://support.strikeforcecp.com/kb/article/10-will-guardedid-work-with-any-keyboard/>

<sup>74</sup><https://support.strikeforcecp.com/kb/article/15-hide-my-keystrokes-remote-desktop-programs/>



The second interesting aspect is that it confirms the identified need to manage keyloggers rather than trying to detect them. Certainly the proposed architecture for the solution is perfectible but such solution is sold and it means there is a market. This solution illustrates a technical possibility. Finally, the table 5.5 below summarizes the different characteristics that can be attributed to the GuardedID as it stands.

Advantages	Disadvantages
	Protection similar to KeyScrambler
Self protection system	Obsolete modules for self-protection if based on signature
Correct documentation	Close source and paid software
	Based on Dll injection — not ideal for stability

Table 5.5: Evaluation of GuardedID as an anti-keylogger solution.

#### 4.3.6 Conclusion

In the end, we propose to summarize in Table 5.6 all the elements specific to each anti-keylogger software presented here. The goal is to have a look at the security they offer, the advantages they offer and the disadvantages they suffer. We have tried to summarize in a synthetic way the characteristics of different software in their architecture. Globally, we have also tried to highlight the features that seem positive to us in green (efficiency, open-source, documented, stability...) and those that we believe should be corrected in red. More than a simple summary, this will help us to determine all the characteristics required to meet the initial needs: how to detect and fight keyloggers. Furthermore, identifying drawbacks and missing features of the existing products helps us building a more reliable solution to fully protect the user.

	Spys Shelter	KeyScrambler	Zemana	LMT	GuardedID
First release date	2009 (2013?)	2006	2007	2020	2001
Open source	No	No	No	No	No
Free	Limited in time	Basic version is free	Limited in time	Yes	No
Operational	Yes	Yes	Not really	Yes	Yes
Still maintained	Yes	Yes	Partially	Yes	Yes
Documentation	Partial information	Partial information	Commercial only	Highly limited	Correct
User experience	Require user's knowledge	Splash screen not elegant	Not applicable	Virtual keyboard for specific accents	Toolbar is obsolete
Stability	Experimental feature	Should be correct (module specific per process)	Bad due to design choices	Based on Dll Injection	Detected as malware by antivirus
Dll injection	Yes	Yes	Yes	Yes	Yes
Bypassed by ring 3 threat	Yes	Yes	Yes	Yes	Yes
Bypassed by ring 0 threat	Yes	Yes	Yes	Yes	No or Hard
Require admin to bypass	No	Only for andim process	Only for andim process	No	Only for andim process
General protection provided	Low	Admin process only	Dubious	Very Low	Admin process only
Keystroke management	Exclusive access to keyboard	Side-channel Useless cipher	Side-channel Useless cipher	Exclusive access to keyboard	Side-channel Useless cipher
Keylogger keystroke access	Denied or numbers	Random keystrokes (from a large set)	Random keystrokes (from a large set)	None	Numbers
SDK	No	No	Yes	No	No
Use driver	Yes	Yes	Yes	Unknown	Yes
Self-protected	No	No	No	No	Yes

Table 5.6: General resume of the different industrial solutions.

From another point of view, it may be interesting to summarize the strengths of each software. In addition to justifying the diversity of the solutions presented here, some of their characteristics will help us guide our future design choices regarding our own solution.

- Spys Shelter: ciphers keystrokes with a driver and decipher in an injected user-mode Dll. User-mode API filtering access to prevent malware activity ;
- KeyScrambler: uses an alternate or parallel communication path to broadcast keystroke to protected applications only ;
- Zemana: provides a SDK to help antivirus or security software to protect from keyloggers ;
- LMT Anti Logger: is designed as a user-mode only anti-keylogger technology ;



- GuardedID: provides self-protection technology used to prevent malware to fool the protection solution.

#### 4.4 Conclusion about anti-keylogger solutions

##### Key Point 5.30:

- ☞ There is no perfect solution against keylogger so far.
  - ☞ Each strategy generally has advantages and disadvantages.
  - ☞ Some strategies offer very interesting results but are limited by implementation constraints (hypervisors in particular — Key-Point 5.23).
- ☞ The different strategies are often guided by the technical constraints coming from the different level where they had been implemented.
  - ☞ Kernel-mode solutions (drivers) tend to setup parallel communication channels and ciphering keystrokes.
  - ☞ User-mode solutions (applications) tend to hook imported API from Windows to decoy the data stream or preventing any access to it.
- ☞ Generally speaking, it is not possible to neutralize threats that obtains administrator rights.
  - ☞ In the latter case, it is always possible for the malware to try to uninstall the security solution.
  - ☞ Even if victory is impossible, there are still some possibilities to make things harder for an attacker.
  - ☞ It is possible to try to detect a possible attack on the defense system (Key-Point 5.29).
- ☞ The most interesting solutions are usually those that are focused on a particular form of the keylogger threat (password management, only user-mode and not administrator threat, etc).
- ☞ It must be understood that it is only possible to protect against an "acceptable" threat.
  - ☞ For example, it would not make much sense to fight a threat that would be administrator on the system (Key-points 5.8 and 5.26).
  - ☞ Hence the importance of clearly defining the threat that we are dealing with.

It should be seen that beyond the simple detection of keyloggers recognized as classic malware, there are solutions that aim to neutralize or manage the activity of these threats. Why this approach? Simply because the problem of detection presupposes to be able to differentiate between applications listening to the keyboard for a legitimate purpose and those that do it for a malicious purpose. And in the case where we would like to perform a detection, one should be interested in what is done with the data retrieved from the keyboard and not about to know how this data has been received. This problem is more philosophical than technical. More generally, it is about the undecidable problem of malware detection, as presented by Fred Cohen in 1987 [165, 164, 1293]. It is thus in front of this impasse that solutions of neutralization or circumcision of the threat rose.

From what we have seen, many approaches have been proposed. From academic world, decoy strategy is maybe the most used one. This strategy has been taken by NoisyKey [42] and other projects with more or less success. Contrary to the classic use of decoy techniques in computer security, which are generally aimed at attracting threats (as in honey pots) [1294, 1295], here we are dealing with a jamming. The aim is to disrupt the proper functioning of the keyloggers while impacting as few as possible legitimate applications. That is to say, maximizing the impact on keyloggers while minimizing undesirable effects for legitimate applications. Finally, it is in the implementation that these academic solutions could be improved. The lack of driver or poor driver's knowledge can tend to make these solutions vulnerable.

In a similar category are solutions based on the keyboard layout of the protected application. From visual solution reinventing the *keyboard* on the screen to regular layout update, efficiency is variable depending on the solutions chosen. These solutions are in many cases one step behind the attacker and they have to deal with the balance of the user's experience. This user's experience should not be impacted too much to keep a minimum chance of being really usable.

An elegant and really promising solution lies in the use of hypervisors (projects DriverGuard [1234] and KGuard [44]). But here again, the devil is in the details. While in theory this technology allows a certain isolation and therefore a security that no longer depends solely on the operating system in which the malware (keylogger) is running, in practice, the reality is a little different. On the one hand, there is all the work of interfacing with the different (and multiple) mechanisms involved in the management of the keyboard keys. Under Windows very little parts are documented<sup>75</sup> about keyboard management and with the various security mechanisms (ASLR, memory protection, to name a few), it is very complex to interface reliably over time (i.e. automatically guaranteeing the efficiency and reliability of the solution despite the various Windows updates over time). In addition, it is necessary to take into account the protection mechanisms resulting from Hypervisor-Protected Code Integrity [1243]. Today, this security installs by default a hypervisor in the system of Windows 10, which prevents a second one to operate easily. Under Linux, the problem is barely the same because we have to modify the Linux kernel (and thus adapt the patches to take into account the updates and recompile, which cannot always be automatic in case of conflict). And that is not to mention the specific strategies and architectures that need to be implemented in order to keep correct performances — and therefore the user experience. Nevertheless, this type of solution should continue to be studied in the future since they represent a high potential.

In terms of industrial solutions, apart from their small number (but this type of market in security is highly targeted), we have basically two main different strategies (which are not incompatible with each other). The first is to look at the user-mode API used by keyloggers. As shown in this study, there are a finite number of API functions to interface with the keyboard (Key-Point 5.5), which ensures that ruling these functions effectively rules the access to the keyboard (both in the formal authorization of access than in the content that passes through it). Of course, this can only be fought against user-mode malware. But on the one hand, they are the majority, and, on the other hand, the use of kernel-mode malware under Windows is more complex today because of the signature policy on 64-bit systems [1123] (Key-Point 5.7). It is still possible to override security by exploiting a vulnerability that allows the execution of arbitrary code in the kernel, which on the one hand is complex to find and to exploit and, on the other hand, that is increasingly limited by new Windows protection technologies [1301, 1240].

However, focusing on the API is not a perfect solution. Why? Because Dll injections [1172] and detour mechanisms [419] used for this purpose are not good enough. In addition to the instability they can induce or the incompatibility with some antivirus vendors (Key-Point 5.29), it is possible to escape from this type of protection (Key-Point 5.25) once this detour mechanism is detected [1289] for well-written malware. This is why it is a better option to have a solution that involves a driver installed in the device driver stack of the Windows keyboard. And this is what some of the studied solutions do. The strategy adopted in this case is always the same. It aims to use a different channel of communication for keystroke management, on which a layer of cryptography is added, for commercial purposes since this one has been proven to be useless by us (Key-Point 5.26). From there, two tactics are possible for key reception on the auxiliary channel: centralized or decentralized. In the first case, a central process receives the ciphered keys, deciphers them and sends them in clear to the application that is supposed to have the keyboard focus. The reception of the key in the application is done via an appropriate Dll which *reinjects* the keystroke into the message system queue and the asynchronous key management subsystem. The other tactic is that the driver communicates directly with each of the injected Dlls to send the keystrokes. It is then the injected Dll's role to decipher the keystroke and to *reinject* it in clear

---

<sup>75</sup>And even if they would be documented, it does not change the fact that it would not be a good idea to interface them directly. Indeed, undocumented structures or features in Windows is not about keeping secret some parts of the operating system (reverse-engineering allows to discover such secrets). It is because documenting a feature creates a contract [1296] between developers that must be kept forever... And consequences can be relevant [1297, 1298]. In addition, keeping undocumented features allow future next features to be embedded without having to maintain a complex backward compatibility. Of course, there still are some seamless software which used undocumented stuff [1299] for more than dubious reasons... And it is sometimes Microsoft which keeps them working despite such bad behaviors [1300].

in the process. Note that the second tactic is much more complex than the first, because it needs to manage one cipher key per process and, above all, to be able to know, from the kernel, which application has the focus of the keyboard — and this is hard to do since there is no direct documented way to do this under Windows. There is an hybrid strategy where a central process only has a dealer role with injected Dlls. But it is exactly the same consequences for the security provided by the solution.

We have proven that this strategy is far from providing the security that is so highly promoted by some commercial websites from some companies. Technically speaking, such a strategy is more or less equivalent to DirectX library which re-implements a dedicated communication channel with the keyboard (Key-Point 4.58). The difference lies in the way the communication channel managed by the raw input thread is handled. But with such a strategy, under certain conditions, it is possible to completely bypass the security solution and read the original keystrokes. In the worst case, it is perfectly equivalent with the security naturally provided by Windows without the security solution being deployed. In this case, such solutions could be seen as useless, like the use of cryptography for the dedicated channel of communication. Of course, this has the merit of defeating the most basic keylogger threats, as shown in some youtube videos. But there is plenty of room for the threat to adapt itself and totally neutralize these weak security solutions.

In fact, if the idea of using a parallel communication channel may appear to be a good solution to neutralize the keyloggers (because they seem to be starved from keystrokes), this cannot work in practice with advanced threat. Simply because, in practice, the solution proposed does not deprive keyloggers of means to retrieve keys. Instead of, it adds an additional mean to retrieve keystrokes while it tries to turn off the historical means. More directly, only a new hidden channel is created while already existing ones are jammed. Security only holds because the attacker ignores this new *hidden* channel. Technically, this ignorance is the only point to provide security, which cannot be decently enough for being considered as strong and efficient security. And that is not to mention the bad consequences induced by the creation of this parallel channel and the deactivation or distorted usage of the original ones. From the inhibition of keyboard shortcuts, which are useful to the system or specific software, to the loss of keyboard features (specific layout, sonar mouse, etc.) for the user, and the lack of support for keystrokes simulated by applications (with `SendInput` [940] function — Key-Point 5.2.6), the disappointment is great.

This is also why we decided to propose our own solution. It may also be imperfect too. Nevertheless, it aims to make the best use of the good ideas proposed in various academic articles and industrial solutions while avoiding the pitfalls of certain solutions. And this is only possible through the (sometimes very) critical analysis we have made of the academic papers and industrial solutions presented in this state of the art. The next chapter presents our solution called *GostxBoard* and its comparative security analysis.

## 5 Research contributions

### Contribution 5: State of the art about keylogger and anti-keylogger threats.

- ☞ We have made a technical and complete state-of-the-art on keylogger threats.
  - ✍ We have proposed what we believe to be the most effective method of classifying this threat.
  - ✍ The latter is divided into two main levels (*hardware* and *software*), themselves divided into sub-levels that differentiate the different technical means employed by the threat.
  - ✍ To the best of our knowledge, this state-of-the-art, which is technical in its explanation of the means used by keyloggers, is the most complete that is available.
- ☞ Once the threat is identified and understood, it makes sense to analyze the existing methods to defeat it.
- ☞ We have conducted a state of the art on all anti-keylogger solutions focused on *threat management* (not specifically on *detection*, which is a problem with no real solution [165]).
  - ✍ We have divided our analysis between solutions from the academic world and those from the industrial world.
  - ✍ We have written a literature review from all academic solutions fighting against key-loggers to detail their different strategies and results.
  - ✍ In the academic world, we distinguish between *active* (i.e. *detection*) and *passive* (i.e. *threat management*) solutions.
  - ✍ In the industrial world, we have documented the different strategies proposed by commercial software available on the market.
  - ✍ The objective was to assess the advantages and disadvantages of each solution in order to define the specifications for our own solution.
  - ✍ To the best of our knowledge, such an exhaustive review of the various anti-keylogger solutions, analyzing together both academic and industrial solutions is unprecedented.
- ☞ In the end, our study allows us to know the objectives and means used by the keylogger threats while making a survey of the good ideas in the existing solutions and the mistakes to avoid.

THIS PAGE INTENTIONALLY LEFT BLANK

---

# Chapter 6

## Gostxboard solution

### 1 Problem and definition of the need

#### 1.1 Objectives sought

From a pragmatic point of view, our problem aims to fight against the threat of keyloggers. The previous sections have made two points that are absolutely essential if we hope to counter this threat. The first is a fine understanding of the technical context in which we operate under Windows. This context allows us to remove solutions that are not efficient enough to only keep the ones that offer interesting possibilities. Then it provides us possibility to know how to do it. The second point is the state-of-the-art about the threat and the countermeasures against this threat presented in Chapter 5. The first state-of-the-art helps us to understand what we have to fight against. From the strategies used by malware, it is possible to define possible countermeasures. This leads us to the second state-of-the-art which helps us to visualize what is being done to neutralize keyloggers.

Generally speaking, there is no perfect solution against keyloggers (Key-Point 5.30). Nevertheless, it may still be possible to provide a solution capable of being effective in a given context. A bit like some solutions that only deal with passwords, we could propose a specific context to fight against keyloggers. It is precisely from a well-defined situation that we must define the objectives of our project. We do not wish to limit ourselves to the case of password capture. This is a very specific case, abundantly covered in the academic literature [14, 12, 1213, 1215, 1223, 1224, 1225, 44] and generally only able to process small amounts of data or with a very specific user interface. We prefer a more generic solution, capable of handling any keyboard device, on long texts (including short texts such as passwords) in specific contexts. This context for us is to limit ourselves only to the protection of *voluntary software* which desire to receive keyboard input in a secure way.

Overall, we seek to minimize the negative impacts and maximize the good ideas of already existing and presented solutions. Considerations presented here are based on the synthesis of Table 5.6. Substantially, our objectives are :

- Free, open-source and correctly documented ;
- Be able to process any text captured from the keyboard ;
- Be able to secure both administrator and non-administrator applications ;
- Be able, whenever possible, to ensure security at the kernel level ;
- Be able to ensure self-protection of the defense solution ;
- Unprotected applications cannot reconstruct what was provided by the keyboard while protected applications are processing it ;
- Do not introduce a source of instability for the protected application ;
- Do not impact the user-experience — preferably *fire and forget* software.

## 1.2 About to secure administrator applications requirement

### Key Point 6.1:

- ☞ We propose to define a *maximum protection criteria* to deal with the case where a malware is running with administrator rights.
  - ☞ Anti-malware activity is a battle lost in advance if malware has administrator privilege.
  - ☞ But we can try to complicate the fight for the attacker.
- ☞ We propose to force the attacker to perform an action that is at least equivalent to removing our solution.
  - ☞ It is a visible action (from users's eyes) that can be audited and potentially reported.
  - ☞ It also means allowing the user to voluntarily choose not to use our solution anymore.

It is appropriate to make a clarification on *administrator specification* established previously in section 1.1. First of all, we need to explain who could be concerned by this notion of administrator. On the one hand, there are malware that may have such rights for some reason. On the other hand, there are protected applications that could have these rights too. This last case is easy to handle in our context since it is an additional guarantee of security against potential threats that would not be administrator. This brings us back to the case of a threat evolving with administrator rights.

Protection from an application running with administrator privilege may appear to be futile. Indeed, we have already demonstrated how holding administrator privilege is enough to bypass most security (Key-Points 5.8 and 5.26) when it is not directly discussed by Microsoft [1130]. When malware runs with administrator privilege, the war is already lost because the worst has already been done with this gain of privilege (knowing if such right has been acquired in legitimate way or not does not change the situation). At worst, with such rights, the malware can simply uninstall the security software. Thus, it will not be bothered by the last anymore. This argument is true for all situations, with a nuance however.

Indeed, uninstalling a security software (for instance an antivirus software) is not transparent to the user's eyes. An icon will be missing in the *Taskbar* [1302], as well as invasive commercial pop-ups will be missing and more generally the security functionality usually provided will no longer be ensured (which will certainly have consequences). Assuming that the user is physically in front of the machine is not an assumption that is out of limits in our case. Indeed, fighting against keyloggers is about protecting the keyboard keys that the user (in front of the machine) presses. It is thus inevitably in front of the machine. And this may be where anti-keylogger research may be different from other research in malware threats.

Obviously, the protection cannot be perfect but it can be possible to enhance the security. On the one hand, by making sure that if a malware with such privileges needs to perform a visible action. More directly, a visible action by the malware means to perform something visible to user's eyes. For instance, removing our security solution is something that can be easily observed by any user checking which process or services are running on Windows. On the other hand, we can try to restrict the access of a malware to a protected process. More directly, a protected process should not be bothered without its own consent. This includes access to its memory, both for reading (to read a potential cipher key or the content of the keystrokes) and for writing (to change the behavior of the protected process).

Of course, it is possible to object that even if these two elements increase the security, it is still possible to uninstall our security solution. It is assumed and we want give the user the choice to use our product and to stop using it. How could it be otherwise? Assuming this is not the case, this would lead to an arms race [1280, 1303] with users who would like to uninstall our solution (for whatever reason: boredom, malfunction, better choice, etc.) but who could not. This is the same choice that has been made by GuardedID software (Key-Point 5.29). But this is the only logical choice if we want to provide security at the cost of constraints that is not worse than those already induced by malware.



That way, it is possible to uninstall our solution. But the removing our software would result in a display of specific alerts (displayed to user or reported to the machine's administrator). Of course, for the sake of argument, it is always possible to try to imitate the interface displayed on the screen by our solution or fake activity journals, prevent alerts and reports with administrator privilege... But this requires a lot of work, especially the possibility to remove a driver from the system. Such an action that can be monitored by our security solution in some cases and sometimes notified to the user's eyes.

From our point of view, the security with threats evolving with the administrator rights must ensure that malware perform an action that would be equivalent to uninstalling our solution completely. If it is not perfect, it is the best compromise that can be achieved in a context where the situation (an attacker already has all the rights on the system) is clearly hopeless.

### 1.3 General considerations about our solution

#### Key Point 6.2:

- ☞ We recall good ideas from the projects presented in the state-of-the-art.
- ☞ We will cipher the content of keystrokes and act only on the applications that need to be protected (identified by the use our a dedicated library provided by us).

From a technical point of view, many solutions are possible. The solution of using an auxiliary channel to reroute legitimate keystrokes has often been discussed, both in academic (with virtual keyboards [14, 1214]) and industrial worlds (with KeyScrambler, or GuardedID). Even if the solution is attractive, the use an auxiliary communication channel at the kernel level is not perfect. By adding hidden communication channel, we propose a new way to retrieve keystrokes for an application, but not real security. Indeed, if the malware knows how to interface with the hidden channel, it would be able to get access to its content. The hypervisor [44] is more interesting because it allows to manage keystrokes in a secure way in order to better reroute them in the application to protect. But this solution faces technical implementation difficulties nowadays (mainly under Windows) and the fact that it is necessary to interact with the OS in a reliable way. Without having to create a new communication channel, we still have to use the existing one. Of course, we must improve its security. Surprisingly enough, this solution — the simplest one — is neither discussed at the academic level nor present in an industrial solution. We will try to explain why.

Protecting the existing communication channel means dealing with the problem of distributing keystrokes only to application(s) that must be protected. Identifying which application is legitimate for original keystrokes is a hard problem. On the one hand, from a conceptual point of view, it comes down knowing whether an application is malicious or not. On the other hand, from an operational point of view, this is equivalent to distributing keyboard's keys to only one subset of applications at a time. And in practice, only one at a time. Usually, it is based on the application that has access to the keyboard focus. Internals about this property is not documented by Microsoft and it is dangerous to blindly trust it (since it can be updated programmatically at any time — Key-Point 4.41). Note also that access to the keyboard is not always conditioned by the keyboard focus (Key-Point 4.50).

From our point of view, the idea of providing a SDK seems to be a good solution. Like Zemana's SDK solution, but with the difference that we only interact with protected processes and not with all the others. Hence, the use of a SKD aims to have a most intimate interaction between the software to protect and the library used. That is to say, we can target the processes that need (by design) to get a secure access to the keyboard. This first solves the problem of identification of protected processes versus regular ones. Indeed, this is the software that wishes to be protected that uses our SDK library... In addition, it also prevents from on-the-fly modification via Dll injection. That way, the SDK is inserted at compilation time in the protected software and only for itself. That way, it does not impact badly other software with Dll injection...

An application to be protected could use our library to secure its access to the keyboard while taking

benefits from the protection offered by a defense system designed by us. Our philosophy is to protect only software designed to interact in a secure way with the keyboard. More directly, it would be the designers of the software who would have the choice to secure some of their actions on their software, guaranteeing the stability and integrity of their applications. By consequence, this solution has the merit of no longer depending on Dll injection mechanisms, which are not rarely stability-oriented. Of course, since our solution is based on a library, it is still possible to inject it into a process as needed. But it is now an option rather than a potential feature.

## 2 General architecture of the solution

### Key Point 6.3:

- ☞ Our solution ciphers the keystrokes with a kernel-mode driver to be deciphered only by applications which use our protecting SDK.
  - ☞ Identification of protected application is easy since this one indicates by itself the need to be protected.
  - ☞ Our solution keeps compatibly with the raw input thread by keeping ciphered keystroke in the windows message system.
  - ☞ Of course, only protected a process can get access to the content of original keystroke.
- ☞ There is only two function calls (at initialization and to decipher) with our library. The rest is completely transparent, both for developers and for users of the protected process.

From a practical point of view and to be really effective, our solution must be embedded deep into the kernel. User-mode solutions only are not efficient enough (Key-Point 5.28). More directly, we will use a driver whose responsibility is to receive the keystrokes and modify them so that they can only be understood by protected applications. The modified keystrokes will be routed through the usual channel, that is to say the one managed by the raw input thread with message system. It means that keystrokes will pass through the usual filters previously setup by the intermediate drivers on the device stack to reach the applications. From there, two possibilities. The first case is that the application is not legitimate to receive the contents of the keyboard when a protected application is running. In this case, nothing is done to help the application that receives the modified keystroke. In the other case, the application is protected and must receive the code of the key pressed. In this case, it is up to our SDK to provide the necessary support to proceed.

The objective here is to describe the defense architecture implemented by our solution and mainly by our driver. A general representation of our solution is given in Figure 6.1. The idea is to retrace the process, step by step, by clearly identifying each of the actions involved. In the following paragraphs, we will attempt to describe each of these points and refer to them in relation to Figure 6.1.

The benefits provided by our solution are based on the protection of the keystrokes codes provided by our driver. Since we do not use an auxiliary communication channel, we have to cipher the stream of data passing through this channel. Therefore, ciphering data implies *cipher key management*. In practice our driver is started at boot time, when the operating system is starting. It does not need to be active in the keystrokes protection because no application is launched except the kernel and hardware drivers. The protection is only activated when an application that needs to be protected is launched. In this case, as we provide a SDK, the developers of the protected application can request the protection to be started thanks to a dedicated API. This operation is performed in several steps, as given in Figure 6.1.

1. The first operation is to deal with cipher keys. Our API requires a cipher key that will be saved, via our SDK API, securely in memory. This should be done either at the start of the application or — in a more optimized way — as soon as the application needs the user to type text on the keyboard. This is why the protected application requires a cipher-key from the protection driver.
2. The driver receives the request for protection sent from our SDK. After a few security procedures (authen-

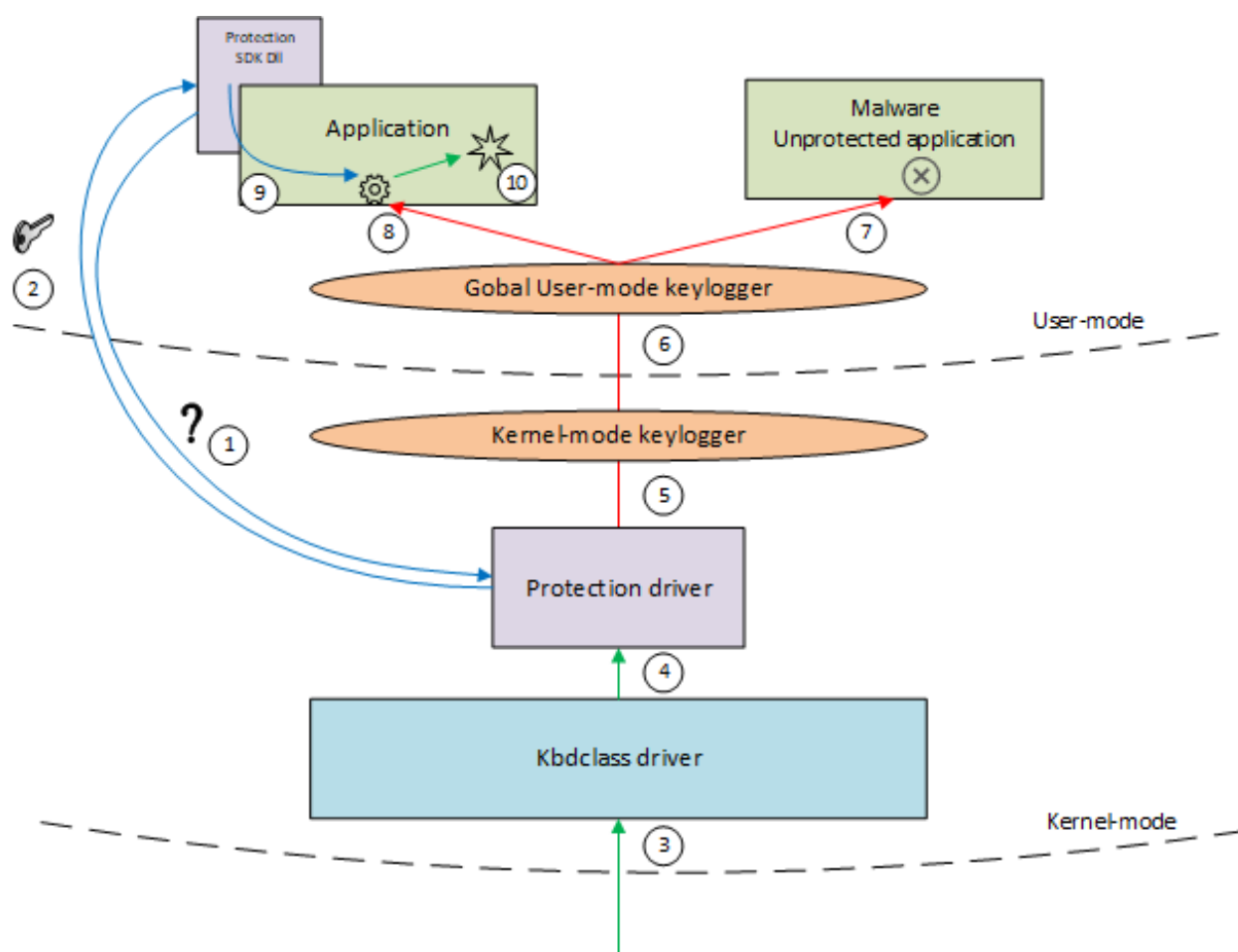


Figure 6.1: General architecture of our protection solution.

tication of the SDK in memory and setup of the general protection of the requesting software, to name a few), it exchanges a driver’s generated cipher key with the application. It means that the application using our SDK and the driver are now connected together since they are sharing the same secret key.

3. Information coming from the keyboard is received by the operating system in *kbdclass.sys* driver (Key-Points 4.5 and 4.26). This driver deals with scan codes and not virtual key code. This information is transferred to our protection driver.
4. Our driver ciphers the content of the scan code to make it not readable for applications that do not have the cipher key.
5. Once this cipher operation is performed, the modified scan code is delivered to the rest of the drivers of the device call stack.
6. The procedure then follows its classical flow as described in Chapter 4, section 5 to reach user-mode processes. Information about a key pressed event can be obtained in various ways (Key-Points 29 and 4.46).
7. If we have to deal with (and potentially malicious) application, the last one receives information that is irrelevant because it is ciphered. And since it neither uses our SDK nor is it referenced as an application to be protected, it cannot contact the driver for help.
8. When we are dealing with a protected application receiving a ciphered keystroke, the SDK retrieves the scan code sent and deciphers it.

9. From the application point of view, the decryption is simply performed thanks to our SDK API via a direct function call.
10. The content of the ciphered scan code is retrieved thanks to usual API means by the developers. Then, they just have to call our API with the ciphered scan code to get the clear content. Our API is responsible to manage cipher-key and deciphering procedure in the background.
11. From there, the application can resume to come back to its normal activity.

From the point of view of unprotected application, there is nothing very interesting to do with this keystroke information received. With this approach, we might think about the approach proposed by NoisyKey solution [42], with the difference that we are our security is directly embedded into a single protected process, providing security from kernel-mode and without any Dll injection.

From the point of view of the protected application, only two points are added in the implementation for the keyboard interaction. The first is about to signal to the driver that the application is a protected one (at initialization time). And the second is to add a function call once the scan code has been read in order to decipher it. Notwithstanding these two points, the rest of the procedure is totally transparent for the protected application.

### 3 Genesis of the project

#### Key Point 6.4:

- ☞ The project has been originally presented at DEF CON 23 conference in 2015 with Paul Amicelli.
  - ☞ At this point, it was based, on the architecture of the Ctrl2Caps project (Key-Point 5.10), which subsequently evolved in the light of our study.
  - ☞ The original idea of our work was to generalize the concept of Ctrl2Caps with key swapping to apply it to the whole keyboard randomly.
  - ☞ It was a proof of concept but it partially handled some special issues (especially low level hooks — Key-Point 4.56).
- ☞ Our solution has been improved over time, especially from the Defcon conference.
  - ☞ Over time, we have used several driver technologies, including WDF and WDM and kept the last one.
  - ☞ A better knowledge of the Windows system (Chapter 4) and the threat as well as existing solutions (Chapter 5) allowed us to achieve a more stable and efficient result.

A few words on the genesis of the project could enlighten some choices and implementations for our solution. Historically, this project was carried out with Paul Amicelli to be presented at the DEF CON 23<sup>1</sup> conference in 2015 [1304]. Our solution has been named "GostxBoard". The name comes from the willing to offer a secure cipher key entry module for the GostCrypt software<sup>2</sup>. More generally, the need has been extended to the protection of software requiring an interface with the keyboard (word processing in particular). Originally, the project was first engaged to meet a need (the fight against keylogger threats) with technical means (a driver that ciphers keystrokes). It turns out that our knowledge at the time of the conference about the keyboard was more limited than it is today. In a way, this part is intended to be an extended, updated with a real research approach of our past work.

Technical choices made at former times have been reworked and modified with respect to the solution presented. Nevertheless, the main architectural lines (as presented in section 2) remain. Therefore, it may be

<sup>1</sup><https://www.youtube.com/watch?v=W5B-zjaDzfU&list=PL9fPq3eQfaaBuHqVvDzPoWxznYYmyx5UX&index=54>

<sup>2</sup><https://www.gostcrypt.org/gostcrypt.php>

worthwhile to focus more on concepts than on technical details. More directly, our solution was more a proof of concept or feasibility based on Windows 7 than a general solution. We simply did our best with the knowledge we had at the time and we worked with an engineering logic. We were very pleased with the result obtained, which was functional and we were even very flattered to see our project published at Defcon. But, with Windows 10, the project was not stable enough over time and it could be improved on many points. Here started the resumption of research, justifying this large chapter 4 in our study.

There was no such state-of-the-art or even such an exhaustive study of existing solutions from third-party vendors at the time we started the project. We historically started with the open source Ctrl2Caps driver [1132] from Mark Russinovich (Key-Point 5.10). Indeed, this project has the interesting ability to swap the content of two keyboard keys. The original idea of our work was to generalize this concept of key swapping to apply it to the whole keyboard randomly. If this feature between an application and the driver could be controlled in a secure way, then it becomes possible to provide security. Indeed, third-party applications would only see inconsistent keystrokes when the protected application sees the real ones.

Ctrl2Caps driver is a WDM driver, as explained in Key-Point 5.10. The first proof of concept was performed with this technology. But evolving with the driver and using Microsoft documentation, we switched to a WDF driver technology after getting access to *Keyboard Input WDF Filter Driver* (Kbfiltr) project [1305]. The Kbfiltr sample is an open-source example of a keyboard input filter driver (Key-Point 5.12). In particular, it explains how to interface (to retrieve and to modify) the content of keystrokes received from the keyboard device. In the past, it was written using WDM technology, the same we used from Ctrl2Caps driver. But the WDM version of Kbfiltr has been deprecated and updated with a WDF version. Without any originality, we simply followed this evolution.

What is the situation today? The solution is globally similar in the objectives followed and in the architecture provided. But we have changed (one more time) the technologies on which it is based. We came back to the WDM driver technology. Why this evolution? Firstly, because WDF is an important and voluminous technology for a problem that can be solved simply, in the end. Secondly, it is easier to implement some protection mechanisms with WDM, especially about self-defense. Finally, even if Microsoft prefers to recommend WDF drivers [1161, 1156], WDM is not irrelevant in the sense that it offers a great control over the actions one could undertake — as long as one has a very good control over the Windows kernel technology. In addition, it is removing a certain “magic” from doing things. This is ideal for us, from a pedagogical point of view, to explain and document our solution. Of course, it would not be a problem to port our solution to WDF driver.

## 4 Detailed architecture of GostBoard solution

From this general architecture, we will focus on detailing various mechanisms and implementation details of our solution. We propose a targeted approach on fundamental points for our solution. Why? Because the construction of a driver and a SKD are complex operations, with some details that are not directly related to our main problem. Moreover, technologies evolve but the concepts remain with time. Throughout these subsections, we will focus on describing the different elements in the order of the different steps of our protection procedure, as described in Figure 6.1.

### 4.1 GostBoard WDM driver

Thanks to the knowledge of the Windows keyboard management (Chapter 4), we are going to look at how to interface with Windows to process keyboard data. Of course, the idea is to deal with what has already been presented while specifying how to implement the interaction with the Windows kernel. That way, having documented in detail how Windows works, we can justify and detail our technical choices.

---

### 4.1.1 Entry point implementation considerations

#### Key Point 6.5:

- ☞ A driver must be composed of at least one entry point (`DriverEntry` routine) and one exit point (`DriverUnload` routine).
  - ☞ The exit point is optional. But in case it would not be present, it is not possible to unload the driver (which makes the driver immortal).
  - ☞ This is a possible defense, but it prevents the administrator from shutting down our service.
- ☞ It is possible to select the devices on which our driver wishes to operate in the `AddDevice` routine.
  - ☞ We can filter by device technology (PS/2, USB...), configuration and other criteria...
  - ☞ In our case, we are generic and we accept any type of keyboard device.
  - ☞ We are notified at driver initialization step for each present keyboard device or when one is plugged in at run-time.
  - ☞ We are inserted in the driver device stack as the *highest device object*,
  - ☞ We have to keep the pointer representing the next lower-driver in the device stack to pass it requests.
- ☞ Run-time operation are handled through IRPs, each corresponding to a specific action represented thanks to a dedicated *IRP major function code*.
  - ☞ Some of these operations are mandatory (and must be handled whatever the driver is) and some other are optional.
  - ☞ When the operation is finished, the IRP is passed to the next driver in the driver device stack (known thanks to `AddDevice` routine).

The basic architecture of a filter WDM driver is not complex in itself. The idea is that this driver will be integrated between two drivers in the device stack in order to add its added value, i.e. key ciphering. It is therefore necessary to be focused on this task and let the other requests reach the drivers in the device stack. In a comparable way, we adopt the minimalist architecture of `Ctrl2Caps` [1132]. In a more modern way, we are closer to what is done in `VirtualBox` software [436], especially its mouse driver controller<sup>3</sup>. Adapting this last code for the keyboard is not very complex since the logic remains the same (the difference lies in the management of different events).

At the implementation level, from the driver entry point (`DriverEntry` [1306, 1307]), we have to face requirements when writing a driver interacting with plug and play devices (such as keyboard) [1308]. The main requirement is to supply entry points for the driver's standard routines [1309]. All drivers are able to initialize a system-defined set of standard driver routines [1310]. This is specially required in order to process IRPs. The idea is that each driver manages a particular task assigned to it (and therefore a certain number of operations handled through IRPs) at its level in the driver device stack. Technically, lowest-level drivers that directly control physical devices have more required routines than higher-level drivers, which typically pass IRPs to a lower driver for processing (because requests are going "down" in the driver device stack). Note that all of the standard driver routines have to be handled by drivers. This basic set of standard routines are present to handle most common IRPs. Some routines must be initialized by each kernel-mode driver and some other are optional, depending on the driver type and location in the device stack.

Among the mandatory routines, there is obviously the entry point `DriverEntry` and the `DriverUnload` routine [1311]. This last one is called when the driver is about to be stopped or at shutdown time, to release resources and to stop its activity with the device. If this routine is not present, the driver cannot be unloaded. More directly, it would impact the system if all devices had been removed from the machine. In such a case, it could

<sup>3</sup><https://www.virtualbox.org/svn/vbox/trunk/src/VBox/Additions/WINNT/Mouse/NT5/VBoxMFDriver.cpp>



not be detached properly from the device stack, meaning the driver would be *unloadable* and *unkillable*. This routine can be very simple, often consisting only of a return statement, but it must be present. More information about this concept in [1312, 1313].

Another standard driver routine is the `AddDevice` routine [673]. Previously mentioned in Key-Point 4.24, this routine is called for each device controlled by the driver during system initialization and each time a new device is enumerated while the system is running [674]. This routine is registered in the `DriverEntry` to create device objects representing the driver carrying I/O requests [675]. More directly, in our case, it attaches the device object to the device stack, so that the device stack contains a device object for each driver associated with the device. This procedure is performed thanks to `IoAttachDeviceToDeviceStack` routine [677]. Attachment is done with the highest device object currently layered in the chain (but there is no way to determine whether drivers are being layered in the correct order [677] in the call stack). This allows our driver to filter all or a selected set of devices. In our case, we interact with all keyboard devices since our solution is generic. But it also allows us to attach ourselves directly into the device stack as close as possible to the device where it is interesting to act.

The complete procedure to deal with the `AddDevice` routine for filtering purposes is given in [1314]. For the sake of example, we provide Code 6.1 to illustrate the main steps of an implementation of `AddDevice` routine. Remember, little programs do little to no error checking. This one is only provided for comprehension purposes. The exact version of the `AddDevice` routine implemented by our driver is a bit more complete (especially with error handling) and fully follows the requirements from [1314].

```

NTSTATUS DrvAddDevice(_In_ PDRIVER_OBJECT Driver, _In_ PDEVICE_OBJECT PhysicalDeviceObject){
    NTSTATUS status = STATUS_SUCCESS;
    PDEVICE_OBJECT fDO = NULL, pDOParent = NULL;
    PHIDF_DEVEXT pDevExt = NULL; // Defined by us.

    // This code can be paged (IRQL PASIVE_LEVEL).
    PAGED_CODE();

    __try {
        // Creates a filter device object (filter DO) for the device being added (called fDO).
        status = IoCreateDevice(Driver, sizeof(HIDF_DEVEXT), NULL, FILE_DEVICE_KEYBOARD, 0, FALSE,
            &fDO);
        if (!NT_SUCCESS(status)) {
            __leave;
        }

        // Get access and set to zero the device extension which is defined by our driver.
        pDevExt = (PHIDF_DEVEXT)fDO->DeviceExtension;
        RtlZeroMemory(pDevExt, sizeof(HIDF_DEVEXT));

        // Define flags in the device extension to track certain PnP states of the device
        // and all mechanism for queuing IRPs.
        // (...)

        // Set the DO_BUFFERED_IO flag bit in the device object to specify the buffering
        // used by the I/O manager is buffered I/O requests for the device stack.
        // Set the DO_POWER_PAGABLE flag for power management since our driver is pageable.
        fDO->Flags |= (DO_BUFFERED_IO | DO_POWER_PAGABLE);

        // Attach the device object to the device stack. We keep the returned pointer to
        // the device object of the next-lower driver in the device stack. This one will
        // be used by IoCallDriver when passing IRPs down the device stack.
        pDOParent = IoAttachDeviceToDeviceStack(fDO, PhysicalDeviceObject);
        if (!pDOParent) {
            status = STATUS_DEVICE_NOT_CONNECTED;
            __leave;
        }

        // Keep relevant information the our device extension defined by our driver.
        // The goal is to get access such data in each IRP managed by our driver.
        pDevExt->pdoMain = PhysicalDeviceObject;
        pDevExt->pdoSelf = fDO;
    }
}

```



```
pDevExt->pdoParent = pDOParent;

// Clear the DO_DEVICE_INITIALIZING flag in the FDO or filter DO as expected.
fDO->Flags &= ~DO_DEVICE_INITIALIZING;

// Success.
status = STATUS_SUCCESS;
}
--finally {
// (...)
}

return status;
}
```

Code 6.1: Example of `AddDevice` routine used by our driver to handle each keyboard device plugged in the system.

The last mandatory standard driver routines are the dispatch routines. They are used to proceed any IRP major function code (IRP\_MJ\_XXX<sup>4</sup>) such as "open", "close", "read", or "write" operations to name the most common ones [888]. They transmit requests into the stack of drivers. A driver can initialize one or more dispatch routines with dedicated or mutual dispatch routines defined by itself.

There is no real documented behavior which is required for all dispatch routines. Actually, the required functionality of a particular dispatch routine varies, depending on the I/O function code it handles, on the individual driver's position in a chain of drivers, and on the type of underlying physical device [1315]. But generally speaking, these routines first check that the transmitted IRPs are valid and that they have a specific action to do on them. Whenever an action is required, a dispatch routine must satisfy the request and complete the IRP if possible. Otherwise, the routine passes it on for further processing by lower-level drivers or by other device driver routines [607].

Technically, the dispatch routines are divided into two groups: those required [1316] and those optional [1317]. The difference lies only in the IRP major function codes they interface. More generally, we can consider that the dispatch routines are required when the operations they process are the most common and therefore necessary for the proper functioning of the interfaced device (PnP device recognition, pertaining to the power state, get or release access to the device, transfer data from or to the device, handling I/O control code for specific operations, or system-defined internal device I/O control requests, and WMI [686] requests).

Regardless of the type of IRP that is supported by a dispatch routine, the prototype of this type of routine is always the same. A `DRIVER_DISPATCH` callback routine [1318] is composed by a device object [1319] to represent the target device (previously created by the driver's `AddDevice` routine) and a pointer to an IRP structure that describes the requested I/O operation [530]. Depending on the major function code associated with the dispatch routine, it is possible to retrieve, in the IRP, specific information about the current action to be processed.

In our case, it is normal to initialize all dispatch routines with the same *by default* dispatch routine (Code 6.2). The loop provided in Code 6.2 sets all dispatch routines (stored in the *DriverObject* provided by the operating system as the first parameter of `DriverEntry` routine) to an internal routine called `IrpPassthrough`. This last routine retrieves the device extension object we set in the `AddDevice` while we attached our driver (represented by the device object created thanks to `IoCreateDevice` and linked to the keyboard device in the case of our driver) to the device thanks to `IoAttachDeviceToDeviceStack`. This last routine returns a pointer to the previously highest device object in the device stack (no order is guaranteed about who is supposed to be this device in the stack when our driver is loaded). This pointer has been saved in an *extension* crafted to the device object we created with `IoCreateDevice` routine. Since we kept that pointer representing the next lower-driver in the device stack (we are inserted as the highest device object), it is possible to pass the IRP to the next lower-level driver once our operation has been performed on the IRP. Since our *default* dispatch routine has nothing to do with the IRP, it is mandatory to call `IoSkipCurrentIrpStackLocation` macro [1320] before passing it with `IoCallDriver` [1321] to the next lower-driver in the device stack [1322, 1323]. Note that if required, it is possible to define a completion

<sup>4</sup>Where "XXX" represents any code order used by windows. A complete list of system-defined major function codes is given in [606].

routine to be notified once all lower-driver have completed the IRP thanks to a completion routine [690].

```

for (int i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++){
    DriverObject->MajorFunction[i] = IrpPassthrough;
}

// (...)

NTSTATUS IrpPassthrough(_In_ PDEVICE_OBJECT DeviceObject, _In_ PIRP Irp){
    PDEVICE_OBJECT_EXTENSION DevObjExt;

    DevObjExt = (PDEVICE_OBJECT_EXTENSION) DeviceObject->DeviceExtension;

    IoSkipCurrentIrpStackLocation(Irp);

    return IoCallDriver(DevObjExt->PDONextLowerDriver, Irp);
}

```

Code 6.2: By default initialization of the dispatch routine from our WDM protection driver.

Of course, all dispatch routines are not necessary a default IRP *pass-through* routine. For some operations which require to be handled specifically by a driver, it is possible to define a dedicated routine able to handle a given operation. The procedure is exactly the same by provided in the array *MajorFunction* (which belongs in the *DRIVER\_OBJECT* structure) a pointer to the dispatch routine. The selected index of *MajorFunction* array corresponds to the IRP major code function [606] that the dispatch routine should handle.

#### 4.1.2 Handling keystrokes

##### Key Point 6.6:

- ☞ In practice, there are two main technologies to interact with keyboard keystroke at driver level.
  - 👉 Old school style as Ctrl2Caps through *dispatch read routines* (Key-Point 5.10).
  - 👉 In a more direct way by updating *KeyboardClassServiceCallback* callback routine as *Kbfiltr* (Key-Point 5.12).

At that point, our objective is very clear: we need to change the content of the keystrokes scan codes in our driver. To do this, we need to describe how to access the keystrokes processed by our keyboard. Technically, we have two possible approaches to deal with keystroke management. On the one hand, as Ctrl2Caps driver (Key-Point 5.10) at *dispatch read routine* level. On the other hand, as Kbfiltr Driver (Key-Point 5.12) with a provided hook procedure provided for each keyboard in the system (Key-Point 4.26). The two systems present similar results despite the fact that they are based on a different logic. There is, however, a difference in the way actions are faced between the two approaches. We propose to explain these two systems since they are critical.

### 4.1.2.1 Read dispatch routine

#### Key Point 6.7:

- ☞ In this case, the reading procedure is performed in two steps.
- ☞ First in the *read dispatch routine* recording to handle any read operation for the driver.
  - ☞ The read IRP engaged by the raw input thread is handled by our driver with a regular read dispatch routine.
  - ☞ The reading order has been passed but there is nothing to read yet (a key must be pressed).
  - ☞ This is why a completion routine is registered to be automatically called when the read operation will be completed.
- ☞ Then in the completion read routine registered by the *read dispatch routine* and notified whenever a read operation completed.
  - ☞ Several keystrokes can be returned as an array of `KEYBOARD_INPUT_DATA` structures.
  - ☞ We have access to the scan code of each keystrokes as provided by the device (normalization is performed in the raw input thread — Key-Point 4.37).
  - ☞ Here we can read and modify the content of the scan code as provided by the device.

The simplest solution is to reproduce the architecture of what was done with Ctrl2Caps project. USB keyboard is historically managed by a pooling system lying on a read IRP. In a more marginal way, the information transmission chain of a PS/2 keyboard on modern versions of Windows also uses a read IRP to retrieve the content of the keyboard. This IRP processing is used once the interrupt has been processed by `i8042prt.sys` driver and handled by `kbdclass.sys` driver.

Thus, all we have to do is to deal with the reading IRP to achieve our goal. As explained in Chapter 4, section 5.1.4.3 (Key-Point 4.34), this IRP is initialized from the raw input thread. That way, the order begins at the top (the raw input thread, in a way) of the device driver stack and it ends at the level of device itself, passing through all drivers, including our driver. The read dispatch routine is notified when the read IRP is dispatched to all drivers in the stack. In this case, the IRP is empty because it has been armed in order to wait for information to come from the keyboard. This information will come on the way back, when the keyboard provides information (when a key is pressed), the IRP goes back up to the raw input thread to be processed. On the way, it can be intercepted by any driver in the call stack drivers if and only if a dispatch routines has been engaged thanks to an *I/O completion routine* via `IoSetCompletionRoutineEx` routine [688]. In the context of a I/O completion routine, a callback is registered [689] and notified when the IRP has been completed by lower-drivers [695]. For the sake of simplicity, this mechanism can be compared to a *handmade* pre and post-callback operations subsystem [692]. An example of read dispatch routine registering a completion routine (called `DispatchReadComplete` in our case) is provided in Code 6.3.

```

NTSTATUS DispatchRead(_In_ PDEVICE_OBJECT DeviceObject, _In_ PIRP Irp) {
    PHIDF_DEVEXT devExt = NULL;
    PIO_STACK_LOCATION currentIrpStack = NULL;
    PIO_STACK_LOCATION nextIrpStack = NULL;

    // Gather our variables.
    devExt = (PHIDF_DEVEXT) DeviceObject->DeviceExtension;
    currentIrpStack = IoGetCurrentIrpStackLocation(Irp);
    nextIrpStack = IoGetNextIrpStackLocation(Irp);

    // Push params down for keyboard class driver.
    *nextIrpStack = *currentIrpStack;

    // Set the completion callback, so we can get access to the keyboard data.

```

```

IoSetCompletionRoutine(Irp, DispatchReadComplete, DeviceObject, TRUE, TRUE, TRUE);

// Return the results of the call to the keyboard class driver.
return IoCallDriver(devExt->pdoParent, Irp);
}

```

Code 6.3: Dispatch read routine used to register a completion routine called whenever the read request completes (meaning something has been read from the keyboard).

It is precisely in the return operation (the completion routine) that the IRP has been completed and where it is possible to access and modify the content of the keystroke. In this case, the dispatch routine registered for an IRP\_MJ\_READ operation [733] is handled in a specific way. Following the prototype of a regular dispatch routine [1318], the IRP has specific initialization as documented in [734] (section 5.1.1). In this case, the IRP\_MJ\_READ request transfers zero or more KEYBOARD\_INPUT\_DATA structures [737] from kbd-class.sys internal data queue. If there is no data in the data queue, a read request remains pending until it is completed or canceled. But when data is present, it can represent one or more keystrokes (the case of key combinations can trigger scenarios with multiple keys received in a single IRP). In this case, the buffer containing the IRP's data (IRP->AssociatedIrp.SystemBuffer) contains an array of KEYBOARD\_INPUT\_DATA structures. The number of elements in this table is given by dividing the value of Irp->IoStatus.Information by the size of the KEYBOARD\_INPUT\_DATA structure. From there, a simple loop allows to browse the table and to find all the scan codes in the *MakeCode* field of the KEYBOARD\_INPUT\_DATA structure. The modification can then be directly applied to this field to achieve our goals by ciphering it. The rest of the procedure is compliant with IRP handling management [607]. An illustration is provided in Code 6.4.

```

NTSTATUS DispatchReadComplete(_In_ PDEVICEOBJECT DeviceObject, _In_ PIRP Irp, _In_opt_ PVOID
Context) {

    PIO_STACK_LOCATION      IrpSp = NULL;
    PKEYBOARD_INPUT_DATA    KeyData = NULL;
    ULONG_PTR               numKeys = 0, i = 0;

    // Context could be used to retrieve cipher-key used for a given device, for instance...
    UNREFERENCED_PARAMETER(Context);
    UNREFERENCED_PARAMETER(DeviceObject);

    // Request completed - look at the result.
    IrpSp = IoGetCurrentIrpStackLocation(Irp);
    if (NT_SUCCESS(Irp->IoStatus.Status)) {

        // Get access to the buffer holding the list of keystrokes
        // provided by the stream of data coming from the keyboard.
        KeyData = Irp->AssociatedIrp.SystemBuffer;

        // Get access to the number of keystrokes (stored as a memory continuous array).
        numKeys = Irp->IoStatus.Information / sizeof(KEYBOARD_INPUT_DATA);

        // For each key...
        for (i = 0; i < numKeys; i++) {

            // Display purpose (debug only).
            DebugPrint(TRACE_LEVEL_INFORMATION, DISPATCHER, ("[i] ScanCode: %x %s"),
                KeyData[i].MakeCode, (KeyData[i].Flags ? "Up" : "Down")
            );

            // Process content of each keystroke here (cipher operation)...
            // (...)
        }
    }

    // If necessary, mark the IRP as pending.
    if (Irp->PendingReturned) {
        IoMarkIrpPending(Irp);
    }

    // Return.
}

```

```

    return Irp->IoStatus.Status;
}

```

Code 6.4: Read completion dispatch routine used to retrieve the content of the keystroke data retrieved from the keyboard device.

#### 4.1.2.2 Hook by keyboard filters provided by kbdclass.sys

##### Key Point 6.8:

- ☞ Access to the keystrokes is done by registering a callback routine when receiving the IRP identified by the code `IRP_MJ_INTERNAL_DEVICE_CONTROL`.
  - ☞ The driver needs to be registered retrieves and save the data provided (to call the callback initially registered).
  - ☞ It replaces the data with its own (in particular the address of the callback routine called for each keystroke).
  - ☞ The first callback registered by the system is `KeyboardClassServiceCallback` routine.
- ☞ Unlike the dispatcher routine (Key-Point 6.7), there is no notion of completion routine here.
  - ☞ Callback is called directly once the read operation from the device has been completed.
- ☞ The callback can read, modify or delete the data provided from the keyboard.

This method is more modern than the traditional IRP interception at the device call stack level. The idea is to take advantage of being in the device driver stack of the keyboard to attach our driver directly in the chain of the `kbdclass.sys` driver. To proceed, we need to act with two complementary procedures. The first is the link to the devices attached via our `AddDevice` routine registered in the `DriverEntry`. This operation is called first for each device registered as a keyboard in the system. Once the device is detected and attached, the system notifies the driver with an `IRP_MJ_INTERNAL_DEVICE_CONTROL` code called `IOCTL_INTERNAL_KEYBOARD_CONNECT` [705] (Chapter 4, section 4.2.8, Key-Point 4.26). This control code is handled in its dedicated dispatch routine handled by the dispatch routine registered for `IRP_MJ_INTERNAL_DEVICE_CONTROL` code. Technically, this request connects the `kbdclass.sys` service to the keyboard device. This is `kbdclass.sys` which transfers this request down to the keyboard device stack before it opens the keyboard device and interacts with it.

This mechanism is used by `Kbfiltr` [1305] project to introduce a *callback routine* in the keyboard chain. In practice, in the IRP referenced by `IOCTL_INTERNAL_KEYBOARD_CONNECT`, there is a buffer (stored in `Parameters.DeviceIoControl.Type3InputBuffer` from the IRP) which references a `CONNECT_DATA` structure [706] (Key-Point 5.12), allocated by `kbdclass.sys` driver. This structure has two members. One is a pointer to the upper-level class filter device object called *ClassDeviceObject* and the second is a callback routine (`PSERVICE_CALLBACK_ROUTINE` [707] — Key-Point 4.26) that specifies the class service routine (and called *ClassService*). This structure provides a template for a filter service callback routine to be registered in `ClassService` field. Originally, this is the `KeyboardClassServiceCallback` routine [529] which is exported in the structure by `kbdclass.sys` driver (Key-Points 4.25, 4.5 and 5.12). By replacing this pointer to a filtering routine (using the prototype given in [529]), it is possible to supplement the `KeyboardClassServiceCallback` routine to filter the input data that is transferred from the device input buffer to the keyboard's class data queue. As explained in Key-Point 5.12 with `KbFilter_ServiceCallback` routine [1162] in `Kbfiltr` project, this callback can delete, transform, or insert data.

For the sake of completeness, the *ClassDeviceObject* field must also be replaced with the device object representing the driver providing the callback referenced in the `CONNECT_DATA` structure. Note also that the call chain between the different callback filters is managed manually. Indeed, each filter modifying the *ClassService* field is supposed to keep a copy of its original value. This is done in order to call, once the current callback

actions are performed, the callback routine provided in `CONNECT_DATA` structure when the current callback has been registered. Code 6.5 illustrates the procedure used.

```

NTSTATUS IrpInternalIOCTL(_In_ PDEVICEOBJECT DeviceObject, _In_ PIRP Irp){
    PIO_STACK_LOCATION Stack = NULL;
    PHIDF_DEVEXT DevExt = NULL;
    ULONG ioCtlCode = 0;
    NTSTATUS status = STATUS_SUCCESS;
    PCONNECT_DATA ConnectData = NULL;

    // Get access to any parameters for the current request.
    Stack = IoGetCurrentIrpStackLocation(Irp);
    if (Stack == NULL) {
        return STATUS_INVALID_ADDRESS;
    }

    // Get access to our device extension structure (defined by our driver).
    DevExt = (PHIDF_DEVEXT) DeviceObject->DeviceExtension;
    if (DevExt == NULL) {
        return STATUS_INVALID_PARAMETER;
    }

    // Get the current iCtlCode.
    ioCtlCode = Stack->Parameters.DeviceIoControl.IoControlCode;

    // Hook into connection between keyboard class device and port drivers.
    if (ioCtlCode == IOCTL_INTERNAL_KEYBOARD_CONNECT) {
        // By default.
        Irp->IoStatus.Information = 0;

        // Check we are dealing with a buffer at least as large as CONNECT_DATA structure.
        if (Stack->Parameters.DeviceIoControl.InputBufferLength < sizeof(CONNECT_DATA)) {
            Irp->IoStatus.Status = STATUS_INVALID_PARAMETER;
            IoCompleteRequest(Irp, IO_NO_INCREMENT);
            return Irp->IoStatus.Status;
        }

        // Retrieve CONNECT_DATA structure provided in the IRP.
        ConnectedData = (PCONNECT_DATA) Stack->Parameters.DeviceIoControl.Type3InputBuffer;

        // Keep original data stored in CONNECT_DATA structure in our device context.
        DevExt->OriginalClassDeviceObject = *Data;
        DevExt->OriginalClassService = (PSERVICE_CALLBACK_ROUTINE) Data->ClassService;

        // Update content of CONNECT_DATA structure to register our "hook" callback.
        Data->ClassDeviceObject = DevExt->pdoSelf;
        Data->ClassService = (PVOID) CallbackKeyboardClassService;
    }
    // (...)

    return IrpPassthrough(DeviceObject, Irp);
}

```

Code 6.5: Register the callback routine provided by the driver to handle every keystrokes from keyboard device thanks to `IRP_MJ_INTERNAL_DEVICE_CONTROL` IRP notification.

The callback call is used in two different situations. The first situation is when ISR dispatch completion routine of the function driver notifies `kbdclass.sys`. The second is when `kbdhid.sys` driver notifies `kbdclass.sys`. With such architecture, our driver's callback routine is notified before the original `KeyboardClassServiceCallback` routine defined by `kbdclass.sys`. Indeed, with this design (calling the original callback once the filtering callback has finished to proceed), this one is called the last since it is the lower-driver class from filter drivers point of view. This allows to modify, in a structured and documented way, what `kbdclass.sys` driver is about to receive. This is true for both PS/2 and USB/HID technology used by modern keyboards.

The `PSERVICE_CALLBACK_ROUTINE` [707] has a prototype which can handle different parameters

size. More directly, it is designed to retrieve an undefined structure able to take different sizes, depending on the service callback registered for it. In practice, this type of callback is called only in the context of the keyboard, but the prototype is designed to be generic. To do this, two of its parameters are used to define the start address and the end address of the data that is provided. It is up to the driver to structure everything between these two addresses. In our case, provided data corresponds to an array of `KEYBOARD_INPUT_DATA` structures [737]. It is possible to calculate the total size of the data in memory by subtracting the value of the end address from the base address. Then, we divide by the size of the `KEYBOARD_INPUT_DATA` structure to obtain the number of elements in the array which can be processed in a loop. This is similar to what is practiced in the case of completion context of a read dispatch IRP processing... The access to the content of the structure is the same and the modifications to be made for ciphering purposes are similar, as illustrated in Code 6.6.

```

VOID CallbackKeyboardClassService(_In_ PDEVICE_OBJECT DeviceObject ,
                                _In_ PKEYBOARDINPUT_DATA InputDataStart ,
                                _In_ PKEYBOARDINPUT_DATA InputDataEnd ,
                                _In_ PULONG InputDataConsumed
){
    PHIDF_DEVEXT DevExt = NULL;
    ULONG_PTR diff = 0, i = 0, nbPackets = 0;

    UNREFERENCED_PARAMETER(InputDataConsumed);

    // Retrieve device extension (from our driver).
    DevExt = (PHIDF_DEVEXT) DeviceObject->DeviceExtension;

    // Compute the number of packets to target.
    diff = (ULONG_PTR) InputDataEnd - (ULONG_PTR) InputDataStart;
    nbPackets = (diff / sizeof(KEYBOARD_INPUT_DATA));
    for (i = 0; i < nbPackets; i++) {
        // Display (debug only).
        DebugPrint(TRACE_LEVEL_INFORMATION, CALLBACK_READ, (" [K] \\Device\\KeyboardPort%d - 0x%04x
- 0x%04x. "),
            InputDataStart[i].UnitId ,
            InputDataStart[i].MakeCode ,
            InputDataStart[i].Flags
        );

        // Cipher procedure.
        // (...)
    }

    // Call the original class service.
    if (DevExt->OriginalClassService != NULL) {
        DevExt->OriginalClassService(DevExt->OriginalConnectData.ClassDeviceObject , InputDataStart
        , InputDataEnd , InputDataConsumed);
    }

    return;
}

```

Code 6.6: Callback routine registered to manage keystrokes in the chain of `kbdclass` interface callbacks.



### 4.1.2.3 Initialization of driver interfaces for both PS/2 and USB devices with kbdclass.sys

#### Key Point 6.9:

- ☞ Even if the initialization order is not documented (and subsequently prone to change), it can be observed that PS/2 keyboards are initialized before USB ones.
  - ✍ This does not change much from the point of view of the user who cannot really use the keyboard at this step of the system start-up.
- ☞ The initialization of a device is completed before moving on to the next device.
  - ✍ All IOCTLs are sent and processed by the system before initializing the next device.
  - ✍ Note that in the case of PS/2 devices, the ISR hook procedure (which manages the interface with any PS/2 device — Key-Point 3.3) is performed via specific IOCTLs (Key-Point 5.11).
- ☞ Whatever the keyboard device is, a `IOCTL_INTERNAL_KEYBOARD_CONNECT` is sent to register a keyboard filter for kbdclass.sys (Key-Point 6.8).

For the sake of consistency, it may be interesting to observe the keyboard initialization from our driver's point of view. The latter is realized from the VirtualBox virtual machine software on a multi-core machine, Windows 10 up-to-date and composed of two keyboards (the default one and one connected one by USB). Note that the default keyboard is embedded in the virtual machine solution of VirtualBox as a PS/2 keyboard. The use of two keyboards is not so common, although it can be used for laptop mounted on a KVM switch and sharing a USB keyboard with other machines. In this case, two keyboards are present (the one embedded on the machine and the other connected by USB with the KVM switch). The trace extracted from our driver is given in Code 6.7.

This trace shows that the initialization is done first for one type of device and then for another. Some IOCTL exchanges are performed according to the nature of each device. The requests for the USB device are not surprising. They are in the line of the description of the USB protocol (Key-Points 4.11 and 4.12 from Chapter 4, section 4.1) when there are multiples interfaces (Key-Point 4.11). In this case, the keyboard driver loads all HID interfaces referencing a keyboard (Key-Point 4.15). For the purposes of the experiment, the keyboard we used has two USB/HID interfaces (one is for keyboard usage and the second is for a *vendor-defined consumer control* usage). The last control code (`IOCTL_KEYBOARD_SET_TYPEMATIC` [1324]) is about switching off LEDs by default at starting time. Latter in the start-up procedure, LEDs will be switched on according to the configuration stored in the registry of Windows [1325].

```

1 AddDevice -> New device crafted (PS/2).
  Dispatch controller -> IOCTL 0x000b0203 (IOCTLINTERNALKEYBOARD_CONNECT) -> update
    ClassService callback.
3 Dispatch controller -> IOCTL 0x000b3fc3 (IOCTLINTERNALI8042_HOOK_KEYBOARD).
  Dispatch controller -> IOCTL 0x000b3fcf (IOCTLINTERNALI8042_KEYBOARD_START_INFORMATION).
5
7 AddDevice -> New device crafted (USB/HID).
  Dispatch controller -> IOCTL 0x000b0203 (IOCTLINTERNALKEYBOARD_CONNECT) -> update
    ClassService callback.
  Dispatch controller -> IOCTL 0x000b0000 (IOCTL_KEYBOARD_QUERY_ATTRIBUTES).
9 Dispatch controller -> IOCTL 0x000b0000 (IOCTL_KEYBOARD_QUERY_ATTRIBUTES).
  Dispatch controller -> IOCTL 0x000b0004 (IOCTL_KEYBOARD_SET_TYPEMATIC).

```

Code 6.7: "Trace from our protection driver at initialization time on Virtual Box with two keyboards."

The case of PS/2 is interesting because it illustrates how a PS/2 keyboard driver can interact with the last. The IOCTL `IOCTL_INTERNAL_I8042_HOOK_KEYBOARD` [1147] illustrated in Key-Point 5.11 gives an access into the `INTERNAL_I8042_HOOK_KEYBOARD` structure [1148] to register specific hook routines for PS/2 keyboards.

It is not relevant for us to use this type of callback since it is only specific to PS/2 keyboard (and we would

like to manage other types of keyboard too) and unnecessarily complex to manage taking into account the IRQ to which callback routines are called with this technology. The second code is `IOCTL_INTERNAL_I8042_KEYBOARD_START_INFORMATION` [1326]. This code is used to pass a pointer to a keyboard interrupt object. `I8042prt.sys` driver sends this request synchronously to the top of the device stack after the keyboard interrupt object is created so that upper-level filter drivers can synchronize their callback operation with the PS/2 keyboard. The interrupt object is provided through an `INTERNAL_I8042_START_INFORMATION` structure [1327]. This last operation is only relevant for specific cases where using hook callback routines in the `INTERNAL_I8042_HOOK_KEYBOARD` structure.

#### 4.1.3 Inserting the driver in the device stack

##### Key Point 6.10:

- ☞ We insert our protection driver just below the generic `kbdclass.sys` keyboard driver level.
  - ☞ Any lower level would expose us to having to manage all USB/HID or PS/2 devices (not just keyboards) manually.
  - ☞ A lot of work for neither functional gain nor real security improvement.
- ☞ The way a driver is installed in the system matters at least as much as the implementation of the driver itself.
  - ☞ It is during the installation that the driver device stack where the driver belongs is configured.
  - ☞ This configuration is stored in dedicated and documented keys in the registry of Windows.
  - ☞ Even if the operation can be done manually, it is usually done through an `.inf` file.
- ☞ Our solution called *GostxBoard* uses the same filtering technology as `KbFiler` project (Key-Point 5.12).
  - ☞ Our driver is registered as an *UpperFilter* of `kbdclass.sys` driver (the keyboard device driver).
  - ☞ Reusing *Ctrl2Caps* project (Key-Point 5.10), to allow an efficient filtering, registration would be "GostxBoard, kbdclass, Ctrl2Caps".
  - ☞ In this last case, our driver *GostxBoard* is notified first when a key is pressed on the keyboard.

In driver development, features implemented in the driver's source code are not the only relevant points. The way the driver is installed in the system matters because it defines how the last is integrated into the operating system. Usually, the installation of a driver is done using an `.inf` file [566, 567, 1328] — even if it can be done manually in a dedicated program. Generally speaking, as we are simply trying to be part of the keyboard's device drivers stack, there is not so much to do regarding a classic `.inf` file. We are updating `Class` and `ClassGuid` fields to "Keyboard" and its associated GUID {4D36E96B-E325-11CE-BFC1-08002BE10318} [568]. The rest of the installation procedure is quite classical and it concerns the registration with the list of services (in [1136]) to be started automatically with the operating system at boot-time.

But this boot-time procedure is not enough to be linked to the keyboard. It is necessary to indicate somewhere that the driver is linked to the keyboard device. Ideally, we try to be as low as possible in the device call stack of the keyboard. There are, however, some limitations. Too high and we could be manipulated by malicious kernel-mode drivers or be victim to low-level filters able of bypassing us. Too low and we may only filter certain types of devices (PS/2 or USB/HID only). Being too low does not imply design or safety issues. It is indeed possible to filter the two types of devices differently and to mutualize the code that can be mutualized (especially security features). But the amount of work to achieve this design is very important.

Indeed, within the USB example, going lower than `kbdclass.sys` implies to be at the level of `kbdhid.sys` (Key-Point 4.26) or `usbccgp.sys` (Key-Point 4.13) and thus to have to parse the HID protocol or worse the USB protocol in addition to the HID one. And what would be the benefits? Of course we are very low and we can modify the information at a very low level (as close as possible to the device), but the risk of error during

parsing (not to mention managing the difficulty of HID or USB parsing) is not zero. And in the end, the same security is introduced: the content of the scan codes from provided keystrokes is changed. Indeed, the change is made as close as possible to the keyboard device. But if there is a threat at the driver level, it can also be — with a great amount of work — as low as we are. The attacker’s task would be much more complex such as the defender’s job, without any guarantee that this security is definitively efficient.

It is therefore necessary to find an acceptable compromise in terms of ease of development and efficiency while maintaining a good security. Being at the level of `kbdclass.sys` is the best choice from our point of view. This choice can be justified because `kbdclass.sys` driver is the lowest element that gathers all the information related to keyboards, regardless of the type of device. Below, we are specialized in a given type of keyboard (PS/2 or HID) or in an information transport technology used (USB, Bluetooth...). In addition we have to make the distinction between is coming from a keyboard device (if there is one) and all other devices that are not keyboard (and thus irrelevant). Just being on top of transport specific drivers and specific technology is enough. That way, all keyboard devices can be managed by our driver whatever they are today or they could be tomorrow. In addition, if we look at Microsoft’s documentation [1116] about keyboard filtering in kernel-mode, this is the driver level that is explicitly recommended. Figure 6.2, inspired by Microsoft’s documentation [1116], illustrates where to put it (that is to say on the third-party components in Figure 6.2).

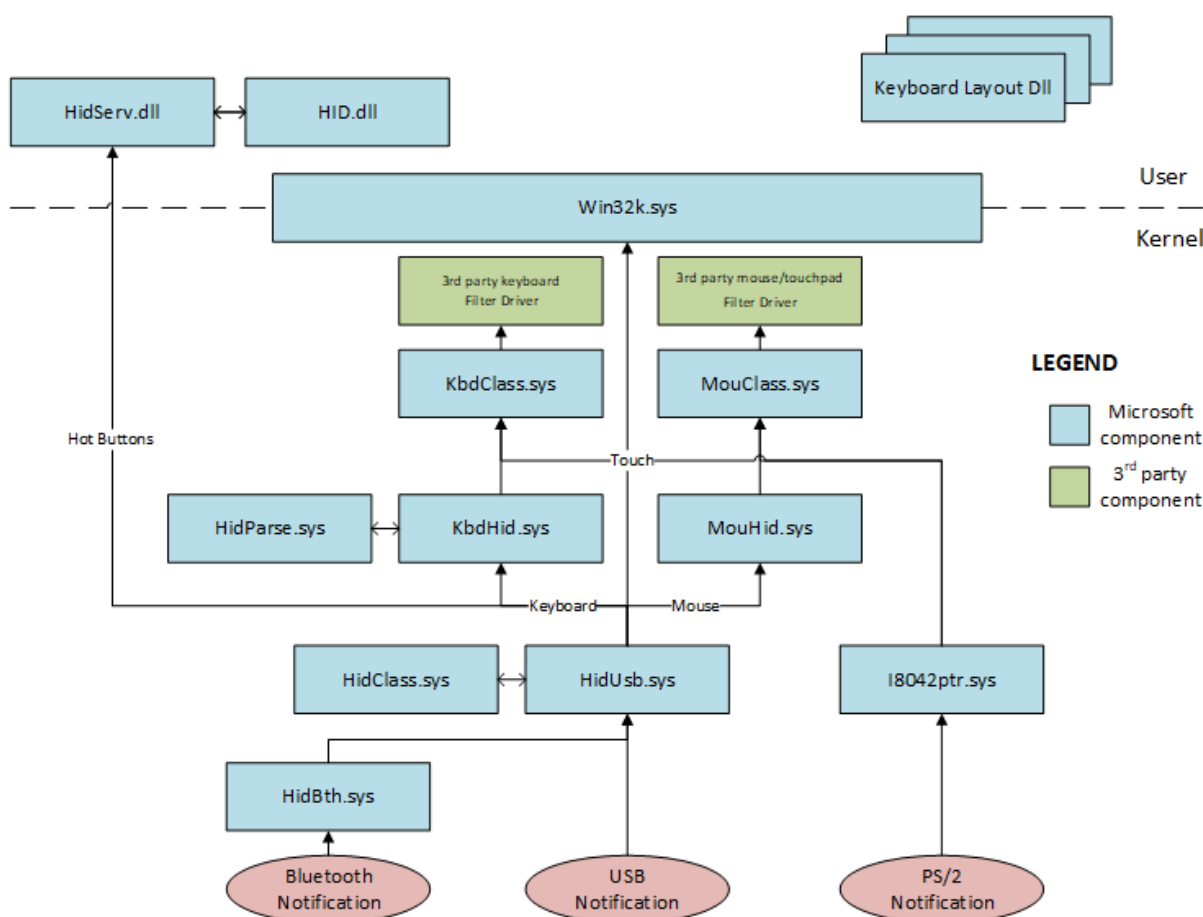


Figure 6.2: System-supplied driver stacks for USB keyboard and mouse/touchpad devices.

The objective is therefore to stay as close as possible to `kbdclass.sys`. To install a filter driver on any device, the procedure is documented in [1143]. In practice, it is enough to register in the registry of Windows “HKLM\System\CurrentControlSet\Control\Class” [1141] whose key is the GUID [568] corresponding to the type of device targeted. Once the device is targeted, we create a registry value (if it does not already exist) named *UpperFilter* or *LowerFilter* (depending on the desired objective) to be above or below the target device.

The difference between *LowerFilter* and *UpperFilter* is not as simple as it seems.

To define this notion of “upper” or “lower” correctly, we must specify that it depends from the point of view we have on the driver call stack. More directly, it requires to define the direction of data flow between the device and the application. For instance, when pressing a key on the keyboard, data comes from the device and it goes to an application, via the operating system — Chapter 4, section 4.2.4.3 (Key-Point 4.20). As explained in [1329], assuming requests are going down from the software to the device, an upper filter receives requests *before* the device it filters receives them. In the same way, a lower filter receives requests *after* the device being filtered receives them. Of course, if the point of view is now from the device sending information to the software, lower filter receives requests *before* the device driver while upper-filter receives them *after*.

In general, driver filters are rarely lower-filters but more upper-filters<sup>5</sup>. Why? Because being a lower filter obliges to manage the interface of the driver that provides the information to the filtered one. In our case, as we interface with `kbdclass.sys`, it means that we should process information coming from `kbdhid.sys` or `i8042prt.sys` drivers directly. As a result, we lose the API provided by `kbdclass.sys` (such as the filtering system given in section 4.1.2.1 — Key-Point 6.7) and we end up having to act in a more basic way as in section 4.1.2.2 — Key-Point 6.8. Note that in the last case, the operation is totally equivalent for an upper filter driver.

Thus, in our case, we will insert our driver — called *GostxBoard* — as an *UpperFilter* in relation to `kbdclass.sys`. This driver uses the same hook technique than `KbdFilter`, that is to say, the one using *ClassService* and presented in section 4.1.2.2 (Key-Point 6.8). Taking into account that our driver is called *GostxBoard*, the *UpperFilter* value is defined as a `REG_MULTISZ` [1330] containing exactly “GostxBoard, `kbdclass`”. The order of the elements matters. The filter drivers are loaded<sup>6</sup> from left to right. This means that `kbdclass` is loaded *after* *GostxBoard* in our case. And this is what we expect since we want to benefit from the `kbdclass.sys` API to get access and replace its `KeyboardClassServiceCallback` routine.

If we would like to use the read dispatch routine to filter the keyboard (Key-Point 6.7), it would also be possible too. In this case, it is necessary to not be below<sup>7</sup> but above `kbdclass.sys`. Indeed, reading operation by the keyboard is done via an IRP. An IRP that is initialized by the raw input thread, in Win32k for the sake of simplicity — Key-Point 4.34. This IRP<sup>8</sup> is engaged on a physical device created by `kbdclass.sys`. In fact, what we are trying to filter is the read IRP which is handled by `kbdclass.sys` and which will be completed by the whole device driver stack above (ultimately by the raw input thread). Therefore, in this case, we create a registry value, still called *UpperFilter* but in append mode in order to add our driver at the end of the existing chain. As we reuse the filtering technology as in the `Ctrl2Caps` project, we propose to call it in the same way here, for the sake of illustration. Thus, we have in *UpperFilter* value: “GostxBoard, `kbdclass`, `Ctrl2Caps`”. For the sake of readability, an illustration of what we discussed is given in Figure 6.3.

This system of hierarchy between filters is a bit archaic and lacks flexibility (especially compared to a more modern system such as mini-filter drivers [1279] for the file system, for instance). Since Windows 10 version 1903, it is possible to order the different filters involving in filtering a device [1144]. This new system does not allow to do more than the old system presented here. It is just a better organized way of doing the same operations, with a bit more complexity but offering a real ease of interpretation.

---

<sup>5</sup>For instance, both `Kbfiltr` [1305] or `Ctrl2Caps` [1132] projects are referenced as *UpperFilter* drivers. `VirtualBox` is also another example: <https://www.virtualbox.org/svn/vbox/trunk/src/VBox/Additions/WINNT/Mouse/NT5/VBoxMouse.inf>.

<sup>6</sup>Only the name of the filter driver is provided. How does the system know the path where the file of the driver belongs? Simply by checking the content of the service list previously setup by the `.inf` file and where all services belong in the registry of Windows. With the *service name* provided in the *UpperFilter* key, Windows knows which driver to load by opening the corresponding key. In this key, a value named *ImagePath* gives the full path to the driver’s file to execute.

<sup>7</sup>Still with the point of the device sending information to its device driver (`kbdclass.sys`) and ultimately to user-mode software, from bottom to top.

<sup>8</sup>This IRP is *different* from the one used to read from the USB keyboard device. Indeed, `kbdclass.sys` does not read from the keyboard device with an IRP. More directly, the stream of data is supplied by `i8042prt.sys` or `kbdhid.sys` through `KeyboardClassServiceCallback` routine it exports to these drivers (Key-Points 4.5 and 4.26). In the case of `i8042prt`, keystrokes are retrieved through interruptions (Key-Point 4.5). But in the case of `kbdhid.sys` driver, the last (re)-engages by itself an IRP (Key-Point 4.24 and Figure 4.57) to process the USB keyboard data — this follows requirements from the USB documentation (Chapter 4, section 4.1 and more directly Key-Point 4.10). But this IRP is out of scope for `Ctrl2Caps.sys` driver since this one aims to manage all keyboards and not only USB/HID ones.

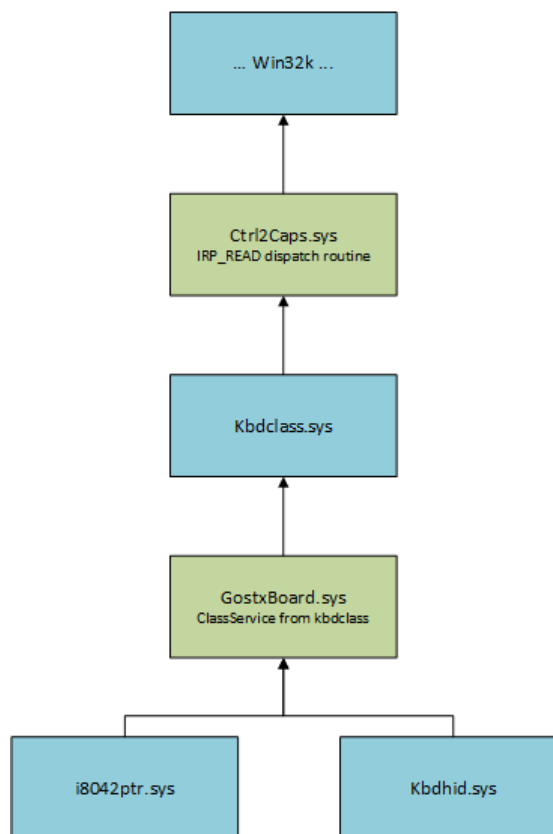


Figure 6.3: Filtering architecture as given where the *UpperFilter* value for keyboard is defined with "GostxBoard, kbdclass, Ctrl2Caps".

The interception of GostxBoard is lower than Ctrl2Caps. Notwithstanding the fact that it is recorded below kbdclass.sys (and by extension below Ctrl2Caps), it is also the essence of the hook procedure described in section 4.1.2.2 (Key-Point 6.8). In practice, we usurp the communication mechanism between specific keyboard device drivers and kbdclass.sys. We are the new intermediary transporting the information to kbdclass.sys from i8042ptr.sys or from kbdhid.sys.

We can check our setup and the architecture of the drivers to know which driver is filtering at which level. To proceed, it may be a good idea to use the DeviceTree software (version 2.30) from *Open Systems Resources Inc.* (OSR) company [1331]. This software lists the entire PnP enumeration tree of device objects, including relationships among objects (filters included) and all the device's reporting PnP characteristics. It is presented with an interface that gives a tree structure by driver where the objects and devices are attached to it. In Figure 6.4, we have detailed the different elements with which we interact on our virtual machine (which always has two keyboards, one PS/2 (keyboard 0) and the other USB/HID (keyboard 1)). The two first drivers are our two filters drivers. They both have created unnamed device filters for both keyboard devices.

Next, in Figure 6.4, details about the view of USB devices (HidUsb). One of these USB devices corresponds to the keyboard (`\Device\KeyboardClass1`). It is first driven by kbdhid.sys which is linked to GostxBoard. This one is attached to `\Device\KeyboardClass1` (PDO created by kbdclass.sys) filtered finally by Ctrl2Caps.sys. We find the same logic when we explore i8042ptr.sys and kbdhid.sys drivers, both displaying the same filtering architecture with first GostxBoard and then Ctrl2Caps at the end of the line.

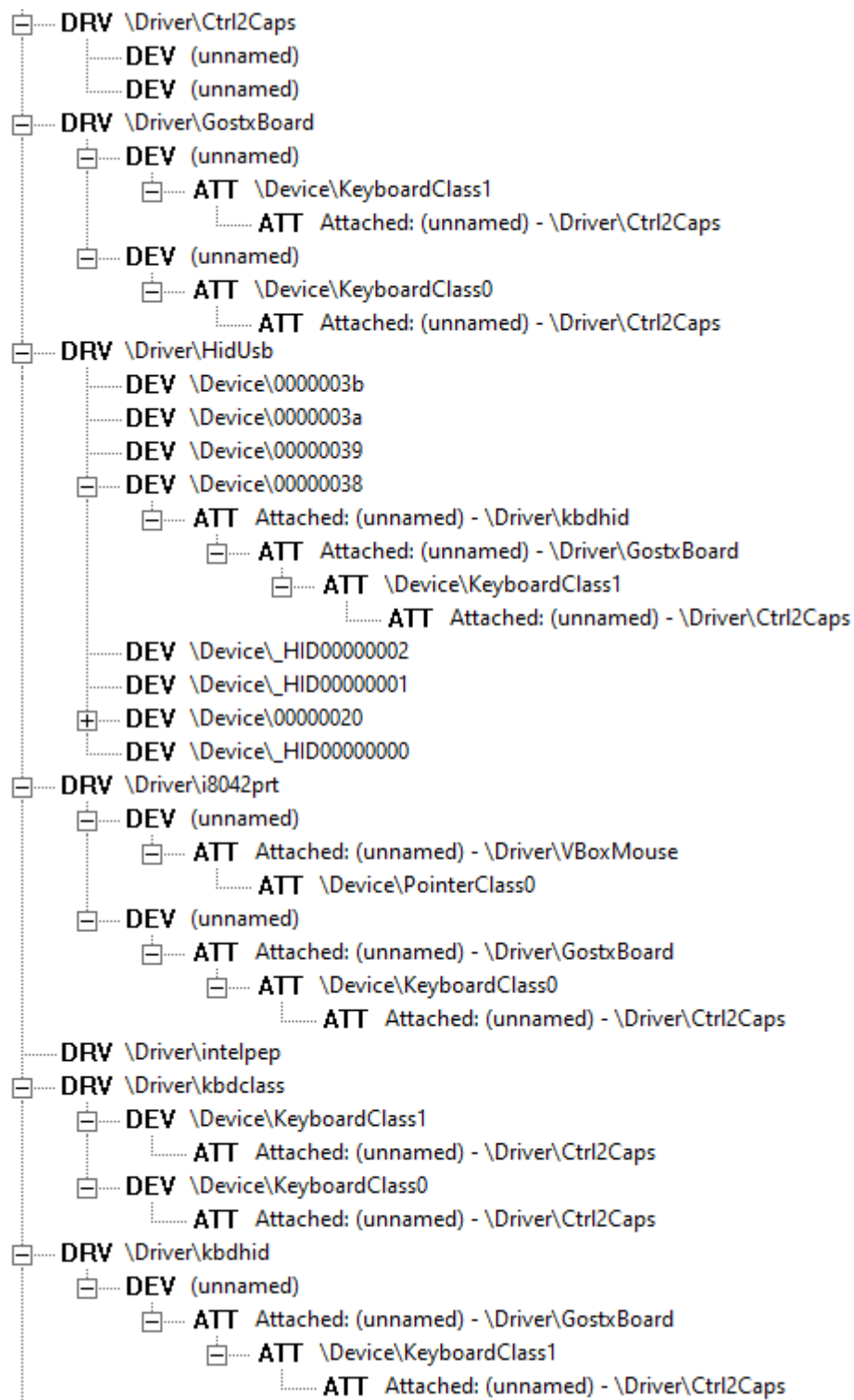


Figure 6.4: View from OSR’s DeviceTree software on our virtual machine, configured with two keyboards (PS/2 & USB/HID) running Ctrl2Caps and GostxBoard drivers.

#### 4.1.4 Conclusion about our solution's keystrokes management

##### Key Point 6.11:

☞ GostxBoard has precedence over Ctr2Caps driver, meaning that we are as close as possible to keyboard device's keystrokes while being filtered at kbdclass.sys level.

If we test these two drivers together, a simple trace showing data processed by both of them is enough to confirm that GostxBoard has precedence over Ctr2Caps. The trace is given in Code 6.8 whenever a key is pressed. In this case, GostxBoard is notified first and then Ctr2Caps is notified. Note that Ctr2Caps is notified about a reset of the read IRP after a read operation occurred. This is done in order to rearm the read IRP operation. The same happens when the key is released on the keyboard. In this case, the key is seen as up. Note that modifying keystroke content from GostxBoard impacts Ctr2Caps that sees the key's content updated and not the original one. This confirms that kernel-mode keylogger solutions using the architecture of the Ctr2Caps project (Key-Point 5.10) would be fooled by our technique.

```

1 [1]1084.1754:-06:46:00.047 [GostxBoard][K] \Device\KeyboardPort0 - 0x0010 - 0x0000 (Down).
2 [1]1084.1754:-06:46:00.047 [Ctrl2Caps][i] CTRL2CAPS -> ScanCode: 0x0010 - 0x0000 (Down).
3 [0]0204.0270:-06:46:00.048 [Ctrl2Caps][i] CTRL2CAPS read IRP request
4 [1]0000.0000:-06:46:00.120 [GostxBoard][K] \Device\KeyboardPort0 - 0x0010 - 0x0001 (Up).
5 [1]0000.0000:-06:46:00.121 [Ctrl2Caps][i] CTRL2CAPS -> ScanCode: 0x0010 - 0x0001 (Up).
6 [1]0204.0270:-06:46:00.121 [Ctrl2Caps][i] CTRL2CAPS read IRP request

```

Code 6.8: "Trace from our protection drivers when a key is pressed and the two drivers implementing each a different interception technique are running."

Knowing the technique presented in section 4.1.2.2 allows to intercept lower than the one presented of section 4.1.2.1, we prefer to use the lowest level one (Key-Point 5.12). It provides a result in line with what is expected for a simple and generic implementation (management of an IOCTL emitted by kbdclass.sys and for which a routine pointer is replaced).



## 4.2 Cipherng scan codes for keystrokes

### 4.2.1 Problem of direct cipherng on the window's system message queue

#### Key Point 6.12:

- ☞ Most protection solutions using encryption do it with a parallel communication channel.
- ☞ Why not just use the existing communication channel managed by the raw input thread?
  - ☞ The experiment shows that transmitting encrypted keystrokes on this channel is impossible.
  - ☞ The RIT that translates device's scan codes into its internal scan code (Key-Point 4.43) and if the translation is impossible, the keys are dropped and never forwarded — Key-Point 4.37.
- ☞ This could be an explanation for the use of parallel communication channels by other software vendors.
  - ☞ Security is not enhanced by the use of cryptography (Key-Point 5.26), it could be just a technical convenience.

More than the technical way of modifying the scan codes of keystrokes received by a driver, it matters to see how to set up the cipherng procedure on them. In our state-of-the-art (Chapter 5), it was common to see products claiming to have added military-grade encryption systems (GuardedID — Key-Points 5.29) or more reasonably algorithms such as Blowfish (KeyScrambler — Key-Point 5.26) to secure their exchanges. Of course, these algorithms are used to protect their communication channel and not the one used by Windows. Why not using it directly on Windows instead of creating a new one? The question could be answered with a small experiment.

Let us suppose we are using a cipher system directly on our driver to modify the scan codes from keystrokes. Regardless of the generation, the secure sharing and management of the cipher key that is assumed to be optimal here, we use a powerful and modern encryption algorithm. All else being equal, the system is operational both at the driver level and at the protected application one. When we press any keys on the keyboard, the result is immediate. But generally, nothing happens in the protected software. Why nothing is happening? Is there any technical issue? Is there any implementation flaw? Does Windows have a bug preventing our solution to work?

A few observations allow us to answer this mystery definitively. On the first hand, the driver receives the scan codes emitted by the keyboard. On the other hand, the driver is able to modify the content of these scan codes. But the protected application receives literally nothing (or almost), as if no key had ever been pressed on the keyboard device. The same applies to applications that are not protected. The most logical explanation is that the ciphered codes were lost between the driver and the application. However, a benign scan code modification (for instance exchanging two scan codes, as with Ctrl2Caps project) is very well received by all applications. Why is there any difference?

The answer is in the way Windows handles scan codes. From the moment where the keystrokes are read (Key-Point 4.34) to the moment they are broadcast in the system (Key-Point 29), there is a translation of scan codes to virtual key codes (Key-Point 4.37). This translation uses mechanisms based on `InternalMapVirtualKeyEx` routine (Key-Point 4.45). At the application level, when an invalid scan code is provided, for instance to `MapVirtualKeyEx` function [906], the return value is zero. It means that, internally, Windows does not match the scan code provided with any virtual key code.

When the kernel processes scan code directly, the same conversion routines in the kernel are involved in the conversion. And the conversation starts in particular at the level of scan code normalization with `MapScanCode` routine. This routine aims to normalize the scan code with the set used by Windows and to handle prefix and modifier key state codes with `MapFlexibleKeys`. When these routines are not able to normalize the scan code provided, they return zero. And the kernel does not continue handling a key if one of the normalization routines returns zero, as explained in Chapter 4, section 5.1.4.7, Key-Point 4.37. If we are looking for the source code of Windows XP (Chapter 4, section 5.4.2, Key-Point 4.59), this one confirms this information by returning and

hence stopping the keystroke procedure if `MapScancode` fails.

Thus, the ciphering system implemented deals with a set of scan codes (encoded on two bytes but using one byte for nearly all scan codes in practice) as input and with a larger set of possible codes as output (two bytes), much larger than the initial set. In addition, the distribution of the different codes in the output space is not uniform in the case of the Windows scan code set (one of the two bytes is almost always zero), forcing many codes generated from the cryptographic system to be invalid. This explains why so little codes are ultimately transmitted to applications.

## 4.2.2 Proposed solution

### Key Point 6.13:

- ☞ To use communication channel managed by the raw input thread (and thus be transparent for each application), our ciphering system must produce only acceptable values as output.
  - ✍ More directly, we have to design a cryptosystem able to output only values in the Windows internal scan code set.
  - ✍ While keeping the initial security provided by the cryptosystem.
  - ✍ Several solutions are proposed to illustrate how to efficiently design such a cryptosystem.

From the observation made previously, it is not possible to use encryption directly on the data coming from the keyboard to meet our needs. This may also explain why other solutions use a communication channel of their own and not the one usually used. Nevertheless, if it is not possible to use classical ciphering solution, we can try to adapt our requirements defined for our solution with the constraints coming from the operating system. The problem is that the output space of our ciphering system produces too many invalid codes. The solution is to design a cryptosystem that only outputs valid Windows scan codes. To proceed, we propose two solutions.

### 4.2.2.1 Shuffle solution

#### Key Point 6.14:

- ☞ One solution is to fix the set of input and output values and randomly mixing the connections between these two sets.
  - ✍ This solution requires a minimum number of permutations (even random).
  - ✍ This solution uses a random generator not able to guarantee the security in our case.

One method is to produce a simple cryptosystem whose input and output space are fixed by design as a bijective function. To proceed, we propose to realize a permutation system driven by a secure pseudo-random number generator. That is to say, we will start from a set of given scan codes that will be transformed through a secure encryption mechanism (and then translated into a smaller set of authorized scan codes). This is the encryption mechanism in the middle, using a secret cipher key, which provides the security of our solution. A simplified view is proposed in Figure 6.5.

In practice, we draw up a table of authorized scan codes. All codes that could have an impact on the whole system (sticky keys, system hot keys and so on) are removed from it. The idea here is that any scan code received by the application are not in the list of those intercepted by the operating system to be interpreted as a specific command. In practice, this concerns quite a few keys and especially control keys state (CTRL, ALT, SHIFT). In addition, usually this kind of shortcut commands requires to press several keystrokes at the same time (for instance, CTRL+ALT+DEL) to be efficient. In our case, we manage each key separately. We just need to make sure that the output from our system does not produce such undesirable combinations which

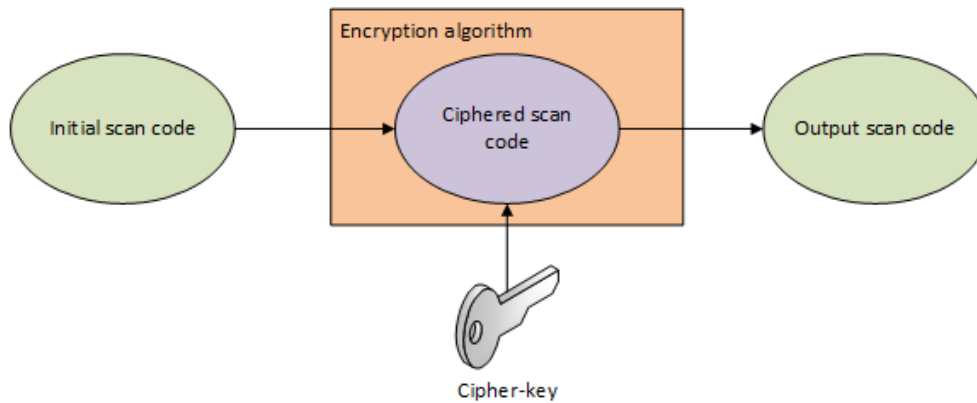


Figure 6.5: Simplified view of the cipher transformation on scan codes used by our solution.

would ruin the user experience. The simple way is to remove the control key codes always present in such keyboard shortcuts. An illustration of kept keys on a general qwerty layout is given in Figure 6.6.

~ `	1 !	2 @	3 #	4 \$	5 %	6 ^	7 &	8 *	9 (	0 )	- _	+ =	Backspace
Tab	Q	W	E	R	T	Y	U	I	O	P	{ [	} ]	 \ _
Caps Lock	A	S	D	F	G	H	J	K	L	:	" '	Enter	
Shift	Z	X	C	V	B	N	M	< ,	> .	? /	Shift		
Ctrl	Win	Alt							Alt	Win	Menu	Ctrl	

Figure 6.6: Keys that are kept as cryptosystem output are in white. Keys which are in grey should be avoided.

As a reminder, there are multiple codes used by keyboards. For PS/2 devices, there are three major scan code sets (table 4.1 in Chapter 4, section 3.2). For USB/HID devices, such table can be manufacturer defined if it is translated by a driver to be compatible with Windows (Key-Point 4.25). In practice, the scan codes used are compliant to the Usage Page as defined in keyboards HID documentation [591]. But Windows handles the scan codes in its own way by using its custom and normalized scan code set (Key-Point 4.37), inspired from the scan code set 1. Supposed to be mostly for retro-compatibility purposes, this design is documented and explained by us in Chapter 4, section 4.2.7 (Key-Point 4.25).

The table of authorized keys is just an array with the scan code used by Windows. Since some codes can have discontinuities in its mapping set (HID does not but PS/2 has some), we crafted a table which maps an index per scan code (Figure 6.7). We call that table an *index table*. That way, we end up with a code whose values follow each other and which is faster and easier to manipulate by our system.

To perform our ciphering, we propose to mix (ie: *shuffle*) the possible output scan codes. More directly, to each input code corresponds a unique and random output code. To proceed, we use an index table that is randomly shuffled — the same way cards in a deck is shuffled. In practice, this is like swapping two items selected randomly and repeating this operation a large number of times. As the values are the same between the index table and the shuffled table (only the order of elements has been changed), the transposition is done by reading the content of the entry in the shuffled table given by the index from the first table selected with the

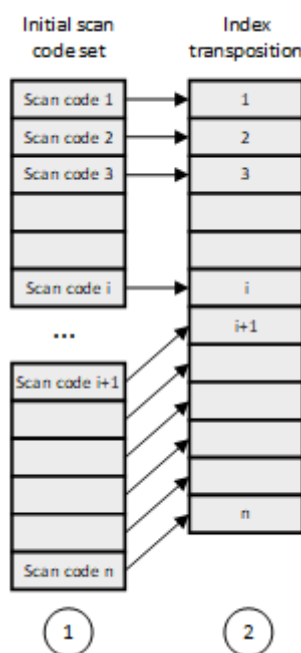


Figure 6.7: Transposition from scan code to index.

corresponding scan code provided. All that is needed is to convert this index into a valid scan code. To achieve that end, we take again the table of correspondence between scan codes and indexes. The same we used at the beginning of the operation. This time, we pass from the index to the scan code, which means we read in the opposite direction with this table.

The procedure can be represented in the form of a diagram. It consists of three numbered steps as shown in Figure 6.8. The first one aims at converting a given scan code into an index. The second one is to obtain the equivalent index in the shuffled table. The third one is to convert this index into a new scan code.

However, this solution is not perfect in terms of security without some improvements. Indeed, we associate a unique output scan code to each input scan code. Of course, the associated code is random. But if it is not regularly updated, it remains possible to break the security by an attacker who is listening keystrokes over a long period of time (or a long text). Supposing there is sufficient statistical material, a frequency analysis [1232, 1231] of the keystrokes pressed would make it possible to guess the association of the main keystrokes and by inference to find those that are less used (Key-Point 5.22). To face this challenge, the table of shuffled indexes must be re-shuffled regularly. The mixing frequency can be variable. Each time a key is pressed in the ideal case, after a few keystrokes if there is any perceptible impact on user experience — which is not what we observed.

The relevant part lies that the security that is induced by operations that depend on the cipher key. Indeed, under the condition that the random number generation algorithm is secure, permutations that shuffle the elements in the index tables cannot be reconstructed unless the cipher key is known. Moreover, the number of permutations performed can be controlled as a random number taken from the pseudo-random generator and added to constant to guarantee a minimum number of permutations (and thus an homogeneous mixing of the index table). If the permutations are performed with a fairly high frequency (ideally each time a key is pressed), it becomes complex for a possible attacker to reconstruct the any random index tables based on the frequency of the keystrokes used in a large text, for instance.

#### 4.2.2.2 Cryptographic chain

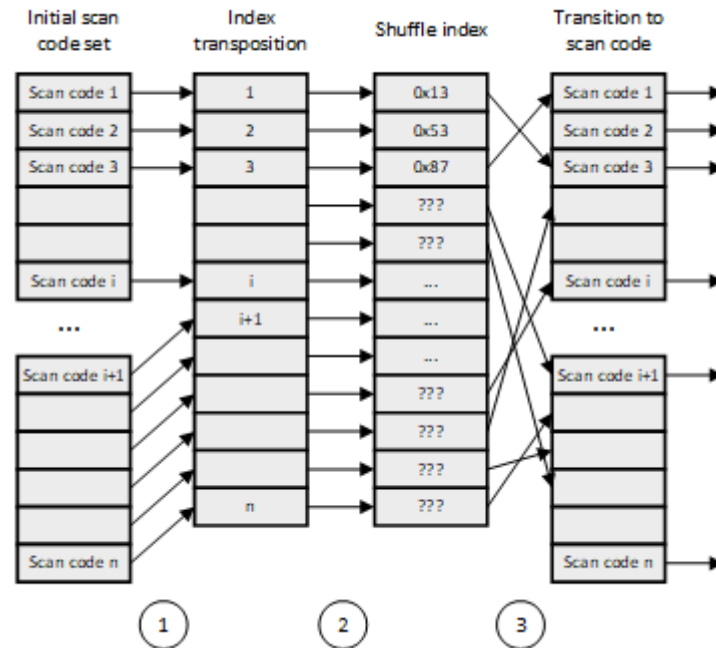


Figure 6.8: Our ciphering system for scan codes based on a random shuffled permutation.

#### Key Point 6.15:

- ☞ One solution is to use a proven cryptographic system over the scan code many times until the output value is in the expected set of values.
  - ☞ The solution is cryptographically secure.
  - ☞ But it is not constant in execution time (which can weaken the solution and ruin the user experience by inducing random wait periods).

It is possible to proceed with another method. The idea is to use a proven cryptographic system to guarantee the security. The problem is that a cryptographic system is designed to produce an output that is as random as possible (in order to not induce a bias that would weaken the security of the cryptographic system) and therefore the output of the algorithm can produce many more values than the output set allows. It is therefore necessary to control the output of the system so that it produces only authorized values while maintaining security. We propose to base our solution on a cryptographic system such as RC4 (although other algorithms such as RC5 or AES can be used without any restriction), which takes as input a buffer of data to be ciphered and a cipher key.

A naive idea would be to directly cipher the content of the scan codes and to check if the output value from the algorithm is an element of the allowed output set or not. If it is, the cryptographic procedure is over. If it is not, we cipher the scan code again (with the same cipher key) until the output value is an element of the allowed output set. Generally speaking, the idea is to ciphering until the output value is within the expected output set (Figure 6.9).

This operation is known in literature as *over-ciphering*, *cascade ciphering*, *multiple encryption*, or *superencipherment*. This can be done in two different ways. On the one hand, we use the cryptographic system in its most classic form with the cipher key and the keystroke which is used as a data to be ciphered. The output is the ciphered scan code and we *over-cipher* it until the output value is in the expected output set. Thus, the ciphering operation is repeated on the code scan (which may then be submitted to several ciphering operations) until the output of the cipher system is in the value set. An illustration of the algorithm is given in Figure 6.10.

The deciphering operation is not very complex to implement assuming that the expected output set is equal

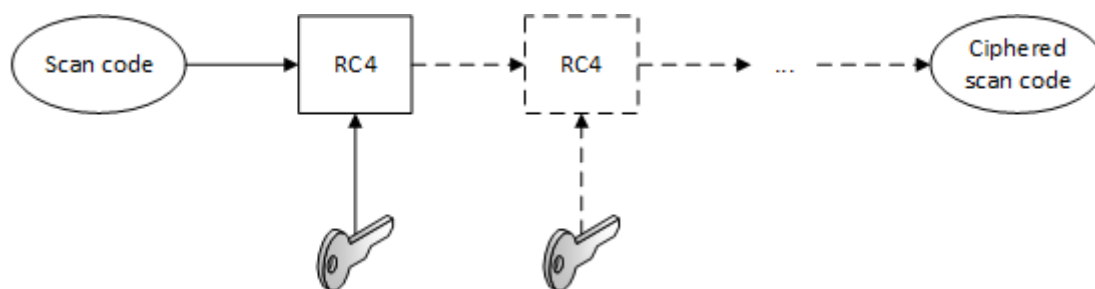


Figure 6.9: Basic principle of the keyboard key cryptographic system with *over-ciphering*.

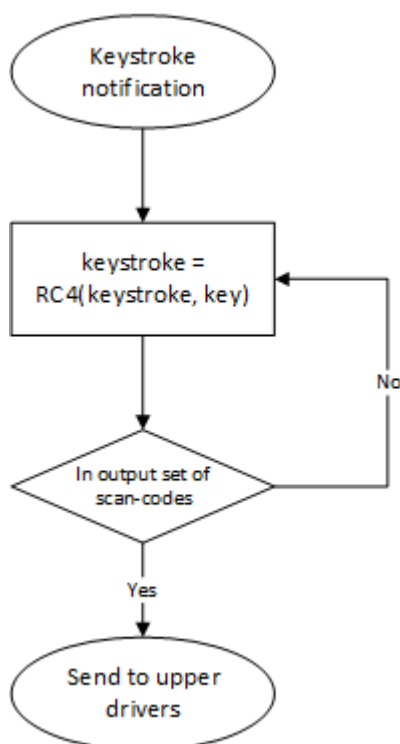


Figure 6.10: Illustration of the algorithm used for the cryptographic chain implementation.

to the input set. More directly, the key scan codes which are produced by our cryptographic system must be relevant for the Windows operating system and do not produce any unexpected actions (with specific shortcuts or keystrokes combinations). Once the ciphred scan code is received by the protected application, this one is deciphered thanks to the cryptographic algorithm (by using the same cipher key) to get a value. At this point, there are two possibilities for the output value. Either the latter gives a value which is conform to the input set (which means a valid keyboard scan code key from the operating system point of view) and the procedure is stopped here (the key is then transmitted to the rest of the protected application). Either the value produced is not in the expected set of values. Then, we apply once again the deciphering procedure on the obtained value to produce another one. From there, the value is rechecked and we continue or not the procedure, if needed. More directly, all intermediate and invalidated values produced during ciphering are deciphered until a valid value is obtained. Such valid value would be the only one that has not required an over-ciphering procedure and therefore the only possible original input value (Figure 6.11).

On the other hand, we can use the cryptographic system as a pseudo-random generator. At this point, the cipher key is also used as input data in the cryptographic system. This system is self-supplied so that the output data produced is the input data for the next iteration of the pseudo-random generator. Thus, we produce in a secure way a stream of random numbers that we will be able to use to protect the scan codes. The

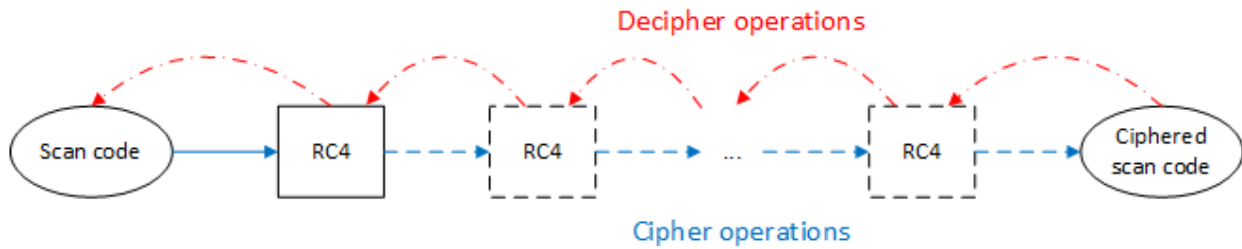


Figure 6.11: Illustration of the cipher and the decipher procedures. Note that intermediate ciphered values are all deciphered until we have a value from the input set.

ciphering of the scan codes is then done by a simple operation of exclusive-or between the bytes of the scan code and those taken out of the pseudo-random generator. Finally, we find here the principle of a simple stream cipher.

Of course, it is possible that the output of the cipher algorithm is not in the expected output set. In this case, a new random number is regenerated and recombined with the previously obtained output value. This new output is then evaluated to know if it belongs to the expected output set. In such a case, the value is transmitted to the operating system. Otherwise, the procedure is repeated, in the same way as with the previous *over-ciphering* method. Illustration of this procedure is given in Figure 6.12.

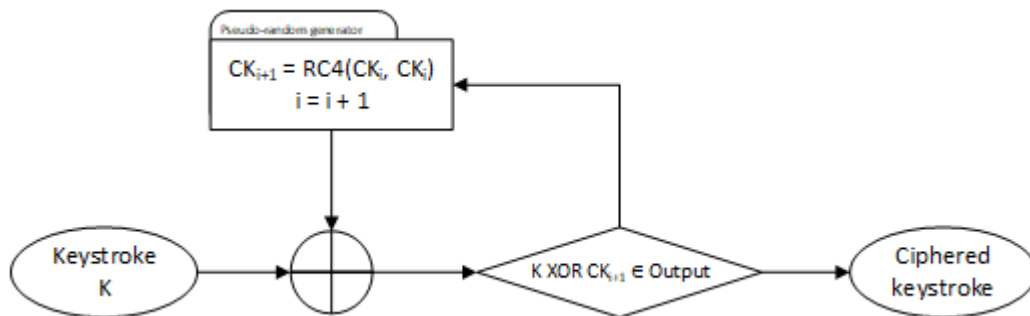


Figure 6.12: Use of the cryptographic system as a pseudo-random generator to protect scan codes.

At the decipher level, the procedure aims to always have a perfect synchronization of the cipher key between the driver and the protected application. Thus, it is possible to produce the same random numbers each time a key is pressed. And it is then possible to re-apply all the exclusive-or operations to restore the initial scan code value. That way, we reuse the same principles mentioned above.

This type of cryptography is similar to the work done on *Shrinking generator* [1332]. This system uses two linear feedback shift registers where the first generates output bits and the second (the control register) rules the first register. In practice, both registers are clocked and if the control register bit is 1, the output from the first register is outputted, otherwise, the output is discarded and a new clock cycle is performed. Such system has an output rate which varies irregularly. Despite its simplicity, the security provided by such a system is quite strong [1333, 1334].

This ciphering system is secure, but it nevertheless entails a certain charge for the system. Indeed, it is quite possible that the algorithm needs to over-cipher many times before obtaining a value that is within the expected output set. In this case, a certain delay may be induced when ciphering and deciphering each keystroke. In practice, the latter is hardly noticeable, but there is a worse case where the release of the output value may be delayed. It is all about the probability of quickly obtaining one of the values of the output set, which is difficult to model with a robust cryptographic system.



### 4.2.2.3 Constant-time implementation solution

#### Key Point 6.16:

- ☞ To solve issues from previous solutions proposed (Key-Points 6.15 and 6.14), we propose a constant time solution that is secure from a cryptographic point of view.
- ✍ In practice, we use a memory mapping representation of all different output scan codes to select them in constant time.

From an operational point of view, both of the cryptographic systems have drawbacks. On the first hand, the shuffle solution must produce random numbers whose value is between two bounds. To proceed, we might use a modulo operation (the remainder of a division) in order to guarantee that the random value does not exceed the maximum bound. Such an operation provides the expected result but it biases the results obtained in terms of the uniformity of the statistical distribution of the output values. And these random values are the central point used in the shuffle solution. On the other hand, the second cryptography solution based on a cryptographic chain is not perfectly deterministic. More directly, the time used to proceed a keystroke can be different for each keystroke. Such difference of timing could be used by an attacker to guess part of the cipher key used in our model.

This last type of attack is close to a cryptographic vulnerability called *timing attack* [1335, 1336]. For short, such attacks allows an attacker who measures the amount of time required to perform cryptographic operations to potentially break cryptosystems. There are some cipher algorithm specially designed to resist to such attacks [1337]. But such algorithms would not be appropriate for our solution, which uses a *cryptographic algorithm* (and any cryptographic algorithm could be used, after all) and not a specific *cryptographic architecture*. Generally, the solution to solve this issue is to implement the algorithm so that it is constant in execution time. *Constant-time* implementations are pieces of code that do not leak secret information through timing analysis [1338]. This is why it is important to design a solution whose execution time can be evaluated as constant [1339, 1334].

Note that the constant time execution issue matters for the shuffle solution in the case where the number of permutation would be selected randomly. This is why the solution proposed here may be suitable to overcome the difficulties of both solutions. The proposed idea is based on the characteristics of the shuffle solution. Indeed, it is not possible (nor desirable) to condition the output of a true cryptographic system to be in a given subset other than with the cryptographic chain solution. The objective is therefore to make the shuffle solution capable of running securely in constant time. A simple way to make this solution constant-time is to force the number of random permutations in the shuffle operation to be constant. Either it is fixed to the implementation or it is randomly initialized<sup>9</sup> once and for all at the initialization of the protection system.

If this solution allows an execution in constant-time, it is also necessary to remove the bias induced on the generation of the random values used in shuffle solution. Technically speaking, we have about sixty authorized codes. By using a translation system already presented, it is possible to return to a continuous interval of values between 0 and 60 (Figure 6.7). A naive solution would be to draw random numbers until one is in the range of the authorized codes. But this would bring us back to using a solution that is not constant time executable. The solution is to use an additional amount of memory to always draw a random value within the range of allowed output values.

For the sake of the demonstration, suppose that our allowed output set consists of 64 values. The few extra values can come from various allowed punctuation marks or be a padding generated randomly (and independently of the cipher key) by inserting duplicates of already existing allowed values selected randomly. The statistical bias induced is negligible in the second case. Our system is driven by generating two random numbers representing the index of values to be exchanged in the shuffle index table. In practice, since we are processing

---

<sup>9</sup>This value is initialized thanks to a purely random value which is not linked to the cipher-key. The entropy source used can come from the operating system or the underlying hardware, such as from the CPU. This random value is added to a minimal number of permutation pre-compiled to enforce the security.

only about sixty values, it is possible to code these random indexes on a single byte. As a result, the random values are between 0 and 255. As we do not want to rely on the random index value anymore, we have to make sure that whatever the index value is, it corresponds to an authorized code. To do this, we propose to copy in memory, four times in a row, the table of allowed values. That way, we have 4 times 64 values, which is equal to 256 possible values, representing all indexes between 0 and 255. This process is given in Figure 6.13.

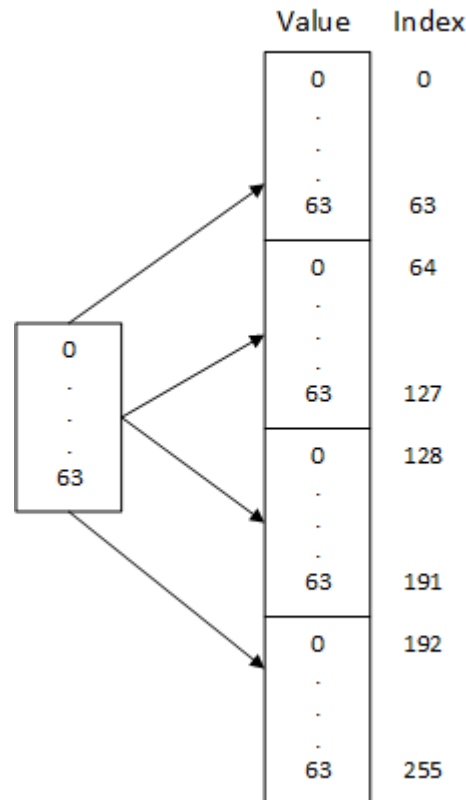


Figure 6.13: Transform the original allowed output set of values to only draw allowed values whatever is the random value index.

In practice, the index is generated randomly and the value is read in the table. Thus and as shown in Figure 6.13, the index can belong to the  $[0, 255]$  range of values but the output value read from the table is always in the  $[0, 64]$  range. Technically speaking, it is enough to insert this mechanism within the shuffle solution (step two in Figure 6.8) to always draw in constant-time random indexes to be exchanged as part of the shuffle operation, without introducing any bias.

#### 4.2.2.4 Conclusion about our solution

It may be interesting to observe here that the proposed solution is built in such a way it solves successively with different approaches the various problems specific to the specifications of our protection solution. Each solution offers a different approach, each with its own advantages and drawbacks. Table 6.1 resumes the different drawbacks that each solution has. Note that from each solution, a particular drawback may appear and that it was possible for us to correct each at the end without giving up the initial objective sought.

Note that it would have been possible to present only the final solution. For pedagogical reasons and to illustrate the process of our approach, it seemed important to us to write the intermediate steps, that is to say the different possible approaches as well as the critical analysis specific to each idea that we implement. For the same objective, there are several solutions where the best one must be kept.

Solution's name	Drawbacks
Shuffle solution	Optionally in constant-time. Shuffle operation based on a biased operation.
Cryptographic chain	Not constant-time operation.
Constant-time implementation	None

Table 6.1: Summary of the drawbacks of the different proposed solutions.

Whatever is the protection method used here, we can guarantee that the output of our protection system will produce only valid codes (which means valid scan codes) that can be correctly transmitted by Windows. That way, because attackers are supposed to ignore our cipher key (which is supposed to be protected), we can provide a strong security. But this is because our cipher key is ignored by the attacker that the security is possible. It is precisely the management of this key that is relevant and presented in the next section.

### 4.2.3 Application level processing

#### Key Point 6.17:

- ☞ We only cipher the scan code value provided by the keyboard device.
  - ☞ The virtual key code is not ciphered by us (since the translation into virtual key code is done by the raw input thread after the action of our driver).
  - ☞ But the translation is made from the scan code ciphered by our driver, which induces the effect of ciphering on the virtual key code.
  - ☞ In most of the cases, there is a one to one correspondence between scan codes and virtual key codes, which means it is generally possible to decipher from a virtual key code (if the cipher key is known).
- ☞ The developer using our protection SDK library is not aware of technical details.
  - ☞ Only calling a deciphering function is required.

From the point of view of the application, our library is responsible for the deciphering operation. To proceed, developers only have to call a single function of our API and to provide it with either the scan code or the virtual key code to decipher in parameter. Technically, this is the scan code that is ciphered but the virtual key code is a translation of the ciphered scan code. Hence, the virtual key code is ciphered indirectly because it is the ciphered scan code that is taken into account by the routines used by the raw input thread. Part of them are supposed to convert the scan code into virtual key code before transmitting it to the applications via the message system (section 5.1.4.7). And since the scan code has been ciphered before, it is converted by the raw input thread into a ciphered virtual key code as well. It means that when the developer needs to convert the ciphered scan code into its clear text version, it is easy.

In the case where we would be about to decipher a virtual key code, the general strategy is to convert the last into scan code and then to decipher this scan code. Technically speaking, it is possible to reverse translation from virtual key code to scan code. The simplest procedure to proceed would be to establish a correspondence table between scan codes and virtual key codes. However, as explained in section 5.2.5, such a correspondence depends on the current keyboard layout. The simplest way to do this would be to use the conversion functions of the Windows API such as `MapVirtualKeyEx` [906] (section 5.2.7). Of course, it is necessary to keep a bijection between virtual key codes and scan codes, which is not always possible (several scan codes can be required to produce a single virtual key code). Hence the importance of properly designing the set of output scan codes of our cryptographic system.

### 4.3 Ciphering keys and exchange procedure

The cryptographic system used is not very sophisticated. Note that it could be replaced with a more powerful one as long as the set of output scan codes remains within the constraints we have defined. From this point, we can take the simplified schema of Figure 6.7 and adapt the cryptographic system to bring another security. The solution we propose here illustrates the principles to be followed.

But more important than the cryptographic system used, the cryptographic key matters. In Chapter 4, section 4.3, we discussed about the unnecessary use of cryptography in some industrial solutions. The industrial solutions (Key-Point 33) do not seem to detail any particular mechanism to protect the secret shared between the protected application and the driver. This is a pity because it is a key point of the implemented security. But is it possible to implement an efficient cipher key management? On the first hand, we recognize that the constraint about securing a cipher key on a single machine also applies to us. This is an almost impossible problem, especially when considering that the attacker can be an administrator<sup>10</sup>. But unlike other solutions, we will try to increase security through various procedures to make this key as secure as possible. Without being perfect, we will try to show the strengths offered by our solution as well as the potential vulnerabilities or unpleasant consequences. The idea is to show that in order to access to the keyboard in an illegitimate way, an attacker has to make such an effort that it is equivalent to uninstalling our solution.

#### 4.3.1 Key exchange procedure

##### Key Point 6.18:

- ☞ The driver is responsible for the generation of cipher keys.
  - ☞ It is more difficult to compromise the code executed from a driver.
  - ☞ The exchange procedure is performed via a mechanism based on IOCTL code.
- ☞ From a security point of view, even if it is possible to intercept the communication during the cipher key exchange, it requires to have at least enough rights to uninstall or attack directly our driver.

In general, cipher keys are more secure if they are not directly present on the machine. The use of a third-party device is a good solution. Otherwise, when it is not possible, it is better to store them in the kernel than in an application. The reason is that the kernel memory is not accessible to user-mode applications. Even if the malicious user is an administrator, it must either exploit a vulnerability present in the system or install a driver by itself (with the need to have it signed for execution) to achieve its goal.

This is why we prefer the cryptographic key to be generated from the kernel. It does not only make it harder to read the key in memory, but it also forces an attacker which would like to modify the code of our key generation system to install a driver. We are on an architecture where the protected application asks the driver, after having generated a cipher key, to provide it. The communication between a protected application and the driver is a documented mechanism when it is based on IOCTL [1340, 1341]. There are three different ways to manage buffers exchanged between an application and a driver: `METHOD_BUFFERED`, `METHOD_IN_DIRECT` or `METHOD_OUT_DIRECT`, and `METHOD_NEITHER` [1133]. The difference between all these methods is not very relevant in our case. What is important to remember is that for some methods intermediate buffers are used and that there are different constraints for each method. It is especially important to pay attention to the security of buffers exchanged between an application (which can be malicious) and the driver (which must know how to protect itself) [1342]. There are tutorials online to have example how to proceed correctly [1343]. Note that some of these tutorials are far from being secure (for instance with [1344])... In our case, we have respected all security recommendations which applied to our system in order to designed a driver [1345, 1346, 1347, 1348].

It is the protected application that asks the driver via our SDK for a cipher key. In practice, it requires to open an access to a named device object [1349] created by the driver. This access can be restricted to adminis-

<sup>10</sup>In this case, and as explained before with many examples [1127, 1128, 1129, 1271], when an attacker is administrator, the game of security is over.

trators or accessible to any user of the system [1350, 1351, 1352]. In general, it can be interesting to create two named device objects. One for administrators and one for all protected applications. The first one is used to control the driver and the protection system in general. As such, it should only be reserved for administrators and optional in the case where security is deployed on client workstations in a company. The second is provided for applications using our SDK to retrieve a cipher key from the driver.

In terms of security, one can wonder about the possibility of intercepting the exchange of cipher keys in memory. Technically speaking, from the point of view of the legitimate application, the mean used to exchange information with the driver does not really allow an intermediary application to eavesdrop. In practice, the exchange is done as if the application would be reading or writing on a device (via `ReadFile` [1353] or `WriteFile` [1354]) or with specific codes (thanks to `DeviceIoControl` function [1274]). In all cases, user-mode communication functions take a handle previously opened on the named device object generated by the driver, so that the communication, once the Windows API is called, is direct. The API function calls an interface function in `ntdll.dll` and via a `syscall`, the code passes the hand to ring 0 which then allows the Windows kernel to transfer the notification to the driver dispatcher routines (in read, write or control code, depending on what is required) registered by the driver itself.

Such an interception would be complex to do for an attacker. There are technically two ways of doing this. On the first hand, in user-mode, the attack aims to act as a debugger and to intercept function calls in memory to interact with the driver. Either we hook the functions used (via a detour mechanism [419] such as *Deviare open source hooking framework*<sup>11</sup>) or we intercept the result directly in memory once the exchange has been performed. This attack only works if it is possible to debug the process to be protected (and applying the case of protected processes [994, 997] allows to fix the problem — Key-Point 6.23), in particular by having a sufficient level of rights (administrator rights when dealing with administrator processes).

On the other hand, it is also possible to act for an attacker at the driver level. There are solutions (`IrpTracker`<sup>12</sup> from OSR or `IRPMon`<sup>13</sup>) to intercept the IRPs transmitted to the drivers. Technically, these solutions act by inserting themselves between the Windows operating system and the driver to be notified. There are two methods to proceed. The first is to register the monitoring driver (via an `.inf` file — Key-Point 6.10) in the call stack of the targeted driver as any other filter driver. It is the cleanest solution in the sense that it is fully documented. But it does not offer a great flexibility because we have to register this monitoring driver for each type of call stack supported, not to mention the fact that some drivers do not belong to any call stack. But this solution is sufficient to filter a single keyboard filter driver. Another solution is to modify the list of dispatch routines that have been registered by a targeted driver. It has been explained in section 4.1.1 (Key-Point 6.5) that driver's entry points is usually used to register dispatch routines in the `DRIVER_OBJECT` [770] structure (as provided in Code 6.2). An illustration is proposed in Figure 6.14.

Monitoring drivers could be tempted to update the content of the driver object to change the list of dispatch routines. To proceed, *IRPMon* uses the undocumented `ObReferenceObjectByName` routine to retrieve the driver object thanks to the object's name<sup>14</sup> and *hook* it<sup>15</sup> by updating its internal dispatch routine with sort of *pass-through* dispatch routines which logs the operation (and arguments provided) before calling the original dispatch routine (in order to keep the original targeted driver's behavior) — Figure 6.15.

This approach is more flexible than the first approach which aims to insert the monitoring driver into the call stack of the targeted driver. But it relies on undocumented mechanisms (access to the `DRIVER_OBJECT` of a driver from its name) in addition to provide a potential instability. Indeed, updating a pointer of routine is not perfectly safe. There is no guarantee that the dispatch routine is not about to be called by a CPU while another

---

<sup>11</sup>More information: <https://www.nektra.com/products/deviare-api-hook-windows/> and <https://github.com/nektra/Deviare2>

<sup>12</sup><https://www.osronline.com/article.cfm\unhbox\voidb{x}\bgroup\let\unhbox\voidb{x}\setbox\@tempboxa\hbox{\global\mathchardef\accent@spacefactor\spacefactor}\let\beginngroup\def{}\endgroup\relax\let\ignorespaces\relax\accent2\egroup\spacefactor\accent@spacefactorarticle=199.htm>

<sup>13</sup><https://github.com/MartinDrab/IRPMon>

<sup>14</sup><https://github.com/MartinDrab/IRPMon/blob/d7340c3af84ebad843b1de54b09562ce3f357ae6/km-shared/utills.c>

<sup>15</sup><https://github.com/MartinDrab/IRPMon/blob/d7340c3af84ebad843b1de54b09562ce3f357ae6/irpmdrv/um-services.c>

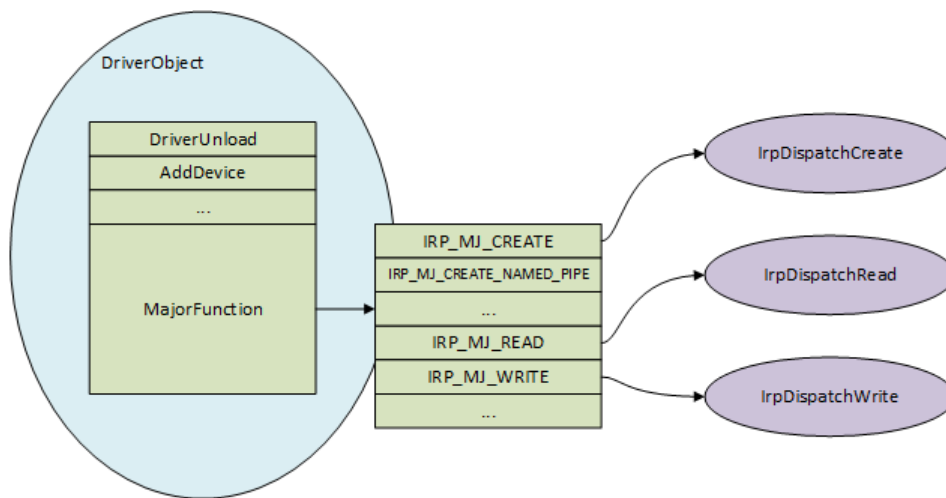


Figure 6.14: Illustration of a regular driver object with dispatch routines set.

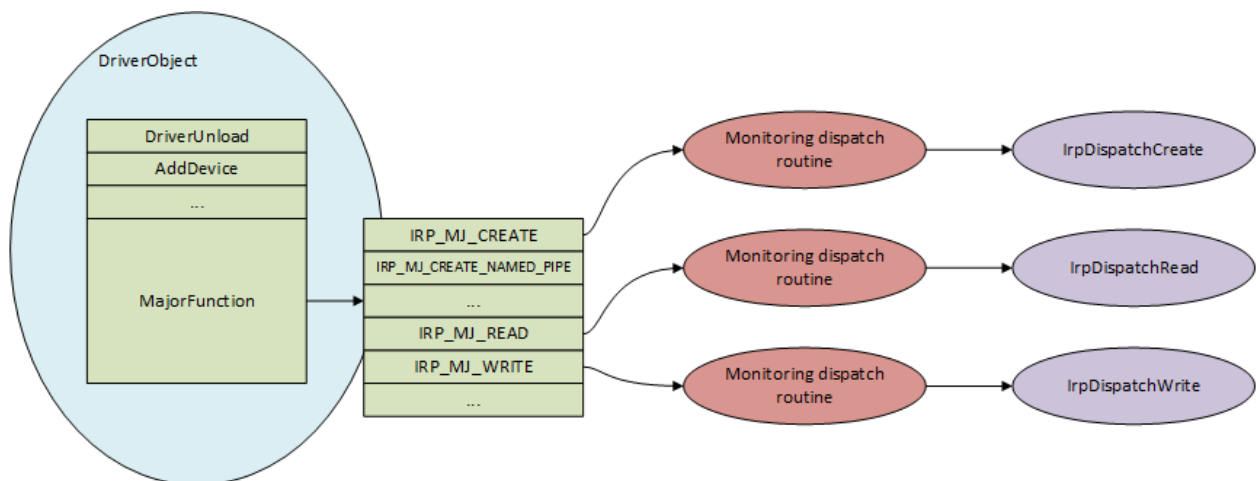


Figure 6.15: Hooked dispatch routines for monitoring purpose by a third party driver (monitoring dispatch routines are in red).

CPU is updating it (even with an atomic operation). There is no real synchronization to avoid dispatch routine to be called while the monitoring procedure is engaged. The probability of a crash remains low but not null.

More directly, this approach requires the use of a driver, which means administrator rights and a signed driver. And at this level of privileges, when it comes to play with another driver's dispatch routines, then it is much easier to simply disable our protection driver. In this last case, the procedure is documented (while hijacking dispatch routines is not), but the result is basically the same: bypassing the security set up.

Thus, apart from the potential debugger attack (explanations in Key-Point 6.23 teach how to limit this one) in user-mode, the rights required to attack the exchange of cipher keys between the driver and the protected process are the same as those required to disable our solution (i.e.: administrator and potentially being a driver). From this point of view, the security of the cipher key in the way it is delivered to the protected process is globally robust.

### 4.3.2 Potential denial of service with cipher key requests

#### Key Point 6.19:

- ☞ A user could try to create a *denial of service* by initiating unjustified connections with our driver.
  - 👉 To defend our solution, we have set up a signature system for executable binaries.
  - 👉 Only signed code can request access to our driver — but even in this case, the solution is still vulnerable.
  - 👉 In fact, nothing can stop a user from making its user's own life miserable.
  - 👉 Our driver would be just another mean to do so. But a complex one compared to other means. Therefore, it is not a real problem.
- ☞ Nevertheless, this signature solution prevents developers to use our solution for malicious purposes.
  - 👉 Indeed, the digital signature associated with the SDK provided to them is unique. It allows them to be identified.

With a communication system open to everyone, it could be tempting for an attacker to monopolize our driver by regularly asking for cipher keys in order to impact the performance of the machine. To address this problem, a double answer must be given. The first one is technical. On the one hand, our SDK requires the application that uses our SDK to be signed [1355, 1356] with a *digital signature* [1357], like a driver [1123]. In addition, the Dll holding our SDK is also signed for a specific SDK developer. That way, copying our SDK from a solution used in software A on a given machine would not allow it to be used on another machine where software B, developed by another company, would be used. Such a design forces the attacker to retrieve the Dll on the machine, which is not impossible but adds a bit of work to find it. Signature checking is performed by our driver before exchanging cipher keys. In addition, the application must be signed in order to use our SDK and the Dll attached to it. The signature is authenticated by a system of certificates that is accredited by us to our SDK's users. In case of a problem (if a developer signs for malicious purposes), we could revoke the certificate, preventing the execution of the malicious product on the customer's machine — not to mention the bad publicity we could make because the certificate is associated with the developer's real identity. It is also possible to develop mechanisms so that an application cannot request a new key for a certain period of time or to force an exclusive access to an application (no cipher key can be requested as long as an existing application is already using a cipher key).

But there is more since it is possible to provide a more definitive answer. The problem we are trying to solve is the case where the attacker is the user. In other words, the attacker is attacking himself. This is not particularly interesting since it is not a security vulnerability. It is just another example that users can make their own lives miserable [1017]. Ironically, with our method of application-driver's communication (Key-Point 6.18), it is the thread calling our driver that, by passing in kernel-mode, that is executing the driver's code. More directly, it means that the induced workload is reported to the attacking application. If one wants to paralyze the system by monopolizing the CPU, it is possible to do it directly via an infinite loop of calculations and by raising the priority of the current thread as high as possible (with `SetThreadPriority` function [1358]).

Finally, we can potentially implement the security features previously presented, not to avoid a denial of service attack, but to ensure that developers using our SDK will not do anything malicious. Such a protection is guaranteed because, if our SDK would be used for malicious purposes, it is possible to identify the responsible developer. In addition, it helps to protect the software executed with our SDK since our drivers knows that it is exchanging a cipher key with an authenticated software, preventing any share with any third-party application.



### 4.3.3 Cipher key and cryptosystem

#### Key Point 6.20:

- ☞ The cipher key can be generated from a random source in the machine by several means.
  - ☞ With the Windows API called *Cryptography API Next Generation* (CNG).
  - ☞ With the CPU itself or a custom source of entropy plugged to the machine.

After presenting how the cipher key has been used and exchanged, we need to see how to generate it. Technically, our cipher key comes from the driver and is only a sequence of random bits in memory. The size of the key can be quite arbitrary, but a minimum of 256 bits seems to be a minimum requirement. In our case, our system is even designed to deal with a much larger key with no maximum limit.

The cipher key is generated for each protected process and different for each instance of the process running if there is more than one. More directly, it means that the generated key does not depend on the process with whom it is shared but only from a random source. Since the cipher key is randomly generated, we need to look at the pseudo-random generator that created it. The generator can be implemented directly as a custom system in our solution or be generated from the Windows API. About the Windows API, we can refer to Cryptography API Next Generation (CNG) [1359] which can be used in kernel mode. This API belongs to ksecdd.sys and cng.sys drivers which both run as a kernel mode export drivers to provide cryptographic services. Random numbers can be generated through `BCryptGenRandom` routine where `BCryptOpenAlgorithmProvider` routine has been previously used to select a pseudo-random generator algorithm supported by Windows. The list of available algorithms supported is given in [1360]. Note that this API could support dedicated devices able to generate hardware secure random numbers.

In the context of the pseudo-random generators from the Windows crypto API, the exported routines do not propose to provide an initialization key. This is not really a problem for us because two solutions are usable. On the one hand, the pseudo-random procedure can be customized to be controlled with an encryption key. On the other hand, it is possible to use another API and algorithms than the ones provided by Microsoft's API. Such choices can also be justified if there are restrictions or trust concerns about using the Microsoft's pseudo-random generator API. In fact, what matters is the use of a cipher key to rule the pseudo random generator or the cryptographic algorithm. In the following parts, we propose to detail the security offered by each of the encryption methods we propose.

### 4.3.4 Security of the shuffle protection

#### Key Point 6.21:

- ☞ In the case of the shuffle solution, updating regularly the cipher key provides more security for the system.

In the case of the shuffle protection (Key-Points 6.14 and 6.16), the cipher key is important because it allows us to control the shuffle of the scan codes index table in a secure and synchronized way between the driver and the protected process. Hence, if this shuffled table is updated periodically, then the key must be derived each time to regenerate a new table. If the protected application and the driver synchronize their update mechanism of the shuffled table (and thus with the cipher key that drives this evolution), protection is ensured and keystrokes will be retrieved correctly by the protection application.

The shuffle operation of table  $T_s$  is called  $S(T, K)$  where  $T$  is a table,  $K$  is the cipher key and  $n$  the number of random swap operations to perform to shuffle the table (note that  $n$  can be a random number). To proceed, we need to drive a pseudo-random generator which takes a seed as input. This pseudo-random generator driven by a seed is called  $G(K)$  where  $K$  is the seed. Technically, the behavior of this generator between two states  $t$  and  $t + 1$  is defined as recursive function such as:  $K_{t+1} = G(K_t)$ . The procedure to shuffle the table  $T_s$  with

the cipher key  $K$  is given as below (algorithm 1):

```

Require:  $T_s, K$ 
 $i \leftarrow 0$ 
 $s_i \leftarrow G(K)$ 
 $n \leftarrow s_i + 1,000$ 
while  $i < n$  do
   $s_{i+1} = G(s_i)$ 
   $s_{i+2} = G(s_{i+1})$ 
   $a \leftarrow T_s[s_{i+1}]$ 
   $b \leftarrow T_s[s_{i+2}]$ 
   $T_s[s_{i+1}] \leftarrow b$ 
   $T_s[s_{i+2}] \leftarrow a$ 
   $i \leftarrow i + 2$ 
end while

```

**Algorithm 1:** Compute the shuffle table  $T_s = S(T, K)$

Let  $K_0$  be the encryption key at the initial time, as generated by the driver from any entropy source. For each step  $t$  where the key is updated, we note it  $K_t$ . Be  $C(M, K)$  a ciphering function (AES-256 [1361] in our case) taking two parameters:  $M$  the message to be ciphered and  $K$  the cipher key. To update the cipher key between two shuffle operations, we proceed as  $K_{t+1} = C(K_t, K_t)$ . It is an efficient way to derive the cipher key in a secure manner [1362]. This way manipulated, the cipher key is updated after each shuffle of the shuffled table. The cipher key request convention by the protected application to the driver includes specifying the step with which to update the key (how many keystrokes before updating). In our case, the step is for every keystroke.

How difficult is it for an attacker listening to the ciphered scan codes to retrieve the original ones? From a practical point of view, everything is done so that each input value has an equiprobability chance of producing an output value. In practice, the input and output set is composed of about sixty keys values (which are not continuous). Thus, the output probability of each value is equal to  $\frac{1}{60}$ . If the index table is changed at each key press, then the output probability of each value becomes independent of the previous value. Thus, it becomes very complicated for the attacker to guess the original value of a keystroke by having only information about the ciphered output value of our system.

## 4.4 Protecting the protected application and its cipher keys

### Key Point 6.22:

- ☞ It is not possible to perfectly protect the cipher key shared between the protected application and our driver if an attacker is an administrator.
- ☞ But this does not mean that nothing should be done.
- ☞ We will make sure that the attacker needs privileges at least equivalent to ours (administrator and driver — Key-Point 6.1) to bypass our security.

The protection we wish to provide is a defense in depth. That is to say for each measure, without being perfect taken individually, the set of security measures represents a major challenge to bypass our security.

### 4.4.1 There is no perfect security but it is possible to do better than nothing

Our protection system does not work if the attacker gains access to the program we claim to protect. We can protect the whole keystrokes transmission chain, if the attacker has access to the program that retrieves them, our defense becomes useless. Generally speaking, this remark could apply to any virtual defense system. Keylogger protection is viscerally broader than keyboard access management for some privileged applications. And since our security against keyloggers relies on ciphering, we need to protect its weakest point: the cipher key shared between the protected application and the driver.

The fact of ciphering the keys on the Windows communication channel leads us, in a way, to wonder about the relevance of our solution. Indeed, what would be the difference with the use of a dedicated channel like some of industrial solutions presented (GuardedID or KeyScrambler — Key-Points 5.29 and 5.26)? As explained earlier, using a private communication channel provides enough security that it is not necessary to use cryptographic systems on the private channel. More directly, bypassing the first security provided by the private channel allows to easily bypass the one provided by the use of ciphering.

As we use the usual communication channel of the keyboard for keystrokes, we need to protect them. And unlike the KeyScrambler and GuardedID projects where ciphering is superfluous, in our case it is necessary because everyone can listen to the data exchanged. We cannot prevent anyone from listening to the keyboard with regular API, but we can try to avoid being understood by illegitimate software. If the attacker cannot directly understand the data stream from the keyboard, it may be interesting to retrieve the cipher key from a legitimate process. That way, as with solutions presented using cryptography, the delicate problem of key management remains. More directly, if an attacker has access to the cipher keys, it is easy to obtain the information from the keyboard and hence bypassing our security.

We have to be clear and concede that it is not possible to perfectly protect our cipher keys on a given system with a purely software solution limited to a single machine. Why? Because a determined attacker could be able, with significant means and time, to get access to the same privileges that our defense system owns. From there, on equal terms, it is possible to fight against our defenses. Of course, this corresponds to the security target we have defined (Key-Point 6.1) and it is likely that such malicious actions may be visible for the user. But this does not mean that we cannot (or we should not) do anything. If it is not possible to provide perfect security, we still have to increase security in such a way that the attacker has to use means at least equivalent to ours. This means requiring administrator rights and driver tools to read our cipher keys or to simply disable our solution.

## 4.4.2 Protection of user-mode application access

### Key Point 6.23:

- ☞ The objective here is to prevent any malicious action that a malware using debugger means could take against a protected process.
  - 👉 The idea is to prevent access to the space of a protected process to prevent cipher keys from being read or modified by a third party.
  - 👉 We try to create a kind of *protection-bubble* (*containerization*) on the protected process.
  - 👉 In practice, this means preventing mechanisms used for Dll injections purposes.
- ☞ Three possible methods are presented in the following sub-sections:
  - 👉 *Protected Process* are present since Windows Vista for this purpose.
  - 👉 It is possible to filter access to any process from a driver thanks to `ObRegisterCallbacks` routine API.
  - 👉 Monitoring from kernel-mode which executable file is mapped to each process (marginal solution).

### 4.4.2.1 What does user-mode application protection mean?

Interacting with a SDK gives us some advantages over competitors' solutions. The first one is to allow to document the constraints carried by the use of our protection system. Immediately, protected processes must not be accessible to other processes. That is to say, it must neither be possible for another process to modify the code in memory, nor it is possible to read the data manipulated by the protected process. *De facto*, this excludes debuggers, like in SpyShelter solution. Is it a real problem to limit debuggers' access? In the case of a work environment where safety is a priority, the answer is no. After all, our solution being a SDK, it remains possible for developers to debug during the development of their secure application using our library. At the end of their development, they only have to activate a specific function from our API that will ensure that when the last is connecting with the driver (to request the cipher key), the security of the application will be guaranteed by our system.

First of all, we need to define what we intend to defend ourselves against. Generally speaking, anything that can voluntarily and legitimately interfere or spy on the process being protected. That is to say the mechanisms of Dll injection [1172] and everything related to the art of debugging. More generally, it is about managing the access to processes by handling process security and access rights [1363]. Nevertheless, it will not be possible to defend against a vulnerability that is already present in the application and that allows it to be manipulated remotely (injection of malicious code via shared and insufficiently secured memory, data manipulation, and so on). It is the developers' responsibility not to implement vulnerabilities. For all intents and purposes, we remind you that Microsoft also provides security to reduce the potential impact of any vulnerabilities (Device Guard [17] — section 4.2.4.3).

#### 4.4.2.2 Protected Process from Windows

##### Key Point 6.24:

- ☞ *Protected Process* is a mechanism that allows Microsoft to restrict access to certain processes running in Windows.
  - ☞ Present since Windows Vista, it prevents Dll injections, reading and writing to the memory of the protected process.
  - ☞ It is even possible to make these processes *unkillable*, although this has potentially harmful consequences.
  - ☞ It is used by Microsoft to guarantee the security of its own processes and those of some other software editors (for instance antivirus).
- ☞ Without being perfect, this security is equivalent to outsourcing the protection to Windows.

How to guarantee the security of an application? Since Windows Vista, Microsoft has already answered this question with a solution called *Protected Process* (PP) [994] that has evolved over time. This technology has been presented in section 5.3.2.2 and details about internals can be found in [1364]. Historically for media protections [995], it relies on a special signature<sup>16</sup> from the executable file [1365] to heavily restrict Dlls loading to a subset of code installed with the operating system [996]. That way, it prevents any injection of Dll or debugger manipulation on a process signed with such certificate. Since Windows 8.1 a new mechanism was introduced: *Protected Process Light* (PPL). This technology ensures that the operating system only loads trusted services and processes [1366]. Unlike PP, PPL acts as a type of security boundary introduced with different signing requirements for the main executable. Less restrictions apply to Dlls possibly loaded and a set of signing levels has been introduced to separate different types of protected processes. Internals about how it works have been documented by Alex Ionescu [1367, 1368, 1369]. The application of this type of security, still in use under Windows 10, is dedicated to Anti-Malware type software (ELAM) [997].

Note that since Microsoft uses this type of protection for its own purposes and for third-party software, it means it should not be considered as a bad thing for customers — at least from Microsoft's point of view. For the sake of completeness, it could be mentioned that such technology of controlling process access rights can be used to create *unkillable* processes. It started with *critical process* responsible to crash the kernel (and get a BSOD<sup>17</sup>) when one was killed. This technology was not really documented. Then comes PPL used to restrict access to `PROCESS_TERMINATE` right [1363] such as it would not be possible to kill it anymore. Recently, Windows started a new type of *unkillable* process with a structure field called *DisallowUserTerminate* [1370] in the `EPROCESS` structure [525] which represents a process in kernel memory. Note that Raymond Chen, from Microsoft company, does not have the same opinion about the advantage of providing mechanisms to create *unkillable* processes [1290, 1292]. His main argument is that these processes create more trouble for users than they provide real solutions — solutions to often poorly formulated problems. In a way, this type of security should be reserved for the operating system only. Moreover, these processes are not really *unkillable* since it is always possible to *kill* them by shutting down the machine...

Why not directly use the PP or PPL technology to protect processes using our SDK? After all, it is possible to run processes from a service running in anti-malware mode [997]. But the created child processes will run at the same protection level as the parent service and their binaries must be signed with the same certificate that has been registered via ELAM resource section. On the one hand, this would imply that our solution embeds a dedicated service to execute the protected processes. On the other hand, these created processes would get a potential and unnecessary gain of privileges. While the second point can be technically controlled, the first point artificially complicates our architecture. In addition, it requires SDK users to get in touch with Microsoft to sign their applications correctly, which simply outsources the security solution.

<sup>16</sup>The signature procedure is performed by Microsoft.

<sup>17</sup>Blue screen of death: a kernel panic from Windows which stops the machine from running with a blue screen displayed, providing information (when possible) about the process and error responsible for this crash.

### 4.4.2.3 Filtering access protection from a kernel-mode driver

#### Key Point 6.25:

- ☞ It is possible to implement a system equivalent to protected processes via `ObRegisterCallbacks` routine (kernel-mode API).
  - ✍ In fact, this allows our driver to filter all access requests to the protected process and its threads.
  - ✍ We can edit the access request (by restricting required rights) or simply reject it.
  - ✍ There is a system of *pre and post callbacks* as well as an *altitude* system to organize the priorities between the various functions of all drivers.

From our point of view, it is possible to achieve this security by ourselves. On the one hand it offers us a certain mastery and sovereignty in the proposed solution, but it also allows us to have an all-inclusive solution. To do so, we will rely on our driver. The first protection is to filter any access to our protected process by another process. Thanks to the `ObRegisterCallbacks` routine [1371], it is possible to register one or more callback routines for any thread, process, and desktop handle operations. It is an antimalware procedure used to check and manage access to resources. The registration procedure requires the use of a specific structure called `OB_CALLBACK_REGISTRATION` [1372]. Since we are about to register a callback routine, this raises the question of the execution order with callbacks already registered. The execution order is managed in the same way as the mini-filter drivers [1279] with the same altitude system [1373, 1374]. To make it short, each driver is declared at a given altitude<sup>18</sup>. The higher the altitude is, the sooner the callback is notified. Conversely, the lower the altitude is, the later the callback is notified. The altitudes are divided into sub-groups of categories, reflecting the purpose of the driver which operates at a given altitude (Anti-Virus, Continuous Backup, Compression, Encryption ...).

The parallel with mini-filters continues by the use of two types of callbacks: *pre* and *post* callbacks [692]. In `OB_CALLBACK_REGISTRATION` structure, there is a `OB_OPERATION_REGISTRATION` structure [1372] referencing two pointers for pre and post callback routines. Such routines can be registered thanks to the `ObjectType` field in the last structure for specific operations happening on the system. Such operations concern `PsProcessType` for process handle operations, `PsThreadType` for thread handle operations, and `ExDesktopObjectType`<sup>19</sup> for desktop handle operations. Pre-operation callback [1376] is called by the operating system when a selected operation occurs. The callback is notified *before* the operating system performs the requested operation. Such a way, thanks to the parameters provided to the callback routine, it is possible to read, update<sup>20</sup> or refuse the request. In our case, most of the work is performed at this level. Indeed, we can check which is the requesting process and the requested process. If any process targets one of our protected process, the pre-callback refuses the access to the handle. Technically, refusing access can be performed by removing all desired access rights. Another solution, to try to maintain the stability of legitimate applications badly designed is to remove all undesired rights to keep the same list allowed by *protected processes* (`READ_CONTROL`, `SYNCHRONIZE`, `WRITE_DAC` and `WRITE_OWNER`). The returned handle in this case is not usable for malicious purposes against our solution. But it could lead to unexpected behavior from the requesting application since this one has a handle used with unappropriated rights, which could lead to access denied with some operations.

The case of the post callback [1379] could be another possibility to achieve our goals. The callback is notified by the operating system after the requested operation occurs. This is maybe a good place to cancel an operation if this one is seen as illegitimate for our defense system and return an appropriate `NTSTATUS` value [797] (for instance `STATUS_ACCESS_DENIED`). But the provided parameter provided is a `OB_POST_OPERATION_INFORMATION` structure [1380] whose content is only informational. More directly, it is not possible to modify it, unlike with the pre callback routine. This is why we use the pre callback only in our driver.

<sup>18</sup>A listing [1374] is maintained by Microsoft, appearing on it is free and just requires to write a mail to Microsoft [1375]. It may take Microsoft up to two weeks to process and assign requested altitudes.

<sup>19</sup>Since Windows 10, this value is supported.

<sup>20</sup>We cannot add more rights than those desired by the caller [1377, 1378].

The same security is applied to *thread* objects. The procedure is exactly the same but it only concerns thread objects, especially handling thread access rights. All these operations are performed for both *create* and *duplicate* [1381, 1382] handle operations. Note that actions performed with pre or post callbacks in our case are restricted for safe calls [1383]. For short, it means that operations must not be performed in the callbacks to avoid deadlocks or instability in the system. More directly, operations allowed should be dedicated to memory access, arithmetic operations and manipulating data in the structures provided, which is generally far enough in our context.

#### 4.4.2.4 Miscellaneous protections

##### Key Point 6.26:

- ☞ Thanks to `PsSetLoadImageNotifyRoutine` routine (kernel-mode API), it is possible to filter any executable file mapped into memory.
  - ☞ We use this mechanism to verify the integrity of the digital signatures of applications using our library (Key-Point 6.19).
  - ☞ It can also be used to check that nothing illegal is loaded in the protected process (but it presupposes to know in advance what is legitimately loaded).

Marginally, another security concerns the verification of DLLs that can be loaded in the memory space of the protected process. The idea is taken from the *protected process* that restricts the loading of DLLs that are not properly authenticated by the system. Thanks to the `PsSetLoadImageNotifyRoutine` routine [1384], we can record a specific callback [1385] notified by the operating system when a driver executable file or a user file (for example, a DLL or EXE) is mapped into virtual memory. The notification is part of the memory mapping of an executable file, before its entry point is called. Actions which can be performed are restricted [1383] the same way as object callbacks described just before. But it is possible to read the content of what is mapped in memory. That way, it is possible to check the signature of the loaded image and to ensure that only trusted code can be loaded by the application using our SDK. Such security can be useful to avoid Dll-side-loading attacks [1386, 1387, 1388]. Note that this security allows to detect whenever a protected application is about to be launched. That way, it is possible to protect any process automatically, before it is accessible to any malicious process.

With the protection designed here, it is possible to have a smooth tracking of the accesses that could be attempted on our protected process. Note that without the ability to read, write or execute anything in the protected process from a third-party application, it becomes complicated to access to the cipher key just as it becomes complicated to retrieve the original content from the keyboard. It would be even possible to further track the security of the objects handled by the protected process by managing kernel objects [1389] with *Callback Object* technology [1390] via the `ExCreateCallback` routine [1391].



### 4.4.3 Other means for direct cipher key protection

#### Key Point 6.27:

- ☞ Many means could be used to improve the security of the cipher key in memory.
  - ☞ The first is about using *virtualized environment* means (*hypervisor*) as KGuard (Key-Point 5.23) solution.
  - ☞ Using hardware such as *Trusted Platform Module* (TPM) to protect cipher keys is a promising solution but this technology is prone to vulnerabilities.
  - ☞ The use of *System On a Chip* (SoC) in the future could be a more efficient solution than TPM.
  - ☞ Storing cipher key in non paged memory (memory which never goes on the hard driver) is a minimum to avoid cipher key forensic.

It may be interesting to review some advanced means to protect the cipher key in memory. One interesting point could be to use virtualized environment and we might think about using a hypervisor to handle cipher key. It would allow to restrict access to the memory part where the cipher key belong to only threads which are referenced in the protected process or in the driver. But this solution could be simplified to directly carry the keystroke via our SDK to the hypervisor, reusing the idea of KGuard (Key-Point 5.23) but applied through a SDK to solve parts of its issues. Indeed, this solution is effective for both protecting cipher keys in memory (the hypervisor guarantees that the cipher key can only be touched by the protected process) and keystrokes. In the latter case, it is possible to capture the event whenever a key is pressed at the hardware level even before the information reaches the kernel. At that point, it would be possible, by implementing an event synchronization system, to redirect this information directly to the protected process that has the keyboard focus. This is equivalent to adding a parallel communication channel, but it is at the hypervisor level and hence impossible for the threat to access. Unfortunately, this solution is limited by two aspects. On the first hand, it bypasses the raw input thread and consequently all the related keyboard interactions (Key-Point 4.44) including the keyboard layout (Key-Point 4.43) — even if it would be possible to redo the translation in the protection library. On the other hand, using a dedicated hypervisor on Windows 10 nowadays is not anymore possible without making a balance with the VBS security embedded (Key-Point 5.24). This is may be the main reason why it is not possible today to base our protection on hypervisor solutions.

One more way, it must be mentioned the crypto API key management [1392] and the use of a secure element such as the *Trusted Platform Module* (TPM) [33] to handle the key is a secure way. A *Trusted Platform Module* (TPM) [1393] is a microchip designed to provide basic security-related functions, primarily involving encryption keys [1394]. This technology is part of the *Hardware Security Module* (HSM) which could be available on a system. The TPM is usually installed on the motherboard of a computer, and it communicates with the system by using a hardware bus [1395]. This security is used in Windows 10 [1396] with *BitLocker Drive Encryption* for device encryption, in the context of *Credential Guard* [1245], boot procedure and so on [1395]. It is generally used to generate, to store, and to limit the access to cryptographic keys. It is possible to specify whether cipher keys that are created by the TPM can be migrated [1397]. In such a case, the public and private portions of the key can be exposed to other components, software, processes, or users. Otherwise, the private portion of the key is never exposed outside the TPM.

Such features are obviously very interesting since they would allow us to process the cipher key with a great efficiency. Regarding the low quantity of data to proceed, using the TPM to store a cipher key and to proceed cryptographic operations in the chip would be a great security. *A priori*, there is nothing that prevents us from using this technology to manage our cipher keys. But, this technology suffers from two limitations in our case. The first is that it requires the chip to be present on the customer's hardware. This is nowadays generally the case, but this still cannot be taken as a prerequisite. In the absence of the microchip, we need at least one alternative, possibly degraded. The other limitation is that the safety of the TPM has been questioned by different research works [1398, 1399]. There have been several flaws in recent years [1400] and even if some of

them have been corrected [1401], it is still<sup>21</sup> possible [1402] to exploit it. Note that new *System On a Chip* (SoC) called *Microsoft Pluton security processor* is currently under development in collaboration with leading silicon partners AMD, Intel, and Qualcomm Technologies, Inc., and Microsoft [1403, 1404]. This type of CPU chip is precisely designed to counter attacks on the TPM by reading the PCI bus used to transmit the data. Everything will be contained within a single chip designed with maximum security while allowing certain flexibilities for updating purposes.

A default and minimalist security would be to prevent memory pages holding the cipher key to be paged to the disk. Indeed, a page file can be present on Windows to allow the system to remove infrequently accessed modified pages from physical memory to be stored on the hard drive [1405]. This procedure allows the system to use physical memory more efficiently for more frequently accessed pages. Thus, if one of these memory pages contains the cipher key, it may appear in plain-text on the hard disk. But there is a way to counteract this phenomenon without having to remove the paging file.

When working with Pages [1406], it is possible to lock some pages in memory thanks to the `VirtualLock` function [1407]. This function locks the specified region of the process's virtual address space into physical memory. It ensures that subsequent accesses to the region will not incur a page fault. More directly, it prevents the system from swapping the locked pages out to the paging file.

Designed to ensure that critical data is accessible without disk access, in our case, it guarantees that the cipher key content will not be written on the disk. But according to the documentation, locking pages into memory is dangerous because it restricts the system's ability to manage memory. Hence, it reduces the available RAM by maintaining specific pages in physical memory and forcing the system to swap out other critical pages to the paging file. Among these pages, it can include code pages, which would impact the performances of the system. But note that with only a single page locked for protection purposes, the impact is almost zero. Such issue was true for 16-bit or 32-bit system with low quantity of memory available and a high use of locked pages. Locked pages remain in physical memory until the process unlocks them (thanks to `VirtualUnlock` [1408]) or terminates.

Locking page is a mechanism which is only for user-mode applications. For kernel-mode driver, where the cipher key belongs, it is possible to use non-paged memory [1409]. Non-paged memory is guaranteed to not be paged on disk (technically, locked pages can be seen as non-paged memory). At the beginning, it was used to deal with memory at different IRQL, we use this memory to ensure that our cipher key will not be paged to the disk. Interacting with non-paged memory can be done from allocation phase. Thanks to `ZwAllocateVirtualMemory` routine [779], it is possible to allocate memory in the non-paged pool of memory.

#### 4.4.4 Cipher key protection in case of hibernation

##### Key Point 6.28:

- ☞ In Windows, there is a special sleep mode called *hibernation* which is used to restart faster.
  - ☞ In practice, this means storing a memory image of the system in a file called `hiberfil.sys`.
  - ☞ It includes drivers and the memory associated with them ... and therefore potentially our cipher keys.
- ☞ However, it is possible to protect our cipher keys from being saved in clear text on the hard disk.
  - ☞ We can be notified when the system is switched to hibernation and when it wakes up.
  - ☞ These notifications are used to erase the cipher key in memory (just before hibernation) and request a new one at waking time.

Another way to collect the cipher key in memory but stored to the hard drive lies in `hiberfil.sys` file on

<sup>21</sup>A recent demonstration on Twitter by Henri Nurmi from F-Secure company claims it is reproducible: <https://twitter.com/HenriNurmi/status/1334514577204195331>.

Windows systems. Since Windows 8, a *fast startup* mode has been introduced to start a computer in less time than is typically required for a traditional *cold startup*. A *fast startup* is a hybrid combination of a cold startup and a wake-from-hibernation startup [1410].

#### 4.4.4.1 Sleeping states and Modern-Standby

Technically speaking, there are several working and sleeping states in the system [1411]. From S0 which is the nominal system working state [1412] to S5 when the system is shutdown [1413] (and G3 called “*Mechanical Off*” where the system is completely off and consumes no power [1414]), there are different states which describe different situations linked to the fact that the CPU loses power. There are transitions from one state to another and generally, the higher is the sleeping state, the longer it takes to return the computer to the working state S0. The S4 (*fast startup*) is the hibernate state. That is to say, it is the lowest-powered sleeping state and it has the longest wake-up latency. Illustration of the system power states is given in Figure 6.16 extracted from [18].

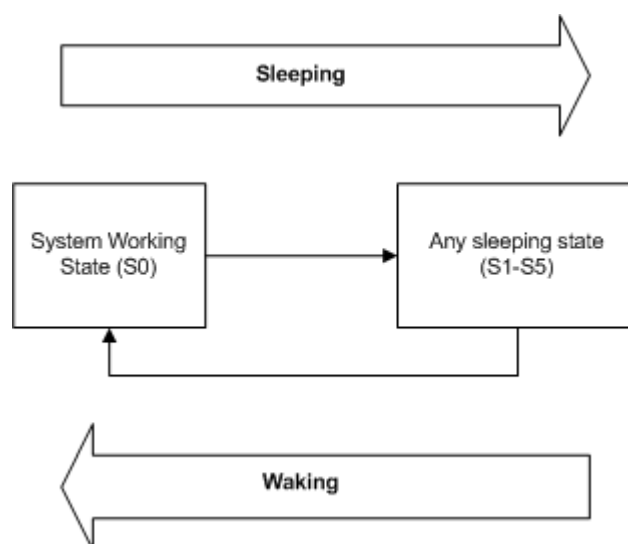


Figure 6.16: Possible system power state transitions, extracting from [18].

Over the history of Windows, the power management model has been improved. Indeed, the old one called “*S3*” is a little bit an outdated standard which is not capable to boot “*instant-on*”. Such feature is nowadays expected from consumers using modern devices. This is why Microsoft introduced “*Modern Standby*” able of leveraging all the capabilities of a modern chipset to be integrated across the breadth of tablets and PCs today [1415]. The Windows 10 Modern Standby system expands the Windows 8.1 *Connected Standby* power model [1416]. The goal of sleeping states evolution is to enhance *instant-on/instant-off* user experience, such as at low power idle states, it enables the system to stay connected to the network. Note that *S3* (with emphasis) corresponds to a standard while S3 is an ACPI power state which corresponds to a low level power-consumption *sleeping-state*.

According to Microsoft’s documentation [1415], the difference between *S3* and *Modern Standby* lies in the path of how it enters and exits low power state. In this state, in both cases, systems may look very similar to systems in the S3 state. That is to say that CPU cores are powered off and memory is in self-refresh. But with *S3* systems, the system is either active or in S3, and nothing else. With *Modern Standby*, the transition from the active to the low power state is a series of steps to lower power consumption. The goal is to keep components powered down<sup>22</sup> when they are not currently used [1417]. With *Modern Standby*, transitions into and out of a lower power state is much quicker than on an S3 system.

<sup>22</sup>There is a notion of *power floor* which refers to the hardware power state in which all devices are idle and inactive, and power consumption is dominated by hardware static leakage.

Microsoft conceptualizes modern sleep as equivalent to traditional S3 sleep, with the added benefit of allowing value-added software activities to run periodically (such as networking devices to allow network notification or managing user input from keyboard device). For the sake of simplicity, the strategy is to maximize low power usage and only waking from the lowest power state when absolutely necessary. That way, it allows software to execute in short period of time for controlled bursts of activity. At lower states (S1-S3), volatile memory is kept refreshed to maintain the system state. More directly, lowest-powered sleep state to support all enabled wake-up devices. That is to say that some components remain powered so the computer can wake from input from the keyboard, LAN, or a USB device [18].

The goal is to wake up instantly when needed, especially when there is real time action required. From Windows 10, this system has been improved to deliver longer battery life by postponing non-critical work and removing unnecessary wake-ups with *Modern Standby*. There are many possibilities to wake from standby in response to certain events (*wake sources*), even if the platform has entered a very low-power idle state [1418, 1419]. All these improvements about standby states [1420] are still using "S-states" but in a better way than with S3 standard.

#### 4.4.4.2 Hyberfile or S4-Sleeping state case management

The S4 state matters in our case. Indeed, since the power consumption is reduced to a minimum and the hardware powers off all devices, the main difference between S4 and S5 is the speed of the system to restart from S4 compared to S5. Indeed, S5 requires to reboot the system while S4 restarts from the hibernate file called hiberfil.sys. Indeed, when the system loses battery or AC power, operating system context is retained in the hibernate file. This file is undocumented but part of its format has been documented by Joachim Metz in 2015 [1421], even if it could have evolved since 2015. When the system goes in S4 sleep mode, a saved image of the Windows kernel and loaded drivers is written (among others) in the hiberfil.sys file.

More directly, in order to perform a fast startup, when Windows is going to S4 sleeping mode, it uses elements of a full shutdown sequence and a prepare-for-hibernation sequence. First, Windows suspends all applications and it logs off all user sessions. At that point, no applications are running but the kernel is loaded and the system session is running. The next step is to send power IRPs to device drivers to prepare them and their devices to enter in hibernation mode. Finally, Windows saves the kernel memory image (including the loaded kernel-mode drivers) in hiberfil.sys and shuts down the computer. Note that, according to Microsoft's documentation [1414], the hibernation file must be large enough to ensure there will be space to save all the contents of physical memory.

And since hibernation is a fast procedure, this write to the disk does not follow standard writing procedure since it avoids part of the traditional file-system filters. The goal is to allow a fast startup which takes significantly less time than a cold startup. This is why, when the memory of the loaded driver is kept on the hiberfil.sys, there is no real notification of drivers used to manage hard-drive's file-system. Subsequently any cipher key stored in kernel memory could be stored in plain-text on the disk in hiberfil.sys. How to avoid such event?

Generally speaking, it is hard to use mini-filter drivers [1279] effectively in this case. Solutions like Bitlocker [1422, 1423] can be useful if and only if the hiberfil.sys file is present on a bitlocker-encrypted disk [1424]. Rather than relying on original (and often undocumented) solutions or third party products (which is just about shifting the problem), it is possible to adopt an original approach here. Technically, it is not possible to cipher the cipher key in memory (because the cipher key of the cipher key would then have to be contained somewhere and that one would be prone to finish in hiberfil.sys). Instead, we propose to simply remove the cipher key from memory when the system is about to go into hibernation. From a technical point of view, our driver is notified through a system set-power IRP (more precisely with an IRP\_MN\_SET\_POWER notification managed through IRP\_MJ\_POWER IRP dispatcher routine) that informs the driver that the computer has update its power state [1410]. Note that is possible to get more information by registering a power-management callback thanks to PoRegisterPowerSettingCallback [1425] routine.

When the notification spawns to inform our driver that the system is about to go in hibernation, we can

remove it from memory with `RtlSecureZeroMemory` routine [1426]. This last routine does *not* make things secure but it just makes them *more* secure [1427]. That is to say, it forces the compiler to set memory to zero and not avoid this operation for optimization purposes. Such situation could happen when zeroing memory before releasing it (since the memory is released, the compiler wonders why wasting time to zero it). Nevertheless, overwriting the cipher key in memory prevents keeping on processing operations at wake-up time with the protected application.

To solve this new problem, it is also possible to be notified, at the application level, to know whenever the system is about to enter in hibernation mode. Applications and services register for power event notifications [1428] by using the `RegisterPowerSettingNotification` function [1429] to be notified via the `WM_POWERBROADCAST` message [1430], which contains the power management event and any associated event-specific data [1431]. Note that such notification can be used to notify any application for a long list of power events (battery capacity, system power source has changed, current monitor's display, primary system monitor state, battery saver state, if the user activity timeout has elapsed with no interaction from the user and so on) [1432].

In the notified events list, `PBT_APMSUSPEND` event [1433] is used to inform that computer is about to enter a suspended state. More directly, we are about to go in hibernation mode. In this case, the notified application has approximately two seconds to handle this notification [1434]. Beyond this time limit, the system may interrupt the application. But it is more than enough time to securely remove the cipher key from memory thanks to `SecureZeroMemory` function [1435].

In the same way that one is notified during hibernation, we are notified during the waking procedure. For example, in the context of a user mode application, always in the context of the `WM_POWERBROADCAST` message, it is the `PBT_APMRESUMESUSPEND` [1436] event that is used to notify that the system is resuming from a low-power state. In this position, it becomes possible to request a new cipher key from the driver, as we did at the start of the application. Once it has been done correctly, we return to a normal situation with a new cipher key holding the stream of received keystrokes. Note also that if a key is pressed before this procedure can be carried out, it is always possible for the protected application, when ciphering a received keystroke, to check whether the buffer of the cipher key is empty (full of zeros) or not. In the case where there would be no key, a request to the driver to get one can be performed.

## 4.5 GostBoard Dll

### Key Point 6.29:

- ☞ This sub-section describes the programming interface (API) offered by our keyboard security library.
  - ☞ We detail the functions used during the initialization and keystroke processing phases of the library.
  - ☞ Our API proposes an optional feature to test the randomness of the cipher key exchanged between the driver and the application.

Since we know how the security is provided by the driver, especially by handling keystrokes and ensuring the security of the protected application, we propose to explain here how the library of our SDK can be interfaced with the software in which it is integrated. Generally speaking, our SDK is composed by a Dll that has been compiled to be loaded by an application wishing to secure its access to keyboards (from a password to full text management with a word processor software, for instance). Of course, there is a documentation that explains how to interface with our library. The latter has been published in open-source<sup>23</sup> from the beginning of the project. Our objective is to ensure that the developer can easily manipulate the API provided by our SDK and that this one does not impact the user experience on the developed software.

<sup>23</sup><https://bitbucket.org/WhiteKernel/gostxboard>

### 4.5.1 Description of the API

With the aim of facilitating the integration of the module on applications, an API has been developed. This API is a new version of the former one presented at DefCon [1304] conference. It takes into account a lot of improvements. But the compatibility remains for a large part.

The API is a set of exported functions used to implement a cipher session with the driver. As the project is open source, it is up to the developer to recompile the library to go further or directly use the compiled DLL provided with our SDK. The following functions are exported by the DLL:

- **GostxBoardInitiateSession**: it ensure the security and smooth functioning of the system. As a check list procedure for checking safety and configuring the defense solution. It checks if the driver is present, if there is a TPM and so on. If the protection of the protected process is not setup by default (through a registry key read by the driver at its initialization phase), the function takes an optional parameter to enable it or not. It also checks that cryptographic functions of the API from the application are correct by initiating a test phase. A test is initiated by sending a `IOCTL_TEST_SESSION` control code to the driver. That way, we are checking if the driver is operational. This function has to be called before any other session control functions.
- **GostxBoardRunSelfEncryptionTest**: tests the cryptographic functions provided by the API (to check that everything is working as expected).
- **GostxBoardStartCipherSession**: starts the cipher session to protect keystrokes by sending `IOCTL_KEY_START` control code to the driver.
- **GostxBoardStopCipherSession**: stops a cipher session by sending `IOCTL_KEY_STOP` control code to the driver.
- **GostxBoardDecipherKeystroke**: allows to decipher a keystroke scan code or virtual key code given in arguments.
- **GostxBoardGetLastReturnedCode**: returns the last return code of the API. This is our internal equivalent to `GetLastError` function [1437] from Microsoft API.
- **GostxBoardGetCodeMessage**: returns a string message associated with a return error code of the API (from `GostxBoardGetLastReturnedCode`). These messages are in English and defined in `defines.common.h` file. this helps the developer to know if a call to an API function failed.
- **PGOSTXBOARD\_VERBOSE\_CALLBACK**: a callback that the client application can register in order to be notified of every action driven by the API. This function is optional and useful if developers do not want to handle keyboard input by themselves but only retrieve from a callback content of the keyboard in clear-text whenever there is something to retrieve.
- **GostxBoardStartMonitorProcessState**: A function provided to create a thread used to notify the protection driver if the protected application has the keyboard focus or not. This one is perfectly optional and depends on developer design choice when using our library.

### 4.5.2 Secure keyboard initialization

As explained previously, to provide security, our library must obtain the cipher key from the driver. This is precisely the objective of the initialization phase. This can be done almost anywhere in the code of the client application, as long as it has been done before retrieving the keyboard keystrokes in a secure way. In practice, we recommend to do this in the entry point of the application or at the initialization of the thread responsible for keyboard processing.

In our API, the initialization is driven through `GostxBoardInitiateSession` function. The description of this function is given in Figure 6.17.



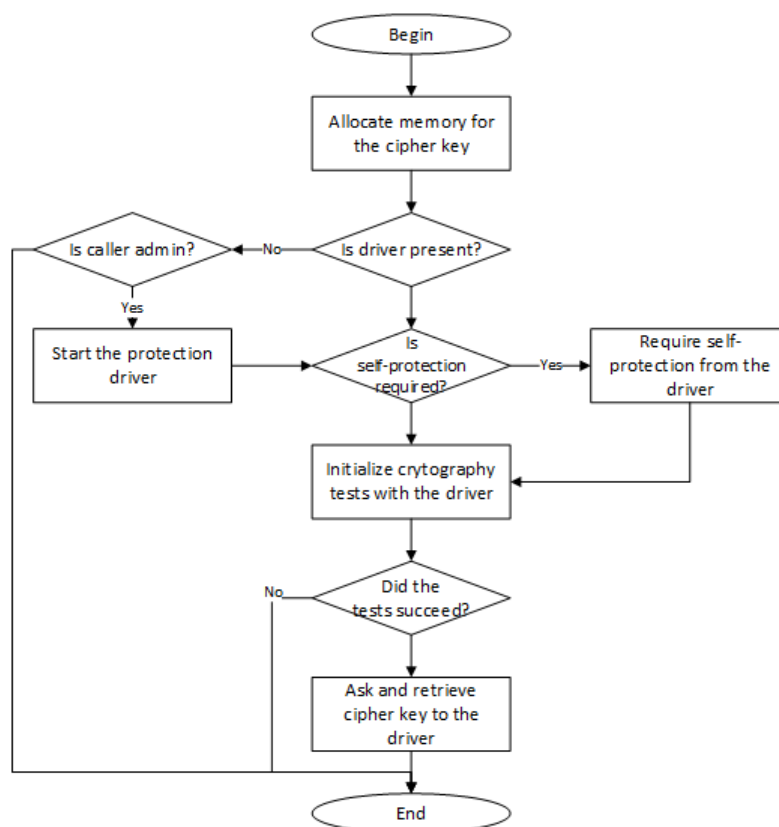


Figure 6.17: Pseudo code illustration of `GostxBoardInitiateSession` function to initiate protection with the driver.

A detailed procedure is given below. In a general way, the goal is to check that everything is correctly functioning before starting the security. And this check is performed in a way that it is the driver's responsibility to prove it is able to protect efficiently. That way, we can detect a malfunction and avoid working in a compromised environment.

1. Allocate a safe memory pool for cipher key structure, as it is possible in user-mode. Memory is reserved, committed, and locked to prevent pagination.
2. Check if the driver is present and running on the system (`IOCTL_TEST_CONNECTION` control code). If it is not and if the application is running with administrator privileges, it tries to start the driver.
3. If it is not already present and if required by the caller, launch the protection of the protected application with the driver (`IOCTL_START_PROTECTION` control code).
4. Check cryptographic modules present with the Dll. Test output of functions with a set of pre-computed input/output (`GostxBoardRunSelfEncryptionTest` function).
5. Request the cipher key from the driver:
  - (a) Allocate safe memory for input and output buffers of the IOCTL communication operation.
  - (b) Initiates the input structure with:
    - i. The size of the cipher key.
    - ii. A pointer to a pre-allocated memory large enough to store the cipher key.
    - iii. Optionally, an array of scan codes supported in the white list (to reduce or extend the authorized scan codes out from our cipher algorithm).
  - (c) Send the request to the driver.



- (d) Retrieve the response and check that everything is correct.
6. Return success if all operations succeeded.

### 4.5.3 Keyboard access management

#### Key Point 6.30:

- ☞ To manage key decryption, two strategies are possible:
  - ☞ The first one aims at using ciphering regardless of whether the protected application has the keyboard focus or not. Efficient but not very compatible with the user experience.
  - ☞ The second one aims to activate the protection according to the keyboard focus by the application. More flexible for the user, detecting keyboard focus must be carefully implemented to be efficient.

#### 4.5.3.1 Interface keyboard with our SDK

#### Key Point 6.31:

- ☞ In the case where the protection depends on the keyboard focus, our protection library must be able to know if the protected application has the focus or not.
  - ☞ Several technical strategies are discussed to know which one would fit the best to our need.
  - ☞ A message management approach with `WM_KILLFOCUS` and `WM_SETFOCUS` in a dedicated thread is privileged by us.
- ☞ The goal here is to notify the driver that the protected application has acquired (or lost) focus.
  - ☞ This information cannot be retrieved (in a documented way) from kernel-mode since this is a user-mode property (GUI).

There are two ways to consider the security induced on the keyboard by the protected application. Is the application critical enough so that when it is used the user is not supposed to do anything else or does the user need to be able to switch between the secure application and others? In the first case, this is easy to deal with since as soon as the application needs to enter text, security is activated and all keystrokes are ciphered. The security applies until the application validates the fact that it has received the secure text (and thus deactivates the protection that has been started, possibly to give the hand back to another application). It has also the advantage for the developer to have the two functions `GostxBoardStartCipherSession` and `GostxBoardStopCipherSession` capable of activating and deactivating secure keyboard input. Only the developer knows when to start this feature (for instance, when a window with text pops up) and when to stop it (for example, when the input is completed and validated via a button by the user).

The disadvantage of this strategy is that if the user switches to another window or another application, the engaged protection continues. And the new window selected by the user, that now gets the keyboard focus, then receives ciphered keystrokes that it cannot correctly interpret because it does not have access to the cipher key. This can be an effect sought by developers the same way it can be considered as an annoying feature. But, to be efficient and not annoying, this kind of configuration should require that the window handling the text is always in the foreground, which is not a good idea in practice [1438]. Hence, it could lead to conflict if two protected processes would run at the same time [1439].

Another possibility is to consider that the protected application can be switched with another application. In this case, the protection must cease, at least until the protected application returns to the foreground and it retrieves the keyboard focus back. This leads us to have to deal with the detection of keyboard focus in order to react accordingly. Keyboard focus has been introduced in Chapter 4, section 5.2.3 (Key-Point 4.41) and its

is a complex notion. There are several ways to detect that an application has keyboard focus.

A first solution is to create a dedicate thread looping on a call to `GetFocus` function [862]. But it is not an efficient way of doing it, since this strategy consumes CPU time by calling in an infinite loop a single function. This pooling method is not a great design comparing to an asynchronous notification. After all, `GetFocus` function is internally based on `WM_KILLFOCUS` and `WM_SETFOCUS` (Key-Point 4.41). Creating a thread looking for these messages (Code 4.27) is a much more valid option to be effectively notified as soon as the application loses or gains keyboard focus, without having to monopolize the CPU.

If developers are willing to be informed if a particular window has taken or released the focus, it is possible to handle `WM_ACTIVATE` message [867] to be notified (through *wParam* message's parameter) that the window's application is active or inactive. This message is linked to `GetActiveWindow` [868] function.

These two *polling* methods of managing keyboard focus are the simplest but least efficient. They both result in the creation of a single thread analyzing the messages input queue from the protected application. This is not efficient since it is not a way of reacting quickly. With polling, we are trying to be as close as possible of asynchronous notification solution, but without being one.

It is nevertheless possible to have solutions that are really based on a asynchronous notification system, at the cost of a greater intrusion into the protected application. To accomplish this, two methods are proposed. On the first hand, coming from Zemana AntiLogger (Key-Point 5.27), we can create a thread calling `SetWinEventHook` [1277] function with `EVENT_OBJECT_FOCUS` event [1278]. This function takes a parameter to register a callback [1440] function which is notified when an object (a GUI window in our case) has received the keyboard focus. The callback is notified with a handle to the identified window that receives the keyboard focus. On the other hand, in a more classical way, we can use `SetWindowsHookEx` function [1] (Key-Point 5.3.2) to process `WH_CBT` type of hook [2]. This procedure has been given in Table 4.18 from Chapter 4 section 5.3.2.1. This hook procedure is notified before setting the input focus, among other possibilities. This allows the application to restore the ciphering procedure with the driver even before the protected application receives keyboard focus back. Note that such a hook callback can be local which avoid to inject it in all applications [976].

Among the four proposed solutions, all are valid and all contribute to produce the (almost) same result. It can be left to the developer to choose which interface is preferred to handle the keyboard focus. By default, we tend to prefer the approach based on `WM_KILLFOCUS` and `WM_SETFOCUS` messages. In a dedicated thread waiting for messages (Key-Point 4.40), we are analyzing this two notifications from window messages to react accordingly. In case where the focus of the keyboard is lost, our thread calls the `GostxBoardStopCipherSession` function until the focus is restored where it calls the `GostxBoardStopCipherSession` function.

Of course, it could be convenient for a malware to mess with the keyboard focus to try to compromise our system. Let's consider the case where a malicious application would send focus messages (such as `WM_ACTIVATE`) to activate the security feature from the protected application. In such case, it would activate our ciphering driver and, at the end, the user will see totally inconsistent keys on its screen when using the keyboard. We can suppose that he or she will understand that something is going wrong and that it is time to look for the cause. But even without this user's observation, one can imagine a system that acts in addition to simply relying on messages. Once a keyboard focus notification is received, it is quite possible to use `SetFocus` function to force the application to keep the focus, ensuring that the focus is acquired and defeating transparently a fake message by any third party application. A possible re-entrance problem (new `WM_ACTIVATE` notification due to the call to the `SetFocus` function) can be treated by keeping in memory the state of the notification (first or second time) and avoiding calling `SetFocus` function the second time.

### 4.5.3.2 Discussion about security of the keyboard interface

#### Key Point 6.32:

- ☞ To evaluate our security, we assume that a malware could steal the keyboard focus fast enough to manipulate the driver.
  - ☞ The goal is to send scan codes intended for a protected application to an illegitimate application.
  - ☞ Some keystrokes would be captured and others lost.
  - ☞ Such an attack is probabilistic and must be performed in a loop and independently for each key.
  - ☞ Such an attack is very theoretical and in practice it would require dealing with a whole bunch of user interaction issues to remain relatively stealthy.
- ☞ For the sake of demonstration, we suppose such an attack is possible. Remembering the attack is based on very fast timing, the malicious application can then receive the data stream in two possible forms.
  - ☞ Received keystrokes are cipher-text, the driver did not have time to switch protection off.
  - ☞ Received keystrokes are plain-text, the driver did not have time to switch protection on.
- ☞ In both cases, the protection is operational.
  - ☞ If the keys are in cipher-text, it cannot do anything with them except send them to the protected application that will be able to read them.
  - ☞ If the keys are in plain-text, malware must send keys back to the protected application that is waiting for them in order not to stop it.
  - ☞ And without knowing how to cipher the contents of the key, the protected application will not be able to process the received key correctly, which will betray the presence of malware.
  - ☞ Note that since the attack is probabilistic and the output of the cryptographic system corresponds to valid keystrokes (Key-Point 6.13), the malware has no way to know whether the intercepted keystrokes are plain-text or cipher-text.

Basing the switch button for security on keyboard focus can be dangerous as SpyShelter solution seems to do (Key-Point 5.25). But in our case, it is possible to implement a relatively safe solution. Two explanations are possible:

- On the one hand, because we do not base keyboard access on a list of legitimate or illegitimate applications, at the opposite of SpyShelter solution (Key-Point 5.25). Our system aims to give access only to applications that directly use our SDK and that are related to the driver. And with the protection of the protected process (Key-Point 6.23), it is not possible to inject a Dll in any protected process to abuse the keyboard focus.
- On the other hand, let us suppose for the sake of the demonstration, that it is possible to take the focus despite the lack of our official SDK. We suppose a malicious application which is now able to get access to the content of the keystroke which was supposed to be sent to the protected application. Two possibilities in this case: either the scan code received is ciphered either it is not. All depends if the driver has been notified soon enough to switch off the cipher protection or not. Always for the sake of facilitating the demonstration<sup>24</sup>, we suppose the scan code can be received in cipher-text or in plain-text.

<sup>24</sup>Such assumption is not as fantastic as it seems. This is just a particular application of "time-of-check is different from time-of-use" (TOCTOU) [1441] applied to the command sent to stop the cipher operation and updating the owner of the keyboard focus.

In this case, the application that was supposed to receive the keystroke did not receive it (at least through windows message system which is impacted by the keyboard focus — Key-Point 29 — unlike the use of `GetAsyncKeyState` function for example — Key-Point 4.50 — where the keystrokes are always ciphered for each application when the protection of the driver is active) because it did not have the keyboard focus. It goes without saying that if the user presses keys on the keyboard and nothing happens on the screen, it is matter of time to discover the trick. To solve this problem, stay stealthy is required in order to not break the user experience (not to say the original behavior of the operating system). That way, the malicious application must resend the received key. And what is send back obviously depends on what it has been received.

For the sake of simplicity, we propose to illustrate the cases where the scan code received by the malicious application stealing the focus is ciphered or in plain-text. The first case where the focus is stolen before the driver has stopped its cipher procedure is illustrated in Figure 6.18. In this case, the scan code received (1) is ciphered (2) before the malicious application steals the focus (3). In such case, after having logged the *ci-phered* received scan code, this one is broadcast to original protected application via `SendInput` (4) before being *deciphered* by the protected application thanks to the SDK (5). That way, the decipher procedure is efficient and the protected application receives the original scan code while the malicious one gets the ciphered scan code.

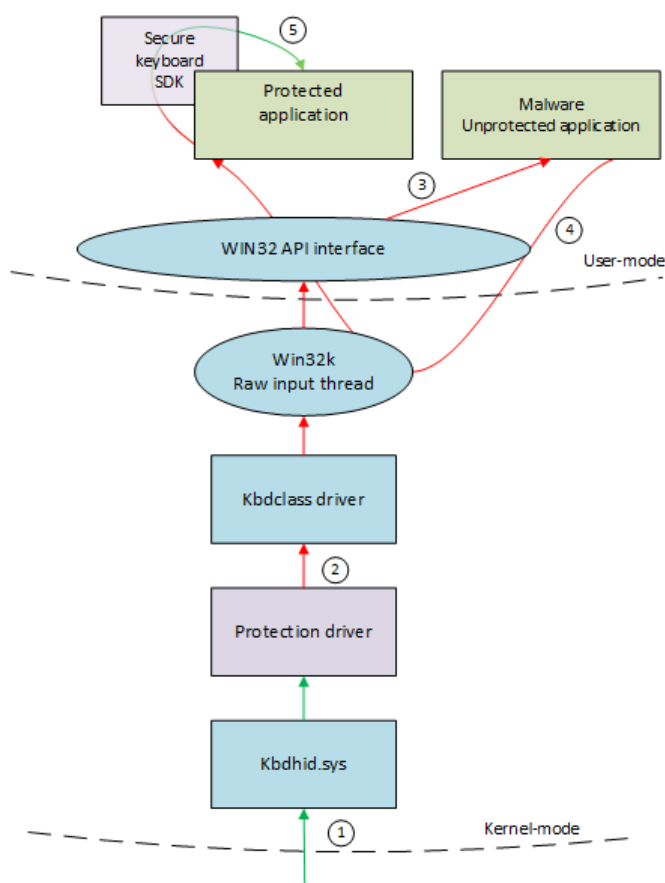


Figure 6.18: Illustration of steal focus when the driver still ciphers.

The second case where the application which has stolen the focus is notified in plain-text scan codes is a bit more delicate. As given in Figure 6.19, a scan code is received from the hardware device (1) but the driver does not apply any cipher procedure since it has already been notified that the protection application does not have the focus anymore (2). In such case, the malicious application receives the plain-text scan code which can be

logged (3). But to not break the original behavior of the protected application, the scan code received must be sent to the original protected application.

This is possible thanks to the `SendInput` [940] function (4). But as explained in Key-Point 4.44, if `SendInput` involves the kernel to handle specific events due to specific keys combinations, it remains that all of this work is performed in the raw input thread. The raw input thread is at the end of the kernel-chain to process keyboard information. However, by design, our driver is much lower in the keyboard device call stack and therefore, it will not see this keystroke simulation, which in any case does not concern it since it handles physical keyboard devices only. The consequence is direct: the simulated key will not be ciphered.

Therefore, even if it would be correctly transmitted to the protected application (via a very clever manipulation to take and to release the keyboard focus in a stealthy way), the protected application would try to decipher the received keystroke via our SDK (even though it is in clear text). It would result in a completely different keystroke than the one entered by the user since the decipher procedure will be applied on a plain-text scan code (5).

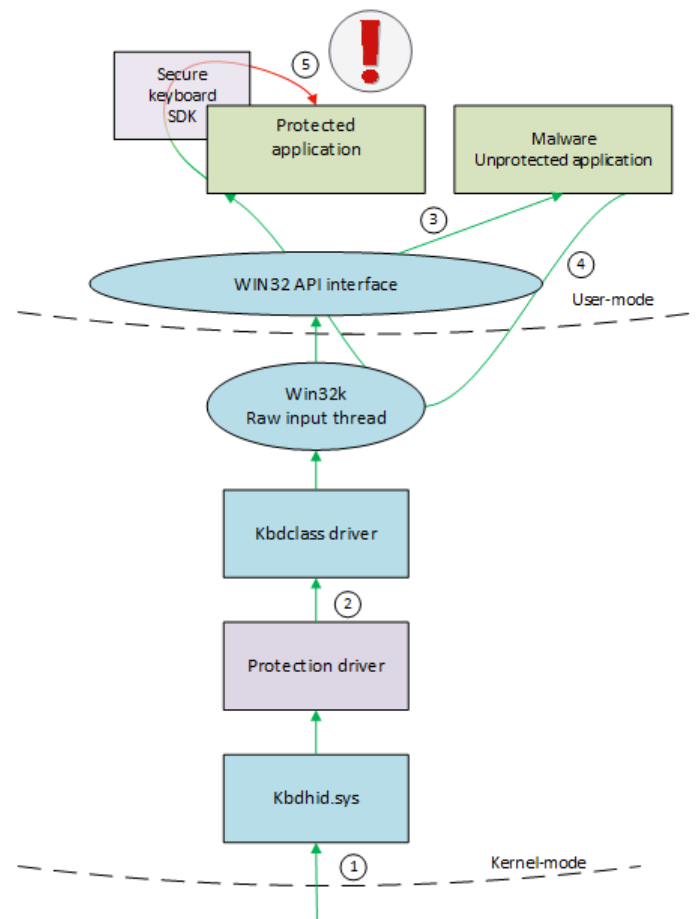


Figure 6.19: Illustration of steal focus with protection stopped.

Even though the user can visually observe the problem, the fact remains that the malicious application has captured some of the keyboard stream in this last case. Even if this is the case, the collected data is not so easily exploitable. Indeed, there is no reliable way for malware to know whether the captured data is a ciphered or an plain-text scan code. Indeed, once it forces to grab the focus (whatever the means), it considers that it has the focus. However, in practice, there is a small delay between the moment the focus changes and the driver stops the protection. In real life, as the user changes windows manually, human interaction time is much longer than this delay, making it imperceptible. But malware software can execute actions much faster in order not to

induce a too long delay on the focus (which would be easily observable by the user since each keystroke would take more than a second to be taken into account by the protected application). Such a short delay induced by programming interaction (and not one from human GUI) when stealing focus and retrieving keystroke allows the abused driver to send ciphered scan codes to an unprotected application.

By the design with our cipher system, outputs correspond to a valid scan codes. That way, it is not possible to differentiate between ciphered scan codes and plain-text scan codes. It means that received keystrokes for an application, despite being meaningless, remains coherent and could have a sense. This obviously complicates the activity of keyloggers which do not really know which scan code keys are correctly received from those that are not. Moreover, in order not to ruin the user experience, in some case where keyboard focus means that the GUI window is directly activated, the focus must be taken and released each time a key is received (otherwise the protected application's window could move, go backward, change color on top bar, and so on...), which makes the keystroke reception operation independent and therefore forces a non-zero probability for each key received to be ciphered.

In the case where ciphered scan codes would be retrieved, the malicious application can only transfer them to the protected application via `SendInput` function. That way, the protected application will be able to decipher it but not the malicious one. In both cases (ciphered or plain-text scan codes), it is complicated for a malicious application to trust the data it can collect. And in the most favorable case for the malware (i.e. plain-text scan codes), it remains to manage the transfer of the plain-text scan codes to the protected application. In that case, the scan code will be misinterpreted by the protected application, showing that there is an issue. Hence, it will not be very complicated to know where the issue comes from (either our driver or a third party software).

#### 4.5.3.3 Conclusion about keyboard interface

In the end, it appears that while it is possible to control the keyboard focus notification, this feature requires the creation of a thread in the protected process to deal with the situation in the most efficient cases. Only the case of the use of `SetWindowsHookEx` function shows that it is possible to dispense with a thread when inserting a local hook, which may have some consequences for developers if they want to avoid the action of this technology within their application.

From a security point of view, this one can potentially be challenged by trying to intercept the focus of the keyboard. Unlike the case of `SpyShelter`'s solution where it is possible to redirect the action to a protected process manipulated through a Dll injection (which is no longer possible with our anti-Dll-injection mechanisms — Key-Point 6.23) to get access to the plain-text scan codes, with our solution, such focus stealing would result in an unreliable access to scan codes. Both ciphered and plain-text scan codes could be received and the original behavior of the protected process is definitely altered.

## 4.6 Self-defense mechanisms

If it is not possible to attack the protected application directly (accesses are filtered by our driver), one can try to attack our driver itself. Attacking a kernel-mode driver requires to be able to run code at the kernel level, unless a vulnerability from the kernel is exploited. At least, administrator privilege is required to launch a driver or to stop our driver. As already explained (Key-Points 6.1 and 6.22), if the attacker is administrator, the game is essentially lost [1129, 1442, 1443, 1444]. But we can nevertheless try to detect or reduce the action of a threat coming from kernel-land.

In the same way than `GuardedID` project (Key-Point 5.29), we propose a self-defense module. What are the threats? On the one hand, there is the possibility of setting up a kernel-mode keylogger that is lower than ours. On the other hand, there is the possibility that one hostile driver is engaging our driver to neutralize it.

### 4.6.1 Handling lower level driver threat

The first threat comes from a driver that could be launched before ours or from a driver inserted in the device driver stack lower than ours. In the first case, such driver is executed before us, which means it is executing

its code before ours. In this case, it is possible for this driver to prevent ours to be loaded. In the second case, it means that the driver inserted in the device call stack of the keyboard will be able to capture the keyboard keystrokes before our defense mechanism would have started. To manage both cases, we need to understand first how the operating system is starting and then the load order of drivers.

#### 4.6.1.1 Operating system boot procedure

##### Key Point 6.33:

- ☞ The security of the Windows start-up is a long sequence of procedures that follow each other and are controlled before being launched.
  - ☞ This starts with the UEFI by launching the operating system kernel and its main drivers after checking their integrity (digital signature and file tampering).
  - ☞ The start-up configuration (and related security) can be controlled through the BCD tool (*boot configuration data*).
  - ☞ The integrity check from UEFI is called *secure boot* while the one of Windows' components is called *trusted boot*.
- ☞ With *Measure Boot*, the firmware logs the boot process in order to send it to a trusted server that can objectively assess that everything has worked as expected.

The Windows boot procedure is quite complex and may depend on how Windows was installed and which security features were enabled. In this part, we will focus on presenting only the main parts and we will refer to the literature for more details. Generally speaking, in modern versions of Windows, the boot procedure can be divided into four parts: the UEFI/BIOS boot, the boot manager initialization, the boot loader initialization and the rest of the system. In former Microsoft's documentation [1445], there are three phases: the UEFI/BIOS boot, the operating system loader and the operating system initialization, as given in Figure 6.20. This one is still more or less relevant, considering that the operating system loader manages both the boot manager and the boot loader initialization.

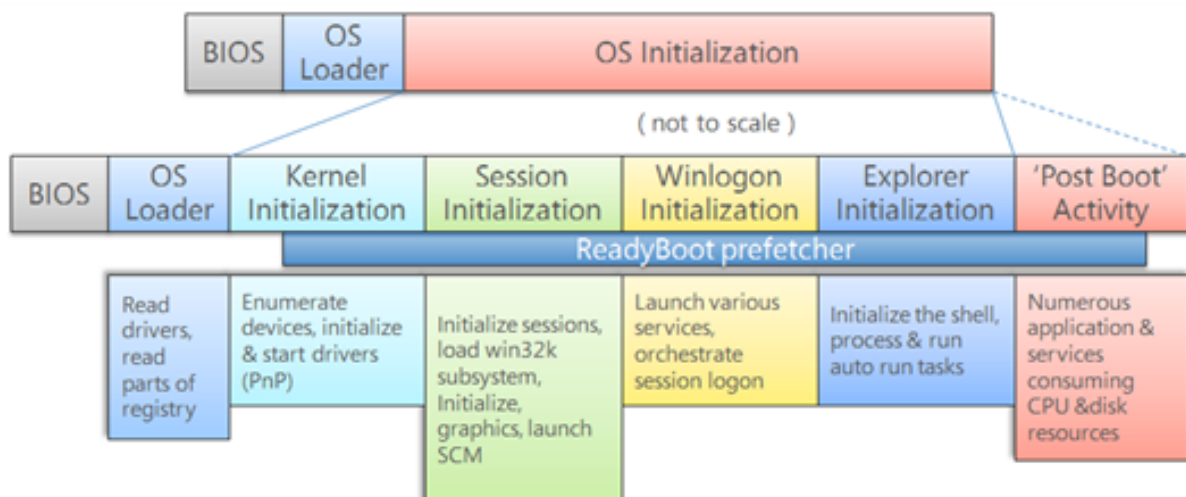


Figure 6.20: Windows boot process consists of several phases (extracted from [19]).

The first part is performed from the UEFI (known as the formerly BIOS initialization phase [1446]). In the phase, the platform firmware identifies and initializes hardware devices. This one is generally embedded on the motherboard and it is responsible for performing power-on self-test (POST) before starting the operating system



itself. At this point, Windows has not booted yet. But the firmware is about to launch it, thanks to the UEFI boot manager. This one loads UEFI device drivers and the Windows boot applications. Technically, Windows is starting thanks to its own UEFI application which is the boot manager. That one is stored in Windows directory, under "`\Windows\Boot\EFI`" directory with first `bootx64.efi` and then `bootmgfw.efi` or `bootmgr.efi` files. These UEFI applications are responsible for loading the Windows boot applications (*OS loader* in `winload.efi`, *Resume loader* in `winresume.efi` and *Memory tester* in `mementest.efi`) [1447].

The Windows boot manager displays the boot menu which lists the boot options the user can select. All these options comes from the *Boot Configuration Data* (BCD) store [1448]. This one is stored on the *EFI System Partition* (ESP) where the Windows boot environment files are located. Technically speaking, the EFI boot volume is on a dedicated FAT32 volume specially formatted to hold all configuration and application files. This is due to UEFI specification which requires to take into account FAT32 formatted partitions [1100] and not directly NTFS partitions. By default, this sensitive partition is unmounted (but a copy is given in `Boot\EFI` directory in Windows) but we can access Windows partitions by two means. The first through *diskpart* [1449]. In *diskpart*, the first operation is to list the different disk on the machine.

```
DISKPART> list disk
```

Disk ###	Status	Size	Free	Dyn	Gpt
Disk 0	Online	1863 GB	1024 KB		*
Disk 1	Online	465 GB	1024 KB		*

Code 6.9: "List all disks on the machine with diskpart."

In our case, the boot sector is on disk 1 which can be selected. From that point, we can list all partitions defined on the disk 1. To find which one is the EFI boot partition, that must be seen as *system* partition and formatted on FAT32 format to be able to manage boot procedure from UEFI.

```
DISKPART> select disk 1
```

Disk 1 is now the selected disk.

```
DISKPART> list partition
```

Partition ###	Type	Size	Offset
Partition 1	Recovery	450 MB	1024 KB
Partition 2	System	99 MB	451 MB
Partition 3	Reserved	16 MB	550 MB
Partition 4	Primary	464 GB	566 MB
Partition 5	Recovery	513 MB	465 GB

```
DISKPART> select partition 2
```

Partition 2 is now the selected partition.

```
DISKPART> detail partition
```

Partition 2  
Type : c12a7328-f81f-11d2-ba4b-00a0c93ec93b  
Hidden : Yes  
Required: No  
Attrib : 0X8000000000000000  
Offset in Bytes: 472907776

Volume ###	Ltr	Label	Fs	Type	Size	Status	Info
* Volume 4			FAT32	Partition	99 MB	Healthy	System

Code 6.10: "List all partitions from disk 1 and get information from the EFI one."

Once we have identified which partition is eligible for the UEFI boot loader, it is possible to mount it by assigning a letter to it. This can be done with the command `"assign letter=b"`. Since explorer does not allow a direct access to this partition by default, it is possible to visit it with a command line launched with administrator privileges. Another way to proceed without diskpart is to use `mountvol` tool [1450]. The last provides a specific option `"/s"` to mount the EFI system partition on the specified drive. For instance, the following code mounts (and removes with `"/d"` option) the EFI system partition on drive "B:".

```
mountvol B: /s
mountvol B: /d
```

Code 6.11: "Mount and remove the EFI system partition with mountvol tool."

Architecture of the EFI partition is given for illustration purposes in Figure 6.21. In this one, the boot directory holds most of the main relevant elements. This is where boot manager executable applications are stored in addition to there database. Note the important list of languages able to manage all version of Windows.

Thanks to the boot option editing tool BCDEdit.exe, it is possible to configure boot options for debugging, testing, and troubleshooting driver on computers running Windows. Of course, editing such options requires administrator privileges. BCD provides firmware-independent boot option interface able to run on any version of Windows since Windows 7 [1448]. Technically, it replaces legacy boot.ini (BIOS) and efinvr.exe (on Itanium). BCD is a container for BCD objects identified by GUIDs or aliases. For instance, each boot application is a BCD object. Each BCD object is a container of BCD elements and each elements contains configuration setting for a boot application. In the case of a simple Windows setup, there is only one entry in the boot manager of Windows, which is by default selected to boot the system. An illustration is provided in Code 6.12 where the first part gives the general configuration of the BCD and the second is the description of each (and here only one) boot loader. Note this is where the Windows' boot loader device path is stored (`osdevice` and `systemroot` elements).

```
Windows Boot Manager
2
3 identifier          {bootmgr}
4 device              partition=\Device\HarddiskVolume5
5 path                \EFI\MICROSOFT\BOOT\BOOTMGFW.EFI
6 description         Windows Boot Manager
7 locale              en-US
8 inherit              {globalsettings}
9 default              {current}
10 resumeobject        {9761eeaa-c696-11e9-8e5d-db8449047b4e}
11 displayorder        {current}
12 toolsdisplayorder   {memdiag}
13 timeout              30
14
15 Windows Boot Loader
16
17 identifier           {current}
18 device               partition=C:
19 path                 \WINDOWS\system32\winload.efi
20 description          Windows 10
21 locale               en-US
22 inherit              {bootloadersettings}
23 recoverysequence     {9761eead-c696-11e9-8e5d-db8449047b4e}
24 displaymessageoverride Recovery
25 recoveryenabled      Yes
26 isolatedcontext     Yes
27 allowedinmemorysettings 0x15000075
28 osdevice             partition=C:
29 systemroot           \WINDOWS
30 resumeobject         {9761eeaa-c696-11e9-8e5d-db8449047b4e}
31 nx                   OptIn
32 bootmenupolicy       Standard
```

Code 6.12: "Display of the BCD configuration on a default machine."

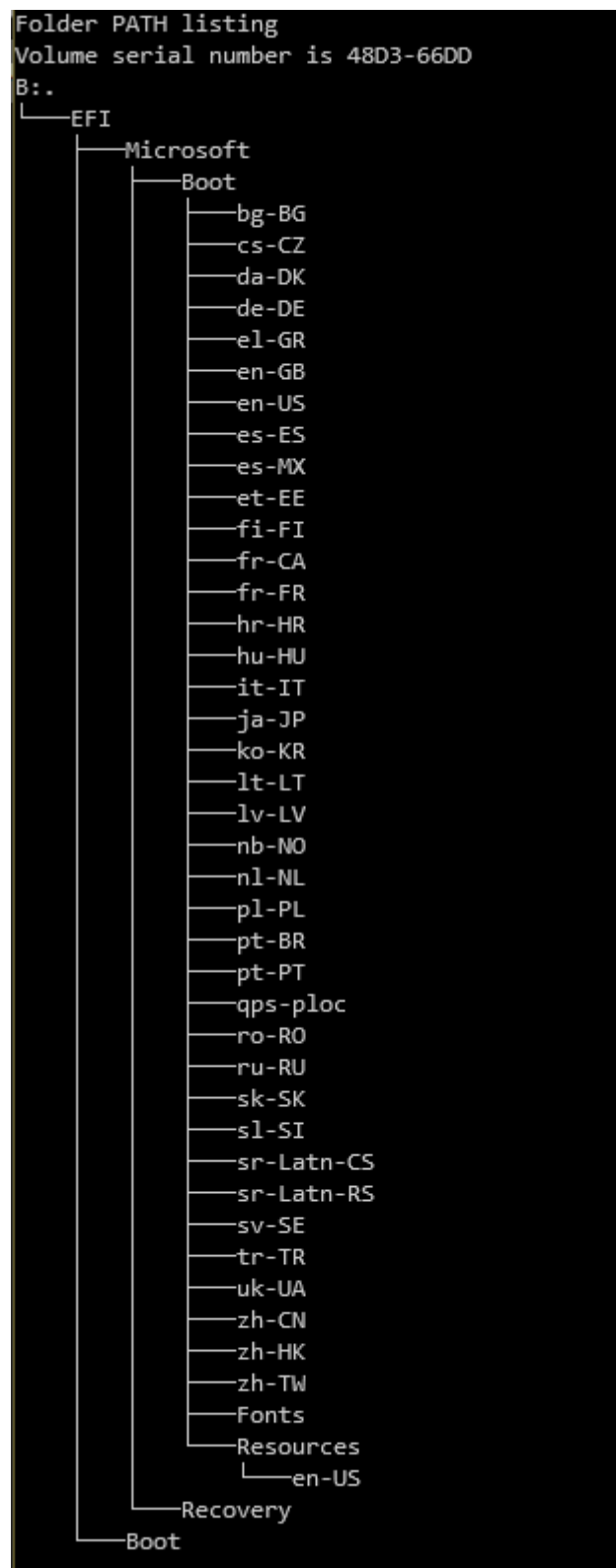


Figure 6.21: Tree architecture of the EFI partition.

The boot manager understands the NTFS file system where Windows operating system belongs. Internally, it has a NTFS parser able to read the disk on which Windows is installed. This allows to locate and to read file

on the disk into memory. The goal of the boot manager when starting Windows kernel is to load all components required for initialization (ntoskrnl.exe, hal.dll, kdcom, etc.), the registry of Windows (by reading the registry file, sometime called a "hive" system [1451] in Microsoft) and all drivers marked as *boot start*. These drivers are generally essential to allow a correct interface between the operating system and the underlying hardware.

This initialization of the operating system starting is partially done by the boot manager. This one is partially responsible for the operating system environment setup. In this case, it performs memory page management initialization, specific architecture initialization [1452] (initialize GDT, IDT and kernel stacks, among other things) and prepare the system to operate in virtual mode<sup>25</sup>. This last operation is performed in the context of the boot loader through `SetVirtualAddressMap` service [1453] in UEFI environment, called after `ExitBootServices` (Key-Point 5.6). Before calling `ExitBootServices`, the boot manager is responsible to load the operating system into memory. In practice, it means mapping the boot loader from disk (winload.efi file, as referenced in Code 6.12) in memory.

Technically speaking, the boot loader is executed in an alternate context than the one of the boot manager. Indeed, the virtual memory space is launched and the context for the kernel has already been initialized. Application winload.efi is still an UEFI application but it is closer to the operating system than any other UEFI application. Boot services and runtime services from UEFI world are still mapped but they are going to hand over drivers of operating system. The goal of this application is to start the kernel, that is to say ntoskrnl.exe. The kernel will then be responsible to setup the operating system environment by loading drivers, creating device nodes, launching smss.exe, followed by the subsystem initialization (thanks to win32k.sys) and the creation of the user session processes (lsass.exe and csrss.exe) and other services. Finally, there is winlogon.exe which is responsible to *welcome* the user login and password. When the user logs in, Windows creates a session for that user, meaning launching explorer.exe and displaying the Windows user's desktop.

Note that the boot procedure can be a bit different, in particular by the consequences of various events or by the presence of particular software. At the level of events that can influence the boot procedure, we can note the management of hibernation. As explained in section 4.4.4, the operating system's context can be restored from the hibernation file (hiberfil.sys). This file holds the state of the physical memory and processors before kernel put the system in S4 power management mode. And all the pages being used by the kernel before hibernation must be restored while avoiding to conflict with the kernel's memory previously mapped.

The case of BitLocker [1423] is also impacting the boot procedure when it is setup and used to protect the boot partition. In this case, the literature [1424] offers an interesting view about how BitLocker is interacting with the rest of the system. Another point is the secure the Windows 10 boot process [19]. In this case, it must be included some security protections which are setup to avoid rootkits to be inserted soon in the boot procedure. From a general point of view, Figure 6.22 (based on [19]) illustrates the modern boot under Windows 10. Note that UEFI motherboard's firmware, the boot manager are both confused in the context of the UEFI.

The secure boot procedure allows computers with UEFI firmware and a *Trusted Platform Module* (TPM — Key-Point 6.27) to be configured to load only trusted operating system boot loaders (*Secure Boot* [1255]). This procedure is usually based on *Intel Trusted Execution Technology* (Intel TXT) [1454, 1455]. In addition, Windows checks the integrity of every component of the start-up procedure before loading it (*Trusted Boot*). ELAM which stands for *Early Launch Anti-Malware* (section 4.6.3) is used to tests all drivers before they load. Finally, the *Measured Boot* (which can be active on *secure boot* and *trusted boot* parts) allows the computer's firmware to log the boot process in order to send it to a trusted server that can objectively assess that everything is correct [19]. In the last case, the check being done on another machine, it ensures that the attacker must potentially to compromise these two machines to bypass the detection.

The case of the *Secure Boot* ensures that no boot loader can be used except the ones authorized by Microsoft or any one that the user would have manually approved its digital signature. In addition, the *Trusted Boot* — which is run once the *Secure Boot* has finished — verifies the digital signature of the Windows 10 kernel and

---

<sup>25</sup>Technically speaking, UEFI applications and drivers are working in real mode memory, meaning that everything is "ring-0" from the point of view of the CPU. Using the virtual mode allows to setup *ring-3* environment for a better application management subsequently by the operating system.

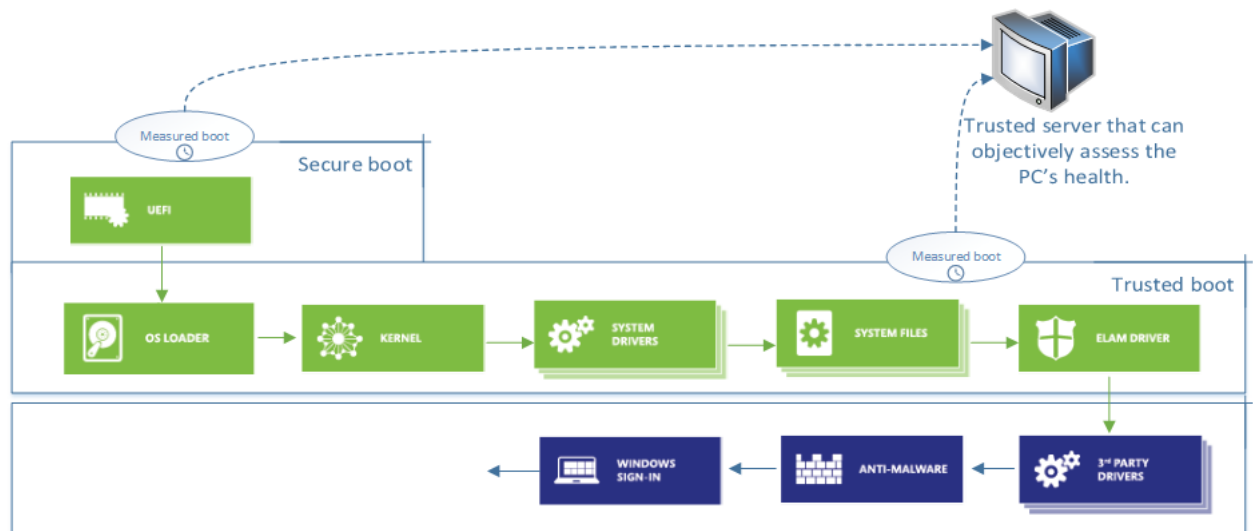


Figure 6.22: Secure Boot and Trusted Boot illustrates the boot process (freely inspired from [19]).

every other component of the Windows start-up process before starting them. Among the component checked, there are the boot drivers, start-up files, and ELAM drivers. In all these cases, these components can only be signed by Microsoft and except for ELAM which are a bit different, they all belongs to Microsoft company. In case of one of these components would have been corrupted, Windows will refuse to load in and in general, will try to repair it with an integrity restoration procedure.

Once all the Microsoft's assets have been correctly checked, loaded and launched, this is the turn of the other drivers on the system. On Figure 6.22, it corresponds to "*third party drivers*". These are drivers potentially written by companies other than Microsoft. They can therefore be malicious. And this is probably where a non-UEFI malware could lie to be loaded the soonest in the system. To understand how it is possible to start at such stage, it matters to understand how it is possible to select drivers' execution order.

### 4.6.1.2 Load order of drivers and protection

#### Key Point 6.34:

- ☞ Understanding the load order of the drivers in Windows allows us to understand how to start as soon as possible to be efficient.
  - ☞ It explains how a potential threat could disrupt our driver too.
  - ☞ The registration of a driver is done in the Windows registry.
- ☞ The load order of drivers is not documented as being perfectly deterministic. But it is possible to have outlines:
  - ☞ All boot drivers (SERVICE\_BOOT\_START) and drivers' dependencies (whatever are their type) are loaded first.
  - ☞ All system start drivers (SERVICE\_SYSTEM\_START) are loaded (with dependencies) followed by auto start ones (SERVICE\_AUTO\_START).
  - ☞ Thereafter, on demand drivers (SERVICE\_DEMAND\_START) are loaded for different purposes.
- ☞ Protection can take place at two levels:
  - ☞ By a notification system at the kernel level (with CmRegisterCallbackEx routine) for each action on the registry.
  - ☞ By checking the content of the keyboard device call stack in real time with a dedicated thread.
- ☞ But our weapons can be turned against us (a driver can use the same filter/notification system to fool us).
- ☞ Generally speaking, there is an important gain for the code that are executed before the others (by controlling the codes executed afterwards).

The load order of drivers [1456] in Windows is quite complex since it does not provide any real guarantees about the real order in which the elements will be loaded, even if some general rules exist. When a driver is installed, that one can try to select where to be set in the driver's loader order. The hard work is performed in the driver's .inf file, more precisely within the *AddService* directive [1457] at driver's installation time. This section can be initialized with relevant values that driver vendors should specify in the *service-install-section*. Specifically, the relevant values are the *StartType*, *BootFlags*, *LoadOrderGroup*, and *Dependencies* entries.

For short, the *StartType* value is the most important since this is the major point used to drive the load order. Technically, the .inf file is about to create a value stored in a key representing the driver in the registry of Windows [1136]. It can take different values for any type of drivers<sup>26</sup> SERVICE\_BOOT\_START (0x0), SERVICE\_SYSTEM\_START (0x1), SERVICE\_AUTO\_START (0x2), SERVICE\_DEMAND\_START (0x3), AND SERVICE\_DISABLED (0x4) [1459]. Lower is the value, higher is the probability to be started soon after the machine has booted. But this simple rule is too simple to not suffer from a lot of exceptions...

Drivers which are registered with SERVICE\_BOOT\_START are required to start with the computer. There is no real order inside boot-start drivers except the relative order which is specified by each driver's *load order group*<sup>27</sup> [1459]. Technically, within each load order group<sup>28</sup>, drivers are generally loaded in random order. This normally results in drivers being loaded based on the order in which the driver was installed [1459].

<sup>26</sup>For specific types of driver, there may be recommended values [1458].

<sup>27</sup>According to official documentation [1459], a complete ordered list of load order groups can be found under the *ServiceGroupOrder* sub-key of the HKLM\System\CurrentControlSet\Control registry key [1139].

<sup>28</sup>Note that if a driver does not specify a load order group, it is loaded after all the other drivers of the same start type that do specify a load order group [1459].

Once all boot drivers are loaded and their entry points executed, the PnP manager configures the rest of the PnP devices and loads their drivers. If any boot driver has created a device object without starting its driver, the system loads it. Technically, the PnP manager walks the device tree [723] and it loads the drivers for the *devnodes* (that is to say those defined in the registry "Enum" tree [1460]) that are not yet started, *regardless* of the drivers' *StartType* values (except if the service is `SERVICE_DISABLED`). Many of these drivers are `SERVICE_DEMAND_START`. The load ordering is based on the physical device hierarchy. The idea is to load the drivers according to the needs of the boot drivers, generally linked to the hardware presence of some devices.

At that point, according to the documentation [1456], all the devices are configured, except devices that are not PnP-enumerable and the descendants of those devices. This is why the PnP manager loads any remaining drivers with `SERVICE_SYSTEM_START` that are not yet loaded, reusing the *LoadOrderGroup* entries for these drivers to launch them. In the end, the service control manager loads drivers of *StartType* `SERVICE_AUTO_START` that are not yet loaded, using *DependOnGroup* and *DependOnServices* values to manage dependencies. More information in [1457]. Finally, a PnP driver that has a start type of `SERVICE_DEMAND_START` value can be loaded by the PnP manager at *run-time* when it finds a device that needs services from this driver.

As another rule which is relevant in our case, it is guaranteed that a driver in the device stack can rely on the fact that any drivers below it are loaded [1456]. For instance, a function driver can be certain that any lower-filter drivers are loaded. However, a driver in the device stack cannot depend on being loaded sequentially after a driver supposed to be lower in the device stack. Indeed, such lower driver might have been loaded previously when another device has been configured. Still about filter drivers, the load order of drivers belonging in the same filter group cannot be predicted. For instance, if a device has three registered upper-filter drivers, those three drivers will all be loaded after the function driver but could be loaded in any order within their upper-filter group. In practice, from our own observations (Key-Point 6.10), it seems that this order relies on the fact that the first defined in the list is the first loaded. But this behavior is not officially documented.

In the end, it appears that the order of loading is something complex and that if there are guidelines (*StartType* and *load order group*), it is necessary to note that the internal order of loading in these guidelines remains random. From the documentation point of view [1456], this random loading is true for filters (upper or lower) defined for a given device. How to proceed under such conditions?

The way to proceed is multiple. On the one hand, it is possible to monitor in real time what is happening on some registry's keys managing the keyboard [1142]. The one managing the *UpperFilter* value (Key-Point 6.10) is particularly valuable. This is possible thanks to the *Filtering Registry Calls* API [1285] using `CmRegisterCallbackEx` routine [1286] (the same API supposed to be used in *GuardedID* solution — Key-Point 5.29). This last routine allows to register an `EX_CALLBACK_FUNCTION` callback routine [1461] at a given altitude<sup>29</sup>. The callback routine registered is notified for any type of registry operation [1462] and a specific parameter is provided to identify the operation (another parameter allows to handle this operation). Notifications happens before and after the targeted operation has been performed — the same way than with pre and post callbacks [692]. In the case of registry filtering, this is the callback routines which are responsible to check which operations must be handled on the registry. Generally, usual implementation design is to use a dedicated routine as a giant switch/case to handle and dispatch all supported operations to dedicated set of routines.

With this registry filter technology, it is possible to monitor if anyone is trying to add, remove or update the content of highly critical registry keys. In particular, the registry key that holds the service and the one that registers *UpperFilters* for the keyboard are top priority. More generally, we aim to check that a driver is not inserted as a dependency of a driver from the keyboard device call stack. Of course, all this technology is valid if and only if the operating system is considered as safe at the time our security solution is installed. Indeed, our defense is only valid at running time. Hence, our solution is powerless if the threat is already present in the system.

Against an already existing solution that would be able to have an asset driver running, there would be very little things to do. We can of course try to make an inventory in the registry of the drivers already installed in the system during the installation of our product and remove all undesirable elements (not to say unknown).

---

<sup>29</sup>The same way than with altitudes [1373, 1374] used in the context of mini-filter drivers [1279].



But, of course, a malicious driver could then deceive our research (by filtering by itself the registry to hide its presence).

In addition to reading the registry, it is also possible to try to build the tree structure between the device objects representing the keyboard (as in the OSR's DeviceTree software given in Figure 6.4). We can even monitor this device tree structures with a dedicated thread to check if an unexplained new object does not appear in the keyboard's device call stack. This could prevent some attacks that would manually "recreate"<sup>30</sup> the Windows API to interface into the keyboard device call stack.

But we have to be honest and recognize that the ability to create a malicious driver and execute it on the victim's machine requires some specific skill (except if it is about slightly modifying the drivers presented in Chapter 4, section 3.2 — Key-Point 5.7). Such skills allow possible implementation of countermeasure mechanisms able to prevent the threat from being detected or removed. The technicity here is the same as the one presented for our security mechanisms. We are fighting on equal terms but not necessarily with the same advantages. There is a question of timing here, knowing which code will run before which one. With the advantage for the code executed first. Indeed, such a malware could prevent or sabotage the installation of our driver. Against a system that is already compromised, there is practically nothing to do.

#### 4.6.2 Handling direct attack on our driver

##### Key Point 6.35:

- ☞ An effective method to neutralize our solution is to use a driver that will attack our solution head-on.
  - ✍ From the neutralization of callback routines used to be notified of system activity to the simple modification of the opcodes of the drivers loaded in memory, the possibilities are endless.
  - ✍ But Microsoft tends to prevent this kind of factious behavior, first on itself (Patchguard — Key-Point 5.8) and maybe in the near future with (Virtualization-Based Security — HVCI) kernel in general with VTL0 and VTL1 (Key-Point 5.24).
- ☞ Attacking a driver directly is a bad idea because it uses undocumented techniques (therefore prone to crash in case of update) but also potentially covered in the future by Microsoft HVCI technology.

Our solution is vulnerable to the fact that a malicious driver could try to neutralize our security directly. Hence, it would be possible, through undocumented (and therefore unstable) mechanisms, to modify or disable all of our callbacks. Of course, such procedure is not documented since they are not legit at all. But doing so would only suppose a minimum of reverse engineering on Windows to disable third-party callback routines, but it is perfectly doable. Generally, callbacks using altitude notification technique are managed through a double linked list [1463, 1464]. Finding the list (and the lock which manages the access to that one) is not a hard task. Removing entries (hence callback routines) from the list is quite common. But this procedures is dangerous since Windows can update at any time its internals (routines and structures) managing what is manipulated here. To avoid this point, a malicious driver could be even more direct.

If it has the ability to be loaded before us, it can register a callback (such as PsSetLoadImageNotifyRoutineEx [1287] routine) to be notified when our driver is loaded in memory. At that time, it has the ability to modify on-the-fly op-codes [419] or data to update our driver's behavior. For instance, to neutralize our solution, it could rewrite entry-point's op-codes. Simple and efficient.

But this last operation supposes that the memory can be executed and written at the same time, which is not advised for driver development [1465], even if it is possible. But an improved version of such a security could naturally come in the future, one day, from Windows operating system. Indeed, with the ability of a

---

<sup>30</sup>Note that such attack, based on totally undocumented structures and routines would be highly unstable over time regardless of its efficiency. But for highly targeted threat provided by high level malware developers, such threat could be likely even if it has never been publicly seen at best of our knowledge.

driver to run in a Hypervisor-protected Code Integrity (HVCI) environment [1466], there are restrictions about modifying on-the-fly running code in a driver [1467]. More directly, it is not possible to run a driver that would not conform to specific requirements (including non executable memory, which means dynamic code in kernel or attempting to directly modify executable system memory) on Windows 10 with HVCI environment [1466]. With the rise of this technology, we might hope that such attack from a driver to another would be restricted. In the worst case, it would be a possibility to ask Microsoft to load our protection driver in VTL1 (Key-Point 5.24) to avoid any corruption by a third-party driver. To the best of our knowledge, we note that today, there is not driver from another company than Microsoft allowed to be executed in VTL1.

Note that whatever happens, this type of attack needs two important requirements. The first one is to be an administrator and the second one is to have a signed driver able to be executed on Windows [1123, 1124]. It means that the identity of the attacker can be known in addition to see the malicious driver banned from Windows kernel's environment by Microsoft. The other possibility is to exploit a vulnerability which allows an access to code execution in kernel mode (as explained in Chapter 4, section 3.2.1 — Key-Point 5.8). Such a vulnerability is far from being common and hard to find. And in a more general way, if these requirements are already owned by any attacker, we can consider that no security solution can resist and all applications are vulnerable in such a case, following our requirements (Key-Point 6.1). This also explains the vanity of trying to protect oneself from it.

### 4.6.3 Preventing threats before they operate

#### Key Point 6.36:

- ☞ *Early Launch AntiMalware* (ELAM) are drivers loaded before all other drivers.
  - ☞ They are used to check the subsequently loaded driver by the system, allowing to refuse to load some of them.
  - ☞ They are part of the *trusted-boot* (Key-Point 6.33) procedure of Windows 10, authenticated by Microsoft.
  - ☞ They are the drivers loaded as soon as possible in the system (Key-Point 6.34), allowing a real security chain.

Our main challenge is to ensure the security of our solution with a correct level of efficiency is to be launched before all possible threats (Key-Point 6.34). There is a specific way to be able to do this, called *Early Launch AntiMalware* (ELAM) [1468]. Starting with Windows 8 [1469], ELAM is a set of drivers that are initialized first and allowed to control the initialization of subsequent boot drivers. They are started before other boot-start drivers and that ensure that subsequent drivers do not contain malware, potentially by not allowing initialization of unknown boot drivers. Of course, the quality and the security (and the trust) of an ELAM driver are really very important, much more than for any usual driver.

### 4.6.3.1 Context of ELAM driver

#### Key Point 6.37:

- ☞ To run ELAM drivers on Windows 10, we must be part of *Microsoft Virus Initiative* (MVI), a highly selective club led by Microsoft.
  - ☞ It prevents anyone to be ELAM and to be launched before any third party driver in the system.
  - ☞ There are few members in MVI and ELAM drivers must follow much stronger requirements than any other driver to be able to be run on Windows 10.

If this technology has a great potential because it guarantees us to start very early when the machine boots, this is one is not really accessible to all developers. Developers of Early Launch Antimalware drivers must be members of the *Microsoft Virus Initiative* (MVI) [1470]. According to Microsoft [1471], this membership ensures that the vendors are active antimalware community participants with a positive industry reputation. Exception could be performed if a company believes and could justify it would need to use a ELAM driver. But such exceptions are at the discretion of Microsoft company, which is free to decide whether or not to grant the right to be loaded in this particular context. In addition to reputation, drivers must be signed by WHQL [1472] to be loaded by Windows<sup>31</sup> and follows strong quality requirements [1473].

An example of a ELAM driver is provided by Microsoft [1474]. This one is given to illustrate the development of such driver, since it is far from being a mass sport. Note that restricting the access to ELAM driver for developers is both a good and a bad thing. The good point is that malware authors would normally not be able to run in such context, preserving the advantage for antivirus products only. The bad point relies in the fact that Microsoft has the full control to chose who is eligible to ELAM program to who is not. This is a great power.

### 4.6.3.2 Installing an ELAM driver

#### Key Point 6.38:

- ☞ Installing an ELAM driver requires additional fields in the .inf file.
  - ☞ It is a boot-driver where the load order group is referenced as "*Early-Launch*".
  - ☞ Additional digital signatures are required to be loaded at boot time (*SignatureAttributes* filed).
  - ☞ A backup file must be recorded in the registry (*BackupPath*) to restore the driver in case of file tempering.

ELAM driver installation is performed as any other driver, usually with .inf file. Since an ELAM driver is loaded early in the boot process of Windows, this one advertises itself as a boot-start driver in its *StartType* value (Key-Point 6.34). This is similar to all other boot-start drivers. To be loaded before other boot-drivers, it advertises its load order group as "*Early-Launch*". All ELAM drivers are not PnP drivers, which means it does not own any devices. Hence, an ELAM driver does not need other registry keys to be launched other than the ones used for registering a regular service [1136]. Last but not least, it is required to include a *Signature-Attributes* section [1475] in the .inf file for the ELAM driver. This last section ensures additional signatures required for ELAM drivers are correctly setup.

As an ELAM driver is loaded very early in the Windows boot process, it is important that no file corruption can occur during boot-time. Technically, if such a situation would occur, it could not be corrected without re-installing Windows. For the sake of resilience, it is therefore required to provide a recovery mechanism. A backup location path is referenced in the *BackupPath* value in the registry key "HKLM\SYSTEM\CurrentControlSet\Control\Early

<sup>31</sup>Such drivers are codes signed by Microsoft, using a special certificate indicating that it is an Early Launch AM Driver.

In this path, a certified copy of the original driver is stored to restore an operational version if the driver file would have been inadvertently corrupted.

Note that an ELAM driver is not started just after the Windows' kernel. Indeed, to load the ELAM driver, some Windows system and hardware driver components must be already present. For instance, it includes the device drivers that need to be initialized before the disk stack has been initialized. These drivers include, among others, the disk stack and volume manager, the file system driver, and bus drivers for the operating system device.

#### 4.6.3.3 Implementation of an ELAM driver

##### Key Point 6.39:

- ☞ The purpose of an ELAM driver is to (quickly) check whether a driver loaded in memory is malicious or not.
  - ☞ It uses a set of APIs provided by the kernel that is exclusive to ELAM drivers (except for filtering the registry).
  - ☞ The verification time allocated to each driver loaded in memory is extremely short.
  - ☞ In practice, this is often limited to a verification of the driver certificate loaded in memory.
- ☞ What does an ELAM driver do to refuse to load a malicious driver?
  - ☞ In the ELAM notification procedure, there is a structure with a *Classification* field initialized by ELAM driver.
  - ☞ Many values are possible [1476], for short: *Unknown*, *Good Image*, *Bad Image* and *Bad Image Boot Critical*.
  - ☞ Impact of such detection depends of the configuration of Windows in the registry.
  - ☞ Usually, in case of a malicious detection, Windows refuses to load the driver and try with the next driver, hoping the lack of the skipped driver would not make the system crash...
- ☞ ELAM drivers are only active during the ELAM procedure.
  - ☞ They are stopped and unloaded at the end of the procedure.
  - ☞ The idea here is that the ELAM driver would check all the drivers before the usual security drivers does.
  - ☞ There is a continuity between the ELAM driver (which is used only to check boot drivers) and the boot security driver ensuring that no malicious driver has been run before it.

The goal of an ELAM driver is to check that loaded boot-drivers are not malware [1477]. To proceed, ELAM driver uses callbacks which provides from PnP manager a description of every boot-start driver and dependent DLLs. From these callbacks, ELAM driver can classify every boot image as a known good binary, known bad binary, or an unknown binary. By default, only bad drivers and associated DLLs are not initialized when they are detected. But this by-default policy can be configured<sup>32</sup>.

We might think that ELAM driver could use the `PsSetLoadImageNotifyRoutineEx` [1287] routine to register a callback and be notified when a boot-driver is loaded. If the idea sounds good, in practice, it is not the case because this callback does not allow to reject the loading of an executable image in memory. As an alternative, we use the `IoRegisterBootDriverCallback` routine [1478] to register a `BOOT_DRIVER_CALLBACK_FUNCTION` callback routine [1479]. This callback provides status updates from Windows to an ELAM driver, including

<sup>32</sup>This configuration is present in the registry under the value "HKLM\System\CurrentControlSet\Control\EarlyLaunch\DriverLoadPolicy". This can also be configured through Group Policy on a domain-joined client. It is possible with this value to configure to only load good drivers only or keep all drivers, whatever is the detection [1476] from ELAM driver. Of course, it is possible to have intermediate policy [1477].

when all boot-start drivers have been initialized and the callback facility is no longer functional.

Before going deeper in this callback, it must be noted that any error returned from a status update callback is treated as fatal and that it leads to a system bug check. Additionally, if the callback returns an error, the driver's image is treated as *unknown*. More directly, the code executed in ELAM context must be able to respond efficiently and precisely to any demand it is notified for. Any error would compromise the boot of the machine, making harder to correct any problem. This shows all the criticality and skills required to develop this type of driver.

The boot-driver callback [1479] is notified for two types of reasons which are provided through a dedicated parameter called *Classification*<sup>33</sup> [1481]. This classification informs the driver for which purpose it has been notified. The two possible values are *BdCbStatusUpdate* and *BdCbInitializeImage*.

The first value provides status updates from Windows to know at which stage of the boot process the action of the driver is (and if it is therefore still required). This information is provided in a specific parameter (shared with *BdCbInitializeImage*). Two status (*BdCbStatusPrepareForDependencyLoad* and *BdCbStatusPrepareForDriverLoad*) indicates that a boot-driver or a dependency from this driver is about to be loaded. The last status (*BdCbStatusPrepareForUnload*) informs ELAM driver that Windows has completed the initialization of all boot-start drivers. In this last case, the driver should initiate its cleanup procedure by removing callbacks (especially the boot-driver callback thanks to *IoUnregisterBootDriverCallback* routine [1482]) and be prepared to be unloaded. Deregistration cannot occur during a callback; rather, it has to be done during the *DriverUnload* routine [1483], which a driver can specify during its *DriverEntry*.

In addition to prepare cleanup procedure, this last status received can be used by to check the presence of a runtime antivirus driver. Indeed, by initializing a regular boot-driver, it is possible with an ELAM driver to check if this driver was loaded and initialized or not. If it is not the case, an ELAM driver can fail the prepare-to-unload callback to prevent Windows from booting without the antivirus driver. In a way, it ensures a chain of trust since the ELAM driver can correctly ensure that the regular anti-malware security has been correctly loaded.

The second value (i.e. *BdCbInitializeImage*) provides information about the boot-driver image loaded in memory. This notification is performed with a *BDCB\_IMAGE\_INFORMATION* structure [1480] which holds a lot of information about the loaded boot-driver image. In this one, we have access to the full path name on the disk, the registry where the service is registered, and information about the certificate used to sign the loaded driver (publisher, issuer, hash algorithm used, content of the hash and so on). This is in this structure that *Classification* field [1476] is dedicated to register the classification decision made by the ELAM driver. To ease the verification procedure, it is possible to use the CNG API [1359, 1484] which has been loaded before ELAM drivers.

To be ELAM-compliant, the driver must handle malware signatures database in a specific way. First, this database must include, at a minimum, an approved list of driver hashes. And as explained, ELAM drivers should not refuse to load Windows' drivers... The signature database is stored in the registry in a new "Early Launch Drivers" hive under "HKLM\ELAM\\\" where "<VendorName>" corresponds to a unique key per vendor in which to store their database. The database is a *binary large object* (BLOB) which is a way to say that it is antivirus vendor defined. For performance reasons, this database stored in the registry is unloaded from memory after its use by Early Launch Antimalware. Of course, if this database is still required subsequently, it is possible to reload it if necessary by the antivirus boot-driver. This database must be use part of the detection procedure to certify or to refuse boot-drivers.

What is happening when a boot-driver is skipped due to ELAM driver detection? In this case, the kernel keeps on attempting to initialize the next boot driver to be loaded. Such procedure repeats until there is not more driver to load (meaning the boot start-up procedure succeeded) or the boot failed because the boot-driver

---

<sup>33</sup>The *Classification* parameter provided to a boot-start driver's *BOOT\_DRIVER\_CALLBACK\_FUNCTION* routine must not be confused with the *Classification* field [1476] used in the *BDCB\_IMAGE\_INFORMATION* structure [1480]. The first one is used to describe the notification purpose of the callback and the second one to classify the boot-driver analyzed.

that has been rejected was critical and absolutely required. If the crash occurs after the disk stack is started, then a crash dump is generated. The last has some information about the reason or the crash, including information about the missing driver. Otherwise, information can be displayed on the screen, hoping it could be meaningful for the user.

#### 4.6.3.4 General security provided by ELAM driver

Technically speaking, before Windows 8, it was possible to create a bootkit malware able to update the boot procedure of Windows. Attacking directly Windows boot procedure has been hardened, version after version of Windows. This enforcement of driver loading policy and kernel protection has made more difficult to insert a third party driver in the boot chain, compared to the situation in the past. This old principle is illustrated on Figure 6.23.



Figure 6.23: Booting procedure before Windows 8.

With the rise of ELAM technology, it is now possible to be notified for each boot-driver about to be loaded by the operating system. That way, ELAM driver can check which driver will be running on the system, before these ones are executed. This is a real security which theoretically ensures that malware could be detected before any code can be executed by *regular*<sup>34</sup> developers. This security can be illustrated as given on Figure 6.24.

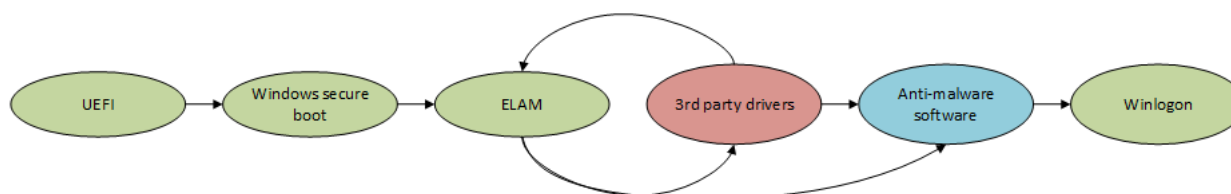


Figure 6.24: Booting procedure with secure boot and ELAM technology able to control third party drivers and ensures that antivirus driver has been correctly loaded.

<sup>34</sup>Regular developers as opposed to ELAM drivers developers, whose companies are specially selected by Microsoft.

#### 4.6.3.5 Extra implementation with an ELAM driver

##### Key Point 6.40:

- ☞ In practice, ELAM driver should be limited almost exclusively to the analysis of boot-drivers loaded in memory.
  - ☞ But loaded driver could have an impact on the configuration stored in the registry of Windows.
  - ☞ This is why ELAM driver is allowed to filter registry operations thanks to `CmRegisterCallbackEx` routine.
  - ☞ This allows us to monitor if there is not threat inserted at run-time in the registry (Key-Point 6.34).

Operationally, an ELAM driver should not do anything else than exclusively analyzing boot-drivers loaded in memory. Since we are launched very early, the API actually available is quite limited. Nevertheless, we have access [1477] to the *Windows Registry Filtering* API [1285] through `CmRegisterCallbackEx` routine [1286]. That way, it is possible to implement our monitoring solution as proposed in section 4.6.1. This allows to activate the registry run-time protection very early to prevent the execution from any unexpected driver already installed. In addition, if a driver tries to register itself into the keyboard device driver stack, we will detect it and we would have the possibility to reject this action. Otherwise, even if it is not proposed in the documentation, if necessary, the driver file on the hard disk can be accessed directly. Of course, an ELAM driver has extremely high performance constraints, which could restrict such operation.

Such an operation allows us to position ourselves in such a way that we can start our monitoring procedure before the other drivers and thus controlling their access to the keyboard device. What we have to see here is that with ELAM technology, it is possible for us to activate our protection very early (and even before any non ELAM boot-drivers) to check that no bad driver is inserted in the keyboard stack. Moreover, it allows us to make the link between our ELAM driver and the protection driver that will be launched among the boot-drivers as a keyboard upper filter. Thus, there is a real continuity of protection between the ELAM driver and our defense system.



## 5 Going further current limitations

Our research was limited to what could be observed in different situations or put into practice directly. Due to lack of time or simply insufficient support, it was not always possible to fully implement all of our ideas. We mean that some of the proposed solution might have been partially implemented but insufficiently tested and some others are just theoretical proposals. This is why we propose in this section various approaches to go further or to explore other solutions. It is also a way for us to draw up the current limitations of our system. What is possible for our system to do and what is not.

### 5.1 Improving cipher key protection against crash-dump

#### Key Point 6.41:

- ☞ During a system crash (*Blue Screen Of Death* — BSOD), a crash dump file is generated containing information about the crash and all or part of the system memory.
  - ☞ It could include the cipher keys used by our protection system.
  - ☞ An attacker could collect the ciphered stream, cause a crash, retrieve the data (and thus the cipher keys) on reboot and decipher the data stream.
- ☞ It is possible to protect against this type of attack with the *Bug Check Reason Callback Routine* used to process the crash-dump before it is sent to the hard disk.

A flaw in the security of our system is to not take *crash-dumps* into account. A crash dump is the dump file that is produced when a system crash<sup>35</sup> happens, that is to say when Windows cannot run correctly anymore. In addition to the *blue screen of death* (BOSD) displayed in such situation [1486], there is a produced file [1487] which holds information resulting from the crash context.

Technically, there are two types of *crash-dump*. The first is a *kernel-crash-dump* (generally called *crash-dump*) generated in the context of a BSOD. There are three possibilities to select what should be recorded. The *complete memory dump* option which records all the content of system memory when the system has crashed. It means it records data from processes that were running and data coming from the kernel. Of course, the total amount of memory recorded can be important. To only keep relevant information, it is possible to select *kernel memory dump* option to only keep kernel memory in the crash-dump file. Finally, the tiniest option is *small memory dump* (64 KB) to only record the smallest set of useful information that may help to identify the reason of the crash. All information can be configured in the Windows' registry [1488]. To generate a *crash-dump*, notwithstanding the involuntary case where a bug is triggered from a kernel-mode component, it is possible to generate it from specific Sysinternal's program [1489] or from the keyboard [1490] if the operating system has been correctly setup before the crash in the registry.

The second type is rather reserved to user-mode applications and it is called *mini-dump* [1491, 1492]. Since Windows Vista, thanks to the Windows Error Reporting (WER) [1493], it is possible to generate full user-mode dumps collected and stored locally after a user-mode application crashes. All configuration where this crash dump is generated can be configured through the registry of Windows [1494]. Note that it is possible to generate such crash by calling `MiniDumpWriteDump` function [1495] and to analyze its output directly [1496]. Another way to generate a crash is to use a Windbg debugger with the `".dump"` command [1497].

In both cases, the targeted process (mini-dump) or the entire system (crash-dump) can be written to the hard disk. In a malicious context, even if the operation would be nothing but stealth from the user's eyes, it would be possible to retrieve the cipher keys in memory used by our protection solution. That way, if a keylogger has recorded ciphered keystrokes, with the cipher key retrieved, it could be able to decipher them.

But there is a way to prevent this issue thanks to the *Bug Check Reason Callback Routine* [1498]. Indeed, a driver can register a `KBUGCHECK_REASON_CALLBACK_ROUTINE` callback routine [1499], that will

<sup>35</sup>Such event is also known as a *bug check*, or a *stop error*, or a *kernel error* or *BSOD* for *blue screen of death* [748, 1485].

be called by the system in the context of a *crash-dump*. This callback can be registered with `KeRegisterBugCheckReasonCallback` routine [1500] by selecting one of the possible reason [1501] to be notified. Among all the different reasons (which are quite technical), `KbCallbackTriageDumpData` and `KbCallbackRemovePages` are the most relevant for us. The first is used in the context of *mini-dump* generation. The second is used to remove memory pages to be dumped on the crash-dump file. In the callback, there are strong restrictions<sup>36</sup> about what can be done. But it is possible, in our case, to remove the pages known to hold cipher-key from our driver. Technically, a context can be provided to the callback routine at registration time. In this context, we can store a list of addresses where cipher-keys or any sensitive data belong. Note that it is possible to evaluate if the bug check callback has been correctly taken into account thanks to Windbg debugger [1502].

## 5.2 Multi-processes management

### Key Point 6.42:

- ☞ In practice, our solution can handle several protected processes, but each at a time.
  - ☞ For us, it does not make sense that two applications can receive the same (textual, since shortcuts are allowed) information from the keyboard at the same time.
  - ☞ This observation directly follows Microsoft's GUI guidelines, internally driven in Windows with the keyboard focus.
  - ☞ Nevertheless, several processes can be protected, in such a case, the cipher key is updated whenever a protected process has the focus.

Technically, our driver can handle only one cipher session at a time. Why? Because it does not makes no sense that a user could write on two applications at the same time since the keyboard focus is unique. If a session is open from an application A, another application B would not be able to decipher keystrokes handled from the driver. Somehow, Windows does not allow the keyboard to be shared synchronously between two applications (except with background reading with `GetAsyncKeyState` — Key-Point 4.50 — but this case is quite different since it does not question the general window message system). The impossibility for our driver to cipher a key pressed with two valid but different cipher keys, for each process, is only the continuity of this design choice made by Windows.

In practice, it is nevertheless possible to manage several applications at the same time, as long as they are alternatively used. As described in section 4.5.3, our SDK handles automatically keyboard focus acquisition or lost. Internally, our API provides the `GostxBoardStartMonitorProcessState` function (section 4.5.1) to create a dedicated thread in which the keyboard focus monitoring is performed and for each case, a notification is sent to the driver (Key-Point 6.31). This is the driver's responsibility to switch the cipher-key used in the cryptosystem to interface the correct protected process holding the keyboard focus.

---

<sup>36</sup>These restrictions come from the fact that the callback is executed at `IRQL = HIGH_LEVEL`, which prevents to allocate memory, to access pageable memory (which explains also why it makes sense to allocate the cipher key in non-paged in kernel-mode context), to use any classical synchronization mechanisms and to call any routine that must execute at `IRQL = DISPATCH_LEVEL` or below.

### 5.3 Crash of the protected process

#### Key Point 6.43:

- ☞ It is possible that the protected process crashes or is terminated without using our API.
  - 👉 In this case, the driver registers a callback (with `PsSetCreateProcessNotifyRoutineEx` routine) to track the creation and shutdown of protected processes.
  - 👉 The driver can act without necessarily being notified by the API of the library (even if it would be better).

It cannot be excluded that the protected application may crash. This may be due to a programming error coming from the protected application itself or from our SDK (even if everything is done to prevent it). If the crash occurs while the driver is protecting the keyboard's communication, the driver must be able to detect that the protected application is no longer running. There is no communication link computed between the protected process and the driver to exchange keystrokes. Otherwise, it would allow the driver to be notified about the absence of the process during a possible exchange. Instead, it is possible to know which process requires a protection when that one is requesting a cipher key. When this request happens, it is possible for the driver to retrieve the `EPROCESS`<sup>37</sup> [525] of the process to be protected. The address of the `EPROCESS` structure can be kept in memory in a linked list (in which the cipher key is also included, among other data) to memorize which are in the list of protected processes.

Keeping the `EPROCESS` address from the protected process does not allow to be notified when the last disappears. To proceed, it is mandatory to use the `PsSetCreateProcessNotifyRoutineEx` routine [1503] that allows a driver to register a callback routine `PCREATE_PROCESS_NOTIFY_ROUTINE_EX` [1504] which will be notified for process creation or destruction. This callback is notified with many parameters, including the `EPROCESS` address and the process ID of the process notified. In our case, we only monitor the case where the process is exiting. We can use the `EPROCESS` address provided in the callback to compare it with the one stored in the linked list of protected processes. The process ID is a bonus check which can be derived from `EPROCESS` thanks to `PsGetProcessId` [1505]. This last check is not perfect since one process could retrieve the same process ID from a former process. It is unlikely but such situations could occur, this is why it is relevant to check with the `EPROCESS` address.

### 5.4 Protection at deeper level in the device stack

#### Key Point 6.44:

- ☞ As already explained in Key-Point 6.10, being lower in the device stack implies a lot of work for very limited (not to say no) gain.
  - 👉 If a threat inserts a driver at a lower level than ours, it means that it has already passed our ELAM driver and its driver is parsing all devices using the filtered transport technology (PS/2 or USB/HID).
  - 👉 It means such a threat has already enough rights to pass our security (Key-Point 6.1).

It may make sense to wonder if and how it could be possible to provide a protection system lower in the keyboard device stack than we are. Technically, there is no conceptual means to prevent a driver keylogger to be lower than our driver. It is technically complex to write such low level driver (Key-Point 4.57) and our solution is a correct balance between complexity, efficiency and genericity (Key-Point 6.10). But it may be interesting to see if it is possible for our security system to work lower in the device stack than it is now.

---

<sup>37</sup>The `EPROCESS` structure is used internally by the kernel to represent and to store all information about a given process. Even if this structure is only partially documented, it is supposed to be unique per process. That way, keeping its address is enough for our driver to identify a process.

---

If we focus on the case of the USB/HID keyboard, it is quite possible to write a special solution for each of this type of device. As explained in Chapter 4, section 4.2.4 (Key-Point 4.18), it is possible to insert a driver in the HID device stack to interface it. In such situation, it would be required to create a device driver handling all the dispatch routines used by HID (Key-Point 4.19). In addition, it would be necessary to parse the HID requests to be able to deal with the keyboard only (to not interfere with other devices) — Key-Point 4.21. In this case, our operations would be focused on understanding the HID report descriptors of the keyboard and all subsequent reports. It is possible to use the kernel HID API already presented (Key-Point 4.22) to facilitate this operation.

But it is possible to go below at the USB level. We can be in the USB device stack as given in Figure 4.28. At this level, we do not deal with HID but with the USB packets that are exchanged. Depending on where we belong, we have to process the packets that deal with the communication with the USB device. The lower we are, the more packets we have to manage and higher is the complexity of the parsing procedure (since we are not only focused on keyboard activity but on all USB devices activity). Once we are able to process USB packets, it is possible to consider interpreting their contents to finally retrieve the HID data. In this case, the Chapter 4, section 4.1 allows us to know how to interface data transiting through our driver at this level. The same principles apply in the case of PS/2 keyboard devices with ISR routines management (Key-Point 4.5).

In the end, it is possible to answer that our solution could be implemented at a lower level in the device stack. However, the complexity of the driver to be implemented would be more important (Key-Point 6.10). And we must also see that we would not only have the keyboard to process but all USB and HID devices to manage. This means that our action must not impact too strongly other devices, when some of them may need an optimal response time. It is also for these reasons that our solution aims to be at the level of the keyboard driver. To be generic (and do not dependent on the transport technology), but also to be efficient by being notified only and the sooner for keyboard devices actions and not for other devices.

## 5.5 Limitation with low level keyboard hook procedure

### Key Point 6.45:

- ☞ At the time where we presented our solution at the DefCon conference [1304], we had trouble managing low level hooks (WH\_KEYBOARD\_LL) — Key-Point 31.
  - ☞ There was retention by the system of the scan codes as provided by the keyboard before the notification of our driver.
  - ☞ This *saved* scan code value was provided directly in low level hook context, justifying the notion of “low” but bypassing our driver.
  - ☞ At this moment (and without any glory), the only solution we could find used undocumented mechanisms to solve this issue.
- ☞ Today, we are no longer vulnerable to this problem.
  - ☞ We have updated the architecture of our driver and there have been changes since Windows 7.
  - ☞ In fact, the problem only concerned PS/2 keyboards only and it could have been solved more efficiently if we would have had the knowledge given in Chapter 4.

As part of the development of the driver presented at Defcon [1304], we had observed a surprising result. When a low level hook (WH\_KEYBOARD\_LL [2]) is installed using SetWindowsHookEx [1] function, we can access the content of the scan code thanks to the KBDLLHOOKSTRUCT structure [1019] which is provided to the callback function [983] notified when a key is pressed (Key-Point 4.52). If we look at the *scanCode* field in the structure, we can access the scan code (as well as the virtual key code which is held in the structure too) of the keystroke. The surprise came from the fact that when our protection system was active, this scan code was not modified and it held the original value provided by the keyboard device. As a result, this keyboard management technique was not processed properly, causing a security hole.

How can we explain this? The execution context at that time was about Windows 8 operating system and Virtual Box virtual machine software. The driver was a proof of concept mainly focused on processing information coming from a PS/2 keyboard. PS/2 keyboard connection was by-default the one available on our virtual machines. This means that we were using PS/2's hooks techniques, as described in Chapter 5, section 3.2.2.2 (Key-Point 5.11). However, when checking the execution chronology (by using Windbg debugger), we easily observed that our driver was notified long before the application's hook callback notification. This meant that we were processing correctly by ciphering the scan code but that the low level hook (and only this one) was able to retrieve mysteriously the original scan code despite our ciphering. The only explanation is that the low level hook notification uses a scan code value stored in another memory location that the one provided to our filter driver. This extra-memorized scan code escaped from the eyes of our protection driver.

To solve this mystery, we needed to know how the notification procedure worked for low level keyboard hook. First, we reversed the `SetWindowsHookEx` function in Windows 8. Our analysis was close to the one proposed in Chapter 4, section 5.3.2.3 (Key-Point 4.54) but it did not help us to know how the notification procedure worked. The solution came by analyzing `NtUserCallNextHookEx` (kernel interface for `CallNextHookEx` [1005] function). Indeed, this function is called part of `SetWindowsHookEx` procedure to give parameters to the next hook in the hooks chain. Internally, this routine was calling `xxxCallNextHookEx` which itself called `xxxCallHook2` routine. In this routine, `xxxInterSendMsgEx` routine is used to send the message to the system, part of the hook notification. In a way, this procedure is similar to the one described in section 5.3.2.4. This analysis confirmed us that the procedure did not go lower than our driver. More directly, it meant that there was neither *extra*-communication channel bypassing our driver nor any hidden way to communicate with the keyboard device.

With hindsight and observing that the `KBDLLHOOKSTRUCT` structure provides a virtual key code field (`vkCode`), knowing that the latter comes from a translation performed in the raw input thread (Key-Point 4.37), we could have reached this conclusion faster but we were ignorant about this point. But we did know that such observation confirmed the hypothesis that a copy of the scan code was made by the `i8042prt.sys` driver just after it read the keyboard port and before it called our protection driver. Internally, `i8042prt.sys` uses `I8042KeyboardInterruptService` routine to handle the interruption coming from the keyboard (Key-Point 4.5). This routine has called `I8xGetByteAsynchronous` routine to read the byte associated with the interrupt on the keyboard port. This routine has used `READ_PORT_UCHAR` routine from the hardware abstraction layer (HAL) library [1506] to read a single byte from the interruption port handling the keyboard. This byte read is the scan code provided by the hardware keyboard device. And when analyzing the rest of the `I8042KeyboardInterruptService` routine, we observed that two copies of the scan code read was made in a specific (and totally undocumented) structure in memory. This are these copies, from this structure, which have been used by the low level keyboard hook to retrieve the scan code. In a way, the notion of *low-level* is quite exact since it directly provided information from the interruption port, bypassing any subsequent modification potentially performed by any third party driver in the keyboard device call stack.

How did we correct this issue? Without any glory by modifying the contents of these addresses holding the copy scan code value in memory. It goes without saying that manipulating undocumented structures with offsets that may change during a Windows update is neither a guarantee of quality nor stability over time. But it was the only solution since there was no way to interact with this part of the memory in a documented way. In hindsight, it might have been a better idea to work on hijacking the user-mode callback (via function hooking mechanism [419]) recorded in all the applications using this low level hook, but it was taking us away from our driver's architecture, not to mention it was only about a proof of concept and not an industrial software.

Does this problem still persist with our updated solution? Fortunately, the answer is no. When dealing with USB/HID keyboard devices, the internal procedure are very different than the one uses for PS/2 keyboard devices. There is no copy of scan code kept for the purpose of low level hook. This is directly the scan code outputted from our driver which is used, even for low level hook procedure. In addition, we do not use anymore ISR PS/2's hooks techniques to manage the keyboard input. Instead, we prefer to use `KeyboardClassServiceCallback` routine (Key-Point 6.8) to handle the keyboard. And surprisingly, the case of PS/2 keyboard devices is no longer a problem. Maybe it is due to our keyboard management that acts low enough to act before the `i8042prt.sys`

driver has had time to make a copy or maybe it is an architecture evolution between Windows 8 and Windows 10 that standardizes the way the data from the low level keyboard hook is retrieved (and the saved value vanished).

We did not take the time to explore the issue further as the problem was no longer present with our new solution. In any case, with the state of the art achieved in Chapter 4, we are now perfectly able to understand each action performed by the system to manage the keyboard. Moreover, it did not make sense for Windows to keep the original value as provided by the keyboard. Why? Simply because driver filters, provided by the keyboard manufacturer, can naturally modify the content of the scan codes (bug fixes, added features depending on the user's keyboard layout, and so on), not to mention the automatic correction provided by Windows itself (with the *Scan Code Mapper* Key-Point 4.45). The information as it was saved at that time (under Windows 7) made little sense. Finally, this mechanism only concerned PS/2 type keyboards and after observing the operation of this type of keyboard (Chapter 4, section 3), it would have been possible to deal with the problem more effectively by inserting an ISR routine directly on the interrupt managing the keyboard. But this problem from the past is a good example of why it is important for a security driver to be low enough in the device call stack not to be bypassed.

---



## 5.6 Original protection based on HID source driver

### Resume 34:

- ☞ We propose a security project based on a parallel channel using the *Virtual HID Framework* technology in Windows.
- ☞ The idea is to inhibit the original keyboard (by having detected it at the HID level) and present it as a non-specific device, to take exclusive access to it and to process its original HID reports as if they would come from a virtual HID device.
  - ☞ A secure application would not listen on the keyboard but on this virtual device.
  - ☞ Access to this virtual device would be restricted to protected applications only (which would prevent eavesdropping to the channel).

We propose in this subsection an original idea to protect the keyboard. This idea is based on Windows documentation (meaning it is legal) but it has never been tested. It is also for us a way to show the originality of the solutions which can be used when dealing with this problem of keyloggers. It is also to show that it is always possible to design original solutions on the subject... The idea is to reuse the principle of the parallel communication channels, but enhanced with the latest Windows technologies and the possibility of using several types of keyboards.

One idea that has had some success in the literature [14, 1211, 1228], especially for password entry, is to use another shape of keyboard (Chapter 5, section 4.2.3, Key-Point 5.22). The solution must also be compatible with existing keyboards. For example, there should be a central point that collects inputs of *original input devices* and more traditional ones such as keyboards.

### 5.6.1 HID source driver

#### Key Point 6.46:

- ☞ Thanks to *Virtual HID Framework*, it is now easier to develop drivers that allow to realize a virtual HID device.
  - ☞ A *HID source driver* can emulate the source of the HID stream data from a virtual device.
  - ☞ In practice, this means using a particular API (contained in Virtual HID Framework) and interfacing with `vfh.sys` driver.

Usually, HID input devices (keyboard, mouse, joystick, and so on) send various reports to the operating system. More directly, HID Usages [712] and HID Collections [631] allow the operating system to understand the purpose of the device and take necessary action when receiving specific reports. All this information is provided via various transports technology (USB, Bluetooth ...) where some of which are supported by Windows and some are not (Key-Point 4.2.4.1). If a specific transport (real hardware or software) is not supported or in the case the HID data stream would not come from a real hardware, before Windows 10, a *HID transport minidriver* [1507] had to be written to interface the HID class driver, `Hidclass.sys`. Writing such driver can be particularly challenging since it is a complex task.

Since Windows 10, it is possible to write a specific *HID driver* by using *Virtual HID Framework* (VHF) [20]. This framework eliminates the need to write a transport minidriver and a HID driver may rely on KMDF<sup>38</sup> or WDM programming interfaces. With this technology, it is possible to allow a specific driver to report a stream of HID data to the operating system. Such a driver which acts as a *source* of HID stream data is called *HID source driver*. Such a driver allows to support HID reports that might not directly map to real hardware. More directly, it is possible to simulate with or without any underlying real hardware device a HID device. In a way, this driver can produce a stream of HID data to represent a virtual hardware component. Documentation from

<sup>38</sup>Such driver is part of the Windows Driver Frameworks (WDF) [1155].



Microsoft [20] proposes an example by considering an accelerometer from a phone that is behaving as a game controller and which sends data wirelessly to a computer.

Extracted from official Microsoft’s documentation [20], Figure 6.25 represents the virtual HID device tree architecture. One important point is `vhfkm.lib` which is part of Windows Driver Kit (WDK) [1508] and that is used to develop drivers. This library exposes the *Virtual HID Framework* API, including routines and callbacks that are used by a HID source driver. This one is responsible to forward the requests from the HID source driver to the VHF driver. This driver (`vhf.sys`) must be loaded as a lower filter driver below our HID source driver in device stack. That way, this driver makes the communication between our driver and the usual HID class driver. More precisely, `vhf.sys` allows to dynamically enumerate and create a physical device object for each child device (if any) that are specified by the HID source driver. It implements the HID Transport mini-driver functionality to provide the stream of HID data. Finally, the `hidclass.sys` and `mshidkmdf.sys` drivers constitute a pair of drivers able to manage HID input (for both WDM and WDF drivers). They are responsible to enumerate top-level collections [624] similar to how it enumerates those collections for a real HID device.

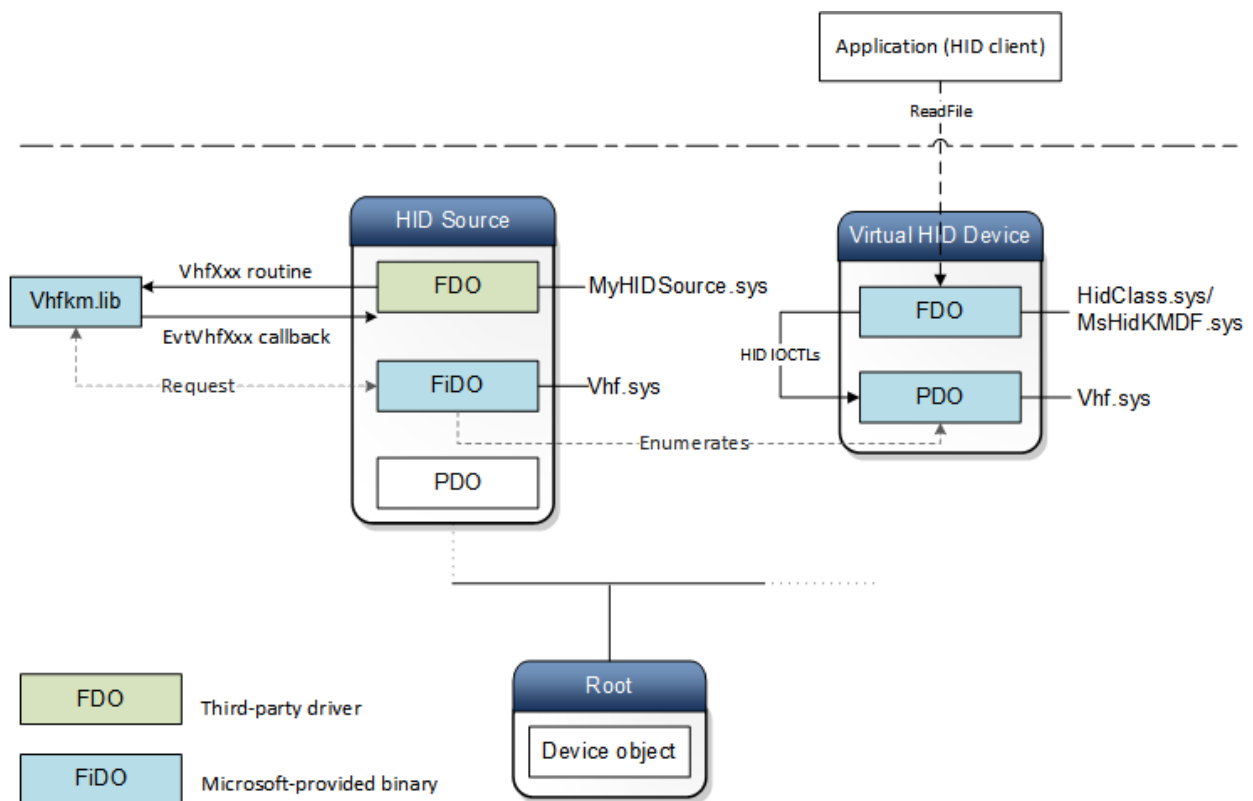


Figure 6.25: Device tree representing the drivers and their associated device objects with a HID source driver (from [20]).

## 5.6.2 Proposed HID source driver solution

### Key Point 6.47:

- ☞ Our solution aims to redirect the stream coming from a HID device (keyboard or other) to redirect it to a HID source driver driver used as a privileged communication channel.
  - ☞ The protected application notifies an user-mode service or our kernel-mode driver to require a protected keyboard access.
  - ☞ The redirection is set up at the kernel level and the access is secured by Windows (Key-Point 4.23).
  - ☞ The received keystrokes (if they do not concern a system shortcut) are broadcast by the message system.
  - ☞ Only applications using the message system are concerned here (since we bypass the *raw input thread* and thus its internal buffers used by the asynchronous keyboard access — Key-Point 4.50).

From this technology, we propose to create a HID source driver able to handle real HID hardware. The goal is to make a mix between regular keyboard devices connected to the machine and original devices able to handle secure input. In the context of password protection, we are reusing the example provided by Microsoft documentation and we imagine an input coming from the accelerometer from a phone that is used as a pointer selector on a picture displaying an alphabet on the phone's screen. Such picture could be provided to the phone by our defense system, and there is nothing wrong trying to make it random for each use. By moving the hand, it is possible to move the cursor and select a character on the screen. Data from accelerometer is transferred wirelessly (in a cipher form to avoid interception by indirect hardware keyloggers — Key-Point 5.3) to the computer and handled by our driver. Another example is to use a RFID device which would be acting as a password provider when the user is close enough from the RFID reader connected to the computer. A web-cam, a microphone, a joystick, a USB dongle, or even a remote control device would be sufficient to imagine a new protection system (as in Key-Point 5.22)...

Once one of these devices has been used as a password provider, this is our driver which is handling it. Otherwise, such devices are handled by the operating system and existing drivers able to manage them, if there are any. The same goes for classical keyboards. When a password<sup>39</sup> is requested, our system is notified by the application. This can be done by the requesting application via a call to a specific function from one SDK provided by us. If it is required to use our SDK *aggressively* for any application requesting text (by detecting that the focus keyboard is taken), this can be done via a Dll injection system, but this is not something we would like to do.

The architecture of our solution is given in Figure 6.26. This is one detailed with numbers representing different elements we reference with parentheses in this part. Our architecture is divided in three components. Two drivers and a user-mode service which is used to manage messages representing keystrokes for applications. The first driver is a HID source driver which handles stream of HID data coming from third-party HID device drivers (1). In practice, it may represent specific hardware device used to process "passwords" or "text to secure" or regular keyboard devices. By design, our security is activated if an application requests protected input or if a device dedicated to password management is sending information.

In our case, even if it is possible to use different devices than regular keyboard, keyboard devices are privileged. In this case, HID keystroke information is provided by the underlying device drivers (2). Usually, this information is handled by `kbdhid.sys` (3) which transfers it to `kbdclass.sys` after having parsed HID keyboard data. When the security is disabled, this path is not modified. But when the security is activated, another driver installed between `kbdclass.sys` and `kbdhid.sys` drivers reroutes the HID data from `kbdhid.sys` to our HID source driver. This switch might be implemented the same way as using `KeyboardClassServiceCallback` routine (4). In this case, we are dealing with keyboard information and no more HID information. If it is more comfortable to

<sup>39</sup>Password are used here to illustrate short text input. But this illustration can be extended for long text management if the input interface is convenient enough with a given device.

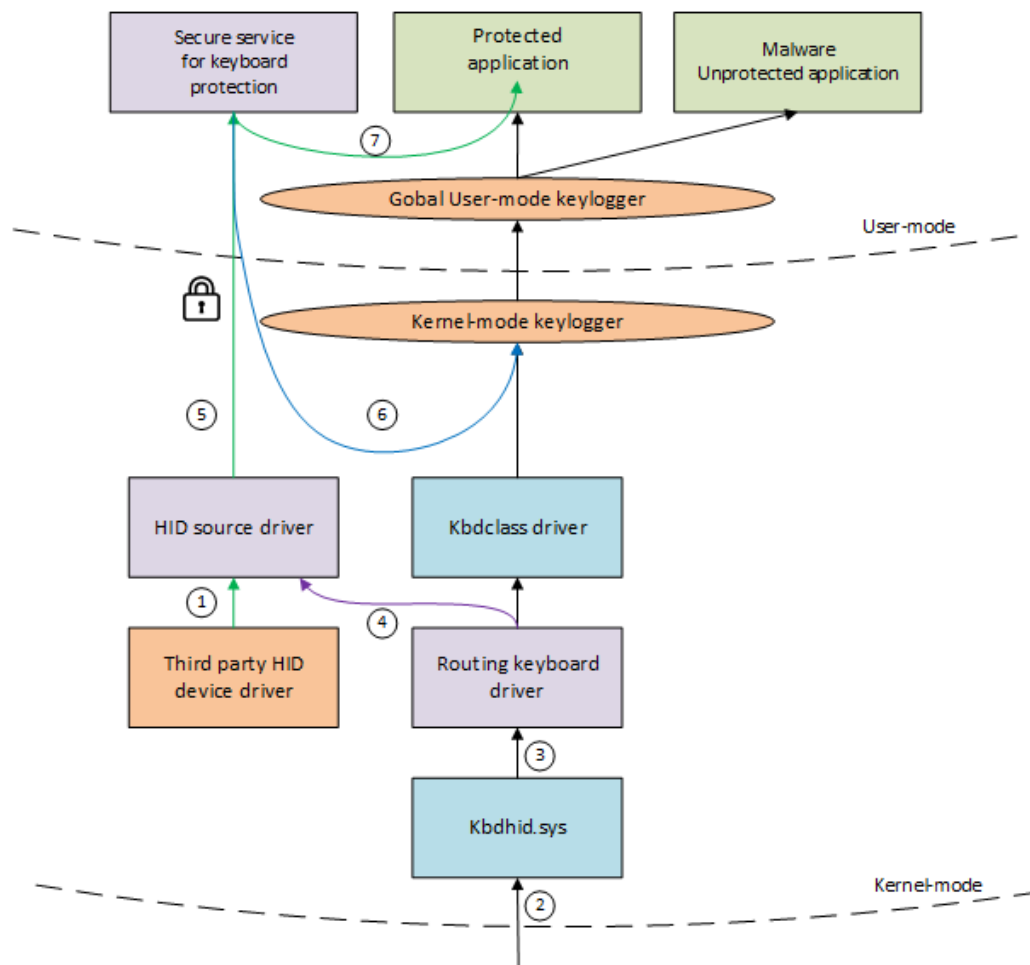


Figure 6.26: Illustration of the proposed solution.

work with raw HID data (since HID data is processing by kbdhid.sys) from a development point of view, a HID pass-through driver could be inserted below kbdhid.sys driver instead of above. That way, when the switch is not activate, the driver transfers transparently data to kbdhid.sys. But when protection is required, the switch is enabled and the raw HID data coming to kbdhid.sys is rerouted to our HID source driver. This one can handle this HID data like it does with any other HID devices connected to it.

The data is processed by our HID source driver to send it directly to an application in user mode. This application gets access to our virtual HID device by enforcing secure read for that device [667] (Key-Point 4.23). That way, it is not possible for another application to get access to our dedicated communication channel except if it owns *SeTcbPrivilege* privilege [668]. In addition, it is possible to only restrict the connection to one client authenticated during the request and checking the digital signature of the running process in memory (Key-Point 6.19). With these securities, it becomes very complicated to listen to the communications between our driver and our service.

The role of our service is precisely to retrieve the content of keystrokes (or any data coming from a device used to manage text input) and dispatch them (5). The latter has a list of key codes that can be used for passwords (or codes exchanged with a non-keyboard device). Anything that is not in this list should be considered useful for the system and broadcast to all applications. This concerns usual shortcuts such as CTRL+ALT+DEL or others. At that point, there are two possibilities. The first is when there is no application to protect or when a key code is not in the list. In this case, keystrokes received by the service are simulated (6) by using `SendInput` function [940]. Hence, provided keystrokes will be correctly interpreted by the raw input thread (Key-

Point 4.44). The second case is when a protection is required. This way, information are directly sent to the protected application via `SendMessage` function [933] (7). This function is not able to reenter in the raw input thread but it has the possibility to directly target the application holding the keyboard focus. As a result, other applications are not even aware that a key has been pressed, reducing the chances that it can be intercepted by a keylogger. The drawback with this operation is that application which are not using system messages to handle keystrokes will be penalized by this design choice. Indeed, asynchronous keystroke management is not taken into account (Key-Point 5.14). In a way, our solution must be reserved to applications using message system as keyboard management...

### 5.6.3 Conclusion about our protection based on HID source driver

#### Key Point 6.48:

- ☞ We are close to the philosophy proposed by the new authentication mechanisms in Windows 10 called *Windows Hello*.
  - ☞ However, our solution would allow a larger volume of text to be inputted by the user.
  - ☞ For example, we can imagine the virtual keyboard of a smartphone used as an HID source and redirected by our driver.
  - ☞ There are interesting possibilities to implement some keyboard layout based protection solutions (Key-Point 5.22).

When we look at this solution, it is possible to see a generalization of *Windows Hello*. Microsoft Windows Hello [1509], part of Windows 10, gives users a personal, secured experience where the device is authenticated based on their presence. In addition, Windows Hello for Business [1510] aims to replace passwords with strong two-factor authentication on PCs and mobile devices. This authentication uses a device which handles biometric or PIN. It is possible to interact with Windows hello by using Windows Biometric Framework [1511, 1512] to handle third-party sensors.

In a way, our proposed solution could be more generic than Windows Hello. Of course, it is mainly dedicated to passwords, but it can be used more generally for general text input. A smart-phone could use a dedicated application to enter text, which would then be transmitted ciphered and wirelessly to the computer, to retrieve the text typed. And as explained at the beginning, our solution has never left the drawing board and may, for a variety of reasons, never work. But this solution offers another approach to meet the needs of secure text input. It might be interesting to continue the work by trying to implement and realize the idea proposed here in the future.

## 6 Conclusion

### 6.1 General conclusion about GostxBoard solution and the research work produced to secure keyboard

This conclusion is intended to be a final word on Chapters 4, 5 and 6. It seems important for us to recall the articulation between the different chapters. It is about justifying the research approach used in this study and deployed through these three chapters. Rather than writing a conclusion only on our GostxBoard solution, it seemed appropriate to recall here what was our approach and how we achieved this solution.

#### 6.1.1 Study of the keyboard management under Windows 10 — Chapter 4

First of all, there is the technical state-of-the-art explaining the functioning of a keyboard under Windows (Chapter 4). This state-of-the-art is not simply a compilation of various sources presenting the knowledge about Windows' keyboard management at a given moment. The Microsoft documentation explains how to interface with the keyboard (via the Windows API) for a third party program and in general how the keyboard works. But nowhere is the internal detail about how the keyboard works internally. There are only a few articles here and there [743, 720, 731] but none of them deal with the whole subject. They can be focused on a given technology (the *raw input thread* in this case [731], but it could have been about HID or PS/2 keyboards) or talking about keylogger and presenting the technology used by keyloggers and therefore some internals from Windows [1513, 1114, 41, 1115].

To the best of our knowledge, no book or article talk about the keyboard as a whole, that is to say being able to explain from the moment where a key on the keyboard is physical pressed to the reception of the character correctly translated according to the user's layout preference in a given application. The reason for this lack may lie in the length of this manuscript: the subject is as vast as it is complex. Vast because the keyboard is a central element in a desktop computer and it is linked to almost all the components of Windows. Complex because there are a lot of elements dealing with different internal topics (device management, power management, security, GUI, USB and PS/2 protocols, HID, system interactivity, session configuration, worldwide alphabets, user-mode, kernel-mode — just to mention the main ones) and usually carrying backward compatibility since Windows is a construction based on former versions of the operating system.

This is one of the major contributions of our work: to give a rather precise and detailed vision (although sometimes summarized, for the sake of brevity, not to say simplicity, and because we cannot always detail everything) about the behavior of this device. Performed in the context of a reverse engineering activity, our own analysis is based on our own observations. Despite all precautions taken, it is not impossible that an observation error would have happened or to misinterpret one reverse-engineered element. Reverse engineering is a field that sometimes requires making some assumptions and trying to verify them as best as possible with what we observe. We have sincerely tried to do the best based on our current capabilities. But we are fully aware that a mistake is always possible. In the same way, we know that what we are presenting in this study is just a snapshot of Windows 10. It is a *freeze frame* from a project that is constantly evolving. We therefore inform our reader that some of the information presented in this study is subject to obsolescence. We are merely trying to acquaint our reader with the fact some of the information presented in this study is subject to obsolescence. This is the tough rule of reverse engineering. Despite this, we enjoyed understanding in a fine way how the keyboard works in Windows 10 with our current version.

But besides the pleasure to know how the keyboard works, it is also about the ability to understand the environment on which our security solutions are based. Why? Because we cannot build an efficient and reliable projects if we do not understand the physics behind the stage. Because it is not possible to have a solution that can be proven to be the best if we do not know how it works. And because it is fundamentally not possible to design a system if we do not know which constraints any solution is facing. A certain idea of science is at stake, but also independence in our ability to design and deploy solutions by ourselves. Of course, the goal is not to use undocumented mechanisms to design a security solution. That would be suicidal because it would add instability to the system more than security. But it is to understand why a solution works and also why some solutions cannot work.

As a reminder, our state-of-the-art in Chapter 4 starts with the history of the keyboards and the electronic circuits used by these systems which have not particularly evolved in their overall design since the beginning. Once the hardware keyboard devices have been detailed, it was necessary to document how they communicate with the computer that is hosting them. To do so, there are two main possibilities. On the one hand the old method with the PS/2 protocol and on the other hand, the new one, based on the combination of USB and HID. Once the communication protocols driving keyboard devices are understood, it is necessary to consider how the Windows kernel receives the signal generated by the keyboard. The reception (which usually happens at the USB and HID device stack level or by the hardware interrupt system with PS/2) allows a translation of the communication protocol to be understood by a driver in charge of the keyboard via a system of scan codes. From there, the latter can provide to the *raw input thread* in charge of the management of window events under Windows (and more generally of the responsiveness of the system as a whole) the content of the scan code. But because there are several sets of scan codes and to facilitate application development, Windows uses a standard universal alphabet (at least on Windows world) to designate the different keys on the keyboard. This is the role of the *virtual key codes* that are translated from the scan code received by the raw input thread.

All that remains is for the raw input thread is to send the content emitted by the keyboard as a message to the applications that need keyboard's input. Many possibilities are available, depending on how the applications are listening to the keyboard. For short, the application displaying a GUI at foreground should have the keyboard *focus* (which is constantly tracked by the raw input thread) to receive the stream of data as a *virtual-key-code*. This code can be translated into displayable character if necessary. This is finally the purpose of the kernel to send the content emitted by the keyboard as a message to the applications that has the focus of the keyboard. In the end, we have detailed how an application can receive and process the keyboard's content.

### 6.1.2 Study of keylogger threats and anti-keylogger solutions — Chapter 5

Coming from our technical state-of-the-art allowing us to understand how the keyboard works under Windows, we can study how to interface with it. And those who are particularly good at interfacing with keyboard devices are keyloggers. In practice, there is not much technical difference between a legitimate software that is listening to the keyboard and malicious software like a keylogger that is also listening to the keyboard and usually in a similar way. By knowing exhaustively how to interface with the keyboard allows us to know exactly what possibilities are given to malware. Thus, we can design effective and flawless strategies against them — since they cannot surprise us.

To confirm and detail the threat we want to face (i.e. keyloggers), another state-of-the-art based on literature reviews has been written on this particular topic in Chapter 5. If this one is still a bit technical, it is designed to be exhaustive in order to detail the different threats in its whole diversity. The technical part having already been covered in Chapter 4, we can delve into the particularities and the strategies implemented in malware, doing it at different levels (hardware or software).

At the hardware level, several strategies are possible. On the one hand, in an active way by being physically between (when it is not in) the device and the machine. On the other hand, passively (i.e. remotely while being close, but not physically in contact) with the device to be remotely intercepted. Hardware keylogger is generally the most effective strategy from an operational point of view. Indeed, a very little fingerprint is left from the system point of view (even none in the passive case) because software protection solutions are totally bypassed. In fact, software solutions act after the hardware interception, when the stream of data goes in the computer while the hardware keylogger has already intercepted the data on the way. More directly, they cannot do anything if not sometimes trying to warn of a possible interception (Key-Point 5.4).

Hardware solutions are cheap and easy to find online. These are often small hardware devices that look straight out of a bad spy movie. But they are restrictive in a daily use. Indeed, they presuppose having physical access to the target keyboard device (or the targeted machine) in the case of active solutions. For passive keyloggers, however, this presupposes having an access close enough to the target and that there is not too much electromagnetic or sound interference. It is not an attack that can therefore be carried out on a large scale (on thousands if not millions of individuals) and that requires human resources (to deploy the device). This is usually used in the context of a targeted attack on a well identified person. Against this type of threat, there



is unfortunately only physical security at the access to a machine that can really prevent this type of threat. It is also for this reason that our study has been focused on the case of software keyloggers.

The difference between hardware and software keyloggers is blatant. These act as regular software within the machine on which they are installed. They can act at different levels (firmware, kernel or user-mode), close to the hardware or as a classic user-mode application. Generally, malware has no choice but to use the Windows API to get access to the keyboard. The idea of totally re-implementing the keyboard management procedure would require a lot of work and a very privileged access (administrator and drivers) to the underlying operating system. This is not the strategy used by the majority of the threats observed. Nevertheless, we should not minimize the ingenuity of malware threat and the various strategies setup. The lower the malware is in the system (firmware, kernel-mode), the more complex its development is, but the more effective its action is (both operationally by recovering the data stream from the keyboard and by guaranteeing its stealth and survival within the system).

It is from the study of keylogger threats that it is possible to draw some plans to fight against them. It is an eternal adaptation from security software (and more generally antivirus software) to try to thwart this type of threat. Presenting the various solutions that exist to address this threat was the logical continuation of our study. We divided our analysis between existing solutions coming from academic and industrial worlds. Without falling into the caricature of opposing these two worlds (which are in fact closely linked but with different aims), we have sought to identify the different possible strategies to fight against keylogger threats.

The academic world proposes an approach divided between, on the one hand, active detection solutions (in the manner of certain antiviruses based on a knowledge base) and, on the other hand, passive solutions. It is this last case which particularly interested us because it starts from a simple postulate. In practice, it is complicated to identify the threat (active solution) while it is perhaps more interesting to neutralize it at run-time. The idea is then to starve their keyboard stream sensors or to interfere with them. Starving sometimes presupposes being able to distinguish between legitimate and illegitimate software. This is an interesting approach but sometime complex to implement without disturbing too much the user experience. Neutralization uses a different logic. It aims to provide a false data stream to the applications while the protected (and pre-identified) one acquires the data provided by the device. Thus, keyloggers are fooled and the impact for other applications (which anyway did not have to consider the keyboard while the protected application was active) is relatively small. This is a strategy that can be found, in various shapes and forms, in the case of industrial solutions.

Industrial solutions are often commercial, close-source and not always volunteer to document their internal strategies. Rather than using reverse-engineering (we already used it widely in Chapter 4), we prefer to use a similar approach than the one used with academic solutions, i.e. a documentary (literature review) approach. In a way, comparing the two different worlds with a globally similar approach is also a guarantee of consistency in the critical analysis of existing solutions. In addition, using a documentary approach with industrial solutions is an original evaluation methodology. We based our analysis on vendor software's own public statements to evaluate these different products. Although it may seem a bit dangerous to trust only the software vendors, our objective is not so much to judge the quality of some of these solutions (even if sometimes it might be possible with what they say about their own software), but to detail the different possible strategies that can be implemented to defeat or neutralize keyloggers. And it is usually an argument used for commercial reasons (to assert the efficiency or the seriousness of the solution). The objective in our case is to measure the advantages and disadvantages of each strategy, as well as the important points highlighted to fight against keyloggers. From this analysis, it was possible to conclude that there was room to potentially do better than what exists. Of course, this final analysis will be the basis for the specifications of our anti-keylogger solution too. With the objective of maximizing the observed advantages while minimizing the disadvantages.

### 6.1.3 GostxBoard solution as a new approach against keylogger threats — Chapter 6

In Chapter 6 we have detailed our solution based on the two precedent chapters. As described at the beginning of Chapter 4 in section 1.1, Figure 4.2 gives the general link between all of our chapters. Hence, on the one hand, we capitalize on our advanced technical knowledge about Windows 10 operating system and keyboard management in particular (Chapter 4). On the other hand, we use our knowledge of the threat to successfully



plan a defense while taking into account existing solutions to try to do at least as well, if not better (Chapter 5).

Starting with the objectives, specifications and technical constraints have been drawn from the observation of the threat and the existing solutions. The precise knowledge of the operating system also helped to guide certain choices, in particular to focus on what it was possible to do while ensuring that the code written was interfaced with documented Windows mechanisms. Keeping the user experience intact was also one of our requirements. This includes to not visually disturbing the user experience, minimizing as much as possible our own visual impact (we are not graphic designers) while keeping the stability of the system and the protected applications intact. Nothing is worse than a system that is unstable or open to vulnerabilities by a third-party product.

Subsequently, the most direct conclusion was to favor an approach where application developers of “*keyboard required interface*” would be able to be volunteer in order to protect the input text provided by the user. This dispenses us to decides which applications must be protected from those which should not. By anointing an application with a dedicated library provided by us, any developer would be able to allow a text to be securely captured. Of course, we are under no illusion that a volunteer-based solution could have a limited future. Hence, it remains possible to “inject” our solution into third-party applications, but this is not the design we would like to promote, although this identified need has been addressed by us. Such a use keeps the problem inherent in many existing solutions with the possibility of inducing a potential instability in the applications protected.

From the requirements definition, it is necessary to propose a solution able to meet them. We have been able to detail the architecture of our solution and especially to justify it. From the history of the project to the technical realization, it was possible to detail the technical part of our solution with the appropriate level of details. Still relying on the advanced technical analysis from Chapter 4 and, if necessary, on specific contributions from the Windows documentation, we were able to illustrate the security provided by our solution.

This security is provided on two levels. On the first hand, at the protected application level. This aims to provide the keyboard keystrokes via the classic Windows keyboard processing system but in unintelligible form. More directly, we will cipher the content of the keystrokes stream while allowing it to be always taken into account by the raw input thread and without harmful or unexpected results (such as involuntary keyboard shortcuts). It is therefore a piece of security added to an unmodified user experience. But with cryptography comes the management of cipher keys. There was of course a whole security protocol setup to prevent a third party application from accessing the memory of the protected process (to recover or modify the ciphering procedure) but also to prevent any literature-known way to force the cipher key to be stored in clear on the hard disk. This is done in order to avoid any forensic analysis of the potentially captured ciphered data stream. A potential vulnerability that few existing solutions seem to take into account, at the best of our knowledge.

On the other hand, a self-protection mechanism (as with GuardID solution — Key-Point 5.29) has been implemented to protect our solution against more or less direct attacks that could target it. For the sake of simplicity, to the best of our knowledge and to be truly effective, such an attack must satisfy a few prerequisites. On the one hand, it must necessarily have administrator rights on the machine. This is a requirement which should not have any exception. On the other hand, the attack must require a driver to run or an obvious observable event to the user’s eyes (such as uninstalling our own protection solution, which should be easily visible to the user’s eyes). It is an enhancement of usual security requirements that we propose to provide with our defense system.

Of course, we are under no illusion that this enhancement is relatively precarious. Being an administrator is enough to uninstall any software (and even reinstall a fake software that would mimic the interface of ours). And even if it would be technically possible (and it is possible in a way — Key-Point 6.5) to avoid being installed when Windows is running<sup>40</sup>, it would be against the user rights and the user experience. It is a choice that we have made and that we fully accept. The solution we propose is not perfect. But we assume that uninstalling our software requires sufficient administrator rights and that it would be potentially possible to warn the user

---

<sup>40</sup>The case where Windows is turned off is very complicated to handle since none of our code is running. In such a case, a solution based on full disk encryption such as *BitLocker* [1423] would be required. Indeed, in this case, even with another operating system mounting the disk where Windows belong to modify it would require the secret cipher used to cipher the whole disk, definitively preventing any *post-mortem* attack on Windows [1514].

that such an operation is in progress (and why not potentially refusing it).

In the end, the ultimate goal could be to show that it would be possible to design a software at least as efficient (if not better) as the existing security solutions (presented in Chapter 5). This is ultimately the role of this subsection. Reusing the table with the strengths and weaknesses of each of the existing industrial solutions (Table 5.6), it is possible here to establish our own entry for our own solution. This resume is provided in Table 6.2.

	GostxBoard
First release date	2015
Open source	Partially (original version only until now)
Free	Yes
Operational	Yes
Still maintained	Partially (in the context of this study until now)
Documentation	Correct — this study
User experience	Close to zero disturbance
Stability	Stable by design since it complies with Windows standards
Dll injection	No
Bypassed by ring 3 threat	No — at best of our knowledge
Bypassed by ring 0 threat	No or Hard
Require admin to bypass	Yes
General protection provided	High for volunteer processes
Keystroke management	Cipher keystrokes over RIT original message system
Keylogger keystroke access	Random keystrokes (from a restricted set)
SDK	Yes
Use driver	Yes
Self-protected	Yes

Table 6.2: General resume of the GostxBoard solution compared with similar requirements from Table 5.6.

From Table 6.2, some conclusions can be drawn. On the one hand, there is an important evolution in the security aspects provided by our solution contrary to all other solutions. This is realized through two essential aspects. First, the difficulty of bypassing the security solution. Indeed, in addition to protecting the keyboard content, we protect the integrity of the protected process in memory. Thus, it is no longer possible for a malware to act as a debugger to recover the cipher keys or directly the keyboard content in clear text. This defense of the protected processes is extended in the concept of *defense in depth* to the protection system itself. This last feature is implemented within a driver responsible to handle the self-protection of our system. Thereafter, the protection provided can really be effective to all the processes and not reserved to a subset (such as administrators only). Another important point is that our solution does not basically use a Dll injection mechanism which is generally considered as a potential source of instability for the host process[1515].

Taking a more nuanced view, our solution brings partial improvements on certain points compared to other solutions. This is especially the case about open-source property of the solution, so far. Indeed, the project is now published in a completely open way in its historical version. The new version could follow a more or less similar path in the future, depending on our own needs. It should be noted, however, that the technical documentation effort is greater than existing solutions today, especially with this study.

## 6.2 Limits and future work

At the end, it was also admitted to talk about the limitations of our project and the possibilities to improve it, or even propose other innovative approaches as a future work.

### 6.2.1 Limitations of our solution

Following the observations written in Table 6.2, the solution we propose is not perfect. One of the points where our solution remains perfectible compared to existing solutions is in its capacity to be deployed to protect any type of software. In its original philosophy, it is designed to be integrated at source code level in the software to be protected. Of course, it could be possible to use it via a Dll injection, but it loses one of its major benefits. It is therefore a factor that fundamentally limits its diffusion. The targeted audience is about developers and not software users. Our solution therefore provides a technical solution based on clear functional aspects, but also on usual and organizational aspects.

Another important point is that our project is founded on concessions and compromises. If we wanted to minimize the impact in terms of user experience (at least in the field of what is immediately visible), we restricted some possibilities. The use of debuggers is no longer possible because of the guarantee of integrity provided to protected processes. More directly, this means that it is no longer possible to directly debug the processes that our system protects. What are the consequences? None for the standard user who is not an IT or software development specialist. For the developer point of view, the problem is more important. Nevertheless, it is possible to mitigate this problem in this last case.

First of all, it is important to recall that our arbitration is nothing but new. For example, Microsoft uses a very similar mechanism in its protected processes (Key-Point 6.24). In addition, other existing solutions may potentially allow such protection under certain conditions (Key-Point 5.25). But rather than justifying ourselves by citing similar cases, we should perhaps provide an operational solution for developers who would use our protection system. One solution is to allow development with our SDK without this integrity protection being active. This can be a registry key but it would then be possible to reduce security by reactivating the key on a client system. More simply, in our case, any protected software must be signed to be authenticated by our system (Key-Point 6.26). That way, all unsigned software will not be subject to be protected by our system as defined in Key-Point 6.23. In order to facilitate the evaluation of the implementation of this security for developers, a registry key can then be used to restore this security for unsigned software only. That way, the registry key is used to introduce security that was not present already (and never to remove it).

Finally, most of our technical limitations have been described in section 5. More generally, the project currently lacks a little bit of maturity on certain advanced key points. For some very specific threats, specific improvements, developments and quality evaluations would still be needed. But this is a common observation performed on many software in the security industry. Nevertheless, it should be noted that most of the limitations have a proposed solution, generally tested as a proof of concept, which is based on technical elements and the official documentation of Microsoft. We are facing issues of time and the ability to test in enough different environments and contexts.

### 6.2.2 Future work

Generally speaking, our security solution was developed by integrating from the beginning the constraints specific to the development of drivers and security software. In practice, this means using all the solutions proposed by Microsoft to develop drivers [1516] (in particular *Driver Verifier* [1517]) but also deploying all the quality evaluation tests from *Windows Hardware Lab Kit* [1518]. In the last case, there is not specific test for *anti-keylogger* solutions (as there is one for antivirus called *Filter.Driver.AntiVirus*), but we can use *Device.Input.Keyboard* which is not specific to security but to keyboard devices. In this last case, we check especially if our driver does not bring instability within the drivers keyboard device call stack.

Although some developments have been carried out with strict safety and quality specifications, the fact remains that some features are now close to the proof of concept or prototype feature. The reason can be technical as with ELAM drivers (Key-Point 6.36) where you have to be authorized by Microsoft to be used in

real conditions. In this case, everything is implemented and functional, but it is not possible to deploy it on a real product. Indeed, we are not an antivirus vendor recognized by Microsoft.

Another reason may be that either a given feature is still experimental (such as with Key-Points [6.44](#) or [34](#)) or a feature might be complex to evaluate in all situations regarding the time we have (Key-Points [6.41](#), [6.42](#) and [6.42](#)). Without really being a justification, we must see here that evaluating certain of the proposed future features requires time and means that go far beyond the context of our study focused on research to reach the field of industrialization. This is a slightly different job and we have to emphasize the design of our study with realizations taking into account from the beginning the requirements of the industrial world to allow an easier integration subsequently. In a way, this is the finality of a research work that was intended, from the beginning, to meet an industrial problem specific to the antivirus security industry.

---

### 6.3 Research contributions

#### Contribution 6: Contributions of GostxBoard solution (1/2).

- ☞ GostxBoard solution proposes to use the original Windows communication channel for the secure transmission of keystrokes.
  - ✍ We evaluated the keystroke transmission mechanism through the message system driven by the raw input thread.
  - ✍ Since any application can listen in on this channel, we use ciphering techniques effectively.
  - ✍ We have explained where the problem is in the management of ciphered keystrokes by the raw input thread thanks to our technical state of the art on the management of the keyboard by Windows (Chapter 4).
  - ✍ We have tried to summarize the main advantages (and minimize the disadvantages) of existing solutions (Chapter 5).
- ☞ To our knowledge, this is the first operational solution using this mechanism.
  - ✍ We show that this design of the security solution can be effectively implemented.
- ☞ Our solution aims to provide security while keeping the user experience optimal and disturbing developers as little as possible (only two functions to call).
  - ✍ We have shown that, without any third-party vulnerabilities, an attacker cannot read the content of keystrokes within a protected application.
  - ✍ We have shown that an attacker cannot directly interfere with our security system without being detected by the system or the user.
  - ✍ More generally, we have tried a lot of known or specially crafted attacks against our system and tried to correct them as best we could.
- ☞ We provide several additional securities to ensure the reliability of our solution.
  - ✍ We attempted to manage security with enhanced requirements including administrator rights owned by an attacker.
  - ✍ We are keeping the possibility for the administrator to disable the protection solution anyway.
  - ✍ All these additional securities can be used in the context of other solutions.
- ☞ The security is enhanced on the protected application side.
  - ✍ The sharing of cipher keys (between the driver and the protected application) is improved (authentication of the applicant application with digital signature).
  - ✍ We have designed an algorithm capable of securely implementing keystroke ciphering in constant time.
  - ✍ Protection of the memory integrity of the protected process (reading and writing).
  - ✍ Hibernation and memory pagination on disk are taken into account to not leak cipher keys (vulnerability potentially shared by all existing solutions).
  - ✍ The activity of the protection can be controlled from the protected application.

**Contribution 7: Contributions of GostxBoard solution (2/2).**

- ☞ The security is enhanced on the driver protection side.
    - ☞ These protections aim to protect the driver against a possible malicious driver.
    - ☞ The protections use existing Windows mechanisms: ELAM, registry filtering API and crash-management.
    - ☞ We have proven that any presented bypass solution requires at least to be administrator and/or the ability to run a driver.
  - ☞ We have tried to propose a theoretical and innovative approach based on *HID source driver*.
  - ☞ We are aware that the solution is not perfect.
    - ☞ It remains possible for an administrator to uninstall the solution.
    - ☞ There may be attacks on our system that we are unaware of. Improvements can always be considered.
  - ☞ Part of the purpose of this study is to demonstrate the research process.
    - ☞ Studying the whole technical background where the considered problem belongs (Chapter 4).
    - ☞ Writing a state-of-the-art on all the existing threats and solutions proposed nowadays (Chapter 5).
    - ☞ Proposing a new solution taking into account our technical study allowing the study of the threats and the existing solutions (Chapter 6).
-

# Chapter 7

## Miscellaneous projects

### 1 Introduction

Within the context of this thesis, many research work and projects have been carried out, more than those presented in this thesis. It would not have been realistic to detail everything in this document. Why not? First of all because this document is already quite substantial. Secondly, because not all projects have been completed to the same degree. Some are fully completed and have been published, some have not been published, and others have been left as prototypes for further research. These are sometimes research trials, one-time achievements or improvements. They are also sometimes projects led with students that we have been in charge of. Finally, it is the inventory of the projects realized within the context of a doctoral stay in Saint Petersburg with the antivirus editor DrWeb which welcomed us during a stay divided into two three-month periods.

The presentation of the projects' achievements is intended to be minimalist. We first present the objectives and the context in which the requirements for a project arose. Then we will present the main achievements and the ideas or techniques used to do so. The explanations are succinct but detailed enough to understand the main lines. Finally, we will detail the conclusion the project as well as its interest and possible impact if there was one.

The structure of this chapter is divided into three main sections. At first, we have the whole of the achievements realized on the two doctoral stays. Then we have all the work done with the students. And finally, we have the personal or collaborative work with other researchers.

### 2 Doctoral stay achievements

In practice, our doctoral stays was articulated over two periods of three months each. The first one was held at the very beginning of our studies in summer 2017. The second was carried out over approximately the same period the following year, in 2018. In both cases, the stay was conducted with DrWeb Ltd in Saint Petersburg under the local direction of Igor Zdobnov, Chief Malware Analyst.

Our stay was realized on a voluntary basis with this antivirus editor, the funding of our thesis covering our needs. All the projects were carried out with a research approach and therefore with the possibility of evoking, if not the technical details, at least the main outlines of what was achieved. In the following, we divide the projects in two groups. On the first hand, published projects and on the other hand, unpublished projects.



## 2.1 Published projects

### 2.1.1 New way to inject Dll and evaluation antivirus detection resilience

#### Key Point 7.1:

- ☞ A Dll injection can be summarized as the injection of code (through a Dll) from a process A to a process B.
  - ✍ There are a limited number of Dll injection techniques.
  - ✍ They are known to antivirus software that may try to prevent them.
- ☞ We propose a new Dll injection technique to escape detection.
  - ✍ This technique uses the *Delay-loaded Dll* mechanism.
  - ✍ Legitimately, it is possible to load a Dll only on the first call to a function when developing with Visual Studio.
  - ✍ In practice, this means calling the `LoadLibrary` and `GetProcAddress` functions.
  - ✍ We hijack in memory the structures managing this mechanism to realize our new Dll injection.

In the context of the internship, an evaluation of antiviral products was considered. In a practical way, this answered the talk we had submitted to the "15<sup>th</sup> Nuit du Hack" conference in Paris 2017 [1519]. The objective of the presentation was to show various malicious attacks (from the simplest to the most elaborate ones) capable of bypassing the security carried by various antivirus software. First, we proposed different techniques to deactivate antivirus software, at different levels. On the first hand, using kernel-mode (ring-0) rights, we reversed internal filtering API provided by Microsoft (such as mini-filter drivers [1279], registry filtering with `CmRegisterCallbackEx` [1286] and so on) to silently remove callbacks setup by third party drivers. On another hand, we used application-level techniques to disable antivirus software by manipulating the registry. At the end, using different methods, we were able to implement some malware attacks (inserting triggers for malware automatic execution at start-up) on systems with no antivirus software able to detect us [1520]. The main objective was to show that although antivirus products are still necessary to fight against malicious threats, they were far from being completely sufficient.

This presentation was an opportunity to show a new Dll injection technique. A Dll injection technique is one of the camouflage techniques used by malware to execute code in another process than the one holding the original malware. The objective here is multiple. On the one hand, it allows to divert the malicious action to a third party process (contributing to the stealth of the malware by blaming another process). On the other hand, it allows to design actions "only in memory". Such actions are sometime called "*fileless attacks*". In the latter case, it allows to bypass some scans performed by antivirus software. It is therefore a malicious trick regularly used by malware.

Formally speaking, there are many different ways to perform a Dll injection. Such techniques can be divided into two main categories. On the first hand, by creating a process from scratch in order to inject it with malicious code. In this case, the innovation is generally focused on the way to inject the code into the targeted process. Historically speaking, *Process Hollowing* [1521] is the method that will later inspire to new techniques. The principle with *Process Hollowing* is to create a process (for instance the Windows calculator) in "suspended" mode so that nothing runs from the beginning. Then, by removing its content, we replace it by the content of another executable file before giving the hand back to the suspended process. That way, the process executes the code stored in another executable file without touching it directly. In practice, we are using `VirtualAllocEx` [1174], `ReadProcessMemory` [1175] and `WriteProcessMemory` [1176] functions. New methods called *Process Doppelgänger* [1522], *Process Herpaderping* [1523] and more recently *Process Ghosting* [1524] reuse the same principle with the difference of how to inject the code into the newly created process.

On the other hand, the historical Dll injection uses an existing process for the injection. In this case, the innovation is generally focused on the way to execute the injected code. Again, there are many approaches

to proceed [1525, 1526]. In general, the procedure always follows the same logic. First, a malicious process gains access to a targeted process (for instance with `OpenProcess` [1173]). Then, it modifies the target's process memory (for instance with `ReadProcessMemory` and `WriteProcessMemory` functions) in order to inject code or parameters able to be executed latter in a remote function call. Finally, the malicious process find a way to execute the injected malicious payload. Generally, to proceed, it is `CreateRemoteThread` function [746] executing `LoadLibrary` [753] function to load a Dll path in a dedicated thread [1172]. This procedure is resumed in Figure 7.1.

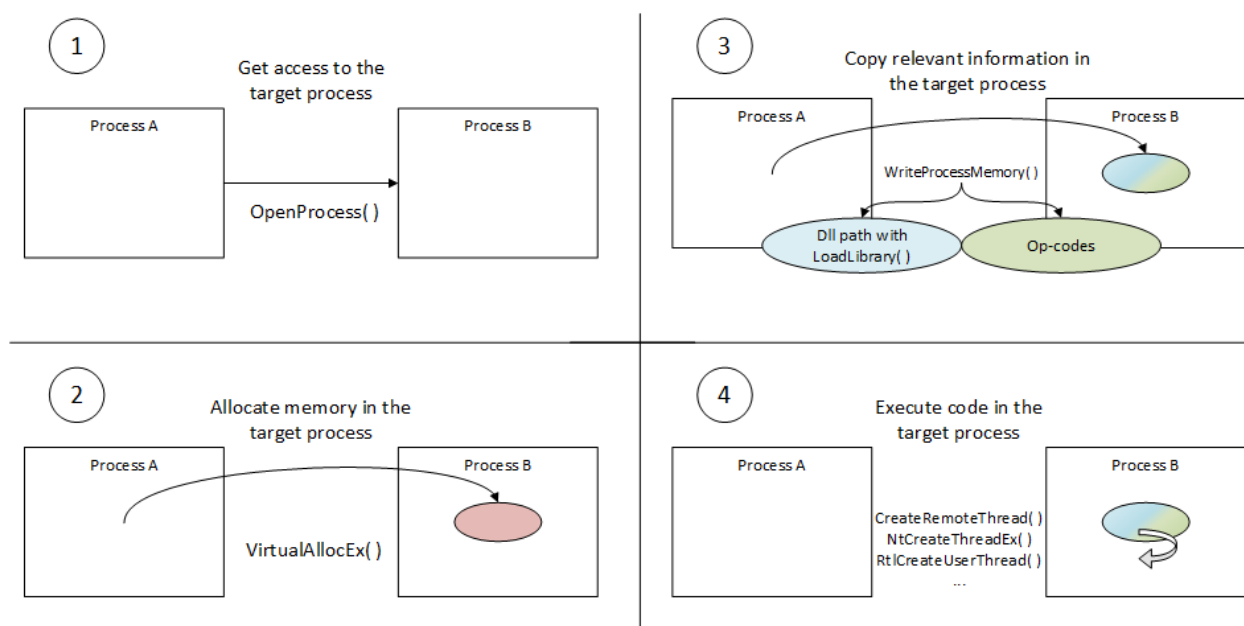


Figure 7.1: Illustration of the different steps to perform a *classical* Dll injection procedure.

This generic approach is well known to antivirus vendors who can look for it. In order to escape detection, original variants are introduced to break the detection patterns. Our new approach aims at proposing a solution using a different mechanism to provide an alternative to the already known procedures. From a technical point of view, it is complex to offer a new approach on how to modify the code injection in a targeted process (contrary to what is performed with *Process Hollowing*). Therefore, we sought to offer a new way to execute previously injected code. To do this, we hijacked the "delay-loaded DLLs" mechanism [1527].

Delay-loaded Dll is a compilation option [1528] introduced by Microsoft Visual C++ 6.0 [731]. It allows to load the specified Dll only on the first call by the program to a function in that Dll. More directly, it means that the developer writes in its own source code a regular call to a Dll exported function (Figure 7.2). But internally, at linking time during compilation process, this function call is modified so that a Dll loading is performed the first time an exported function from that Dll is called (the other times, the call is almost transparent to a normal exported function call). In practice, to reproduce such a behavior, it would be required to write the code given in Figure 7.3.

```
OneFunction(arg1, arg2); pFunction = GetProcAddress(LoadLibrary("DelayedDll.Dll"), "FunctionName");
                          pFunction(arg1, arg2);
```

Figure 7.2: Code written by the developer.

Figure 7.3: Equivalent code the first time the function is called.

How does it internally work? Executable files, under Windows, follow the MZ-PE benchmark defined by Microsoft [1529]. To keep things simple, the file is divided in headers which internally reference sections and directories. The "Delay Load Import Descriptor" directory is the one which rules the delay loading proce-

ture. Accessible through the 14<sup>th</sup> (called `IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT`) entry in the `DataDirectory`, this directory drives the mechanism of delay loading of a Dll. The `IMAGE_DELAYLOAD_DESCRIPTOR` structure is referenced in this directory. This structure is not formally documented by Microsoft but it is accessible in the header files provided with the Windows Kit [1508] (in `winnt.h` file). We reproduce the content of the structure in code 7.1.

```
typedef struct _IMAGE_DELAYLOAD_DESCRIPTOR {
    union {
        DWORD AllAttributes;
        struct {
            DWORD RvaBased : 1;           // Delay load version 2
            DWORD ReservedAttributes : 31;
        } DUMMYSIRUCINAME;
    } Attributes;

    DWORD DllNameRVA;                     // RVA to the name of the target library (NULL-
    terminate ASCII string)
    DWORD ModuleHandleRVA;                // RVA to the HMODULE caching location (PHMODULE)
    DWORD ImportAddressTableRVA;          // RVA to the start of the IAT (PIMAGE_THUNK_DATA)
    DWORD ImportNameTableRVA;             // RVA to the start of the name table (
    PIMAGE_THUNK_DATA::AddressOfData)
    DWORD BoundImportAddressTableRVA;     // RVA to an optional bound IAT
    DWORD UnloadInformationTableRVA;      // RVA to an optional unload info table
    DWORD TimeDateStamp;                  // 0 if not bound,
                                           // Otherwise, date/time of the target DLL
} IMAGE_DELAYLOAD_DESCRIPTOR, *PIMAGE_DELAYLOAD_DESCRIPTOR;
```

Code 7.1: Content of the `IMAGE_DELAYLOAD_DESCRIPTOR` structure.

The delay load descriptor structure is a special *Import Address Table* (IAT) distinct from the official IAT which is supposed to list all imported functions for a delayed loaded Dll. Internally, an array of `IMAGE_DELAYLOAD_DESCRIPTOR` structures references all the delayed loaded Dlls referenced in the program. Among the structure's fields, there is `DllNameRVA` which references the name of the Dll to load. Comparing with a regular IAT structure, the field `ImportNameTableRVA` corresponds to *OriginalFirstThunk* field and `BoundImportAddressTableRVA` field acts as *FirstThunk* which is used to store address of imported function at runtime. The field `ImportAddressTableRVA` points to the official IAT of the executable. The `ModuleHandleRVA` is a spot which fits the size of an address to store the handle (in fact, the base address) of the Dll which will be loaded. Explicitly, it corresponds to the return value of `LoadLibrary`. About this structure, more information can be found at [1530]. A simplified view of the structure is provided in Figure 7.4.

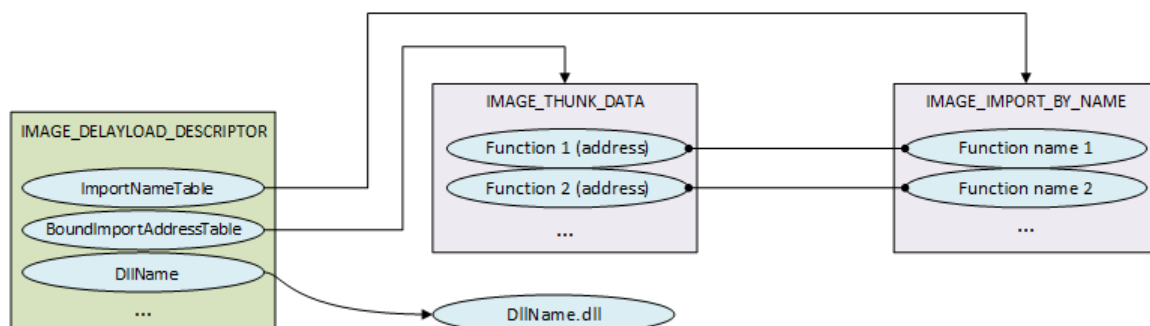


Figure 7.4: Simplified view of the interactions present in `IMAGE_DELAYLOAD_DESCRIPTOR` structure.

This set of structures is used by code added behind the call to the delayed function. More directly, C code is inserted directly and silently at compilation time when a function of a delayed library is called. It is not like a macro which replace source code by other. It is more about a generic function called instead of the requested one in order to load the delayed Dll and resolve requested function's address to call it after. This code is included

in Visual Studio and freely readable by anyone in `delayhlp.cpp` and `delayimp.h` [1530].

During a call to a delayed function, the compiled code jumps to another call which uses the content of a variable to know where to jump. Technically, this variable is used in the IAT to store the resolved address of targeted function. This is where the address of the delayed function will be stored once it will have been resolved. Of course, in the case where the Dll would not have been loaded yet, this address does not refer to the one of the exported function. Instead, it refers to an unconditional jump which finally calls `__tailMerge_Xxx_Dll` (where "Xxx" is used for the delayed loaded Dll name) function. This function is responsible to save all relevant registers from the original call of the function and most specifically those used as parameters (according to x86/x64 architectures — the floating-point registers are not saved on any platform) [1527].

The loading procedure is made by `__delayLoadHelper2` function which is supposed to load the Dll and resolve the required function. With the address of the exported function returned, the initial call can be performed once all of the original registers have been restored. There is no secret about `__delayLoadHelper2` function since its code is available thought `delayhlp.cpp` file. Formally speaking, it uses the regular API (with `LoadLibrary` and `GetProcAddress`) to resolve the Dll importation and initialize internal structures described previously for subsequent delayed function calls.

Finally, the Delay-loaded Dll mechanism is a "hidden call" to the `LoadLibrary` function using a structure inserted at compilation time in the executable file. Different injection techniques presented here enjoin to hijack this procedure to replace a delay-loaded Dll by a malicious one. Technically speaking, it will be necessary to write into the memory address space of the targeted process, as with any historical Dll injection techniques. The main difference with existing techniques is where the writing operation is done. In practice, two possible locations are available.

On the one hand, the easiest way to proceed is to replace, at runtime, the name of the delayed Dll stored at the address under the field `DllNameRVA` in `IMAGE_DELAYLOAD_DESCRIPTOR` structure. This technique is more or less equivalent to a classic IAT hooking technique [1531], with the difference that the loading of the Dll is targeted rather than the access to some given functions. In practice, the use of `VirtualProtectEx` [394] allows to change the rights of a memory page at a given address in a targeted process. Once read and write rights have been set (and can bet restored at the end), the use of `WriteProcessMemory` allows us to change the Dll name.

With this first technique, two restrictions can be observed :

- The first is about the length of the Dll name. Since we are modifying the original Dll name with no reallocation<sup>1</sup>, we must respect the maximum size of the original name. In addition, only the Dll name is stored in the executable file, not the path where the Dll is located. In practice, the Dll is located on the disk by an algorithm specific to Windows [1532]. To artificially specify the location of the Dll, it is possible to change the environment of the remote process (such as the current directory, for example), an operation that does not require any additional right than modifying the memory, as is the case for the Dll name.
- The second restriction involves the functions exported by our injected Dll. Because we replace an existing Dll supposed to be called through a given function, we should provide the same names and prototypes for functions. That way, this situation can perfectly suit to use this injection as a proxy Dll.

On the other hand, it is possible to provide a more flexible technique by hijacking the whole `IMAGE_DELAYLOAD_DESCRIPTOR` structure in the executable file. The idea is to provide a new structure to load our injected Dll and keep the original structure in a pre-allocated memory to load the original Dll thereafter. This operation requires to allocate memory in the targeted process (with `VirtualAllocEx` [1174] function) in order to save the original `IMAGE_DELAYLOAD_DESCRIPTOR` structure (with `WriteProcessMemory` function [1176]). This

---

<sup>1</sup>Of course, it would be possible to allocate memory with `VirtualAllocEx` [1174] function. But such operation would require to change requested rights when obtaining access to the targeted process with `OpenProcess` [1173]. Without requiring extended rights and remote allocation memory function, our method could be harder to detect by regular antivirus software since it does not follow usual requirements of existing Dll injection methods.

structure will be used during the injection to load the original delayed Dll, keeping the original behavior of the targeted process and ensuring the stealth of our operation.

The advantage of our method is that it is no longer necessary to have at least as many exported functions as the targeted Dll (unlike the first method). This requirement forced to plan an injection Dll with many (and potentially empty) exported functions (such a Dll could be suspicious) or to target only a delayed Dll whose number of exported functions was known beforehand. In practice, by updating the `IMAGE_DELAYLOAD_DESCRIPTOR` structure, it is possible to perform a Dll injection with only a single exported function (more is possible, but not necessary). Technically speaking, we design a fake `IMAGE_DELAYLOAD_DESCRIPTOR` which references as many functions as the one exported by the delayed Dll hijacked but all referencing only a single function in the injected Dll. That way, when calling a delayed function in the process, the procedure loads our Dll and resolves the function name as expected. But whatever is the function name provided in our fake descriptor, this one always matches the same generic function in our injected Dll. Resolution of the function is represented in blue on Figure 7.5.

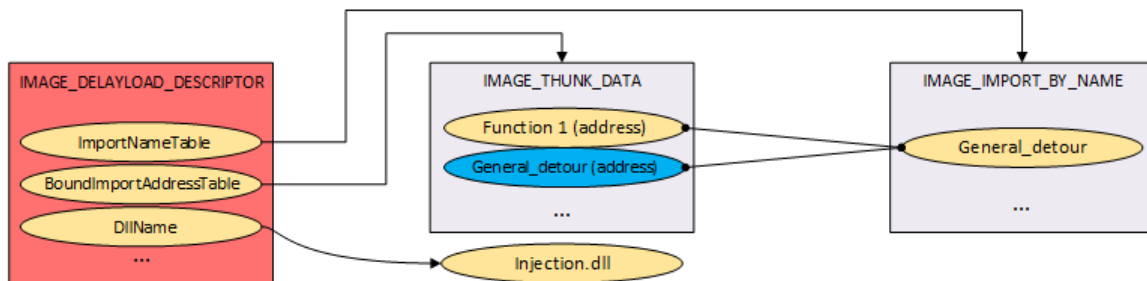


Figure 7.5: Replacement of the original `IMAGE_DELAYLOAD_DESCRIPTOR` structure by the one used for injection purpose. Note that the generic exported function by the process is resolved in blue on the figure when resolved by the process.

Our generic function is supposed to load the original delayed Dll, which could raise a question. We are loading the original delayed Dll in this generic function and not in the entry point of our injected Dll to follow the requirements ruling Dll entry point management [1001, 1002]. This choice is perfectly justified because we have the guarantee that our generic exported function will be called whatever happens by the delayed procedure compiled in the process (thanks to `_delayLoadHelper2` function). The main steps of the generic function are reported in Figure 7.6 :

1. The objective of the generic function is to find the original descriptor in memory. Then, the procedure aims to load the original referenced Dll and solve the initially targeted function, even without knowing its name (because it has been deleted for the call of our generic function). In practice, this means going through the delayed IAT to identify which function has been initially solved (normally, all other addresses should be set to zero since they have not been resolved yet).
2. The location of the address resolved corresponds to the same location in the original delayed descriptor, allowing the retrieve the original function name called. That way, we are able to reset the address of the original function in the delayed descriptor.
3. Once the original function address has been resolved, the list of functions imported in the current delayed descriptor is updated with the backup of the original descriptor. That way, it allows the following calls to be correctly performed.
4. For stealth reasons, it is possible to update the name of the imported Dll to appear as the original delayed Dll.
5. At the end, the original function targeted by the process is called to be transparent.

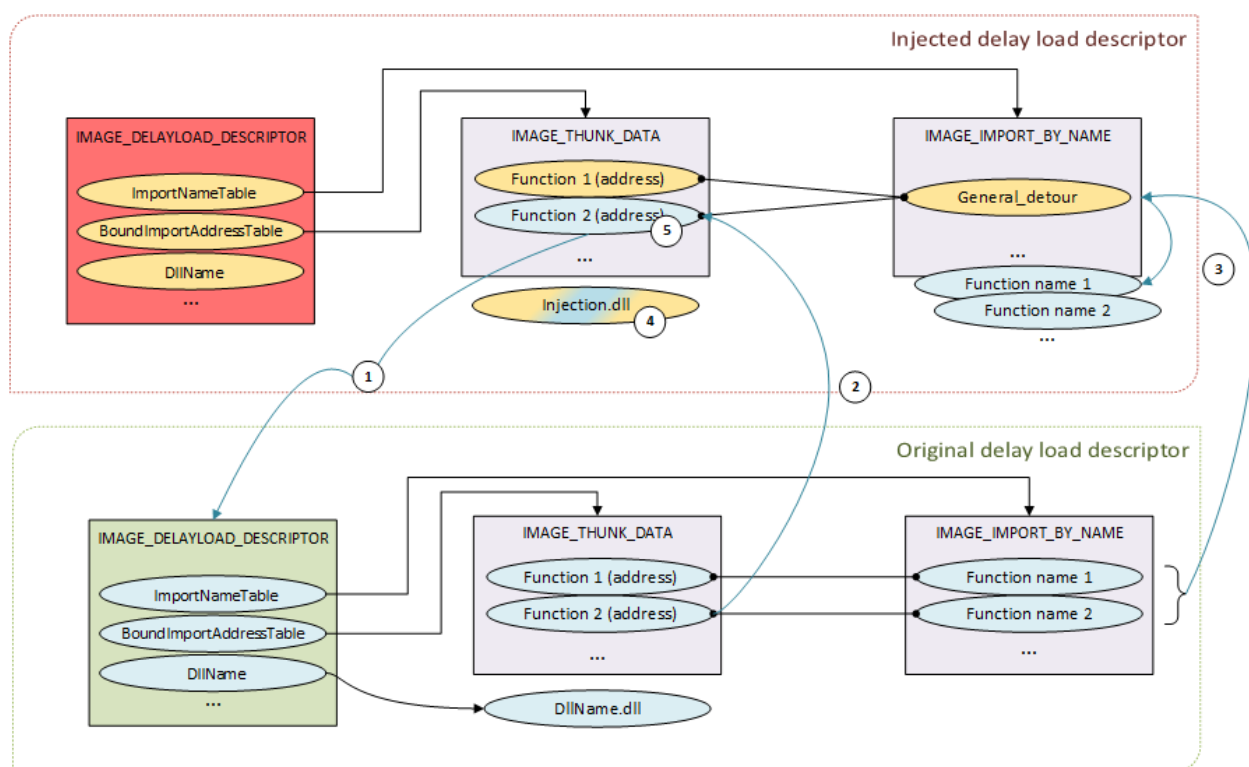


Figure 7.6: Injection procedure based on the generic function used to interface the first delayed function call.

The use of this method requires to get additional rights (possibility of allocation in addition to writing to the memory of a remote process) but offers much greater flexibility (target of exported functions and a reduced list of exported functions). In itself, this injection method does not revolutionize the concept. But it provides a modern way to do it by exploiting specific structures in executable files. It is in fact a new possible path that it is now possible to take. In terms of the security it provides, the result remains ambivalent. Tested with *Virus Total*<sup>2</sup>, an executable file holding our injection method is rarely detected (only 5 detections<sup>3</sup> for more than 60 antivirus software tested).

But the same applies for other Dll injection methods. The explanation may be twofold. On the first hand, Dll injection is not in itself a malicious mechanism and it can sometimes be used for legitimate (though rare) reasons. Only specific antivirus are trying to detect them directly and they are more likely to perform a detection if a Dll injection is followed by another suspicious action (as a conjunction of suspicious actions). On the other hand, the way we tested our injector may have reduced the actual detection rate. That is to say, we tested it without a really malicious Dll used for the injection (indeed, we should avoid confusion of the detection between the malicious payload and the injection technique).

In conclusion, this new injection method has shown that it is possible to find a new way to perform a Dll injection, a mechanism often used by malware. Knowing more about these mechanisms can help to better detect them and therefore potentially detect malicious programs. Moreover, our work has shown the use of delay loaded Dll other than as an ease of writing code for a developer. This is a way to raise awareness about the security of the code that could be sometimes sacrificed by some implementation mechanisms (and not only the delay loaded Dll) by understanding how these mechanisms are working behind the stage.

<sup>2</sup><https://www.virustotal.com>

<sup>3</sup>Only very specific antivirus (reprinting a minority on the market) detected our injection: SecureAge Apex, Cybereason, Cylance, Cynet and MaxSecure.



**Contribution 8: New Dll injection contribution.**

- ☞ We presented a new way to perform an injection Dll.
  - ☞ This injection is not detected by most of the antivirus software.
  - ☞ This injection uses the classical mechanisms of injection but uses *delayed loaded Dll* technology.
- ☞ This work was presented at the "Nuit du Hack" conference in 2018 in Paris.
  - ☞ We also showed many methods to bypass the security of some antivirus software.

## 2.2 Unpublished projects during the doctoral stay

**Resume 35:**

- ☞ During our doctoral stay many projects have not been published.
  - ☞ This may be due to a lack of material to publish (projects are not large or new enough).
  - ☞ This can serve to maintain some competitive advantage for the host company.
  - ☞ Some technical information may have been omitted for the sake of brevity or confidentiality.

During the stay, many projects were carried out without leading to a scientific publication. Instead of, the objective was to bring some gain to the company's antivirus products or to its malware handling procedures. That way, we propose a certain competitive advantage but also a possibility to reinforce the commercial offer of the company by maintaining a certain level of research and development. There is therefore a consensus to keep a certain competitive advantage and therefore not to reveal everything publicly. This also explains why this work has never been published, although rigorous documentation has always been established (for maintenance reasons or to justify certain technological choices). Nevertheless, for confidentiality reasons, some technical details or solutions will not be given, but simply mentioned. The methodology of work being here perhaps more interesting than the final result obtained, it is also there that we will focus.

### 2.2.1 Protect important folders for specific software

**Key Point 7.2:**

- ☞ We have developed a driver to fight against ransomware threats.
  - ☞ The purpose of this driver is to allow writing operation in protected folders only for a list of allowed software.
  - ☞ All targeted folders and software are configurable.
- ☞ Microsoft provided a "*controlled folder access*" [1533] feature in October 2017.
  - ☞ With similar goals and features, Microsoft's solution is native to Windows 10.
  - ☞ This shows that identical needs can promote identical solutions on both sides.

The objective of this project was to respond proactively to the threat posed by *ransomware* malware. A ransomware is a malicious program whose objective is to prevent any access to the user's data (using secure cryptographic means, able to prevent any bypass). Generally speaking, from the first versions raising in 1989 [1534], a ransomware attack takes place in three steps [1535]. The first step aims at infecting a targeted machine (we are therefore in a propagation logic which is not different from the strategy used by some worms [1536]). In a second step, the objective of the malware is to completely cipher the user's files to prevent any access. The ciphering can be done on some targeted files (we talk about *Encrypting Ransomware*) or on the operating system itself (we talk about *Locker Ransomware*). In the last step, a message is displayed to the user's screen urging him or her to pay a ransom in order to regain the initial access to his or her information. A variant is



also to exhort user's money in order to not release any private information taken from the victim's machine. Usually, the ransom is expected to be demanded in digital cash formats such as "bitcoin" [1537]. Such activity is one of the most lucrative one in "malware business" [1538].

It is to fight against this type of threat that we have realized this project. The classic approach in fighting malware is usually to characterize a behavior in order to decide whether a given program is legitimate enough to run. Unfortunately, the preemptive approach does not work efficiently in the case of ransomware threat (as with keyloggers, see Chapter 5). Why is this a difficult family of malware to detect? Maybe simply, like in the case of keyloggers, this type of malware finally uses quite legitimate and low-cost actions to achieve its goals. More directly, notwithstanding the propagation mechanism which is usually the most malicious action (generally based on a zero-day vulnerability or anything close, but independent of the ciphering payload), the malware only seeks to open files, read them and rewrite them. Of course, when rewriting, the originally read data has been ciphered, usually with the Windows Cryptography API [1359]. If the cipher key is ultimately the object of the sale between the attacker and the victim (when it is actually sold, some malware pretend to sell something but they are just taking money), it is usually transmitted over a legitimate internet connection. Basically speaking, there is nothing wrong with reading files or using the cryptography API (although large-scale use of these means over a relatively short time is a possible indicator of ransomware activity, even if the detection would be performed a bit late).

Thus, detection is a complex issue in itself. Note that antivirus editors must also face false-positives (detection of clean programs as malicious) and therefore not incriminate software wrongly. For example, software like 7-Zip or WinRAR (both used for compression purposes and able to generate ciphered compressed archives) behave in a similar way to ransomware and should not be blocked. That is why we have focused on preventing the damaging effects induced by such type of malware. Typically, a ransomware attacks the user's personal documents (photos, movies, text documents, etc). The objective here was to restrict access to the user's documents to a subset of authorized and legitimate software for this purpose. This subset of software and the list of protected directories can be defined by the user through a graphical interface that initializes a configuration in the Windows registry (in a key specific to the driver). The driver then loads this configuration used to manage the access to files (and actions) guaranteed for each software that would try to access a file in a protected folder. From a general point of view, this restrictive access objective is represented in Figure 7.7.

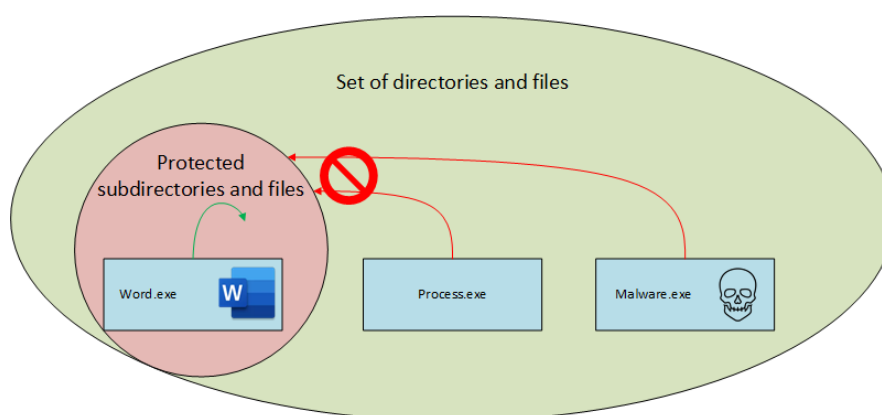


Figure 7.7: General representation of the protected folders against unexpected software trying to access files.

In practice, our protection system is based on a driver using mini-filter technology [1279]. Loaded at "*FSFilter Anti-Virus*" altitude group [1373, 1374, 1539], our driver sets a first callback via `PsSetCreateProcessNotifyRoutineEx` routine [1503] able to be notified each time a process is created or deleted (Key-Point 6.43). More directly, this callback is used to maintain a double linked list of active processes in memory for our own driver (the ones used by Microsoft internally are not documented, this is why we are crafting our own one).

In addition to the list of processes kept in memory, our driver filters all activities undertaken on all the different file systems and volumes that are mounted by the system [1540]. In practice, we are setting several

callbacks able to filter different types of operations [1541] (mostly creation, read, write, delete, query or set information on a specific folder/file). Such callbacks can be notified before (called *pre-callback* [1542]) or after (called *post-callback* [1543]) a specific operation [1544]. On the first hand, in the case of *pre-callbacks*, it allows to potentially modify on-the-fly parameters before the operation happens [1545] or to refuse an access to a given file for a specific operation. On the other hand, in the case of *post-callback*, it is possible to update output parameters or output buffers as setting a *context* [1546] on a specific file to "follow", thereafter, the file for operations subsequently requested.

In practice, we allow to filter files access in a protected directory as a whole (a process can access all files in the sub-directories) or for specific rights with specific file's extensions (for instance, a given process can only read files with a given extension but not modify them). This is done thanks to our ability to identify running processes and matching rights defined in the driver's configuration (setup by the administrator) with the different actions filtered by mini-filters' callbacks. A general summary is given in Figure 7.8.

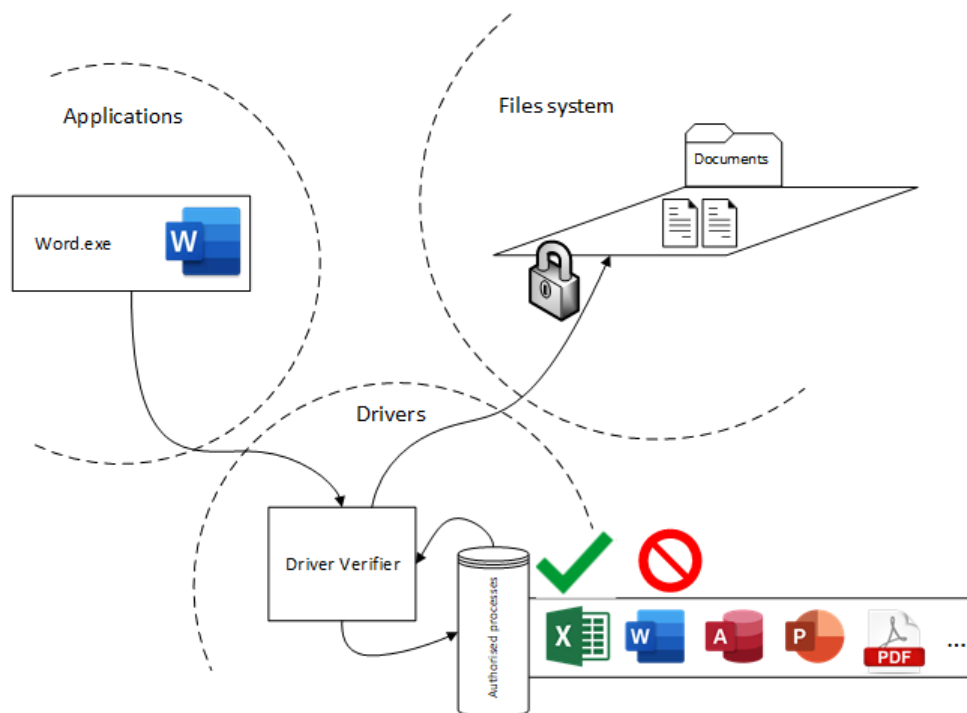


Figure 7.8: General resume of the anti-ransomware directory protection solution.

We can use an example with Word software. In such a case, we are protecting "My Documents" from any third party modifications except by Word. Indeed, the word processing software will be allowed to come and modify ".docx" documents in the "My Documents" folder, as illustrated with Figures 7.9 and 7.10. We can see differences in access rights set up by our driver for Word and any third party software (including malware since they are not properly identified as authorized software).

When we realized this project during our first doctoral stay, we have to admit that ransomware threat was (and still is, in a way) a major threat. More directly, attacks like Wannacry (March 2017), Petya (March 2016) or NotPetya (June 2017) were contemporary to our internship. The active fight against this type of threat and — by default — the reduction of their capacity to cause harm, was a need clearly identified by the entire antiviral community. The ideas were therefore in the air and we were not the only ones to have this approach. If our project was able to see the light of day as early as June 2017 and be integrated into Dr Web's product, Microsoft published a feature that was in every way similar (on its objectives and means) in October 2017 [1547]. This feature called "controlled folder access" [1533] is now in Windows 10 and used to protect valuable data from malicious apps and threats, such as ransomware. The protection is performed by checking applications against

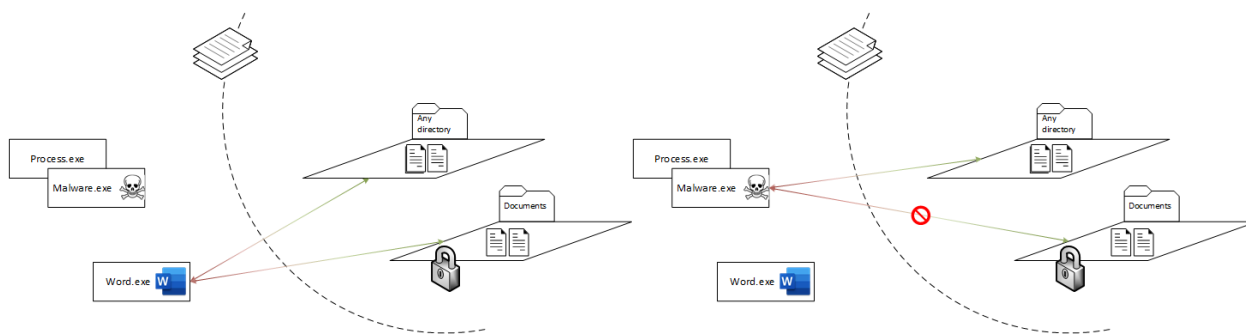


Figure 7.9: Regular access provided to Word.

Figure 7.10: Access denied to third party software.

a list of known, trusted apps, to get access to files in a list of controlled folders. Such a list of protected folders is specified by administrator and typically holds commonly used folders, such as those used for documents, pictures, downloads, and so on...

Without being able to claim anything about this idea (the industry is not always focused on publishing but on meeting customer needs), it is interesting to note that our approach (and probably the outline of its underlying implementation) has been taken up by Microsoft which included it directly in Windows 10. In the end, there was no real need to keep an equivalent feature in the antivirus when it is natively provided by Microsoft. This explains the removal of the feature from the antivirus, although we were able to protect customers for a small slice of time before Microsoft added its feature.

## 2.2.2 Polymorphic packers

### Key Point 7.3:

- 🔊 The objective of this project is to classify different families of polymorphic packers from output samples.
  - 👉 A packer is a program that takes another program as input to produce as output a new program embedding the first program.
  - 👉 It is used to compress some executable files or to make the analysis of programs more complex.
  - 👉 A polymorphic packer is a packer that produces different outputs for the same input.
- 🔊 We have realized a classification system based on an analysis from an execution simulation within an emulator.

A packer is a computer program whose action is to modify a given executable file to store it in another executable file. More directly, a packer is a software that can mutate a binary file into another executable [1548]. The new produced executable is called "*packed executable*" or simply "*packed*". The new packed executable is able to execute the original one by extracting it (in memory or directly on the hard driver) and then running it (Figure 7.11). That way, a packer preserves the original file's functionality while offering another content on the disk.

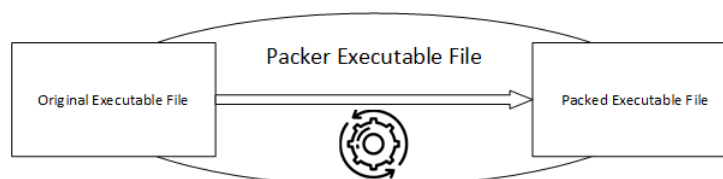


Figure 7.11: **Packing** process with an **original** executable file packed with a **packer** producing the **packed** executable file.

The result is a new executable file holding the original executable file plus a payload, executed at the beginning of the packed file. This payload aims to extract the content of the original file. The extraction procedure of the embedded executable is called *"depacking"*. Note that an additional payload can be including in the packed executable file. This one aims to detect if the current packed process is under analysis, either by a debugger or any analysis tool<sup>4</sup>. Packers which embeds such feature are rarely used for legitimate purposes but instead by malware. A general view of a packed file is provided in Figure 7.12

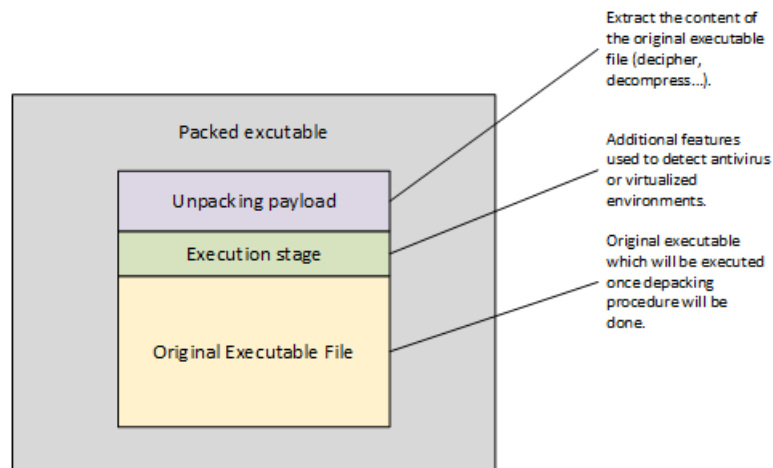


Figure 7.12: General view of a packed file. This one uses the depacking payload to extract the original file in memory before executing it. That way, the original file is hidden in the packed file.

Packer software is used for two main reasons. The first one is to reduce the size of the packed file, since packers originally used compression techniques to store an executable into another one. With such a technique, the depacking procedure aims to decompress the original executable. But in addition to reduce the size of binary file, packers can be used to prevent any analysis of the original executable which is packed. Indeed, because the content of the original file is embedded inside the packed file, this one is hidden from the system point of view. More directly, if an antivirus software tries to analyze a packed file, it only gets access to the depacking payload and not directly to the content of the original file. This can be used for good reason to protect intellectual property or avoid broadcast of cracked versions of a program. But it can be used to avoid analysis, reverse-engineering and other practices by antivirus company too. According to [1549], about 80 % of malware use a packer to escape antivirus detection.

To proceed, antivirus software can try to extract the original content of the packed file. In a way, it aims to mimic or control the depacking procedure in order to extract the original executable file packed. That way, it becomes possible to analyze the original file. Technically speaking, doing this procedure generically can be a complex task, even if solutions could exist. Another approach lies in designing a tool for each packer, able to specifically extract the original executable file per packer.

The problem with this approach is that packer designers are not particularly willing to let it happen. This is why packer authors like to combine many obfuscation methods including anti-debugging techniques and tricks to prevent further analysis of their embedded files. To make a long story short, it is a kind of *"arms race"* between packer designers and antivirus editors who try to break the protections in place. Overall, the war cannot be won by either side. The reason is simple. On the one hand, packers must somehow let the embedded code run, requiring little if any conditions on the user's side. Thus, there is always a path to execute the original file. And not matter if the packed file is stored on the user's hard drive or on a remote server... On the other hand, antivirus editors are forced to follow the evolution of new techniques, being almost unable to anticipate the new tactics and strategies implemented in packers. And any change to an existing packer may force the antivirus vendor to significantly revise its existing work to match the modification.

<sup>4</sup>Such a topic has been already covered in Chapter 3.

To allow the antivirus depacking strategy to work, apart from having a tool capable of performing the extraction, it is necessary to be able to recognize which packer is used by which packed file. The detection of a packer is usually done on its depacking payload, since it is generic to any embedded file. In a way, this is the *fingerprint* of a packer. From there, once the recognition is done, the procedure is simple. Either we have an extraction tool and we can start the procedure to analyze the original file. Or we do not have one and we need to create or adapt the extraction tool. And it is this extra workload that packer editors intend to use on to make life more complex for malware analysts.

A very efficient strategy to proceed is to use a *polymorphic packer*. Polymorphic technique is a technique used by malware to change their identity on each time they attack a new system [1550]. Packers editors reused this approach and adapted it for their own needs. Since it is the depacking payload that is targeted for detection, it is then the depacking payload that is impacted by polymorphic mutation. Thus, each payload at the entry point of the packed file is generated more or less randomly by the packer and it becomes very difficult to recognize the identity of the packer.

The mission was to create a tool capable of recognizing polymorphic packer families from packed files, in order to gather them into many categories. Once the gathering would be done, then it is possible to perform extractions based on generic tools able to manage each *family* of polymorphic packer. Of course, we do not have access to the polymorphic packers directly. These are tools that are carefully guarded by malware vendors<sup>5</sup>. In any case, having access to it is not really necessary. Only packed files are needed. After all, they are available in large numbers because they are generously distributed on the web.

For the sake of brevity, we can summarize the problem in the following way. *Our goal is to create an unsupervised packer ranking system (because we do not know the different packer families and even less their exact number) from packed files only.*

Reverse engineering a few samples of such programs is very instructive. Two things are immediately visible:

- On the one hand, the diversity of strategies before accessing to the original file shows that there is a great diversity of packers and approaches in the wild.
- On the other hand, if the diversity is strong between the families, we note that some packers have certain similarities, although different in their execution. Some strategies are often used, even if they are declined in different ways (spaghetti code, new depacking technique, anti-debugging techniques, self-modifying code, etc). The idea is to use these similarities to create different polymorphic packer families.

In practice, depacking payloads are not tools that like to be analyzed. Thus, they seek as much as possible to play on the whole state-of-the-art of the evasion techniques. One way to try to bypass this security is to put packed files in an extremely special analysis environment: *emulator*. Emulators are programs that mimic the behavior of a regular CPUs, instruction by instruction. In practice, they implement a decompiler that reads every instruction in the program and have a table of functions, each function representing the execution of an instruction. Thus, they can “emulate” the behavior of the CPU, instruction by instruction.

Thus, it is possible for us to process instruction by instruction what is executed by the packed file (and possibly counter some of its evasion techniques). Of course, since the emulators are as close as possible to the CPU behavior (they totally ignore the notion of operating system), they are not easy environments to master. Dr Web teams wanted to test at what was a potentially interesting emulator, a rising star in the field, called *Unicorn* [196]. Presented at Black Hat USA 2015, this emulator is quite interesting, efficient and stable. As a side result of the project, the evaluation of Unicorn emulator was required.

---

<sup>5</sup>As a side note, it is interesting to note that any executable packed with a polymorphic packer should be safely considered as malicious. The justification is that such a packer explicitly seeks to escape antivirus scanning by refusing to disclose the identity of the packer that has been used. Commercial software are not prone to use such packers at the best of our knowledge. But in case of, such software could be considered as highly suspicious. For short, the problem is not about to know whether the packed executable is malicious or not (there is almost no doubt about that point), but how to deal with what has been embedded with it.

---

Hence, the work was to take the emulator in hand, to configure it in such a way that it was fast, efficient and could emulate the virtual execution environment that could make the packed file believe that it was running on a Windows operating system<sup>6</sup> (a minimalist kernel had to be rewritten to proceed). Thus, it becomes possible to execute the instructions of packed programs. Note that emulators is a very good tool against self-modifying code used to hide the real purpose of the depacking payload. Indeed, some advanced packers like to *pack* their own depacking procedure, hidden it with another packing procedure...

The classification is based on the identification of the common points of polymorphic packers. Indeed, to introduce "*polymorphism*", packers use a set of techniques (useless code, depacking sub-functions that can be executed in any order, detection of analysis environment and so on), so that are randomly generated as the initial payload of the packed file. Even if there are variations from one packed file to another one for the same packer, the set of techniques to be shuffled is limited. In addition, packer authors are usually forced to pre-generate the set of codes that they will then mix to create a "new" payload. More directly, they have a finite list of compiled codes, probably written in assembly, that are inserted randomly (position and occurrence) in the payload. The links between the different codes is random but the final result is guaranteed. This architecture is found in many cases since it is one of the few ways to realize such polymorphic packers. And that is what may cause these packers to lose the anti-classification war.

Our classification tool is resumed in Figure 7.13. This one is divided in several steps described as follows.

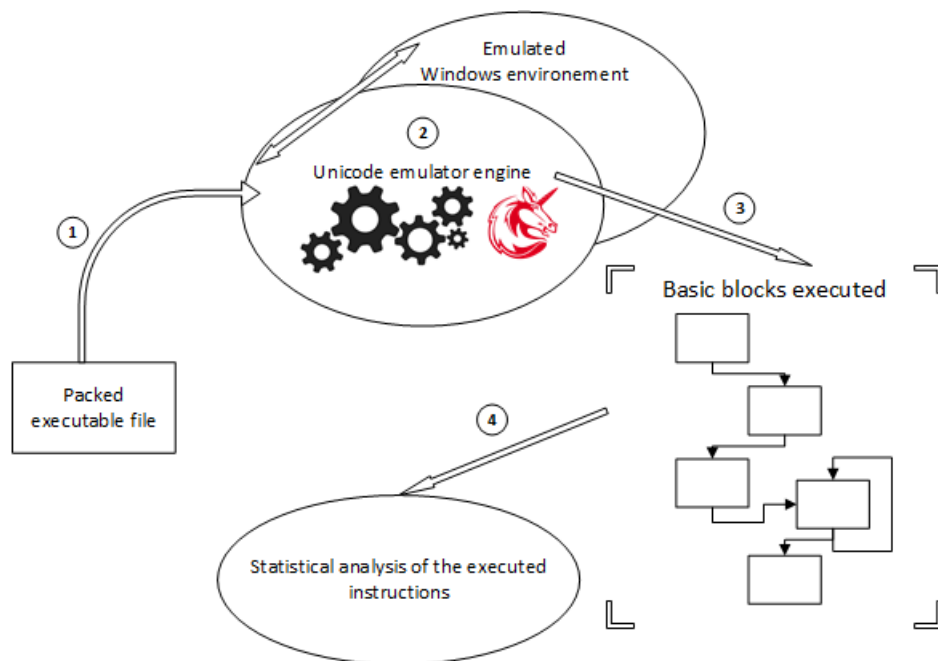


Figure 7.13: General view of our tool able to manage the classification of polymorphic packers.

1. We provide to the tool the packed file to be classified.
2. With the emulator, we trace the execution instruction per instruction at very high speed. There is a minimal Windows environment which is emulated to allow the depacking payload to perform its heavy work, signing its procedure by executing its own instructions...
3. Emulator provides a set of executed instructions, represented as basic blocks (a set of instructions which are unconditionally executed). Our tool is able to work both with a raw list of instructions or with the abstraction of basic blocks.

<sup>6</sup>For the sake of security, the emulator is designed to emulate Windows itself, avoiding to infect the machine where our tool is deployed in a case where a malware would be executed. Note that this property allows to execute our tool in a real machine and not a virtual one which could be detected by packed files at running time — Chapter 3...

4. A statistical management of the instructions provided by the emulator helps to perform the unsupervised classification.

Statistical analysis is the heart of the system because it provides the classification. From what can be said, the latter abstracts the different instructions to propose a more generic model representing what was really executed. Thus, it is possible to only keep what is relevant. These characteristics are statistical, allowing us to reduce the impact of the random aspects and potentially the noise induced by the decoy systems implemented by the packers (dead code, useless code, etc). This is based on robust statistics as provided in Chapter 3 (Resume 22).

After extracting each statistical characteristic of a packed executable file (grouped in a single vector per file), we compare them with those already extracted from other packed files in the database. The classification system is initially configured with a large set of programs packed with polymorphic packers. We perform the classification with quite classical data-mining techniques (this is a multiple vectors distance comparison) to group the closest elements into clusters driven by common (or close) characteristics (for which it is even possible to list the statistical characteristics). Figure 7.14 resumes our approach. With a new element extracted, we are looking for the distance between the vector of the packed file and the vectors that are already stored in the base. If the distance is close (a threshold is automatically computed by our system, avoiding arbitrary decision) to an existing cluster (where a *family* vector can be used to resume the cluster as a speed optimization — but it is also possible to directly interact with samples' vectors in a cluster), we add this packed file to this cluster. If it is not, it means we are with a packed file from an unknown packer (or at least, not identified by our system). In such last case, a new cluster is created with this sample.

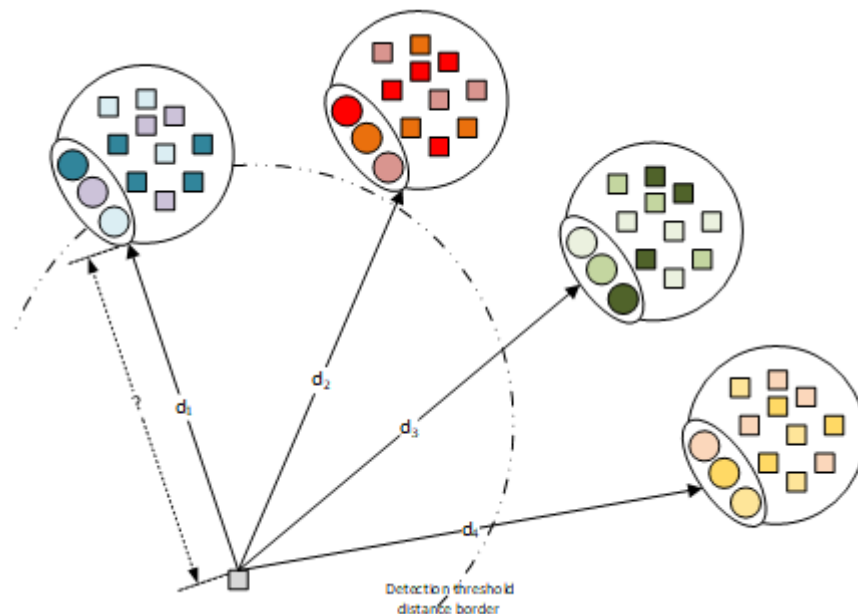


Figure 7.14: Classification of a new element in our unsupervised data-mining clustering system.



### 2.2.3 Stealth virtual analysis environment

#### Key Point 7.4:

- ☞ We enhanced the automatic dynamic analysis environment used by Dr Web to resist against very specific evasion techniques.
- ☞ To proceed, we had to study in detail how works the virtualization technology used on modern CPUs.

As explained in Chapter 3, malware strive to escape its analysis environment. Avoiding to be scanned is one thing, trying to fool malware that seeks to escape is another one. As explained in Chapter 3, the most common mean of analysis is a virtual machine in which the malware is placed. Various tools are present in the virtual machine and in the hypervisor that control this virtual machine to allow malware analysis.

Dr Web uses virtual machine, part of its analysis process. In practice, they have their own hypervisor maintained by the company. And they were faced with a case where some malware managed to detect their scanning system using a well-documented technique on the web. If the technique is based on one heuristic (it is not deterministic and sometimes it requires some calibration to be perfectly efficient, even if in many cases a simple and arbitrary threshold is sufficient), it remains complex to tune. As far as we can tell, this technique is based on time and the procedure which must be performed with specific timing.

The problem was to solve this problem by making the analysis environment stealthy (invisible) to this technique. However, the task is not as simple as introducing a new feature into an existing system. Indeed, to proceed, it will be necessary to modify the hypervisor that drives the VM. And modifying the latter, on a parameter as sensitive as time management, can quickly prevent the VM from working correctly. Freezing, synchronization issue, unexpected crash anywhere in the system, incoherent time management is a plague that is very hard to manage. Finally, the mission was to add this functionality while keeping the existing system fully functional.

The problem was complex, but it was finally solved. The technical solution is not interesting by itself. What is interesting is the approach used. Until now, the solutions that have been tried have been based solely on a technical approach. Starting from a given situation and wishing to reach a different state, the engineers have tried several approaches without much success. Two results were generally present. Either they had solved the problem but the virtual machine became unstable, or they had deployed a solution but the result was not visible and the malware continued to escape recklessly.

The solution will finally come through a different methodological approach. Rather than trying to use hypervisor technology for a given purpose, we took the time to study the technology of hypervisor by itself. Reading everything, technical documentation from Intel to the literature on the subject (a kind of state-of-the-art, in short), the goal was to understand first the technology perfectly. Then, once the expertise was acquired, it became possible to look at the problem from another point of view. Without trying to stick to a given architecture, it was possible to identify the key points of hypervisor technology that could address the problem. And that is finally what we did, with an original algorithmic approach that can fool current malware definitively, even in the case of the most advanced attacks.

In the end, the prototype that we had made based on the Dr Web's analysis system has been updated to take into account the problem for all architectures supported. The project has been deployed in production.

## 3 Third party productions

### 3.1 Superfetch documentation

#### Key Point 7.5:

- ☞ We documented SuperFetch a feature stored in SysMain service in Windows 10.
  - 👉 This service is used to monitor, record and make statistics about user's profile and application used.
  - 👉 It is used for optimization purposes (file pre-loading in memory) of the machine's performance.
  - 👉 But it is also an excellent tool for spying on the user.
- ☞ We documented the files used by SuperFetch.
  - 👉 The databases used are very rich about user's life.
  - 👉 It also becomes possible to read or modify these files.
- ☞ This work can be used for forensic, privacy and malware analysis purposes.
  - 👉 But it is still possible to fake information in the files, impacting forensic analysis.

#### 3.1.1 Introduction about Superfetch project

Prefetcher raised with Windows XP in 2001 as a functionality that has often been the subject of many passions. Originally almost unknown, it is discovered by the public through the American patent "US6317818B1 — Pre-fetching of pages prior to a hard page fault sequence" [1551]. This technology aims to increase the speed of user's experience by two improving two points: faster booting procedure and faster start-up for any process. The idea here is to improve the boot speed of the machine and more generally the responsiveness of the system by learning from user's experience to improve access to the most used resources. And this technology has evolved over time. Under Windows Vista, another component called *Superfetch* is added to the prefetcher and the service is renamed within the name of this improvement, until Windows 10. On this last version, the service is finally renamed *SysMain* but its main purpose did not change. For the rest of this part, both names "*Superfetch*" and "*SysMain*" will be used in an equal way.

In principle, from a security point of view, this performance improvement mechanism should not be relevant for us. It is hard to define how to use it for malicious purposes. Indeed, it is neither possible to interact with it, nor it is possible to access it directly (because it is not documented)... But the real question is not about to know how this service can be used for malicious purposes, but how it works to be used for potentially malicious purposes. Why such an approach? Because although the mechanism of *SysMain* is obscure by design, it improves user experience by studying current user activity. And there is not much difference between studying user activity to improve performance and studying user activity to spy the user. Again, it is not the collection of the data that is the problem (because it can be done for the noblest of motives) but the final use that is made of it.

The approach proposed in this work is twofold:

- On the one hand, it aims to produce a documentation of the mechanisms used by Superfetch (i.e. the *SysMain* service under Windows 10), of how it works, of the data to which it has access, of what it does with them and of how it uses them to improve the system performance.
- On the other hand, it aims to see if it is possible to exploit the information collected to potentially spy on the user.

To what extent does the collection of information allow us to know the user? Could this service be used as a forensic tool? Is it possible that this service could be used for nefarious purposes one day? Is it possible to falsify

the information collected to create false evidence? If the first part is a very classical documentation (based on reverse engineering) procedure, the second part aims at knowing what can be done with the information from the first part. We have in this study a documentary approach to answer operational and fundamental questions in the sense that we potentially touch the privacy of Windows users.

According to the existing literature [1552, 1553, 1554, 1555, 1556, 1557], the *SysMain* service and the mystery that surrounds it has already been partially studied. Globally, these studies are interesting but suffer from two important flaws. On the one hand, they are biased and only express the possibilities of the service, not the way it performs its operations. On the other hand, the internal documentation of the latter is not always accurate, either because the service has evolved since the publication of the previous studies, or because their original documentation was erroneous. Our work aims to improve these two flaws by updating internals of *SysMain* service (both with executable files and database files) and correcting false assumptions or fantasies about this one.

All of this research was conducted with Mathilde Venault. The work was presented at Black Hat USA 2020 conference [462]. If the conference should have been held in Las Vegas in the USA, because of the sanitary context of the year 2020, we had to perform the talk remotely. An extended version of what has been presented during the conference has been published thereafter [463]. This section is intended to be a non-exhaustive summary of our main research in order to present the impact it may have.

### 3.1.2 Technical mechanisms about Superfetch

To boot as fast as possible, *SysMain* will frequently calculate the "optimal layout" which is the files order to launch in memory at boot. More directly, it means to "pre-load" in memory the files that are likely to be used by the user in the near future. This list is established during idle states: whenever CPU, disk and memory utilization are under a certain use, the service will process to non-urgent operations such as the optimal layout calculation.

To proceed, the increasing of applications' navigation is based on the mechanism of reducing page fault<sup>7</sup> [1559] in memory, which is actually an optimization in memory paging. By reducing access to the disk by pre-loading into memory the file about to be read, Superfetch is about to increase the speed of the system. And to guess which file should be pre-loaded in memory, the service uses statistics coming from the system about page-faults. But there is more, because the statistics are organized in databases (details are provided in [463]), *SysMain* is watching and keeping traces of each action done on a computer, including:

- Proofs of software installs;
- Dates and times of application launches;
- Number of executions per program;
- Name and location of files used;
- Links to the content of personal files (cache system).

Technically speaking, the service could be seen as many divisions, handled by groups of functions<sup>8</sup> identifiable by their prefix (Table 7.1). Such functions are the nonstop jobs responsible for the essential features such as processing traces of applications, predicting and pre-launching pages the user might need.

All of these functions are used together for different tasks. To ensure all of these tasks, the work is divided per "agent". An agent is a logical unit which is an independent component dedicated to a specific task. Agents

---

<sup>7</sup>For short, when allocating memory, the memory is not loaded into the physical memory immediately. In fact, this one is placed in RAM only when the process writes into the allocated memory. The same mechanism would apply when reading a file from the disk in memory. The process of moving pages into physical memory incurs page faults [1558].

<sup>8</sup>The name of the functions is usually provided by Symbol path for Windows debuggers [1560] when they are provided. When they were not available, some name could have been provided by us, following the logic of original symbols from Microsoft.

Prefix	Name
PfPr	Prefetch Processor
PfTr	Prefetch Trace
PfSi	Prefetch Section Info
PfHp	Prefetch Heap
PfDb	Prefetch Collector
PfDb	Prefetch Database
PfDi	Prefetch Device Info
Rdb	ReadyBoost
HbDrv	Hybrid Drive
AgAl	Agent Application Launch
AgGl	Agent Global
AgPd	Agent PFN Database
AgRp	Agent Robust Performance
AgTw	Agent Trace Writer

Table 7.1: Function initials and their meaning in SysMain (Superfetch).

are contently active and notified by the system according to their main tasks. A list of agents is provided as follows:

1. Agent PFN (Page Frame Number<sup>9</sup>): it will be aware of page faults, classify the memory page's origin (foreground or background application) and memory state (committed page or not). This agent gathers data to feed statistics modules in order to model pre-launch procedure.
2. Agent Global: it oversees the context per user. it will define the criteria of active days, the limits of daily phases and it might organize *histories*, succession of "*scenarios*" within a certain phase.
3. Agent Application Launch: it is involved throughout the prediction chain creating Markov chains to represent probabilities of uses of a file in different contexts. It is used in order to take decisions and ask to the memory manager to pre-launch certain pages in memory.
4. Agent Context: it is responsible for watching the overall context (hibernation state of the computer — Key-Point 6.28, session information (SID), user token, session switching and so on). The goal is to react to different situation in order to improve user experience.
5. Agent Robust Performance: it oversees SysMain performance. It checks the frequency of accesses to the files referenced by SysMain in order to avoid pre-launching in cache of irrelevant data (for instance, a file opened just once). It ensures that performances are at best and remove irrelevant data from the service.

All of these agents, functions and features are creating a global architecture stored in SysMain service. A schematic representation of this architecture is provided in Figure 7.15. This one shows the different interactions between the different databases, drivers and agents which are definitively the heart of the system.

Internally, SysMain has two major types of supported files. On the one hand, database files (.db or .7db extensions) relative to the agents and on the other hand scenario files (.pf extension) relative to programs. In both cases, they are usually compressed within the XPRESS\_HUFFMAN algorithm from the RtlCompress-Buffer routine [1561].

All databases files are connected to each other and despite their different purposes, they are built on the same basis. This explains why databases construction process (Figure 7.16) and databases reading process (Figure 7.17) involve the same set of functions. We documented these files according to each agent to better understand how each agent is working. But generally speaking, it is about internal data which is relevant only

<sup>9</sup>The page frame number is an array representing each physical page state in memory on the system (Active / Standby / Freed).

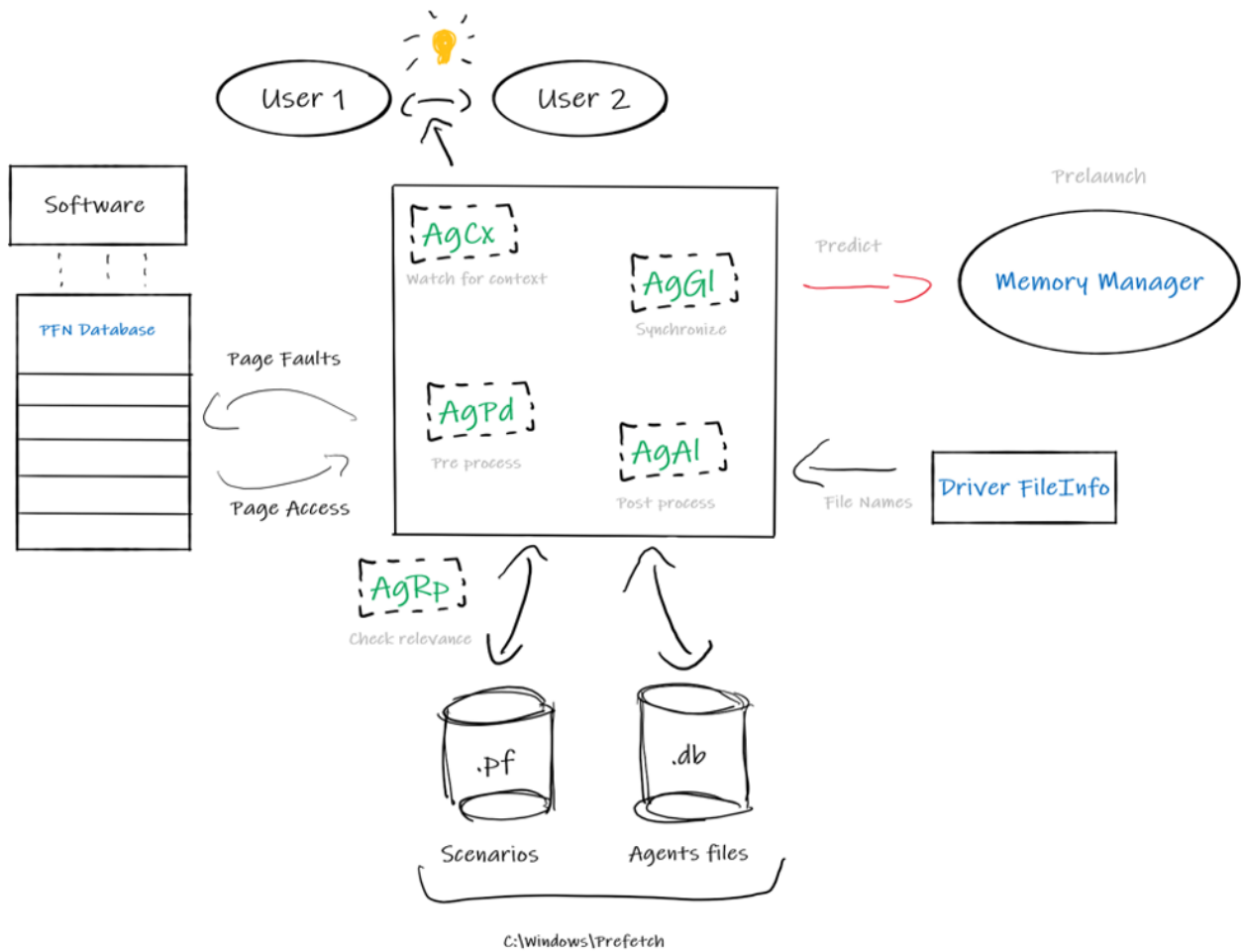


Figure 7.15: SysMain global operation.

for a given agent.

Scenarios files are much more relevant for our short presentation of Superfetch since they concentrate very sensitive data. Indeed, they are the supports for Superfetch to log what happened during a program's execution and to improve future predictions. An application has one or more scenario files attributed depending on the way it has been executed (actually, it takes into account the content of the command line). By default on Windows 10, there are 256 scenarios and the maximum size per scenario file is 10,485,760 bytes. Both values are configurable through the Window registry. Whenever a process begins, the scenario is immediately created or loaded into memory, referencing the page accesses and the page faults that occurred during its execution. The goal is of course to avoid them at the next launch if necessary.

In a scenario file, one can find many information. Among the most relevant ones, there is a list of loaded file and it includes the executable path file and almost all of its Dlls. It makes sense since Dlls are almost always loaded by a given process... In addition any specific file to the application which would be regularly used is recorded in the scenario file (icons, resources, databases and so on). Still, there are also the recent used files. For instance, the scenario of *Photoshop* software holds the name of last photos and directories opened, for *Windows Media Player* the names of any listened songs, for *VLC* the last movie which has been seen... In addition to the full path of the file, the the dates and hours when the files have been consulted are recorded. For short, SysMain knows better than anyone how the user's daily life looks like!

But there is more since it is possible to find sometime in SysMain references to the cache files. The cache

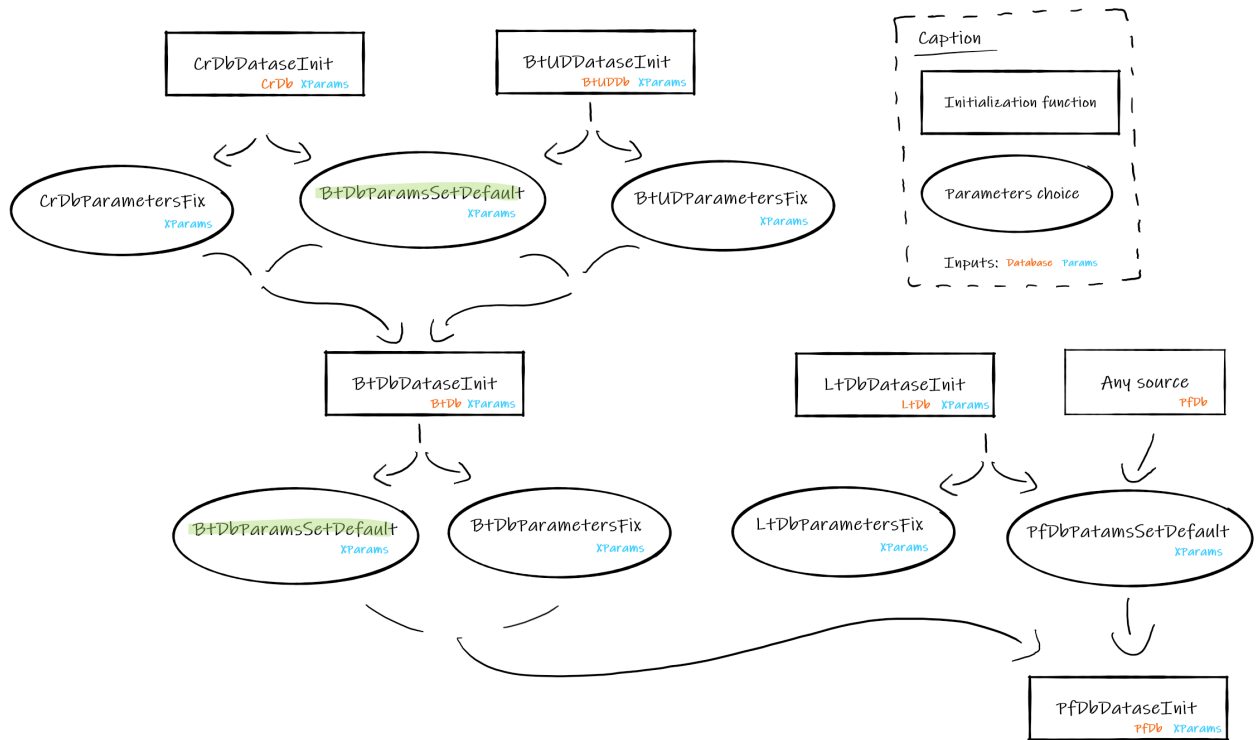


Figure 7.16: SysMain internal databases construction process.

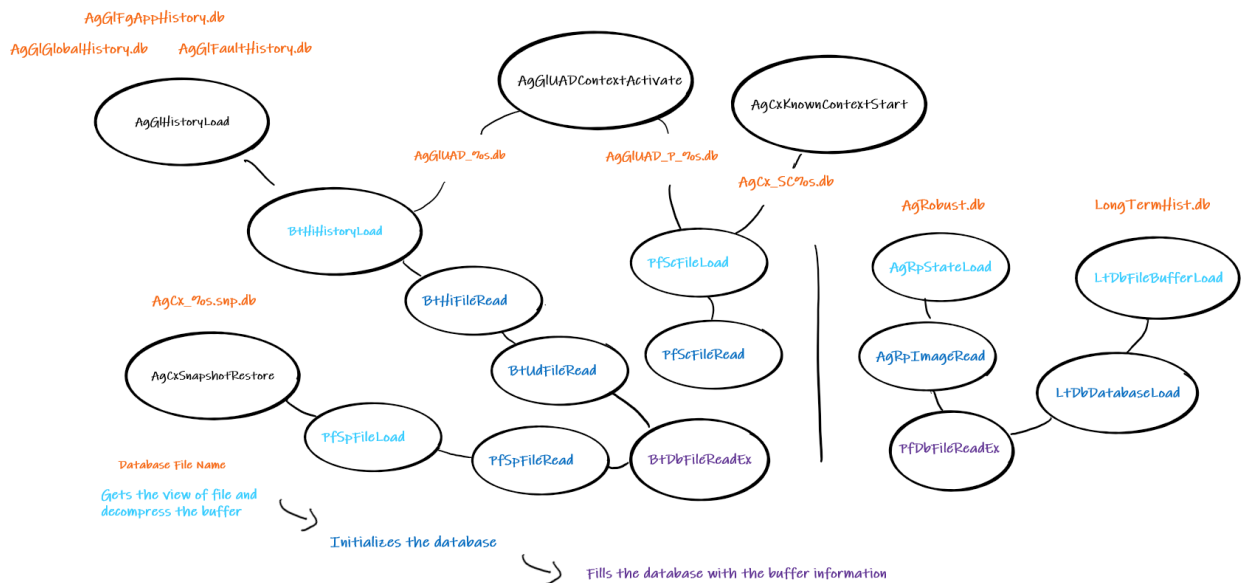


Figure 7.17: SysMain internal databases reading process.

files are the results of the *Cache Manager* [1562] performance, which stores in those temporarily data to reduce the access time next time the data is required. In practice, it corresponds to "`<User>\AppData\Local\Microsoft\Windows\Cache`" where "`<User>`" corresponds to the current user's directory. Since SysMain is referencing the cache files, it gives the possibility to get a direct access to data from the user's applications, including mails or confidential documents... And example concerning WinRAR (a compressor software) is provided in Figure 7.18 where we can see internal document opened by WinRAR, referenced in Superfetch and stored



directly in the cache.

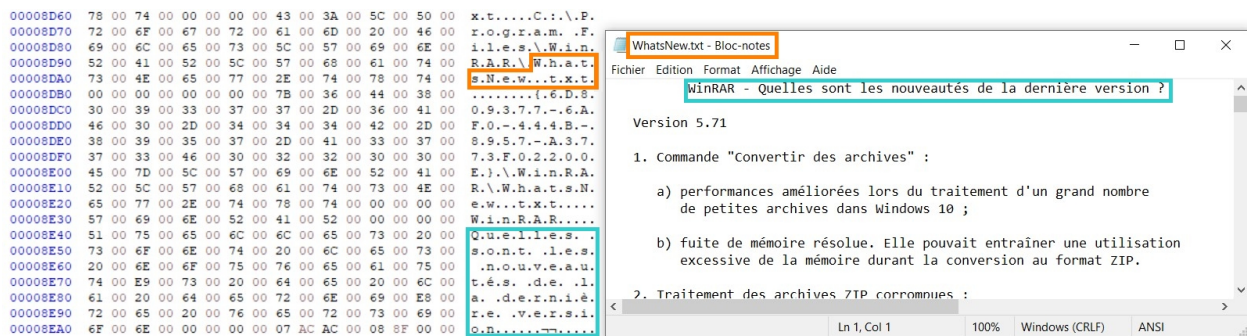


Figure 7.18: Content of the cache concerning WinRAR software, as referenced by SysMain in the Cache Manager.

Finally, the study of the SysMain service was an opportunity to confirm or invalidate certain “fantasies” that could be found in various articles. Some information was sometimes dated (true for a while, but they does no longer correspond to reality), while some may have been totally erroneous. It is also an important contribution of this study to have been able to remove some of the mysteries or dreams about a little known but terribly powerful service.

### 3.1.3 Security review

From a security point of view, the only point we observed concerns PfPrAgentsLoadFromRegistry function. This one loads the content in “HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Superfetch” registry key a value called “Agents”. This value is a string containing a list of Dlls to be loaded thanks to PfPrAgentLoad function. Technically speaking, it is possible to edit the content of the value to add one more Dll, allowing a *dll side-loading* attack. In practice, updating the registry value could load a Dl executed in the context of SysMain service, running within the LocalSystem Account [1563].

While this mechanism may appear to be used as part of a possible elevation of privilege, it is not. In fact, it is a new illustration of the bypassing of an already bypassed security [1127, 1128, 1129]. Indeed, modifying a value in “HKLM” registry hive to perform code injection requires to be administrator. But processes running with administrator privileges already have plenty ways to perform actions normally reserved to SYSTEM [1442]. That way, such code injection is not a security hole since it does not provide any real elevation of privilege. In other words, an attacker would not get any advantage doing this operation, except executing code in another process than the original one. But being an administrator already offered such possibility for cheap. Note that if the registry key would have been located in HKCU (for current user, in order to adapt the service’s behavior per user), the vulnerability would have been real.

After reporting this observation to Microsoft, the company confirmed that this is not a vulnerability for the reasons described above. That way, as it cannot be exploited directly, it was not considered as necessary to remove this feature which is nevertheless not really used by Superfetch anymore. Nevertheless, we thought it would be interesting to point out the mechanism so that it would be more widely known. Thus, it allows to document a mechanism of *dll side-loading* even if it does not provide any offensive possibility from the operational point of view.

In practice, this kind of “weakness” is likely due to retro-compatibility because this function does not seem to be used anymore under Windows 10. It might have been used to load additional agents for Superfetch, especially optional one.

### 3.1.4 Interpretation of Superfetch features

The first essential point to emphasize is that, according to our observations performed thanks to the reverse engineering, the objective of the SysMain service is indeed to promote the performance of the machine by study-



ing the user's behaviour. There are no hidden mechanisms or backdoors here, at the best of our knowledge. Its role matches to the one displayed and there is no ambiguity about it. Nevertheless, if the official objective is laudable and indisputable, it is in potential miss-exploitation that questions should be raised.

Indeed, information flow coming from SysMain represents a significant forensic opportunity for the system. Therefore, *SysMain* knows a lot about the user, from the wake up time to the favorite songs listened on the system. This raises a very serious privacy issue since it tracks lifetime activities. Not only the list of software used is known, the number of times when they have been used, but also the files manipulated by them (when it is not a precise location in the file that is referenced via the cache system). Even if the main purpose is to improve the user's speed experience on his or her computer, limit between spying and profiling is not really clear.

If this feature can be used for spying, it can be used also for forensic purposes. Because all software running on the system are *logged* in SysMain's databases, any malware should leave traces of their execution, including time and location of the executed files. Taking into account that this service is not really famous, at the best of our knowledge, no malware takes time to remove its footprints from Superfetch. Note that, fooling forensic analysis performed from Superfetch to create false evidence on a computer would require administrator rights (since database files are stored in Windows directory). But this is not such a ridiculous hypothesis in the context of a targeted attack... This is why if the data collected by Superfetch is a great source of information, such data should be taken with a certain suspicion and be cross-checked with other forensic sources to have a shred of veracity.

### 3.1.5 Conclusion

A first conclusion about our study is the dual contribution it provides. On the one hand, on a technical level, our work aims to document and understand the internal mechanisms of the SysMain service. This understanding allows us to interface with this service and its valuable databases. We are providing a more efficient interfacing than the existing software whose features are more limited compared to ours (they are reading only a subset of databases and they do not provide any feature to edit these files efficiently) and sometimes not up to date. On the other hand, our study has identified new possibilities of use. Whether these possibilities are good or bad, allowing from one side spying the user in an intimate way or on another side with forensic behavior to document what happened to a machine (and potentially detect malware that would forget to hide from SysMain, because of a lack of knowing this service enough). Note that all this work has been presented to Black Hat USA 2020 [462] to share this knowledge as best as possible.

While these contributions are potentially interesting, they are nonetheless limited in their time frame. SysMain is intended to evolve and it might change dramatically with any next version of Windows. Like any study based on reverse engineering, it is only valid at a given moment and only provides a snapshot of the service at the time we analyzed it. The same applies here as in Chapter 4, which used the same working method on another subject. Although the information provided must be considered as potentially inaccurate in the future, the working method remains the same to update this study with a new version of Windows. Furthermore, Microsoft being very attached to the notion of backward compatibility, it is likely that a large part of our work can be reused in future versions of the operating system.

At the end, we have to discuss what could be the future work with this study. Since our work has been focused on SysMain, it would be interesting to dive further on the external components of SysMain's performance such as the drivers interfacing with it. Drivers are providing a lot of raw data to SysMain. Thus, it would be interesting to look at the information provider. See what is actually being transmitted, actually used by Superfetch, and if it is possible to interface with it in order to feed data to SysMain, for better or for worse. In addition, SysMain's performance management is closely tied to the memory manager, including the cache system. Still, the cache management is not widely documented, and it would be rewarding to understand its mechanisms and precise its exact links with SysMain.

## 3.2 UEFI ciphering system

### Key Point 7.6:

- ☞ This project aimed to achieve a full encryption (100%) of the hard drive of the machine.
  - ☞ The ciphering is performed on the data and system hard disks (including the boot sector).
  - ☞ To do this, the ciphering is performed at the UEFI level.
  - ☞ More precisely, it is performed on the motherboard UEFI firmware ship.
- ☞ The goal is to keep the ciphering key out of the operating system scope.
  - ☞ This is the UEFI code that manages the ciphering process behind the operating system.
  - ☞ A malware with full rights on the machine could not access the ciphering key.
  - ☞ Physical access to the machine would not be sufficient to retrieve or modify the data stored on the hard drives.

### 3.2.1 Context of the project

The protection of the confidentiality of the user's data is done through cryptographic software. The idea is that the ciphered data, without the cipher key, is not accessible to a third party. To proceed, there are several programs available, and among the most effective are those that offer the possibility of full disk encryption. What means full disk encryption? It means that everything on the disk is ciphered. More directly, not only is the user's data is protected, but also is the user's file system.

This last feature is interesting because it offers a particularly robust defense against one of the most effective attacks: physical access to the user's machine. Hence, if an attacker gains physical access to the machine, this one cannot attempt to corrupt the operating system. Indeed, it is in this mode that OS is most vulnerable because before it starts, no security is in place. The only thing the attacker can do is either corrupt the hard disk or format it to zero. In both cases, the victim will quickly see that the machine has been heavily corrupted (data will be missing or incomplete) and will not take long to react.

If the theory is based on this principle, the practice is a little different. When we are talking about full disk encryption, this is not a real 100 % ciphered hard drive. Indeed, all but a small portion holding the boot procedure of the operating system is still in clear text. Such an architecture makes sense since the encryption/decryption process must be initiated somewhere, for instance by requesting the required cipher key. Of course, TPM technology [33] can be used to enhance the security thereafter (Key-Point 6.27).

In practice, this encryption startup system corresponds to an UEFI application (Key-Point 5.6). The latter is executed before the operating system is loaded by the firmware of the motherboard. In Windows 10, security is guaranteed by the Secure Boot system [1255]. Technically speaking, UEFI application cannot run without being duly authenticated by Microsoft [1564]. This prevents potential malware from corrupting UEFI applications to add malicious actions.

While this system is very secure, it is not perfect. The problem we are trying to solve here is twofold. On the one hand, while Secure Boot's security is very important, it is not necessarily perfect. Attacks exist [1565] and it could be possible to find new ones one day. Modifying the content of a UEFI application or successfully inserting one authenticated by Microsoft would be a very effective attack. Of course, this presupposes important means, but it is precisely against this type of attack that we want to fight. On the other hand, this type of system naturally transfers the cipher key to the operating system. Thus, it resides in the kernel memory and could potentially be retrieved by a malware with enough rights. This last point is particularly problematic and if there are solutions to deal with the problem (Key-Point 6.27, Key-Point 5.24 and Key-Point 6.41), they are not always perfect against an attacker with enough rights. In the end, it is significant to note that many cryptographic products are often proprietary software (including BitLocker [1423]) whose operation in UEFI is

rarely documented. This allows little interoperability and even less open alternatives, at least under Windows...

This project aimed to design a system that could offer an alternative to all these limitations on current systems. The goal was to design a UEFI ciphering system that could resist an attacker with full rights on the operating system and even potential physical access to the machine (for a relatively short period of time). The approach adopted here is very prospective, we must admit it. Firstly, because the existing state-of-the-art solutions are really efficient. Then, because the idea was as much to transmit some advanced technical skills to students (in a pedagogical approach) as to try or new experimental approach.

### 3.2.2 UEFI information details

One of the important characteristics of UEFI is that it is more of an execution environment than a geographical area on the hard disk — like the MBR on old hard disks or the BIOS on old motherboards. UEFI combines both of these technologies in two different places. For the sake of brevity, UEFI is located in two places: on the one hand within a dedicated memory chip on the motherboard (sometimes called the “*firmware*”) and the initial boot sector of the hard disk. The transition from one world to the other one is done naturally when the firmware has finished initializing the hardware and after looking for the boot sector on the main hard disk, runs the referenced UEFI application to take the control. Figure 7.19 illustrates this architecture in a graphic way.

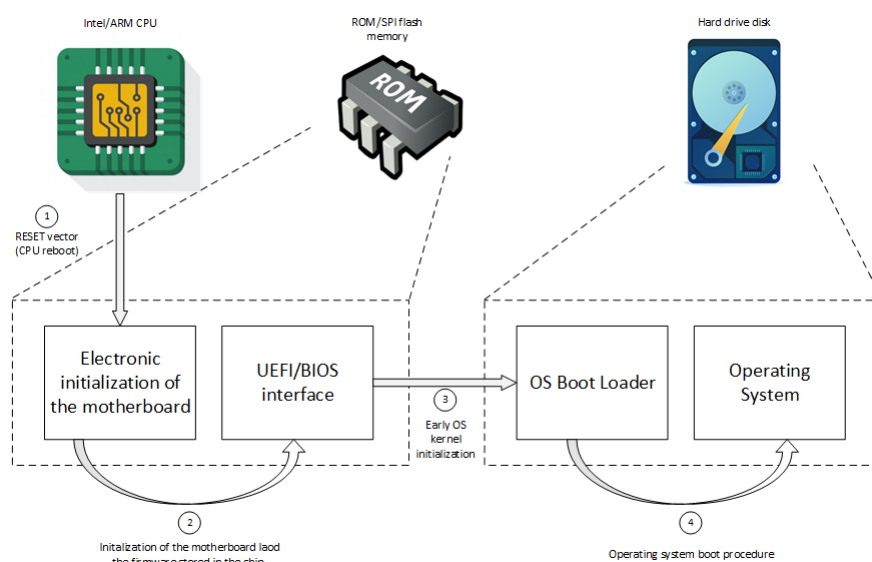


Figure 7.19: Illustration of the UEFI boot procedure in two stages. The first from the ship on the motherboard and the second a boot partition on the hard drive disk.

In practice, it is necessary to distinguish the UEFI firmware from the applications present on the hard disk. If in the last case the modification is relatively easy (it requires to write a file on a certain partition of the hard disk), the first case is more complicated. In practice, each motherboard manufacturer is free to implement its own firmware. What is called “UEFI” (*Unified Extensible Firmware Interface*) is finally more a *standard* than a software fixed forever (same thing for UEFI ancestor called *Extensible Firmware Interface* — EFI). This standard implemented in almost every motherboard in the world allow hardware manufacturer to easily interact between each others. More directly, it is an interoperability surface for firmware components that may originate from different providers.

Technically speaking, an open-source development environment implementation by Intel and called “*Tianocore EDK II*”<sup>10</sup> is the *de facto* UEFI reference for generic UEFI services implementation. A significant number of motherboard manufacturers base their firmware on this project because it follows original UEFI specifications,

<sup>10</sup>EDK stands for “*EFI Development Kit*”.

which has been adopted by over 200 companies<sup>11</sup> and shipped on millions of compute devices.

From a conceptual point of view, this architecture is presented in the official documentation as a whole declined in several stages. In our case, the most relevant point concerns the "Platform Initialization" (PI) which describes the initialization of UEFI. This initialization is provided in Figure 7.20. The first two phases (SEC (Security used to execute authentication process such as SecureBoot and PEI (*Pre EFI Initialization*) used to initialize the motherboard and set the CPU in protected mode) are reserved for motherboard management. Then comes the DXE (*Driver Execution Environment*) used to initialize drivers and all UEFI API used by applications. In the end, BDS (*Boot Dev Select*), TSL (*Transient System Load*) and RT (*RunTime*) are the regular procedure to select a boot partition and running the operating system from this one.

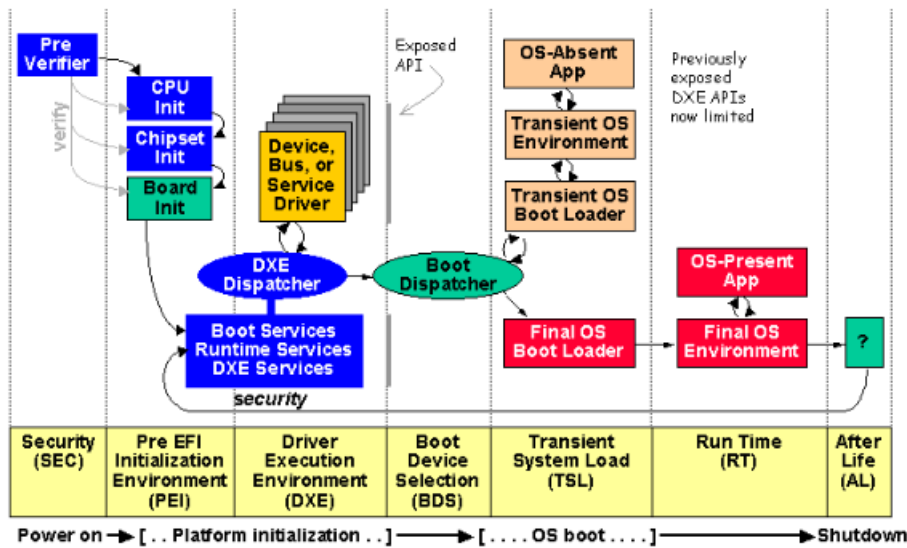


Figure 7.20: PI Architecture Firmware Phases (extracted from [21]).

### 3.2.3 Provided solution

The only solution to design a cryptographic system that allows to fully encrypt a hard disk is to update the firmware of the motherboard. Our idea is to get to the level of the chip on the motherboard to perform our operations. As it has the hand before the hard disk (see Figures 7.19 and 7.20), it becomes possible to set up our cryptographic system at this level.

During our study, we have had to work on different environments. First of all, for ease of development, in a virtual environment with Qemu [71]. The latter allowed us to deploy an EDK II environment. By editing this environment, we have been able to test various encryption solutions at different levels. Two components are essential to develop a valid proof of concept. On the one hand, an application in charge of receiving the ciphering key (whether it comes through the keyboard, a SIM card reader or any other key management media). On the other hand, a UEFI driver that will interface with those responsible for managing I/O access between the hard disk and the file system (precisely to be independent of any type of file system).

In practice, this means being below the file system and therefore at a level that deals directly with communication with a hard disk. This led us to study the PATA (*Parallel Advanced Technology Attachment*), SATA (*Serial Advanced Technology Attachment*) and AHCI (*Advanced Host Controller Interface*) protocols. These protocols are data transport standards between the CPU and the storage media (Figure 7.21). In practice, we had to deal with SATA protocol. Similar to the OSI model, the SATA protocol is composed of several layers of abstractions and packages, as illustrated in Figure 7.22. Note that SATA is backward compatible with the use

<sup>11</sup><https://www.tianocore.org/>

of PATA. The idea is to be able to interface with the hard disk low enough in the system to be able to cipher all the data transiting on it.

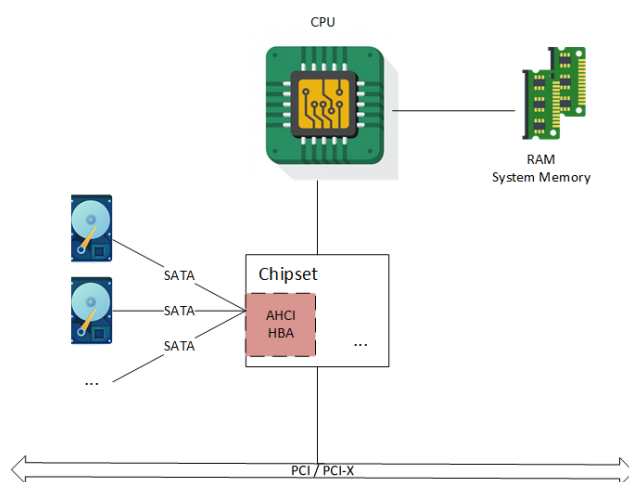


Figure 7.21: Basic architecture of Intel computer with AHCI interface.

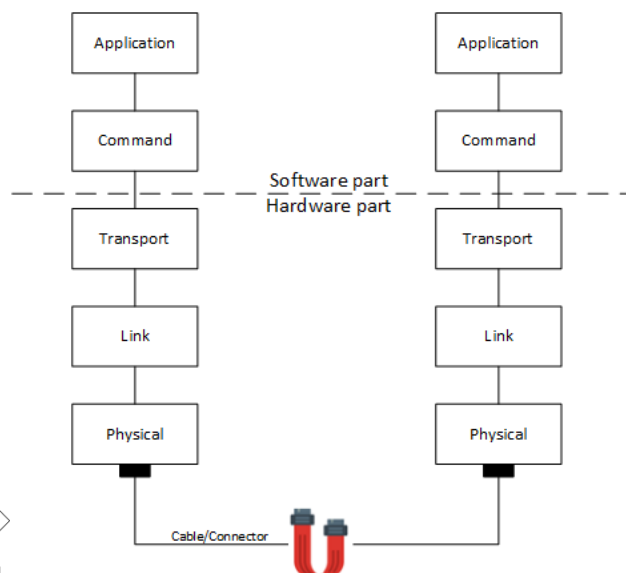


Figure 7.22: Different layers of the SATA protocol model.

More directly, an application gives a write or read order to the hard disk. This order passes through several filters, implemented as drivers. Each driver is responsible for a particular operation. The first drivers support the file system, while the last ones handle the SATA commands so that the order is really perceived by the hard disk. At the end, a transfer operation from memory to physical hard disk is ordered by the CPU. It is within this chain of operations that we have implemented our solution with very satisfactory results.

The results obtained allowed fully operational proofs of concept. First in the virtual environment that is Qemu and then within dedicated hardware such as "Intel Minnowboard Turbot<sup>12</sup>" cards (Figure 7.23) and tests have also been conducted on real hardware (Figures 7.24 and 7.25). Nevertheless, for confidentiality reasons and to preserve future publications, we will not detail here these projects.

### 3.2.4 Publication and work with students

As explained at the beginning of this section, the work was done with three students. We have to admit that mastering UEFI (at the development and architecture level) is not really a mass sport. It requires a deep understanding of the mechanisms driving operating systems as well as hardware protocols and hardware in general. The first part of the project (apart from its presentation) was to take this technology in hand. And if there are some documents, books and tutorials here and there [1093, 1566], practice is still not an exact science.

For pedagogical purposes and to continue the "spirit of research" that was specific to the laboratory where we worked, our PhD Director and us asked the students to regularly produce memorandums. In addition to formalizing knowledge and keeping track of achievements (and sometimes even to save somewhere the *tips* required to operate certain equipment), the idea was also to gradually introduce them to scientific publication. The operation was productive enough with a real quality that we submitted them to *Multi-System & Internet Security Cookbook* (MISC) magazine. The latter is a scientific journal with a peer committee that is open to a certain popularization. Presenting a cutting-edge technology like UEFI with a technical step-by-step approach was of great interest to them. That is why, with the students, we published a series of three articles about UEFI [1567, 1568, 1569], over three papers in three releases of the magazine.

<sup>12</sup>This hardware was easy to manipulate with EDK II firmware.



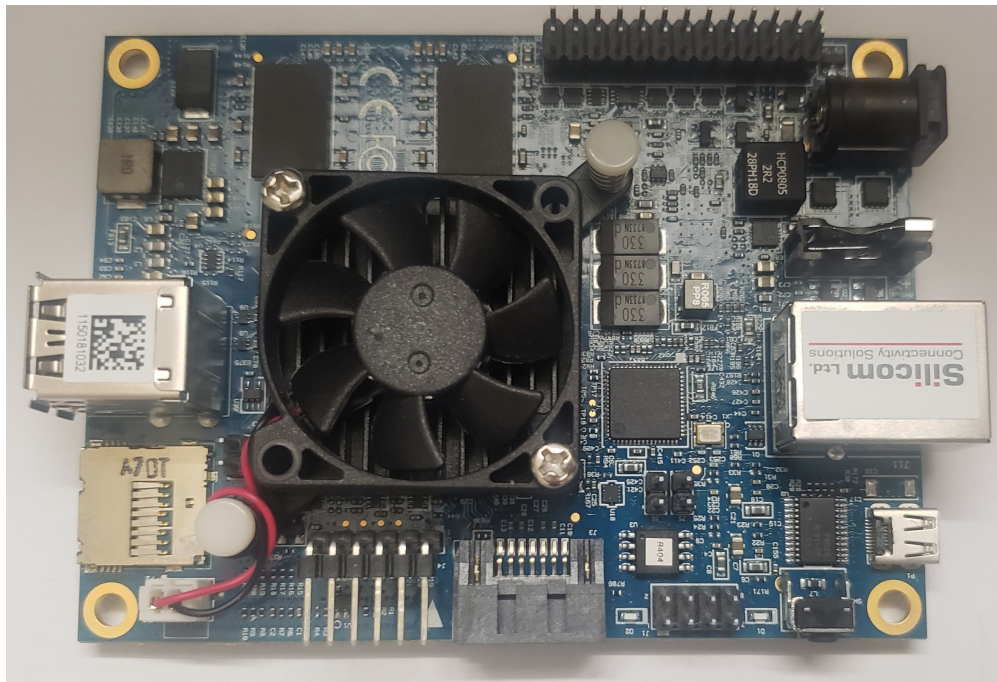


Figure 7.23: Intel Minnowboard Turbot computer.



Figure 7.24: Access to the mother board on a laptop.



Figure 7.25: Electronic setup for flashing the contents of the machine's BIOS/UEFI.

The articles were really intended as an experience feedback of the UEFI technology, resulting from our work for the realization of the project presented in this section. The first article [1567] introduces in a general way what is the UEFI technology and how to deploy a test environment with Qemu. The second one [1568] taught how to program and interact with the UEFI API in order to design small but powerful applications. Finally, the last one [1569] taught how to debug an application with Windbg from Windows to the Qemu virtual environment with all EDK II symbols loaded. This last article was really original in itself since, at the best of our knowledge at that time, there was no technically accurate tutorial on the subject (the existing ones were inaccurate). In the latter case, several bugs have been reported about the interface between Windbg and Intel UEFI Development Kit<sup>13</sup> (Intel UDK) from Intel. The problem came from Windbg software. Last versions were not able to correctly interface with the UDK. We contacted Microsoft and after having recognized problem when interfacing between Windbg and Intel UDK, they never wished to correct the bugs observed. Probably

<sup>13</sup>This is an Intel Debugger tool used to interface with UEFI software. More information at <https://software.intel.com/content/www/us/en/develop/download/intel-uefi-development-kit-intel-udk-debugger-tool-r150-windows.html>.

the interest was too marginal compared to the number of people really interested by this type of tool... This is why we used an old version of Windbg 6.12.0002.633 with Qemu version 0.13.0 to have something functional but downgraded...

Finally, a non-technical article was published in a popular technology journal called "SecuritéOff" [1570]. This article aimed at presenting the UEFI technology (sometimes unknown by computer engineers), where it was present, what it allowed to do and why it was interesting, not to say necessary, to understand it. From there, the article aimed at answering these questions as pedagogically as possible in order to be understood by as many people as possible. It is always an important and beneficial task to make one's research work accessible to people who are not necessarily in the technical field of computer science. This also allows us to take a step back, to contextualize and explain the importance of knowledge, the understanding and the mastery of such a sensitive technology (because it drives the launch of the operating systems of most of the current machines on the planet).

### 3.2.5 Conclusion

In the end, the realization of this work have confirmed the feasibility of the idea that we had initially proposed. Technically speaking, it is possible to design an encryption system for a fully (and really) encrypted hard disk. The solution is to use the firmware of the motherboard with an extended version of EDK II. By interfacing with the drivers controlling access to the hard disk, it is possible to filter all incoming and outgoing data. From there, the cryptographic operation becomes possible.

The technical result made it to produce various proofs-of-concept on several types of hardware. Further work could be done to go beyond and design an industrial quality system. This is an exciting topic and there are many ideas for improvement. It is likely that we will continue this project in the future and that publications may be produced from it. In addition, one the greatest achievement — from our point of view — of this project was the publications with the students. Sharing knowledge, conducting research, but also and above all, arousing the desire and the passion for these highly technological fields and more generally for research has been a most rewarding experience.

## 3.3 Detection of Crawler Traps based on new metric distances for data-mining

### Key Point 7.7:

- ☞ This project aims to detect the crawler-traps mechanisms to better avoid them.
  - 👉 A crawler-traps aims to detect the actions of a bot (crawler) that automatically retrieves data stored on a website.
  - 👉 In the case of the dark web, some crawler-traps randomly generate web pages to fool the bots.
  - 👉 There are various strategies to generate these web pages (from scratch, random words in a dictionary, from an existing page, and so on).
- ☞ The detection of these objects is equivalent to solving a mathematical problem.
  - 👉 We are trying to observe an irregularity in the middle of a more or less coherent set.
  - 👉 This is an on-the-fly and unsupervised machine learning problem.
  - 👉 Each website is different and there are several types of crawler-traps to detect.
- ☞ A classification system measures the distance between several elements to be classified.
  - 👉 The more effective the distance, the more accurate the classification.
  - 👉 We have designed new distances more efficient than the existing ones to solve our problem.
  - 👉 We have proposed a new approach to design and to evaluate our distance.



### 3.3.1 Web-crawlers and crawler-traps war

During our studies, we had to work on data collection and processing from online websites. Of course, it goes without saying that collecting data from an entire website cannot be done manually, because the effort would be significant and time consuming. In practice, this *crawling* of website is about visiting every links for each web-page of a given website to keep all (or only relevant) data. To do this, we have designed from scratch a library capable of processing any website efficiently, safely and very quickly. This library allows us to write autonomous robots, sometimes known as "*web-crawler*" or "*spider-bot*". In practice, our robots are faster and more efficient in meeting our needs than existing commercial tools.

These robots have been used on various websites for various missions. One of them included websites that were hosted on what is known as the "*dark-web*". More directly, the targeted websites were on the Tor network [1571], which is well known for offering a certain anonymity to both the content hosts and the people who visit these websites. With our colleague Maxence Delong, we have adapted our bots to work in this particular environment. More directly, our bot is able to go on this kind of network and to deal with some of the "*special content*" it could host [1572]. Of course, websites hosted on the drak-web are not always very willing to share all their data, so to speak.

To avoid being analyzed this way, websites (and not only those on the Tor network) implements various protections called against bot crawling. On the classic web, this type of protection is very common. This kind of protection aims to prevent malicious bots from archiving "private" parts of websites (collection of email addresses, personal identities, sensitive data, abuse of service and even to prevent denial of service). There are all kinds of them.

On the one hand, the simplest and *official* way to prevent web-crawlers is to create a file named "*robot.txt*" stored at the root of the website. In this file, all the section forbidden for crawling are referenced to prevent legitimate bots from falling into them. This file is only declarative and there is no deeper protection to forbid crawling. In a way, it is a gentleman's agreement that determined bots shamelessly bypass. On the other hand, there are stricter methods and also more intrusive to the user experience. For instance, IP banishment is applied when a web server guessed that a bot is crawling it. That way, an error code can be returned on legitimate webpages thus disallowing content access. In the case where the website has incorrectly guessed and real user has been blocked, it totally ruins the user experience. Another example of common impractical security on the web are captchas [1573]. Indeed, if originally, most of captchas were used to validate specific operation where a human user was mandatory (online payment, registration...), nowadays, basic access to some website is only possible by resolving a captcha first. But captchas are far from being a solution since there are practical ways to automatically bypass them [1574]. As a consequence, other security systems have been developed to be more efficient.

As an intermediate voice, there are softer but generally very effective methods. New trends are focused on systems called "*spider trap*" [1575] or "*crawler trap*" [1576]. The goal is to ambush the bot on a dedicated trap inside the website. More directly, the bot is sent in a specific part of the website where humans are unlikely to go to process meaningless and endless flow of information. There are many ways to create and deploy a crawler trap.

1. **Involuntarily crawler traps:** there are some elements in a web-page that sprang to trap a bot without doing it on purpose. For instance, an interactive calendar where each "next day" is managed by a new url link would be enough to trap a bot. Because a bot visits every link and there is a countless number of days in a calendar, the bot will loop all these days, being stunk inside... Another example stands in some website which generate pages on-the-fly from a general link but with different parameters. Incorrectly managed by the bot, it is possible to make them endlessly loop on this type of link.
  2. **Infinite depth or Infinite loop path:** the widely used solutions stands in the automatically generated pages when the system detects a bot. The content generated does not matter as long as the crawler hits the trap and that it crawls fake sub-pages. In the majority of cases, a loop with relative URL is created and the bot will crawl the same set pages. For instance, we can use two twin directories referencing each other such as bots come to one to go to the other and vice-versa:
-

<http://www.example.com/foo/bar/foo/bar/foo/bar/somepage.php>

3. **Identifiers:** for some website, the strategy is to generate a unique identifier stored in the page URL per user visiting the website. This one is assigned to each client for tracking purposes. This one is usually called a *session ID*. The trap lies in the fact that the user ID is changing randomly at random time. That way, the bot can be redirected to already visited web-pages but with a different ID.
4. **Random texts:** the most efficient solution stands in random generation of text in a web-page. With simple script from web-server side, one page can be randomly generated with random text and random links referencing the same page. Once this page is hit, the bot will treat what looks to it as different web-page, since it comes from different links with a different page's name, over and over. Of course, a human who would come across such a page would quickly see the trap and stop browsing (because the text presented is inconsistent). But a bot, if it knows how to "read", it does not "understand" the text given. Worse, some traps are smart enough to generate random text derivatives from existing texts, keeping a certain consistency of grammar, style or vocabulary.

Generally speaking, the four types of crawler-traps have several advantages. On the one hand, they are relatively effective, at least enough to fool most of the market tools that do not protect themselves against such mechanisms. On the other hand, these mechanisms do not require the use of javascript to work properly. This last point may seem anecdotal, but it is major in the case of the Tor network, which bans the use of this technology for confidentiality reasons. This is also a direct explanation for the success of crawler-trap technologies in the fight against bots on the Tor network (captchas and other protections may require to get access to javascript on client side). These conditions explain why we find this type of mechanism on some websites hosted on Tor network.

In practice, the first three mechanisms of crawler-traps described above can be fixed by various technical procedures. In the first case, it should be noted that some of these mechanisms have regularities that are easy to detect (in names of some of the tags on the page). A bot can be trained to avoid such objects. In the two following cases, keeping a hash of each visited web-page allows to check if we have already crawled the page or not (whether the url is different or not). The last case is definitely more problematic. This is the aim of this study.

### 3.3.2 Context and problem to be solved

Because the pages are generated randomly, it is not possible to archive them. From an operational point of view, our bot visits websites, page after page. The main difficulty lies in the fact that the trap generates random pages after the bot has visited a certain number of pages (based on the speed of the visit, the number of visits, the logic of the order of the links visited, etc). In a way, we can only base our decisions on the already visited pages and the ones we are going to visit next. This is the operational context in which we operate.

A simple solution would be to avoid being detected by the website. That way, our bot would not fall down the trap provided by the defending website. But in such situation, crawling performance would be too downgraded. If it is not possible to avoid being potentially detected, seeking to detect the trap that is being held and get out of it is a more promising approach.

Detecting that we are facing an crawler-trap presenting random pages is a complex problem. In practice, our bot cannot be detected from the first page(s). Our bot needs a "*minimum of speed of crawling*" on the website so that the last tries to defend itself by providing us with a trap. This means that for a given website, we have an initial set of pages that do not belong to those generated by the crawler-trap. The problem is to make the difference between the real pages of the website (the initial one) and those from an crawler-trap. For this reason, we need to measure the distance between regular and irregular web-pages.

Note that the problem can be defined in a more general way (apart from our specific problem about crawler-trap detection). For short, we try to extract from a data set several sub-families (at least two if an irregularity is found). Particularity in our case, we do not know *a priori* the notion of "*regularity*" sought within a cluster. More directly, our method must be applicable to any website (and they are all different: subject, vocabulary

used, language skills, layout, length and so on). We are therefore dealing with an *unsupervised learning system* (since it is not possible to define an initial norm for each website). From there, several *specificities* apply to our case.

On the one hand, our data set is constituted gradually. Unlike a classical big-data approach where the data set is constituted and processed once and for all (which is time consuming), in our case, we can enrich our model element by element (which is much faster). Note that the global approach is always possible by enriching the model with all elements at once. But the advantage of proceeding gradually is that we can quickly have a trained model with relatively few data. As a consequence, our model is built in such a way that it sorts the different elements inserted over time. On the other hand, our system makes no assumptions about the nature of the data taken into account. In practice, we have worked on web pages and more directly on their textual content. But there is nothing to prevent us from generalizing the principle to other sets of text, images, audio, video...

For the sake of understanding, a metaphor can be used to explain our problem. Let us imagine a book written by a particular author. In the middle of the book, pages would have been inserted. These pages can be random, taken from another author's work, from another book written by the author himself or derived from previous pages where couples of words would have been randomly reorganized<sup>14</sup>. Our goal is to find the original pages of the book from those inserted. More directly, this situation can find many applications since it is suitable to find an irregular (or derived) subset in a given population. We can imagine applications in the medical, financial, security, commercial world...

### 3.3.3 Solution and publications

The solution presented here was first published in a short version and then it was presented at the ForSE conference in Malta in 2020 [1577]. Subsequently, an extended publication of the first has been published by the journal JICV in [1578].

From an operational point of view, the crawler processes a website, page after page, and it can fall into a crawler trap at any time. It is nearly impossible to detect a crawler trap with only two consecutive pages. The concept we developed is to train a classifier progressively with  $n$  consecutive pages then to detect whenever a page is too different or too similar from the previous ones. To proceed, we first started by gathering pages from several different websites. This dataset<sup>15</sup> consists of two large families of websites.

On the one hand, those considered as normal (without crawler-traps). These are sites like Wikipedia, forums (Stack Exchange, Stack Overflow) or online media. They are sometimes taken in several languages (Wikipedia is very useful for that), on several different subjects (computer, video game, news, ...) and collected with our bot.

On the other hand, the second family is about crawler-trap websites. We have to admit that it exists really few crawler-traps, at least identified. A lot of crawler-traps implement very basic techniques which are nothing but hard to bypass with a correctly configured crawler. In fact, the real difficulties arise when the targeted website uses random pages method by adapting the content of the web-page when it is close to detect a bot instead of a real human. To achieve such a goal, it exists few scripts or programs able to perform this task. More than a long list, the following one aims to represent all the diversity we have found about:

- **notEvil**: *notEvil* is originally a search engine in the TOR network. On this website, there is an embedded game named Zork. This game is associated with an identifier and, on the page, there are three links to a new game. Hence, a crawler trap is created because once the bot hits this link, it will play games indefinitely. On the same page, a small part is for "Recent Public Channels" and changes every time a

---

<sup>14</sup>Thus, with all these different cases, we have a progression of the "similarity rate" of the pages inserted in the book. From almost no similarity with random pages to a very close one with derived pages. With our method, we expect to be able to sort all different pages in clusters but potentially indicate that the last cases (with high similarity) produce clusters are different from the original book, but closer to the original book than the first cases (with low similarity).

<sup>15</sup>In order to ensure *reproducibility* [1579] of results presented here, the dataset (514.3 MiB uncompressed) is available on <https://github.com/MaxenceD-ConfData/ForSe-2020>.

channel is created or someone responds to an old one. To determine if this change has an impact on your classification, we collected two sets: one is composed of webpages downloaded as the bot does (everything is collected in the same minute) and one set is for a long time gathering process (one webpage per hour).

- **Poison:** *Poison* [1580] is the most random crawler trap we ever met. There is no practical example available online, therefore we have downloaded the software (developed in 2003) and generated our sample with 500 pages. Through this, we had the possibility to play with several coefficients to generate a real random sample of pages:
  - Number of paragraphs [1:10]
  - A CSS background (adding HTML tags and data to create a colorful page) [0:1]
  - Minimum number of words per paragraph [25:75]
  - Number of words added to the minimum number of words per paragraph [25:100]

This crawler trap is based on the operating system dictionary, thus, a lot of words are uncommon and very advanced, even for native speakers (ex:*thionthiolic*). In this system, the probability of picking up the word "house" is the same as the word "chorioretinitis".

- **Tarantula:** This is the name of a PHP crawler trap available on GitHub [1581]. It generates a simple page with 1 to 10 new links and a text that contains between 100 and 999 words chosen randomly in a list of 1000 predefined words. *Tarantula* is not online so we have configured it on our local web server and generated 250 samples from the root address and 250 by following a random link of the page.

Technically speaking, a web page is composed of different parts. A crawler trap can have an impact on all of these different parts. The shape of the page (more precisely the way or the style in which it is written) and the content (words which are displayed to user's eyes) are equally relevant to classify. To proceed, we use the list of words held in the page, all HTML tags and metadata of the page located in the `<head>` tags. Each type of information can be analyzed separately or together as needed. But, for the sake of universalism of our detection method, we have privileged the content of a page as the *de facto* data where to perform the detection.

In our case, we are trying to detect whenever there is a web-page generated from a crawler-trap in a set of regular web-pages or not. According to our operational context, we are aggregating web-pages on-the-fly, which means we cannot know the full data set in advance. It means we have to check the difference of a new web-page from a set of already visited web-pages. This is doable by computing a distance  $\mathcal{D}$  between the new web-page and the current set of visited web-pages.

For optimization purposes, we cannot compute a distance between the new one and all pages of the current set. Since the distance between web-pages of a single family is supposed to be the same or close, we can consider the mean (expected value) as a good estimator. This mean is computed on-the-fly by updating it with each new value which belong to the family. It means that our detector system is based on the fact that the distance between a new sample  $s$  and the mean of a family  $m$  is based on the fact that  $\mathcal{D}(s_i, s_j) \leq \epsilon$  where  $\epsilon \geq 0$  is distinctive for each family.

The fact is that we are dealing with very specific objects that are web pages. Moreover, we have to deal with distance to solve a data-mining problem. In practice, the usual distances (Euclidean, Manhattan, Minkowski, Hamming) quickly appeared to be inefficient or to produce results with a significant variance. This tended to make the results unpredictable and inaccurate. It soon became clear that we needed to produce more suitable and efficient distances ourselves. After all, *clustering* is the process of gathering objects whose measured distance between them the smallest. The direct consequence means that the notion of distance is as central as the clustering strategy used.

In the end, our general procedure of detection has been based on a research work on two points. On the first hand, it has been a design of an unsupervised learning procedure and on the other hand, on evaluating new distances that would fit our needs. The general procedure of this work is schematized in Figure 7.26.

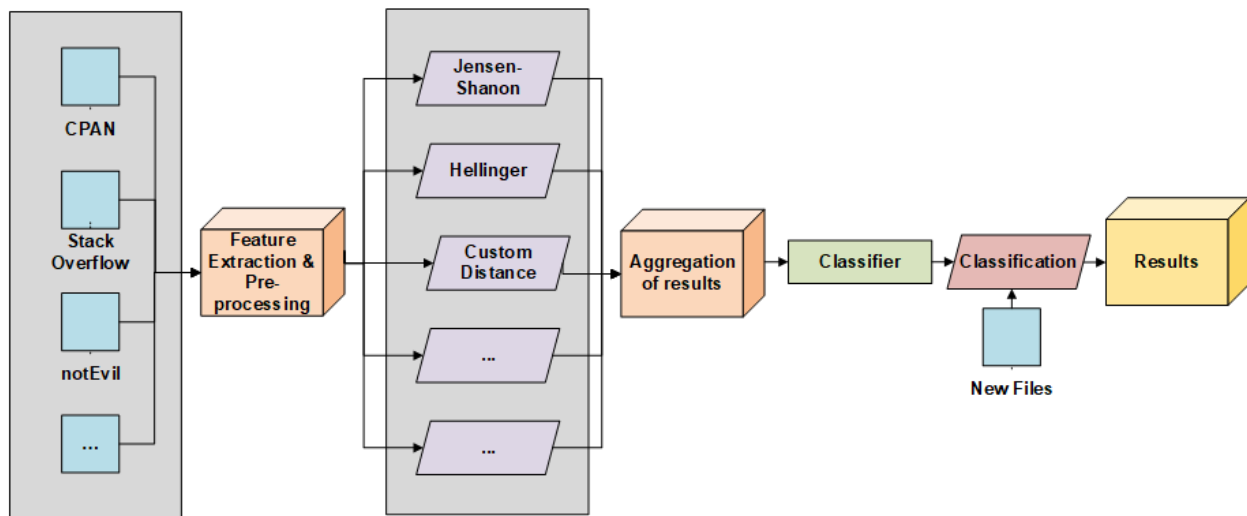


Figure 7.26: General resume of the procedure used to cluster web-pages in different families.

### 3.3.4 About new distances

Before all, we need some prerequisites of mathematical notation. For all distances, we consider the content of two web-pages as random values  $X$  and  $Y$  composed of words. Each word (or group of words) in a page has a probability of occurrence. The set of all of these probabilities constitutes a probability distribution usually noted  $P$  and  $Q$  from the same space  $\mathcal{X}$ .

The study on distances is a work that goes beyond the current context of this document. Nevertheless, for the sake of brevity, we can say that our work is based on information theory [1582, 1583, 1584, 1585, 1586, 1587]. In a way, our problem can be related to the problems treated by information theory as we try to characterize the distance between two objects that can be represented by probability distributions [1588, 1589]. Such objects are called *statistical distances* [1590].

As part of our research, we studied many distances from information theory. Many of these distances had more interesting characteristics and results than the more usual distances. Among the distances that gave us interesting results, we have: the Jensen-Shannon distance (JSD) [1591], the Hellinger distance (HD) [1592] and the Bhattacharyya's distance [1593, 1594].

If these distances provided interesting results, it appeared from their analysis that it would surely be possible to do better, by ourselves. Indeed, the results measured with the classical distances were clearly not precise enough (too many false positives and a low discrimination capacity in detection). This is why we have designed many custom distances to solve our problem more efficiently (with greater precision on the objects manipulated here). Indeed, there is a drawback on the distances used today in information theory which limits the precision in our case.

The main drawback of the existing distances we observed is that the random variables they consider are defined on the same space  $\mathcal{X}$ . It makes sense in simple cases but our one is a bit more specific. Indeed, we are computing the set of words page after page and it means we do not have access to the set of all words at the beginning. More directly, there is not way to preload all possible words from website in our operational context. In a page, we can find words which are not present in the next page (and vice-versa) for the sake of example.

The main issue when we use existing distance is that, to be mathematically exact, we must remove unique words from each page. The reason is explained in [1578], but for the sake of simplicity, we can say it is in order to keep mathematical properties with existing distances. In addition, as we compute the set of words, page after page, on-the-fly and considering unknown words on a page with a probability of zero does not fulfill all mathematical designs. By consequence, such a loss deprives us of information and therefore a loss of accuracy

for our measurements. The idea of our measures comes from this simple observation.

The distances presented here are summarized. Do not hesitate to refer to the article [1578] for more details. The names of all our distances are based on fish names. The reason is historical and lies in the fact that we needed a simple way to distinguish the many attempts that we have sought to evaluate our new distances. Note that the evaluation procedure led us to deal with *simulation* and *high performance computing* issues.

### 3.3.4.1 Custom distance - Shark

The idea of one of our measures, called *Shark* is based on the rate of the entropy (called  $H$ ) of the symmetric difference on the entropy of the two distributions. In other words, we look at the information contribution of the exclusive words from the two pages on all the information embedded in these two pages. Considering  $P \in \mathcal{X}$  and  $Q \in \mathcal{Y}$  we have  $P \setminus Q = \{\forall x \in X \text{ where } x \notin Y\}$ , we define *Shark distance* such as:

$$\mathcal{D}_s(P, Q) = \frac{H(P \setminus Q) + H(Q \setminus P)}{H(P) + H(Q)}$$

### 3.3.4.2 Custom distances - Archerfish and Handfish

We can make the hypothesis that on a web site (but the observation could be used in a broader context) the percentage of differences between two linked pages follow a globalized deviation. In other word, the average percentage of change between two pages from the same website should not be so different from a page to another one. This is why we try to minimize the average difference between two pages of the same family while maximizing the distance between two pages which are either too close or too different.

The arcsine distribution [1595, 1596] is interesting to model a distance which would follow our requirements. In probability theory, the arcsine distribution is known to have a cumulative distribution function such as  $F(x) = \frac{2}{\pi} \arcsin(\sqrt{x})$  and a probability density function  $\forall x \in [0, 1]$  with  $f(x) = \frac{1}{\pi\sqrt{x(1-x)}}$ . This is a rather paradoxical law, since the expectation is the least probable value and extreme values are the most probable [1597]. But for this law, it is possible to design specific distances able to provide interesting results in our case. Starting from the distribution function, we propose *Archerfish distance* such as, from two distributions  $P$  and  $Q$ ,  $\forall p_i \in P$  and  $q_i \in Q$ , we have:

$$\mathcal{D}_a(P, Q) = \frac{4}{\pi^2} \sum_{i=1}^n \arcsin(\sqrt{p_i}) \arcsin(\sqrt{q_i})$$

From the density function, using the same distributions  $P$  and  $Q$ , we propose another distance called *Handfish distance*:

$$\mathcal{D}_h(P, Q) = \frac{1}{\pi} \left| \frac{1}{\sqrt{p_i(1-p_i)}} - \frac{1}{\sqrt{q_i(1-q_i)}} \right|$$

### 3.3.4.3 Custom distances - Dottyback and Dhufish

We can use the arctangent distribution [1598, 1599] to match our needs. In the same way as with Archerfish and Handfish, we design two distances. One is directly extracted from arctangent definition and called *Dottyback distance*. It is defined  $\forall p_i \in P$  and  $q_i \in Q$ , such as:

$$\mathcal{D}_o(P, Q) = \sum_{i=1}^n \sqrt{p_i} \arctan(\sqrt{p_i}) \sqrt{q_i} \arctan(\sqrt{q_i})$$

Another distance is based on the derivative of arctangent function. This one is called *Dhufish distance* and it can be used to measure the distance between two distributions  $P$  and  $Q$ :

$$\mathcal{D}_u(P, Q) = \left| \frac{1}{1 + (p_i(1-p_i))} - \frac{1}{1 + (q_i(1-q_i))} \right|$$



### 3.3.5 Result on the efficiency of the new distances

We are not trying here to evaluate our crawler-trap detection model but to evaluate in an objective way the efficiency of the distances that we could use in the latter. This is why cross-validation [1600] — which is an usual method to evaluate the efficiency of a data-mining detection model — will not be used in this part. Moreover, it is not suitable for our detection model whose data-sets are built on-the-fly (page after page) nor for informing us about some "mathematical flaws" of some distances. More directly, we need a measure which makes sense when the population evolve in the time when the observation is performed. Since pages are collected on-the-fly, at a given time  $t_i$  statistical parameters of population<sup>16</sup>  $\text{Pop}(\mu, \sigma)$  can be different at time  $t_{i+1}$  with  $\text{Pop}(\mu', \sigma')$ .

The solution we propose is simpler and it goes back to the source of a statistical detection test as defined by Fisher [1601]. The detailed construction of the test we propose is quite long and we invite the reader to read [1578] for further details about it. For the sake of simplicity, we can say that it is based on the concept of confidence interval computed from a large number of repeated experiments. More directly, each distance has a confidence interval  $[\mu - \sigma; \mu + \sigma]$  computed from the same dataset shared between all distances. This confidence interval gives an idea of the *uncertainty* carried by a distance for a given measurement.

A distance that has a confidence interval with a small range (i.e.  $\sigma$  is small) is an accurate distance. Conversely, the larger  $\sigma$  is, the more the distance will tend to produce inaccurate results. But this requirement is not sufficient. The standard deviation value matters if and only if some of confidence intervals overlap. And it is this last point that matters. If there is an overlapping, it means there is an *area of interference* which leads to a possibility of misdetection. The goal of our designed distances is to reduce this overlapping so that each family has its own area without being covered by another. More directly, the less overlap there is between intervals, the more our distances produce values that clearly separate the different clusters (regular web-pages from crawler trapped ones, in our case) and the more accurate the detection model will be (since the distances are evaluated with large data sets coming from regular websites and crawler traps).

To ensure a certain efficiency of our distance evaluations, we push the *uncertainty* of our measurements to the limits. According to normal distribution properties and central limit theorem [1602, 1603], the interval  $[\mu - \sigma; \mu + \sigma]$  should contain 68 % of the observed distances applied to elements from a given family. More generally, we can consider the following intervals where we increase the confidence factor value  $t$  applied on the standard deviation:

- $[\mu - 1.96\sigma; \mu + 1.96\sigma]$  should contain 95 % of observations ;
- $[\mu - 3\sigma; \mu + 3\sigma]$  should contain 99 % of observations.

The goal of our evaluation was therefore to create a measure that can evaluate the overlapping rate of distances for an evolving index  $t \in [0, 3]$  driving confidence intervals such as  $[\mu - t\sigma; \mu + t\sigma]$ . With this evaluation, we can measure the impact of regular websites versus trapped ones. In Figure 7.27, the overlapping rate of the different confidence intervals is given on the y-axis. This one goes from 0 to 1 to represent an overlap rate between 0 to 100 %. On the x-axis represents the confidence coefficient  $t$ . Such construction makes possible to artificially increase the length of the intervals to evaluate our distances under extreme conditions. Plots on the chart represent the evolution of the overlapping rate when  $t$  is increasing, as explained previously. Technically speaking, the more efficient the distance is (which means the better the detection with the distance is), closer to the x-axis the plot modeling of the distance's evaluation must be.

In 7.27, we can see that Dottyback distance has good results, even in the worst case (impact of overlapping is 7.28 %) and Shark distance has 3.4 % of overlapping impact rate, which is even better. Hellinger distance has an impact of 17.31 %, Bhattacharyya 20.44 % and Jensen-Shannon 28.08 % which confirms that state-of-the-art distances induce more overlapping than the one we designed. Such impact index helps to predict bad classification by a distance from a cluster to another. Note that in the worst case where  $t = 3$  (from which the rates we quote are extracted), we should in theory cover 99 % of the observations with different websites. The

---

<sup>16</sup>Where  $\mu$  represents the mathematical mean and  $\sigma$  the standard deviation of a random variable from a measured population parameter.



lower this rate is, better the detection will be.

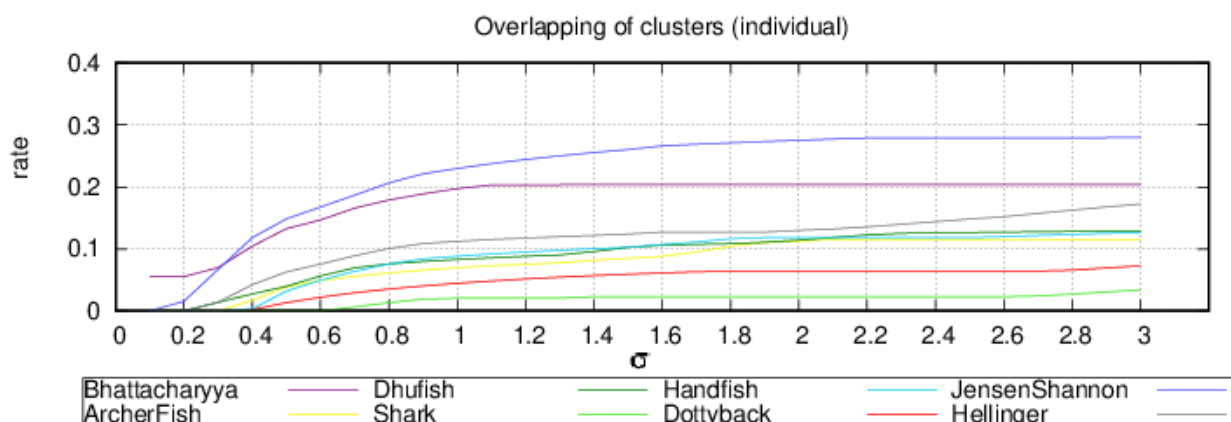


Figure 7.27: Overlapping rates computed between regular web-sites vs trapped ones. It helps to measure the impact of bad detection of cluster by a given distance.

### 3.3.6 Conclusion

With our new distances, our specific detection method, we have been able to solve the crawler-trap issue. The efficiency of the new distances created made it possible to overcome the effects of random generation and the difficulties of measurement inherent in the control of random processes. The consequence was to allow our bots to collect large amounts of data on the Tor network with impunity.

But more than this point, the resolution of an initially very specific problem has opened up multiple applications in various fields. It is an understatement to say that today the themes of artificial intelligence, big data and machine learning are very up-to-date. The creation of new distances can help to better model certain problems specific to the exploitation of these themes. In addition, our on-the-fly data processing method is also an excellent way to propose operational solutions that are relatively economical in terms of computing resources.

But perhaps the most important thing in this study is the way we approach the creation of new distances with a theoretical mathematical point of view. The distances given here are only a sample of what can be done. The research process behind the creation of these distances could (from our point of view) allow new distances to appear. This is a research perspective of particular interest to us. More directly, research is still underway on the subject. With a broader approach, directly embedded in the framework of information theory.

Even if it matters a lot for us, it would not have been reasonable to add one or more chapters on the subject. However, there is so much to say and some notions that have simply been mentioned here deserve relevant demonstrations and more extensive explanations. Moreover, other ideas, other distances, other theorems, built with original mathematical tools have also been created without being published or mentioned here. If our research could continue, our goal is to publish all our work in a separate book in the future. Indeed, the framework of this thesis is already wide enough and to express it more directly, long enough. It is therefore recommended to separate things and to use a dedicated support to further detail research.

## 4 Research contributions

### Contribution 9: Contributions of miscellaneous projects.

- ☞ During the doctoral stay in the Dr Web company, several industrial projects have been carried out.
  - ☞ Publication at the "Nuit du Hack" conference [1519] of new techniques able to bypass antivirus defense systems.
  - ☞ New Dll injection technique via the delayed load Dll mechanism.
  - ☞ Several low-level development projects (kernel and hypervisor) as well as research on polymorphic packed malware classification.
- ☞ Several projects were carried out with our students.
  - ☞ UEFI full encryption project with three articles published in the MISC journal to present UEFI technology.
  - ☞ Reverse engineering analysis of the Sysmain service (Superfetch) in Windows 10 [463].
  - ☞ Documentation and forensics tools presented at Black Hat USA 2020 [462] and JCVHT [463] about Superfetch.
  - ☞ Project of automatic collection of information on the networks (web and dark-net) with anti crawler-trap features [1578].
  - ☞ Mathematical theoretical work on information theory to create new distances for data-mining (more efficient than state-of-the-art distances).
- ☞ Management of scientific R&D projects with students.
  - ☞ Introduction to research with students and support in writing scientific articles.

## Chapter 8

# Conclusion & Future research work

### 1 Conclusion to the methodology used in this study

To conclude this research, it is appropriate to first recall the problematic that was ours at the beginning of this study (Chapter 1, section 3). The main subject was about to know how we could propose more efficient security solutions through an offensive and low-level approach in computer security. Therefore, this study was based on a methodological approach. That being said, the pure logic of research to evaluate such a methodological approach should have ordered a study of the different studies using this approach. But such a study would have been very sociological and outside the technical scope of the study we really wanted to conduct. More simply, the result was also known in advance, just by reading "The Art of War" from Sun Tzu with:

*"If you know the enemy and know yourself, you need not fear the result of a hundred battles..."*

As the interest of the approach seemed to be obvious for us, our approach has been to focus on this method as a means of action and not as an object of study. More directly, it was interesting for us to use this methodology to try to draw results from it and thus to demonstrate its efficiency.

Of course, evaluating a methodology in a global way could have been an approach. But such an evaluation might have lacked two important things. On the one hand, the practice of the experiment and on the other hand, concrete results that could really illustrate our approach. That is why we decided to focus on three different problems, each a given technical level. In any case, the idea was to show how it was possible to bypass the existing security in order to improve the security subsequently.

This way, our research approach allows us to act on two different aspects. On the one hand, from an offensive point of view, we can identify potential gaps between what exists today and what could be done to bypass current securities. In the end, this is a fairly classic approach to research. From a state of the art (coming from existing security or from targeted systems to be attacked), we are able to know very precisely how a system works in order to design new attacks. This first research approach allowed us to make contributions about new attacks. Of course, this study is not about providing new weapons to attackers. The goal is to better understand these new attacks to provide better security systems. Indeed, knowing about new and unpublished attacks allows either to reduce the possibilities for an attacker (because the attack is now identified) or to detect previously unknown ones which would have been rediscovered in this study. In both cases, it is an in-depth knowledge — an expertise — that allows us to design more effective defense systems.

On the other hand, it is precisely on this defensive point of view that our second research is performed. The approach here is the same as before with the offensive point of view, with one fundamental difference, our state-of-the-art is initially enriched by our offensive knowledge. And this is what really makes the difference. Because our state-of-the-art is not only guided by the existing solutions for a given problem, but also by a complete documentation of the problem. This complete documentation of the problem (which ultimately comes down to the art of asking the right questions in order to get better answers) is even boosted by bringing new

knowledge to the existing field of knowledge on the subject. From this state-of-the-art, it is then possible to provide more efficient security solutions. Either by providing a specific solution to the new attack we have proposed (thus increasing the size of the defended system) or by providing a global answer to the problem while covering our new attack.

We therefore propose a twofold research here. On the one hand by applying a research approach in the attack field and on the other hand in the defense area. In fact, we are in a research process that allows us to feed one with another one. Such approach allows us to propose twice more innovation as one responds to the other. In a way, the approach allows a kind of virtuous circle. The knowledge of one allows to progress on the knowledge of the other and so on.

Of course, there is still the requirement to answer directly to the problem to know how to provide security solutions through a low-level offensive approach in computer security. The introduction to this study proposed to focus on three sub-problems to illustrate this approach. This is what has been done here through the various chapters. By carving out a gap in the literature with findings presented in this study, we do hope it sincerely illustrate our approach with this methodology.

## 2 Contribution and significance carried out by our study

It seems important for us to resume to the main contributions and attached significance of this study that we may claim. This operation has already been done for each chapter in detail at the end of each of them (Contributions 1, 2, 3, 4, 5 and 6). In this way and to avoid any repetition, we propose instead to summarize our main contributions by presenting what really fill the gap with the existing literature, what provides an operational or directly useful result while placing the different problems treated in our initial offensive methodology.

### 2.1 Improving security at the software compilation level

One of our main results was to highlight the possibility of deliberately introducing backdoors into compiled software for real. Until now in the literature, such a phenomenon was already described and known, sometimes illustrated but either in a theoretical way (by taking an already trapped compiler) or by using old bugs (thus known and already corrected) or more or less *farfetched* (hard or impossible to implement in practice).

Our first contribution was to find a previously unknown and credibly exploitable vulnerability in a compiler able to introduce silently such a backdoor at compilation time. In fact, we found a decades-old bug in MASM compiler from Microsoft. Used for assembly language, specific to Microsoft but popular, the impact of such a finding should be limited compared to a similar finding in a compiler used for a more widely used programming language. But we show two important things here. On the first hand, that it is possible to find and exploit such bugs. We have shown that what was not fully possible in practice (it always remained more or less theoretical or only possible without strong assumptions) was definitively possible. And if we can do that on a given language, nothing should prevent us from doing it one day with other languages. On the other hand, how complex it is to fix such an issue, especially in software that might have been compiled in a *backdoored* form with this vulnerable compiler.

The offensive approach was clearly about to find a vulnerability in MASM and to show how to exploit it. We have shown how to build a system able to exploit a vulnerability. There is a clear "*weaponization*" approach here. However, knowing this vulnerability allowed not only to fix the problem (and thus prevent future attacks) but also to inform about potential impacted software, without being able to really take them into account. It is therefore a dual research that allows us to improve the overall security. Indeed, Microsoft published CVE-2018-8232 broadcast worldwide for an immediate security-fix update. In addition to promote our own work, it is also a way to improve computer security through the use of that one.

The impact of our work has been recognized through two international conferences (in Russia and in India). Other works followed on other compilers related to the assembly language, notably with a similar flaw (although not exploitable) in the NASM compiler (CVE-2019-6291). This is an effective way to measure the impact of our research, which is read and reproduced for wider application in this area. Of course, it should be kept in mind

---

that this type of research is addressed by a relatively small number researchers. Assembly language is not a mass sport and the review of the associated compilers is even less important. But it is still possible to continue the research here, on other compilers, to find potentially similar cases.

## 2.2 Piece of news about backdoors nowadays

Actuality of these last months is rich to illustrate that this phenomenon of backdoor implemented directly in the tools used for development remains a sensitive subject. Generally, the operations attempted are based on direct modification of source code. It is very similar to what has been as presented in Chapter 2, section 2.3.1 (Key-Point 2.2).

Firstly, according to Rasmus Lerdorf and Nikita Popov [1604], on March 28<sup>th</sup> 2021, two malicious commits were pushed to the *php-src*. The malicious code was not really an offensive code since it aimed to mention Zerodium company (specialized in buying and selling computer vulnerabilities) in the code. This action is more a proof of feasibility than a real attack. Chaouki Bekrar, CEO of Zerodium, replied on twitter<sup>1</sup> that his company has nothing to do with this case. From his point of view, this vulnerability was uninteresting despite the presence of a real bug. He said that nobody in the market wanted to buy this type of vulnerability and guessed that the authors decided to reveal their exploit, for lack of anything better.

This access to the PHP source code was realized by usurping the credentials of the two PHP developers. In fact, this operation had been possible because PHP's autonomous Git infrastructure represented a security risk. This forced the PHP community to give up its infrastructure and migrate to GitHub.

It should be noted, however, that attacking a scripting language to infect the projects that use it can be successful sometimes. Indeed, rather than attacking the central interpreter of a scripting language, it is sometimes more interesting to insert a backdoor in the modules used in addition to the latter. For instance, Andrey Polkovnichenko, Omer Kaspi and Shachar Menashe said to have discovered eight packages in Pypy (the *Python Package Index* where additional modules are present) implementing malicious codes able to steal credit cards and to inject code [1605, 1606]. From their analysis, they estimated that the malicious packages have been downloaded about 30,000 times. Thus, acting on more peripheral projects but used by developers for their own needs, it is still possible to try to introduce malicious code by this means.

Another example of backdoor in source code concerns the Linux kernel. Researchers at the American University of Minnesota have attempted to reintroduce vulnerabilities into the Linux source code. In a similar way to what had been attempted to Linux kernel source code years ago [110], they have created contributor accounts to propose fixes for some known linux kernel problems. These patches did neither really fix the problems nor they seemed to introduce a security hole right away.

But the Linux kernel community detected this fraud and Greg Kroah-Hartman, one of the main maintainers of the Linux kernel, after having observing their "*obviously-incorrect patches*" claimed such patches could only be done on purpose [1607]. In addition, he particularly disliked that all this work has been performed "*in bad faith*" to try to test the kernel community's ability to review "*known malicious*" changes" [1608]. It must be said that researchers were literally publishing about the fact that they managed to poison the Linux community by inserting patches that were not patches [1609]. Proof that the operation was carried out deliberately [1610]. The consequence was to remove all the patches proposed (over 80 different developers helped with the review and fixes [1611]) and to banish this university from the Linux kernel.

This case teaches us two things: on the one hand, it confirms our observations about the difficulty of introducing vulnerabilities into a particularly followed open-source project without being caught (in the middle or long term). On the other hand, the need to conduct scientific research with a clear ethic. The consequences for the researchers at Minnesota University were very clear. Their university publicly disavowed its researchers [1612] and the public apology from the researchers [1613] clearly did not change the situation [1614].

---

<sup>1</sup><https://twitter.com/cBekrar/status/1376469666084757506>

These recent attempts illustrate several points. On the one hand, there is a real will to introduce vulnerabilities from the software development stage. Unfortunately, these attempts are often promise to failure. Indeed, if they are public nowadays, it means they have been discovered. As with the direct modification of the Linux kernel years ago, this type of attack is very complicated to implement. On the other hand, they make us think that more technical attacks (on compilers for example) represent an interesting future. Indeed, more complex to identify, the effects produced are also complicated to detect, unlike direct modifications in the source code.

### 2.3 Improving automatic malware analysis by preventing evasion

There are a lot of claims that could be made in the analysis of malware evasion techniques. But overall, it is possible to promote three important points :

- The first one concerns the survey including all existing threats today in relation with evasion techniques from an automated analysis system. We performed an exhaustive compilation of the entire threat, particularly focused on manual and dynamic analysis evasion tactics.
- For both manual and automated modes, we present a detailed classification of malware evasion tactics and techniques. Factually, there were already articles in the scientific literature presenting the state-of-the-art about such a threat, but most of them trivially surveyed analysis evasion and provided no detailed overview.
- We provide a brief survey on countermeasures against evasive malware that the industry and academia is pursuing.

This survey was carried out within the framework of an international cooperation and recognized by a publication in the Association for Computing Machinery (ACM) journal. It is for us a standard of recognition of our work that is designed to be above all useful to other researchers. The technical explanations have the merit of gathering in a single document all the known methods used by the malware. Of course, the subject is specific to the detection of automated analysis environments, that is to say limited to industry and academic interested by this particular topic.

From this technical survey and still in the perspective of promoting a very technical offensive approach, we wondered if it was possible to improve the technology observed in malware. There is room for new techniques capable of detecting or escaping from an analysis environment. On the one hand, based on what is presented in the literature and by our own knowledge of the system and debuggers, we have proposed different techniques to pass the security of debuggers, in particular Windbg from Microsoft. The main findings show how it is possible to exploit a fine knowledge of assembly language to abuse the debugging tool. On the other hand, we have proposed a generic and universal method able to evade any type of environmental analysis tool (debugger, virtual machine or DBI). While the method proves to be very reliable for DBI and debuggers, it remains relatively probabilistic in the context of Virtual Machines. Still on the offensive side, we have sought to make our universal method more reliable, in particular by reducing the need to calibrate it prior to any use. Although the results are promising, it should be recognized that further investigation is still needed in this area.

To counter this type of attack, solutions may exist but vary from the simplest to the most complex ones. On the one hand, those able to abuse errors from analysis tools can be corrected by updating these analysis tools. By increasing their reliability, efficiency or by covering blind spots, it becomes possible to prevent these new attacks. On the other hand, concerning our universal method, the problem comes from a design flaw in the architecture of Intel or AMD processors. Although less severe in terms of consequences (the attack only has an impact on malware or antivirus scans), the consequences can be comparable to Spectre attack [337]. More directly, this means that the processors from these companies would have to be upgraded to correct the side effect we exploit to perform our evasion. The impact here is relatively important because there is no simple solution (to the best of our knowledge) to solve this problem otherwise (i.e. in a software way).

In the end, the impact of this work has identified some exploitable flaws in certain software and processors to successfully escape the analysis systems used in the antivirus industry. This is of course not such as to

fundamentally question the use of these tools, but it is a useful reminder that analysis tools are not infallible, that a constant watch and continuous research must always try to bypass them in order to enhance them subsequently.

## 2.4 Improving anti-keylogger security

In a more traditional way, this part of the study focused on conducting a more regular research exercise, but always trying to use attacking techniques as a basis for building a solid defense. As a reminder, our problem in this part was to propose a more efficient - if not definitive - solution to the keyloggers threat. The idea was to start from what existed already, to see what is done, what is successful and what is less efficient in order to propose a solution that is at least as effective and ideally better.

At first, we were interested by the internal mechanisms driving the keyboard. This approach allows us to understand in detail the various subtleties of the technology on which the problem we are trying to solve is based. An important contribution of our work was to document the whole keyboard interface. Starting with the mechanical action of a key of the device, following the processing of the signal by the hosting machine, documenting all the possible technologies to finally explain the internal details of Windows in the processing of the keyboard inputs, our work is to date (and to the best of our knowledge) the most complete that exists publicly. Although there are already different works on this subject of keyboard in general, and under Windows in particular, most of these works lacked details or they were not up to date. In practice, there was no comprehensive documentation in the literature that really went beyond the Microsoft documentation and was considered as factually accurate (some were even inaccurate).

Our work has helped to fill this gap. And more than filling the gap, it address the specific need to understand how the keyboard works under Windows operating system. Without this comprehension, how to deal with the keylogger problem? It would be like doing rocket science while ignoring Newtons' law of gravitation... It does not make sens to build a truly reliable system without technically understanding fundamental concepts that drive it. And this is why this important work allows us to better understand how to interface with the keyboard, both for offensive solutions with keyloggers and for defensive ones that aim to neutralize them. Note that this work may have a broader use. In particular, it can be used to document — sometimes partially — other relevant parts of Windows, for a better understanding of this close-source operating system. As a personal note, this was an opportunity for us to gain real expertise on a specific point of the Windows 10 architecture.

Nevertheless, this work should not be considered as a final and definitive work. At first, because our reverse engineering work is not fully complete since it does not include all the details on the subject. The present research work is sufficiently long although it was only focused on keyboard management. We had to make choices and detail the most important parts (and the least documented by Microsoft). But the main reason is simply because the Windows operating system evolves every day. In addition to the regular updates, Windows insiders program<sup>2</sup> provides regularly new versions of the operating system, ahead of the final and official versions. Will these new versions change the way the keyboard works, potentially making our work outdated?

It is of course not possible to give an absolute answer to such a question, as we cannot predict the future. Nevertheless, it is possible to postulate some predictions... On the one hand, if we look at the evolution since Windows XP, there have certainly been changes in the implementation (enhancement of security, more "object-oriented programming", and so on) but the philosophy of the main actions (and the main functions) has not changed so much. It can even be observed that the addition of new features tends to be done by building "on top" of the existing code, in order to preserve the *historical foundations*. This may be due to the ease of development or the need for backward compatibility. On the other hand, it must be seen that this key feature of the kernel is operational today and that there would be nothing<sup>3</sup> that would require to upset the current architecture. In addition, the architecture relies on device communication protocols (PS/2, USB/HID) that are now supported worldwide by most (if not all) hardware manufacturers. Making major changes would question the *interoperability* of existing software (drivers in the first place, but also perhaps the keyboard interface API and thus a lot of user-mode software) and hardware (fixed protocols) used today. To conclude, if changes are likely, they should remain relatively minor and ensure some backward compatibility with what is observed and

---

<sup>2</sup><https://insider.windows.com/en-us/>

<sup>3</sup>Excepting by considering major and unforeseeable events that would question the actual design of the keyboard management.



documented by our work.

Once the analysis of the keyboard operation is done, we studied the existing threat (as a "state-of-the-threat"). Again, our approach starts from the offensive point-of-view to propose improved or new defensive solutions. In our study, we have limited ourselves to a literature review of the threat. There was no need to innovate to better understand the threat. Why? Simply because our study of the keyboard already offered a sufficiently broad vision of the path taken by a key pressed on the keyboard. Making a new type of keylogger is amazingly easy. Similarly to what was done in [1115], every time a keystroke content is available in the system, it is possible to keep this information somewhere. Our up-to-date study of the system facilitates this operation (even if the result would be unstable because it is based on undocumented mechanisms).

The knowledge of the system and the *state-of-the-threat* allows us to consider the existing solutions to deal with it. There is a vast quantity of literature on the subject, both in the academic and industrial fields. On the first hand, we focused on solutions coming from the academic world. In fact, academic publications are often focused on a new innovative solution or on a compilation of existing solutions. There are a lot of "innovative" publications that explain how to protect against keyloggers. But generally, this type of publication reuses an existing strategy or an alternative version of an existing solution. On the other hand, compilations often present a laundry list of solutions without bringing a real coherency between all presented solutions. In both cases, there is no study that summarizes all the existing solutions and presents the main possible strategies. This is the main contribution of our *state-of-the-counter-threat*. We have provided a summary that identifies all possible strategies (illustrated with the different reference articles for each strategy) to address keyloggers in the academic world.

On the other hand, we focused on solutions coming from the industrial world. Our approach here has been to evaluate the set of possible strategies used by commercial software to address the keylogger threat. Our main contribution was to make a compilation about industrial anti-keylogger solutions. At the best of our knowledge, there is no public article providing such a view. Indeed, the absence of really formal documentation from the companies that develop these commercial software does not help to have a vision as clear as the one proposed by the academic world. It is also an opportunity to complete the set of different strategies provided by the academic world. As an aside, this was an opportunity for us to provide *an original documentation methodology* for these software, based on the absence of reverse engineering and therefore the use of their commercial documentation to find or guess their main features.

Another significant contribution that we can claim is the fact that we have taken a critical review with some of the proposed solutions. Indeed, our new findings — mostly in the internals of Windows 10 — seem to contradict or to invalidate some publications or commercial software. It was an opportunity to question the relevance of certain articles and the need to be able to reproduce the results presented in these ones.

From this body of knowledge, we were able to provide our own solution called GostxBoard. By making the synthesis of what has been presented in the defensive solutions against keylogger threat, it was possible to draft specifications. These specifications have included all the strong points observed on the different solutions to reduce their weak points. In fact, no solution (either coming from academic or industrial worlds) can fully address the threat. There are always sacrifices, both with the security provided (we have shown that for some solutions, they did not provide any security at all) as with the user experience. There was a real need to design a solution that was robust and secure enough while preserving the user experience and system stability. We have designed a solution based on ciphering keystroke transiting through the communication system used by Windows to protect volunteers software which would require a secure input. Such a design, based on original Windows mechanisms and integrated directly in the source code from third-party software allows to keep the user experience while improving the security. That way, with our solution, it is possible to provide a broader, more effective and better integrated protection compared to existing solutions. This is finally the interest of our work presented in this study and finally our main contribution to the research on anti-keylogging solutions.

But more than a result, we must consider two important points. On the one hand, the methodology (i.e. how we succeed to design this solution) and on the other hand the universality of the proposed approach to potentially adapt to other problems in the field of computer security. At first, we work to understand in depth

---

the *fundamental physics* on which all threats and solutions to a given problem are based. Regardless of any consideration for the threat, our initial study aims to understand without prejudice that could distort our analysis later. This understanding subsequently allows us to better understand the threat (because we know what threat can and cannot use and the constraints associated with each architectural choice) and the existing solutions (in the same way). This approach allows us to observe the blind spots of existing solutions and the intrinsic limits of malicious software while constituting the state-of-the-art from both offensive and defensive point-of-view. From our point of view, this is the most appropriate way to specify a more (or at least as) efficient solution.

## 2.5 General conclusion about the methodology presented in this research work

The last case dealing with keylogger is finally a global synthesis of all the methodological approaches undertaken within this study. It is just more complete, more specialized on a given technical point and more likely to produce significant results. Performing a preliminary and ultra-detailed study of a given technology before looking at the offensive point of view to design enhanced defensive solutions is also a methodological contribution to the approach that was initially considered when introducing this document.

Note that this initial step of ultra-technical documentation was already more or less present in the two problems discussed in chapters 2 and 3. Indeed, either in the science of compilers or in the science of assembly language with Intel or AMD CPU architecture, a strong prior knowledge of these subjects is required to conduct such research to potentially obtain significant results. This knowledge may have come from our own past experiences or from interactions we may have had with other researchers or students. But the process of formalizing this technical knowledge (in order to better capitalize on it later) is specific in our study to manage keylogger threat. This also shows the reinforcement of the research approach by this preliminary ultra-technical study, only driven by the technological framework in which the studied problem is located — and making abstraction of the hearth of the problem itself.

Of course, we are not here in a conceptual research approach on how to conduct or enhance research as a whole in computer security. Formally speaking, what we are employing here is not exactly new. Learning technical expertise on a given subject before starting a research project is in itself a fairly classic exercise. In a way, this can be seen as following a classical teaching before starting research. The subtle difference with a classic approach is that gaining new skills is totally autonomous and it is built as a research project in itself. In our case, before answering the question of knowing how to reduce the threat of keylogger, we addressed the question to know how the keyboard management works under Windows 10. It was through the answer to this fundamental research question that an applied research question could be subsequently answered.

We wanted to illustrate our approach, as a research experience, diversified on several highly technical subjects. In our case, the originality may come from the fact of applying our methodology step by step, in a concrete way, throughout this research work. The goal was to show that our methodical approach aims to ease the resolution of different problems in computer-security.

THIS PAGE INTENTIONALLY LEFT BLANK

---

# Bibliography

- [1] Microsoft, “Setwindowshookexa function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [2] S. Michael, H. Shawn, W. Tyler, J. Mike, B. Drew, and C. David, “Hooks overview,” tech. rep., Microsoft, 05 2018.
- [3] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual - Volume 2, 2-8*. Intel Corporation, 05 2019.
- [4] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: 8.1.3 Handling Self- and Cross-Modifying Code*. Intel, 2016.
- [5] K. Dhilung, J. Jiyong, and S. Marc Ph., “Deeplocker, concealing targeted attacks with ai locksmithing,” in *Black hat USA 2018*, (Las Vegas - USA), IBM Research, 08 2018.
- [6] C. point research, “Cyber security report 2020,” tech. rep., Check point software technologies LTD, 2020.
- [7] U. Organisation, “Universal serial bus specification 2.0,” tech. rep., USB.org, 4 2000.
- [8] U. I. Forum, “Device class definition for hid 1.11,” tech. rep., USB.org, 06 2001.
- [9] S. Michael, B. Drew, J. Mike, and C. David, “About keyboard input,” tech. rep., Microsoft, 05 2018.
- [10] I. Barankova, U. Mikhailova, and G. Lukyanov, “Software development and hardware means of hidden usb-keylogger devices identification,” *Journal of Physics: Conference Series*, vol. 1441, p. 012032, 01 2020.
- [11] S. Simms, M. Maxwell, S. Johnson, and J. Rrushi, “Keylogger detection using a decoy keyboard,” in *IFIP Annual Conference on Data and Applications Security and Privacy*, pp. 433–452, Springer Verlag, 06 2017.
- [12] M. Rao and S. Yalamanchili, “Novel shoulder-surfing resistant authentication schemes using text-graphical passwords,” *International Journal of Information and Network Security (IJINS)*, vol. 1, 07 2012.
- [13] W. Khedr, “Improved keylogging and shoulder-surfing resistant visual two-factor authentication protocol,” *Journal of Information Security and Applications*, vol. 39, pp. 41–57, 04 2018.
- [14] J. B. Baviskar, S. A. N. Prof Raut S.Y., Kharde Rahul S, and S. Yogesh, “Shoulder surfing and keylogger resistant using graphical password scheme,” *International Journal of Advanced Research in Computer Science*, vol. 5, no. 8, pp. 201–204, 2017.
- [15] A. Parekh, A. Pawar, P. Munot, and P. Mantri, “Secure authentication using anti-screenshot virtual keyboard,” *International Journal of Computer Science Issues*, vol. 8, 09 2011.
- [16] Intel, *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, 2016.
- [17] P. Pillai, “Windows 10 device guard and credential guard demystified,” *Techcommunity Microsoft IIS, IIS Support Blog*, 03 2019.
- [18] H. Ted and S. Tim, “System power states,” tech. rep., Microsoft, 06 2017.

- [19] S. Daniel and al., “Secure the windows 10 boot process,” tech. rep., Microsoft, 11 2018.
  - [20] H. Ted, C. David, M. Mike, G. Eliot, L. Bill, and K. Andrew, “Write a hid source driver by using virtual hid framework (vhf),” tech. rep., Microsoft, 04 2017.
  - [21] M. Kinney, “Platform initialization (pi) specification, pre-efi initialization core interface,” tech. rep., Tianocore, 05 2017.
  - [22] R. Samani, “Rapport de mcafee labs sur le paysage des menaces,” 11 2020.
  - [23] B. Barlowe, J. Blackbird, J. Faulhaber, H. Goudey, P. Henry, J. Kuo, M. Lauricella, K. Malcomson, N. Ng, L. Ragragio, T. Rains, E. Scott, J. Sesso, and F. Simorjay, “The evolution of malware and the threat landscape - a 10-year review, microsoft security intelligence report: Special edition,” 02 2012.
  - [24] McAfee, “2021 threat predictions report,” 11 2020.
  - [25] N. I. of Standards and Technology, “Statistics results for vulnerabilities,” 2021.
  - [26] G. Blokdyyk, *DevSecOps a Complete Guide - 2020 Edition*. Emereo Pty Limited, 2019.
  - [27] L. Thomas and C. Nicolas, “Dod enterprise devsecops reference design,” tech. rep., Department of Defense (DoD), 08 2019.
  - [28] E. Al Daoud, I. Jebiril, and B. Zaqaibeh, “Computer virus strategies and detection methods,” *Int. J. Open Problems Compt. Math*, vol. 1, 10 2008.
  - [29] M. Yusuf, W. Neyole, M. Jacob, Y. Muchelule, and J. Neyole, “Review of viruses and antivirus patterns,” *Journal of Computer Science and Technology*, vol. 17, 01 2017.
  - [30] J. Aycock, *Computer Viruses and Malware*. Boston, MA: Springer Science, 2006.
  - [31] J. Ma, J. Dunagan, H. Wang, S. Savage, and G. Voelker, “Finding diversity in remote code injection exploits,” in *6th ACM SIGCOMM Conference on Internet Measurement 2006*, (Rio de Janeiro - Brazil), pp. 53–64, 10 2006.
  - [32] S. Biswas, M. Sohel, M. Sajal, T. Afrin, T. Bhuiyan, and M. M. Hassan, “A study on remote code execution vulnerability in web applications,” *International Conference on Cyber Security and Computer Science (ICONCS’18)*, p. 8, 10 2018.
  - [33] T. computing Group, “Tpm 2.0 library - specification,” tech. rep., TCG Published, 11 2019.
  - [34] Statista, “Monthly market share held by windows operating system for desktop pcs worldwide from january 2017 to december 2020, by version,” 01 2021.
  - [35] StatCounter, “Desktop windows version market share worldwide - mar 2020 - mar 2021,” 03 2021.
  - [36] A. Bulazel and B. Yener, “A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web,” in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, p. 2, ACM, 2017.
  - [37] Y. Gao, Z. Lu, and Y. Luo, “Survey on malware anti-analysis,” in *Fifth International Conference on Intelligent Control and Information Processing (ICICIP)*, pp. 270–275, IEEE, 2014.
  - [38] J. A. Marpaung, M. Sain, and H.-J. Lee, “Survey on malware evasion techniques: State of the art and challenges,” in *2012 14th International Conference on Advanced Communication Technology (ICACT)*, pp. 744–749, IEEE, 2012.
  - [39] M. Ahmed, M. Shoikot, J. Hossain, and A. Rahman, “Keylogger detection using memory forensic and network monitoring,” *International Journal of Computer Applications (0975 - 8887)*, vol. 177, 10 2019.
  - [40] S. Anandappan, P. S C, J. Mathalairaj, C. Prathap, and L. Vignesh, “Keyloggers software detection techniques,” *2016 10th International Conference on Intelligent Systems and Control (ISCO)*, pp. 1–6, 01 2016.
-

- [41] E. of Ra, “Windows keylogger part 2: Defense against user-land,” *eye of ra blog*, 06 2017.
  - [42] S. Ortolani and B. Crispo, “Noisykey: Tolerating keyloggers via keystrokes hiding,” *Hotsec 12*, 2012.
  - [43] S. Sonal and W. Ujwala H., “Keylogging: A malicious attack,” *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 5, 06 2016.
  - [44] Y. Cheng and D. Xuhua, “Virtualization based password protection against malware in untrusted operating systems,” in *TRUST 2012*, vol. 7344, International Conference on Trust and Trustworthy Computing, 06 2012.
  - [45] E. Ducros, “Daniel benabou (ceidig): ”la cybercriminalit   s’industrialise, les entreprises ne l’ont pas compris”,” *arstechnica*, 04 2021.
  - [46] A.-U.-H. Yasar, D. Preuveneers, Y. Berbers, and G. Bhatti, “Best practices for software security: An overview,” in *2008 IEEE International Multitopic Conference*, 12 2008.
  - [47] M. Dowd, J. McDonald, and J. Schuh, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006.
  - [48] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav, “A survey of static analysis methods for identifying security vulnerabilities in software systems,” *IBM Syst. J.*, vol. 46, pp. 265–288, 2007.
  - [49] E. Penttila, “Improving c++ software quality with static code analysis,” Master’s thesis, Aalto University - School of Science, 06 2014.
  - [50] J. Novak, A. Krajnc, and R. Zontar, “Taxonomy of static code analysis tools,” *The 33rd International Convention MIPRO*, pp. 418–422, 2010.
  - [51] P. Emanuelsson and U. Nilsson, “A comparative study of industrial static analysis tools,” *Electronic Notes in Theoretical Computer Science*, vol. 217, pp. 5–21, 02 2008.
  - [52] A. Arusoae, S. Ciobaca, V. Craciun, D. Gavrilut, and D. Lucanu, “A comparison of open-source static analysis tools for vulnerability detection in c/c++ code,” *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pp. 161–168, 09 2017.
  - [53] R. Mahmood and Q. Mahmoud, “Evaluation of static analysis tools for finding vulnerabilities in java and c/c++ source code,” *ArXiv*, 05 2018.
  - [54] Z. Zhioua and Y. Roudier, “Static code analysis for software security verification: Problems and approaches,” in *Proceedings - IEEE 38th Annual International Computers, Software and Applications Conference Workshops, COMPSACW 2014*, 07 2014.
  - [55] A. Fatima, S. Bibi, and R. Hanif, “Comparative study on static code analysis tools for c/c++,” *2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pp. 465–469, 2018.
  - [56] C. Robertson, *Code analysis for C/C++ overview*, 04 2018.
  - [57] S. Zeisberger and B. Irwin, “A fuzz testing framework for evaluating and securing network applications,” in *SATNAC 2011*, (East London, South Africa), 09 2011.
  - [58] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS ’18*, (New York, NY, USA), pp. 2123–2138, Association for Computing Machinery, 2018.
  - [59] H. Dai, C. Murphy, and G. Kaiser, “Confu: Configuration fuzzing testing framework for software vulnerability detection,” *International journal of secure software engineering (Print)*, vol. 1, pp. 41–55, 01 2010.
  - [60] A. Gosain and G. Sharma, “A survey of dynamic program analysis techniques and tools,” *Advances in Intelligent Systems and Computing*, vol. 327, pp. 113–122, 01 2014.
-

- 
- [61] K. Hazelwood, *Dynamic Binary Modification: Tools, Techniques, and Applications*, vol. 6. San Rafael : Morgan & Claypool Publishers, 03 2011.
- [62] P. J. Guo, “A scalable mixed-level approach to dynamic analysis of C and C++ programs,” Master’s thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, 05 2006.
- [63] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” *SIGPLAN Not*, vol. 40, pp. 190–200, 06 2005.
- [64] D. Panchal, R. Maher, and S. Deshmukh, “Program analysis using code instrumentation techniques,” *International Journal of Advanced Research in Computer Science and Software Engineering*, 10 2015.
- [65] G. Lueck, H. Patil, and C. Pereira, “Pinadx: An interface for customizable debugging with dynamic instrumentation,” *Proceedings - International Symposium on Code Generation and Optimization, CGO 2012*, 03 2012.
- [66] V. Weaver and S. McKee, “Using dynamic binary instrumentation to generate multi-platform simpoints: Methodology and accuracy,” in *HiPEAC*, vol. 4917 of *HiPEAC’08*, (Berlin, Heidelberg), pp. 305–319, Springer-Verlag, 01 2008.
- [67] D. C. D’Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro, “Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed),” *ACM Asia Conference on Information, Computer and Communications Security (ASIACCS 2019)*, pp. 15–27, 07 2019.
- [68] O. Levi, “Pin - a dynamic binary instrumentation tool,” 06 2012.
- [69] D. community, “Dynamorio dynamic instrumentation tool platform,” 02 2009.
- [70] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” (*PLDI ’07*) *ACM*, 06 2007.
- [71] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *USENIX Annual Technical Conference, FREENIX Track*, 2005.
- [72] K. T. Kalleberg and O. A. V. Ravnas, “Testing interoperability with closed-source software through scriptable diplomacy,” (*FOSDEM ’16*), 01 2016.
- [73] J. Fiedor and T. Vojnar, “Anaconda: A framework for analysing multi-threaded c/c++ programs on the binary level,” in *RV* (S. Qadeer and S. Tasiran, eds.), vol. 7687 of *Lecture Notes in Computer Science*, pp. 35–41, Springer, 2012.
- [74] C. Tessier and C. Hubain, “Qbdi - quarkslab dynamic binary instrumentation home page,” 09 2015.
- [75] A. Bernat and B. Miller, “Anywhere, any-time binary instrumentation,” *PASTE ’11*, pp. 9–16, 09 2011.
- [76] F. Pastore, L. Mariani, A. Goffi, M. Oriol, and M. Wahler, “Dynamic analysis of upgrades in c/c++ software,” in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pp. 91–100, 11 2012.
- [77] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey,” *Cybersecurity*, vol. 1, 12 2018.
- [78] A. Aggarwal and P. Jalote, “Integrating static and dynamic analysis for detecting vulnerabilities,” in *30th Annual International Computer Software and Applications Conference (COMPSAC’06)*, vol. 1, pp. 343–350, 2006.
- [79] A. Russo and A. Sabelfeld, “Dynamic vs. static flow-sensitive security analysis,” in *2010 23rd IEEE Computer Security Foundations Symposium*, pp. 186–199, 09 2010.
- [80] H. Myrbakken and R. Colomo-Palacios, “Devsecops: A multivocal literature review,” in *International Conference on Software Process Improvement and Capability Determination*, pp. 17–29, 09 2017.
-



- 
- [81] H. Mason, G. Rumney, J. Neff, J. Jasen, and J. Caraballo-Vega, “Devsecops: Automating security,” 08 2019.
- [82] K. Zetter, “How a crypto ‘backdoor’ pitted the tech world against the nsa,” *Wired*, 09 2013.
- [83] mlk3, “Multiple vulnerabilities in d’link dir-600 and dir-300 (rev b),” 02 2013.
- [84] Craig, “Reverse engineering a d-link backdoor,” 10 2013.
- [85] S. L. Thomas and A. Francillon, “Backdoors: Definition, deniability and detection,” in *RAID 2018, 21st International Symposium on Research in Attacks, Intrusions and Defenses* (Springer, ed.), (Heraklion, Crete, Greece), Springer, 09 2018.
- [86] C. Heffner, “Finding and reverse engineering backdoors in consumer firmware,” in *EELive! Featuring esc - the embedded systems conference*, EE Times, 04 2014.
- [87] M. Malyutin, “Cve-2017-5689 - windows cryptoapi spoofing vulnerability,” 02 2017.
- [88] S. Morgenroth, “Type juggling authentication bypass vulnerability in cms made simple,” *Netsparker Web Security Blog*, 07 2018.
- [89] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmallice - automatic detection of authentication bypass vulnerabilities in binary firmware,” in *NDSS Symposium 2015*, p. 15, 02 2015.
- [90] S. Thomas, T. Chothia, and F. Garcia, “Stringer: Measuring the importance of static data comparisons to detect backdoors and undocumented functionality,” in *European Symposium on Research in Computer Security - Computer Security - ESORICS 2017*, pp. 513–531, Springer, 08 2017.
- [91] S. J. Tan, S. Bratus, and T. Goodspeed, “Interrupt-oriented bugdoor programming: a minimalist approach to bugdooring embedded systems firmware,” in *ACSAC '14, 30th Annual Computer Security Applications Conference*, 2014.
- [92] D. Andriessse and H. Bos, “Instruction-level steganography for covert trigger-based malware,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 8550, pp. 41–50, 07 2014.
- [93] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltzidas, “Implementation and implications of a stealth hard-drive backdoor,” in *ACSAC '13*, (New Orleans, Louisiana, United States), pp. 279–288, Association for Computing Machinery, 12 2013.
- [94] J. Heasman, “Implementing and detecting a pci rootkit,” in *Black Hat USA 2007*, NGSSoftware Insight Security Research (NISR) Publication - Next Generation Security Software Ltd, 2007.
- [95] A. Peterson, “Why everyone is left less secure when the nsa doesn’t help fix security flaws,” 10 2013.
- [96] G. Research and A. Team, “Inside the equationdrug espionage platform,” 03 2015.
- [97] J. Oakley and S. Bratus, “Exploiting the hard-working DWARF: Trojan and exploit techniques with no native executable code,” in *5th USENIX Workshop on Offensive Technologies (WOOT 11)*, (San Francisco, CA), USENIX Association, 08 2011.
- [98] S. Rebecca, B. Sergey, and S. Sean W., ““weird machines” in ELF: A spotlight on the underappreciated metadata,” in *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, (Washington, D.C.), USENIX Association, 08 2013.
- [99] T. Dullien, “Weird machines, exploitability, and provable unexploitability,” *IEEE Transactions on Emerging Topics in Computing*, vol. 8, pp. 391–403, 12 2017.
- [100] Y. Zhang and V. Paxson, “Detecting backdoors,” in *9th USENIX Security Symposium (USENIX Security 00)*, (Denver, CO), USENIX Association, 08 2000.
- [101] C. Wysopal and C. Eng, “Static detection of application backdoors,” *Datenschutz und Datensicherheit - DuD*, vol. 34, 01 2007.
-

- [102] P. A. Karger and R. R. Schell, “Multics security evaluation: Vulnerability analysis,” *Deputy for Command and Management Systems (MC I) Electronic Systems Division (AFSC) Hanscom AFB, MA 01730*, vol. 2, p. 156, 06 1974.
- [103] E. Filiol, “Dynamic cryptographic backdoors,” in *CanSecWest 2011*, 03 2011.
- [104] A. Langley, “Public key pinning,” 05 2011.
- [105] US-CERT, “Borland/inprise interbase sql database server contains backdoor superuser account with known password,” 01 2001.
- [106] A. Fernandes, “Microsoft, the nsa, and you,” 08 1999.
- [107] D. Marshall and T. Hudek, *Symbols and Symbol Files*, 05 2017.
- [108] Microsoft, “Microsoft says speculation about security and nsa is ”inaccurate and unfounded,” 09 1999.
- [109] B. Schneier, “Nsa key in microsoft crypto api?,” 09 1999.
- [110] Corbet, “An attempt to backdoor the kernel,” 11 2003.
- [111] F. Daigniere, “Cisco unified videoconferencing multiple vulnerabilities - cve-2010-3037 cve-2010-3038,” 11 2010.
- [112] Q. S. Advisory, “15 years later: Remote code execution in qmail (cve-2005-1513),” 05 2020.
- [113] E. Bouillon, “Stealing credentials for impersonation,” in *hack.lu*, 10 2010.
- [114] Wallet, “Statement on npm package vulnerability in v5.0.2-5.1.0 of copay wallets,” 11 2018.
- [115] K. Thompson, “Reflections on trusting trust,” *Commun. ACM*, vol. 27, pp. 761–763, Aug. 1984.
- [116] P. A. Karger and R. R. Schell, “Thirty years later: Lessons from the multics security evaluation,” in *Proceedings of the 18th Annual Computer Security Applications Conference, ACSAC '02, (USA)*, p. 119, IEEE Computer Society, 2002.
- [117] D. of Defense, *DEPARTMENT OF DEFENSE TRUSTED COMPUTER SYSTEM EVALUATION CRITERIA*, 12 1985.
- [118] K. Thompson, “On trusting trust,” *Unix Review*, pp. 70–74, Aug. 1989.
- [119] D. Wheeler, “Fully countering trusting trust through diverse double-compiling,” Master’s thesis, George Mason University, Fairfax, VA, 2009.
- [120] J. Thornburg, “?backdoor in microsoft web server?,” 04 2000.
- [121] A. Szegedi, A. Berger, and D. Smith, “Compiler bug, a semi-mythical beastie.,” March 2010.
- [122] V. Le, C. Sun, and Z. Su, “Finding deep compiler bugs via guided stochastic program mutation,” *OOPSLA '15*, June 2011.
- [123] J. Regehr, “Is that a compiler bug?,” February 2010.
- [124] E. S. Raymond, “When you see a heisenbug in c, suspect your compiler’s optimizer,” 02 2010.
- [125] C. Sun, V. Le, Q. Zhang, and Z. Su, “Toward understanding compiler bugs in gcc and llvm,” *ISSTA '16*, 2016.
- [126] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in c compilers,” *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*, June 2011.
- [127] P. Godefroid, A. Kiezun, and M. Levin, “Grammar-based whitebox fuzzing,” *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 43, pp. 206–215, 05 2008.
-

- 
- [128] E. Eide and J. Regehr, “Volatiles are miscompiled, and what to do about it,” in *EMSOFT '08*, (New York, NY, USA), pp. 255–264, Association for Computing Machinery, 10 2008.
- [129] W. M. McKeeman, “Differential testing for software,” *Digital Technical Journal*, vol. 10, pp. 100–107, 1998.
- [130] F. Sheridan, “Practical testing of a c99 compiler using output comparison,” *Softw., Pract. Exper.*, vol. 37, pp. 1475–1488, 11 2007.
- [131] A. Boujarwah and K. Saleh, “Compiler test case generation methods: a survey and assessment,” *Information and Software Technology*, vol. 39, no. 9, pp. 617–625, 1997.
- [132] C. Lindig, “Random testing of c calling conventions,” in *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging, AADEBUG '05*, (New York, NY, USA), pp. 3–12, Association for Computing Machinery, 2005.
- [133] C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang, “Automated test program generation for an industrial optimizing compiler,” in *Proceedings of the 2009 ICSE Workshop on Automation of Software Test, AST 2009*, pp. 36–43, 05 2009.
- [134] V. Le, M. Afshari, and Z. Su, “Compiler validation via equivalence modulo inputs,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, (New York, NY, USA), pp. 216–226, Association for Computing Machinery, 2014.
- [135] L. Simon, D. Chisnall, and R. Anderson, “What you get is what you c: Controlling side effects in mainstream c compilers,” in *2018 IEEE European Symposium on Security and Privacy*, pp. 1–15, 04 2018.
- [136] C. Sun and V. Z. Le, Su, “Finding and analyzing compiler warning defects,” *ICSE '16*, 2016.
- [137] S. Bauer, P. Cuoq, and J. Regehr, “Deniable backdoors using compiler bugs,” *International Journal of Poc - GFTO 0x08*, pp. 7–9, 2015.
- [138] C. Cristian, P. Luis, and R. John, “Multi-version execution defeats a compiler-bug-based backdoor,” 11 2015.
- [139] I. L. C. Team, “Bug 15940 - wrong constant folding,” 09 2013.
- [140] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar, “Compiler fuzzing: How much does it matter?,” *Proc. ACM Program. Lang.*, vol. 3, 10 2019.
- [141] M. Marcozzi, Q. Tang, A. Donaldson, and C. Cadar, “A systematic impact study for fuzzer-found compiler bugs,” 02 2019.
- [142] H. Tipton and M. Nozaki, *Information Security Management Handbook, Volume 6*, vol. 6. CRC Press, 2016.
- [143] C. Hathhorn, C. Ellison, and G. Rosu, “Defining the undefinedness of c,” *ACM SIGPLAN Notices*, vol. 50, pp. 336–345, 06 2015.
- [144] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. Kaashoek, “Undefined behavior: What happened to my code?,” in *Proceedings of the Asia-Pacific Workshop on Systems*, vol. 1 of *APSYS '12*, (New York, NY, USA), Association for Computing Machinery, 07 2012.
- [145] A. Pardoe, “C++ standards conformance from microsoft,” 3 2017.
- [146] M. Luparu, “Bring your c++ code to visual studio,” 5 2017.
- [147] Intel, *Intel(R) 64 and IA-32 Architectures, Software Developer's Manual*, 9 2016.
- [148] C. Robertson, “Microsoft macro assembler reference,” tech. rep., Microsoft, May 2016.
-

- [149] W. Peggy and C. McGeever, “The ibm pc macro assembler was released in december 1981,” *InfoWorld*, vol. 7, January 1985.
- [150] M. Martin, “Macro assembler update adds high-level features,” *InfoWorld*, vol. 13, April 1991.
- [151] S. Hutchesson, “Historical versions of masm,” 2015.
- [152] Microsoft, *ML and ML64 command-line reference*, 09 2020.
- [153] M. Corporation, “Programmer’s guide: Microsoft masm : Assembly-language development system : Version 6.1 : Form ms-dos and windows operating systems,” *Microsoft Corporation*, pp. 141–142, 1992.
- [154] R. Colin, J. Mike, C. Saisang, B. Mike, and G. Hogenson, *LOCAL*, 12 2019.
- [155] Microsoft, *Runas*, 08 2016.
- [156] S. Hutchesson, “Myths about masm,” 2015.
- [157] D. Childs, “Security update review,” 07 2018.
- [158] K. V. Hanford, “Automatic generation of test cases,” *IBM Systems Journal*, vol. 9, no. 4, pp. 242–257, 1970.
- [159] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” *ArXiv e-prints*, jan 2018.
- [160] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown,” *ArXiv e-prints*, jan 2018.
- [161] B. David, “Vulnerability in compiler leads to stealth backdoor in software (extended version),” in *Zero night 2018*, (Saint-Petersburg, Russia), 11 2018.
- [162] B. David, “Vulnerability in compiler leads to stealth backdoor in software,” in *International Cyber Security and Policing Conference C0c0n*, (Kochi, India), 10 2018.
- [163] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, “Dynamic analysis of malicious code,” *Journal in Computer Virology*, vol. 2, no. 1, pp. 67–77, 2006.
- [164] F. Cohen, “Computer viruses,” *Computers & Security*, vol. 6, pp. 22–35, 02 1987.
- [165] F. Cohen, *Computer Viruses*. PhD thesis, FACULTY OF THE GRADUATE SCHOOL - UNIVERSITY OF SOUTHERN CALIFORNIA, 01 1986.
- [166] D. Distler and C. Hornat, “Malware analysis: An introduction,” *SANS Institute InfoSec Reading Room*, pp. 18–19, 2007.
- [167] B. B. Rad, M. Masrom, and S. Ibrahim, “Camouflage in malware: from encryption to metamorphism,” *International Journal of Computer Science and Network Security*, vol. 12, no. 8, pp. 74–83, 2012.
- [168] M. Schiffman, “A brief history of malware obfuscation,” 2010.
- [169] I. You and K. Yim, “Malware obfuscation techniques: A brief survey,” in *2010 International conference on broadband, wireless computing, communication and applications*, pp. 297–300, IEEE, 2010.
- [170] J.-M. Borello and L. Me, “Code obfuscation techniques for metamorphic viruses,” *Journal in Computer Virology*, vol. 4, no. 3, pp. 211–220, 2008.
- [171] A. Moser, C. Kruegel, and E. Kirda, “Limits of static analysis for malware detection,” in *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pp. 421–430, IEEE, 2007.
- [172] F. Guo, P. Ferrie, and T.-C. Chiueh, “A study of the packer problem and its solutions,” in *International Workshop on Recent Advances in Intrusion Detection*, pp. 98–115, Springer, 2008.
- [173] H. Mourad, “Sleeping your way out of the sandbox,” *SANS Security Report*, 2015.
-

- 
- [174] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, “Malware dynamic analysis evasion techniques: A survey,” *ACM Comput. Surv.*, vol. 52, 11 2019.
- [175] F. Plumerault and B. David, “Exploiting flaws in windbg: how to escape or fool debuggers from existing flaws,” *Journal of Computer Virology and Hacking Techniques*, vol. 16, pp. 173–183, 06 2020.
- [176] F. Plumerault and B. David, “Dbi, debuggers, vm: gotta catch them all: How to escape or fool debuggers with internal architecture cpu flaws?,” *Journal of Computer Virology and Hacking Techniques*, 02 2021.
- [177] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer, *et al.*, “A secure environment for untrusted helper applications: Confining the wily hacker,” in *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, vol. 6, pp. 1–1, 1996.
- [178] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, “Ether: malware analysis via hardware virtualization extensions,” in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 51–62, ACM, 10 2008.
- [179] M. Tarral, “Hypervisor-level debuggerbenefits & challenges,” in *hack.lu*, 10 2018.
- [180] Y. Oyama, “Trends of anti-analysis operations of malwares observed in api call logs,” *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 1, pp. 69–85, 2018.
- [181] C. Kruegel, “How to build an effective malware analysis sandbox,” 2014.
- [182] M. Sikorski and A. Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.
- [183] S. Threat Hunter Team, “Threat landscape trends - q2 2020,” 08 2020.
- [184] S. Threat Hunter Team, “Threat landscape trends - q3 2020,” 12 2020.
- [185] N. A. Quynh and K. Suzaki, “Virt-ice: Next-generation debugger for malware analysis,” *Black Hat USA*, 2010.
- [186] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, “Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware,” in *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC*, pp. 177–186, IEEE, 2008.
- [187] R. R. Branco, G. N. Barbosa, and P. D. Neto, “Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies,” *Black Hat*, 2012.
- [188] Apriorit, “Anti debugging protection techniques,” 2016.
- [189] N. Falliere, “Windows anti-debug reference,” *”http://www.security-focus.com/infocus/1893”*, 2007.
- [190] F. Peter, “The ”ultimate” anti-debugging reference,” 05 2011.
- [191] J. Tully, “An anti-reverse engineering guide,” 9 Nov 2008.
- [192] M. V. Yason, “The art of unpacking,” *Retrieved Feb*, vol. 12, p. 2008, 2007.
- [193] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, “Using hardware features for increased debugging transparency,” in *2015 IEEE Symposium on Security and Privacy (SP)*, pp. 55–69, IEEE, 2015.
- [194] H. Shi and J. Mirkovic, “Hiding debuggers from malware with apate,” in *Proceedings of the Symposium on Applied Computing*, pp. 1703–1710, ACM, 2017.
- [195] B. community, “Bochs x86 pc emulator,” 07 2014.
- [196] N. A. Q. . D. H. Vu, “Unicorn the ultimate cpu emulator,” 04 2017.
- [197] Aggarwal, S. Kumar, and S. M. Kumar, “Debuggers for programming languages,” in *The Compiler Design Handbook*, pp. 297–329, CRC Press, 2002.
-

- [198] M. Chourdakis, “Toggle hardware data/read/execute breakpoints programmatically,” 2008.
  - [199] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, 6-13*. Intel, 05 2019.
  - [200] Microsoft, “Debugging functions,” tech. rep., Microsoft, 05 2018.
  - [201] M. Satran, “Creating a basic debugger,” tech. rep., Microsoft, 05 2018.
  - [202] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, 3-17*. Intel, 05 2019.
  - [203] S. Gao and Q. Lin, “Debugging classification and anti-debugging strategies,” in *Fourth International Conference on Machine Vision (ICMV 2011): Computer Vision and Image Analysis; Pattern Recognition and Basic Technologies*, vol. 8350, p. 83503C, International Society for Optics and Photonics, 2012.
  - [204] Y. Oleh, “Ollydbg,” 02 2014.
  - [205] L. community, “The llvm compiler infrastructure,” 05 2019.
  - [206] R. community, “Radar2,” 07 2019.
  - [207] G. community, “Gdb: The gnu project debugger,” 05 2019.
  - [208] Microsoft, “First look at the visual studio debugger,” 07 2019.
  - [209] Microsoft, “Download the windows driver kit (wdk),” 08 2018.
  - [210] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, “Ether: malware analysis via hardware virtualization extensions,” in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 51–62, ACM, 2008.
  - [211] K. Lawton, “Bochs: The open source ia-32 emulation project,” 2003.
  - [212] A. Fattori, R. Paleari, L. Martignoni, and M. Monga, “Dynamic and transparent analysis of commodity production systems,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering, ASE ’10, (New York, NY, USA)*, pp. 417–426, Association for Computing Machinery, 2010.
  - [213] N. Couffin, “Winbagility : Débogage furtif et introspection de machine virtuelle,” in *SSTIC 2016*, 06 2016.
  - [214] K. Chan, “Virtice,” 2017.
  - [215] Microsoft, “Peb structure,” 2018.
  - [216] ntopcode, “Anatomy of the process environment block (peb) (windows internals),” *ntopcode*, 02 2018.
  - [217] L. S. Hub, “Overview of the kronos banking malware rootkit,” 2014.
  - [218] C. T. G. Team, “threat-spotlight-satan-raas,” 2017.
  - [219] Microsoft, “Isdebuggerpresent function (debugapi.h),” tech. rep., Microsoft, 12 2018.
  - [220] Microsoft, “Checkremotedebuggerpresent function (debugapi.h),” tech. rep., Microsoft, 12 2018.
  - [221] K. Oleg, “Anti-debug protection techniques: Implementation and neutralization,” 2016.
  - [222] Microsoft, “Getthreadcontext function (processthreadsapi.h),” tech. rep., Microsoft, 12 2018.
  - [223] McAfee, “The w9x.cih virus,” 2000.
  - [224] McAfee, “W32.mydoom.m@mm,” 2007.
  - [225] Microsoft, “Findwindowa function (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [226] B. Karl, C. David, J. Mike, and S. Michael, “Enumerating all processes,” tech. rep., Microsoft, 05 2018.
  - [227] JOESandbox, “shcndhss.exe,” 2018.
-



- [228] T. Shields, “Anti-debugging—a developers view,” *Veracode Inc., USA*, 2010.
- [229] Microsoft, “Createtoolhelp32snapshot function (tlhelp32.h),” tech. rep., Microsoft, 12 2018.
- [230] Microsoft, “Ntquerysysteminformation function (winternl.h),” tech. rep., Microsoft, 12 2018.
- [231] Virustotal, “vti-rescan,” 2015.
- [232] JOESandbox, “Automated malware analysis report for wdf01000.sys,” 2010.
- [233] C. S. Research, “Inkasso trojaner - part 3,” 2013.
- [234] Microsoft, “Gettickcount function (sysinfoapi.h),” tech. rep., Microsoft, 12 2018.
- [235] Microsoft, “Queryperformancecounter function (profileapi.h),” tech. rep., Microsoft, 06 2020.
- [236] Microsoft, “Acquiring high-resolution time stamps,” 2018.
- [237] G. Pék, B. Bencsáth, and L. Buttyán, “nether: In-guest detection of out-of-the-guest malware analyzers,” in *Proceedings of the Fourth European Workshop on System Security*, p. 3, ACM, 2011.
- [238] Kaspersky, “Virus.win32.hiv,” 2000.
- [239] G. McGraw and G. Morrisett, “Attacking malicious code: A report to the infosec research council,” *IEEE software*, vol. 17, no. 5, pp. 33–41, 2000.
- [240] B. Karl, S. Kent, and S. Michael, “Structured exception handling,” tech. rep., Microsoft, 05 2018.
- [241] Microsoft, “x64 exception handling,” tech. rep., Microsoft, 10 2019.
- [242] I. Institute, “Zeroaccess malware - part 1,” 2015.
- [243] M. Don, C. David, L. Bill, and G. Eliot, “sx, sxd, sxe, sxi, sxn, sxr, sx- (set exceptions),” tech. rep., Microsoft, 05 2017.
- [244] Microsoft, “Outputdebugstringa function (debugapi.h),” tech. rep., Microsoft, 12 2018.
- [245] P. Ferrie, “Anti-unpacker tricks, part one,” *Virus Bulletin*, vol. 4, 2008.
- [246] N. Brulez, “Scan of the month 33: Anti reverse engineering uncovered,” 2012.
- [247] B. Bencsáth, G. Pek, L. Buttyan, and M. Felegyhazi, “The cousins of stuxnet: Duqu, flame, and gauss. future internet, 4 (4), 971-1003,” 2012.
- [248] L. Loobeek, “Ebowla: Framework for making environmental keyed payloads.” <https://github.com/Genetic-Malware/Ebowla>, 2016.
- [249] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, “Automatically identifying trigger-based behavior in malware,” in *Botnet Detection*, pp. 65–88, Springer, 2008.
- [250] J. R. Crandall, G. Wassermann, D. A. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong, “Temporal search: Detecting hidden malware timebombs with virtual machines,” in *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 25–36, ACM, 2006.
- [251] A. Moser, C. Kruegel, and E. Kirda, “Exploring multiple execution paths for malware analysis,” in *IEEE Symposium on Security and Privacy, SP’07*, pp. 231–245, IEEE, 2007.
- [252] J. Wilhelm and T.-c. Chiueh, “A forced sampled execution approach to kernel rootkit identification,” in *International Workshop on Recent Advances in Intrusion Detection*, pp. 219–235, Springer, 2007.
- [253] T. Liu, N. Xu, Q. Liu, Y. Wang, and W. Wen, “A system-level perspective to understand the vulnerability of deep learning systems,” in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 506–511, ACM, 2019.
-



- [254] M. P. S. Dhilung Kirat, Jiyong Jang, “Deeplocker: Concealing targeted attacks with ai locksmithing,” 2018.
- [255] J. J. Dhilung Kirat, “Deeplocker: How ai can power a stealthy new breed of malware,” 2018.
- [256] M. msdn, “Zwsetinformationthread function,” 2018.
- [257] K. Y. Walter (Tiezhu) Kong, “Unlocking lockscreen,” 2013.
- [258] Microsoft, “Suspendthread function (processthreadsapi.h),” tech. rep., Microsoft, 12 2018.
- [259] Microsoft, “Createthread function (processthreadsapi.h),” tech. rep., Microsoft, 12 2018.
- [260] UIC, “Mcrat malware analysis - part1,” 2013.
- [261] CTurt, “Reverse engineering vertexnet malware,” 2012.
- [262] C. Monoxide, “Scyllahide,” 2016.
- [263] XPN, “Windows anti-debug techniques - openprocess filtering,” 2017.
- [264] Symantec, “Trojan.zeroaccess,” 2011.
- [265] Microsoft, “Blockinput function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [266] Microsoft, “Createdesktopexa function,” tech. rep., Microsoft, 12 2018.
- [267] Microsoft, “Switchdesktop function,” tech. rep., Microsoft, 12 2018.
- [268] D. Patten, “The evolution to fileless malware,” *Infosec Writers*, 2017.
- [269] S. Seetharamaiah and C. D. Woodward, “Protecting computer systems used in virtualization environments against fileless malware,” Jan. 31 2019. US Patent App. 15/708,328.
- [270] S. Mansfield-Devine, “Fileless attacks: compromising targets without malware,” *Network Security*, vol. 2017, no. 4, pp. 7–11, 2017.
- [271] Malwarebytes, “Samsam ransomware: controlled distribution for an elusive malware,” tech. rep., Malwarebytes Labs, 2018.
- [272] B. Alex, B. Christiaan, M. Niamh, P. Eric, S. Raj, S. Craig, S. ReseAnne, S. Dan, and S. Bing, “Macafee labs threat report march 2018,” tech. rep., McAfee Labs, 2018.
- [273] L. Ponemon and J. Danahy, “The 2017 state of endpoint security risk report,” tech. rep., Ponemon Institute, 2018.
- [274] A. Cherepanov, “Win32/industroyer, a new threat for industrial control systems,” 2017.
- [275] M. Labs, “Threats report,” 2018.
- [276] R. P. Goldberg, “Survey of virtual machine research,” *Computer*, vol. 7, no. 6, pp. 34–45, 1974.
- [277] Norman, “Norman sandbox,” 2018.
- [278] C. Willems, T. Holz, and F. Freiling, “Toward automated dynamic malware analysis using cwsandbox,” *IEEE Security & Privacy*, vol. 5, no. 2, 2007.
- [279] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser, “The cuckoo sandbox,” 2012.
- [280] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, “A survey on automated dynamic malware-analysis techniques and tools,” *ACM computing surveys (CSUR)*, vol. 44, p. 6, 2012.
- [281] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, p. 46, 2005.
-

- 
- [282] U. Bayer, C. Kruegel, and E. Kirda, “Ttanalyze: A tool for analyzing malware,” *15th European Institute for Computer Antivirus Research*, 2006.
- [283] D. Kirat, G. Vigna, and C. Kruegel, “Barecloud: Bare-metal analysis-based evasive malware detection,” in *USENIX Security Symposium*, pp. 287–301, 2014.
- [284] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan, “Down to the bare metal: Using processor features for binary analysis,” in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 189–198, ACM, 2012.
- [285] D. Kirat, G. Vigna, and C. Kruegel, “Barebox: efficient malware analysis on bare-metal,” in *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 403–412, ACM, 2011.
- [286] T. Raffetseder, C. Kruegel, and E. Kirda, “Detecting system emulators,” in *International Conference on Information Security*, pp. 1–18, Springer, 2007.
- [287] B. Lau and V. Svajcer, “Measuring virtual machine detection in malware using dsd tracer,” *Journal in Computer Virology*, vol. 6, no. 3, pp. 181–195, 2010.
- [288] T. Holz and F. Raynal, “Detecting honeypots and other suspicious environments,” in *Information Assurance Workshop, 2005. IAW’05. Proceedings from the Sixth Annual IEEE SMC*, pp. 29–36, IEEE, 2005.
- [289] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna, “Escape from monkey island: Evading high-interaction honeyclients,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 124–143, Springer, 2011.
- [290] A. Yokoyama, K. Ishii, R. Tanabe, Y. Papa, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, M. Brenzel, M. Backes, *et al.*, “Sandprint: fingerprinting malware sandboxes to provide intelligence for sandbox evasion,” in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 165–187, Springer, 2016.
- [291] Anish, “Reptile malware - behavioral analysis,” 2012.
- [292] Y. Assor, “Anti-vm and anti-sandbox explained,” 2016.
- [293] Sophos, “W32/agobot-ot,” 2015.
- [294] Symantec, “Trojan.peacomm.c,” 2007.
- [295] C. Kruegel, “Evasive malware exposed and deconstructed,” in *RSA Conference*, pp. 12–20, 2015.
- [296] D. R. G. Dreg, “A tool to detect and crash cuckoo sandbox,” 2018.
- [297] Microsoft, “Worm:win32/rbot.st,” 2017.
- [298] Microsoft, “Win32/phatbot.a,” 2006.
- [299] R. Wu, “New emotet hijacks a windows api, evades sandbox and analysis,” 2017.
- [300] V. Kremez, “Let’s learn: Decoding latest ”trickbot” loader string template & new tor plugin server communication,” 2017.
- [301] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. Van Doorn, “Remote detection of virtual machine monitors with fuzzy benchmarking,” *ACM SIGOPS Operating Systems Review*, vol. 42, no. 3, pp. 83–92, 2008.
- [302] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, “Compatibility is not transparency: Vmm detection myths and realities,” in *HotOS*, 2007.
- [303] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, “A fistful of red-pills: How to automatically generate procedures to detect cpu emulators,” in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, vol. 41, p. 86, 2009.
-

- [304] K. Yoshioka, Y. Hosobuchi, T. Orii, and T. Matsumoto, “Your sandbox is blinded: Impact of decoy injection to public malware analysis systems,” *Journal of Information Processing*, vol. 19, pp. 153–168, 2011.
  - [305] J. Blackthorne, A. Bulazel, A. Fasano, P. Biernat, and B. Yener, “Avleak: Fingerprinting antivirus emulators through black-box testing,” in *Proceedings of the 10th USENIX Conference on Offensive Technologies*, pp. 91–105, USENIX Association, 2016.
  - [306] C. Spensky, H. Hu, and K. Leach, “Lo-phi: Low-observable physical host instrumentation for malware analysis,” in *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.
  - [307] B. Dolan-Gavitt and Y. Nadji, “See no evil: Evasions in honeymoney systems,” tech. rep., Technical Report, 5 2010.
  - [308] McAfee, “Mcafee labs threats report,” 2017.
  - [309] A. Singh, “Malware evasion techniques: Same wolf - different clothing,” 2017.
  - [310] N. Falliere, L. O. Murchu, and E. Chien, “W32. stuxnet dossier,” *White paper, Symantec Corp., Security Response*, vol. 5, no. 6, p. 29, 2011.
  - [311] A. S. Yasir Khalid, “Don’t click the left mouse button: Introducing trojan upclicker,” 2012.
  - [312] C. R. Hwa, “Trojan.apr.banechant: In-memory trojan that observes for multiple mouse clicks,” 2013.
  - [313] A. S. Sai Omkar Vashisht, “Turing test in reverse: New sandbox-evasion techniques seek human interaction,” 2014.
  - [314] Microsoft, “Getlastinputinfo function (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [315] A. Singh and C. Kolbitsch, “Not so fast my friend - using inverted timing attacks to bypass dynamic analysis,” 2014.
  - [316] Microsoft, “Getcursorpos function (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [317] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, “Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts,” in *IEEE Symposium on Security and Privacy (SP)*, pp. 1009–1024, IEEE, 2017.
  - [318] GRaT, “The darkhotel apt,” 2014.
  - [319] T. Morrow and J. Pitts, “Genetic malware: Designing payloads for specific targets.(2016),” *Talk at Infiltrate*, 2016.
  - [320] C. Kolbitsch, E. Kirda, and C. Kruegel, “The power of procrastination: detection and mitigation of execution-stalling malicious code,” in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 285–296, ACM, 2011.
  - [321] Microsoft, “Sleep function (synchapi.h),” tech. rep., Microsoft, 12 2018.
  - [322] MalwareTech, “Kelihos analysis, part 1,” 2015.
  - [323] Microsoft, “Sleepex function (synchapi.h),” tech. rep., Microsoft, 12 2018.
  - [324] A. Ortega, “Pafish (paranoid fish),” 2014.
  - [325] A. C. Ben Baker, “Threat spotlight: Rombertik, gazing past the smoke, mirrors, and trapdoors,” 2015.
  - [326] M. Lindorfer, C. Kolbitsch, and P. M. Comporetti, “Detecting environment-sensitive malware,” in *International Workshop on Recent Advances in Intrusion Detection*, pp. 338–357, Springer, 2011.
  - [327] J. Barlow, “Tribe flood network 2000 (tfn2k),” 2000.
  - [328] P. Roberts, “Mydoom sets speed records,” 2004.
-

- [329] Symantec, “Xeram.1664,” 2000.
- [330] McAfee, “W97m/opey.bg,” 2003.
- [331] M. John and M. David, “Wannacry ransomware campaign: Threat details and risk management,” 05 2017.
- [332] V. L. Le, I. Welch, X. Gao, and P. Komisarczuk, “Anatomy of drive-by download attack,” in *Proceedings of the Eleventh Australasian Information Security Conference-Volume 138*, pp. 49–58, Australian Computer Society, Inc., 2013.
- [333] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, “Code obfuscation against symbolic execution attacks,” in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pp. 189–200, ACM, 2016.
- [334] M. Brengel, M. Backes, and C. Rossow, “Detecting hardware-assisted virtualization,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 207–227, Springer, 2016.
- [335] G. Pék, L. Buttyán, and B. Bencsáth, “A survey of security issues in hardware virtualization,” *ACM Computing Surveys (CSUR)*, vol. 45, no. 3, p. 40, 2013.
- [336] C. Thompson, M. Huntley, and C. Link, “Virtualization detection: New strategies and their effectiveness,” *University of Minnesota. (unpublished)*, 2010.
- [337] F. Zhang, K. Leach, K. Sun, and A. Stavrou, “Spectre: A dependable introspection framework via system management mode,” in *2013 43rd Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pp. 1–12, IEEE, 2013.
- [338] A. Vasudevan and R. Yerraballi, “Cobra: Fine-grained malware analysis using stealth localized-executions,” in *2006 IEEE Symposium on Security and Privacy (S&P’06)*, pp. 15–pp, IEEE, 2006.
- [339] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, “Efficient detection of split personalities in malware,” in *Annual Network and Distributed System Security Symposium (NDSS)*, 2010.
- [340] D. Kirat and G. Vigna, “Malgene: Automatic extraction of malware analysis evasion signature,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 769–780, ACM, 2015.
- [341] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song, “Emulating emulation-resistant malware,” in *Proceedings of the 1st ACM workshop on Virtual machine security*, pp. 11–22, ACM, 2009.
- [342] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, “X-force: force-executing binary programs for security applications,” in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 829–844, 2014.
- [343] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, “Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system,” in *Proceedings of the 30th Annual Computer Security Applications Conference*, pp. 386–395, ACM, 2014.
- [344] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen, “Mavmm: Lightweight and purpose built vmm for malware analysis,” in *Computer Security Applications Conference, 2009. ACSAC’09. Annual*, pp. 441–450, IEEE, 2009.
- [345] D. Vostokov, *WinDbg: A Reference Poster and Learning Cards*. Opentask, 2008.
- [346] M. Don, C. David, N. Nitya, G. Eliot, and L. Andy, “Debugging using windbg preview,” tech. rep., Microsoft, 01 2020.
- [347] M. Don, G. Victor, and JamespiWork, “Time travel debugging - overview,” tech. rep., Microsoft, 01 2020.
- [348] T. N. development team, “Nasm,” 12 2018.
-

- [349] T. Joshua, “An anti-reverse engineering guide,” 11 2008.
  - [350] Y. Reiley, “Data breakpoints,” tech. rep., Microsoft, 07 2011.
  - [351] F. Peter, “Anti-unpacker tricks,” *Microsoft Corporation Blog*, 05 2005.
  - [352] Kui and Y. Zhang, *Software debugging*. Electronic Industry Publishing House Pub, 2008.
  - [353] B. Ron, “In-depth malware: Unpacking the ”lcmw” trojan,” 01 2014.
  - [354] R. Colin, S. Matthew, B. Mike, J. Mike, H. Gordon, and C. Saisang, “try-except statement,” tech. rep., Microsoft, 08 2020.
  - [355] Microsoft, “Thread environment block (debugging notes),” tech. rep., Microsoft, 05 2018.
  - [356] Microsoft, “Teb structure (winternl.h),” tech. rep., Microsoft, 05 2018.
  - [357] B. Karl, S. Kent, C. David, B. Drew, and S. Michael, “Using an exception handler,” tech. rep., Microsoft, 05 2018.
  - [358] B. Karl, S. Kent, C. David, B. Drew, and S. Michael, “Using a vectored exception handler,” tech. rep., Microsoft, 05 2018.
  - [359] B. Karl, S. Kent, C. David, B. Drew, and S. Michael, “Vectored exception handling,” tech. rep., Microsoft, 05 2018.
  - [360] M. Czumak, “Windows exploit development - part 6: Seh exploits,” *Security Sift*, 03 2014.
  - [361] Swiat, “Preventing the exploitation of structured exception handler (seh) overwrites with sehopt,” *Security Research & Defense*, 02 2009.
  - [362] Microsoft, “Exception\_record structure (winnt.h),” tech. rep., Microsoft, 12 2018.
  - [363] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual - Volume 2, 2-1*. Intel Corporation, 05 2019.
  - [364] Wikipedia, “x86-64,” 07 2019.
  - [365] B. Krithika and N. Keerthana, “Comparison of intel processor with amd processor with green computing,” in *2013 International Conference on Green Computing, Communication and Conservation of Energy (ICGCE)*, pp. 737–742, 12 2013.
  - [366] C. Igiri, A. Oghenekaro, and T. Olowookere, “A comparative study of two microprocessor based distributed systems: Intel xeon and amd opteron,” *International Organisation of Scientific Research- Journal of Computer Engineering (IOSR-JCE) 2278-8727*, vol. 16, pp. 44–48, 10 2014.
  - [367] G. Christian, P. Rafael, R. I., V. S., F. P., P. Marcelo, and S. Valerio, “Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms,” *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pp. 133–142, 10 2018.
  - [368] L. Peng, J.-K. Peir, T. Prakash, Y.-K. Chen, and D. Koppelman, “Memory performance and scalability of intel’s and amd’s dual-core processors: A case study,” in *2007 IEEE International Performance, Computing, and Communications Conference*, pp. 55–64, 05 2007.
  - [369] W. Swanson, “Understanding intel instruction sizes,” 2003.
  - [370] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual - Volume 2, 2-20*. Intel Corporation, 05 2019.
  - [371] x64dbg community, “x64dbg debugger,” 11 2013.
  - [372] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, “Intel mpx explained: A cross-layer analysis of the intel mpx system stack,” *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, 06 2018.
-

- [373] C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. San Francisco, CA, USA: No Starch Press, 2011.
- [374] M. Don, M. Don, N. Nitya, and G. Eliot, "Setting up kdnet network kernel debugging manually," tech. rep., Microsoft, 12 2018.
- [375] J. PARK, Y.-H. JANG, S. HONG, and Y. PARK, "Automatic detection and bypassing of anti-debugging techniques for microsoft windows environments," *Advances in Electrical and Computer Engineering*, vol. 19, pp. 23–28, 05 2019.
- [376] M. Gagnon, S. Taylor, and A. Ghosh, "Software protection through anti-debugging," *Security & Privacy, IEEE*, vol. 5, pp. 82–84, 06 2007.
- [377] D. Lukan, "Anti-debugging: Detecting system debugger," 02 2013.
- [378] P. Xie, X. Lu, Y. Wang, J. Su, and M. Li, "An automatic approach to detect anti-debugging in malware analysis," in *ISCTCS*, vol. 320, pp. 436–442, 01 2013.
- [379] Z. Qi, B. Li, Q. Lin, M. Yu, M. Xia, and H. Guan, "Spad: Software protection through anti-debugging using hardware-assisted virtualization," *J. Inf. Sci. Eng.*, vol. 28, pp. 813–827, 2012.
- [380] M. Marhusin, H. Larkin, C. Lokan, and D. Cornforth, "An evaluation of api calls hooking performance," *Proceedings - 2008 International Conference on Computational Intelligence and Security, CIS 2008*, vol. 1, pp. 315–319, 12 2008.
- [381] H.-M. Sun, Y.-H. Lin, and M.-F. Wu, "Api monitoring system for defeating worms and exploits in ms-windows system," in *Proceedings of the 11th Australasian Conference on Information Security and Privacy, ACISP'06*, (Berlin, Heidelberg), pp. 159–170, Springer-Verlag, 2006.
- [382] A. Ortega, "Al-khaser v0.79," 11 2015.
- [383] S. Karvandi, "Defeating malware's anti-vm techniques (cpuid-based instructions)," 06 2018.
- [384] J. Rutkowska, "Subverting vistatm kernel forfun and profit," 08 2006.
- [385] D. Quist, V. Smith, and O. Computing, "Detecting the presence of virtual machines using the local data table," *Offensive Computing*, vol. 25, no. 04, 2006.
- [386] J. Rutkowska, "Red pill... or how to detect vmm using (almost) one cpu instruction," 11 2007.
- [387] R. Leon, M. Kiperberg, A. Algawi, A. Resh, and N. Zaidenberg, "Creating modern blue pills and red pills," *European Conference on Cyber Warfare and Security*, vol. 1, p. 9, 07 2019.
- [388] T. Tomasz, B. Mark, Z. Joshua, L. Tamas K., and T. K.J., "Who watches the watcher? detecting hypervisor introspection from unprivileged guests," *Digital Investigation*, vol. 26, pp. S98 – S106, 2018.
- [389] I. Korkin, "Two challenges of stealthy hypervisors detection: Time cheating and data fluctuations," *Journal of Digital Forensics, Security and Law*, vol. X(X), p. 25, 05 2015.
- [390] A. Desnos, E. Filiol, and I. Lefou, "Detecting (and creating!) a hvm rootkit (aka bluepill-like)," *Journal in Computer Virology*, vol. 7, pp. 23–49, 02 2011.
- [391] M. Ali, S. Shiaeles, B. Ghita, and M. Papadaki, "Agent-based vs agent-less sandbox for dynamic behavioral analysis," in *arxiv*, p. 5, 10 2018.
- [392] M. Ben-Yehuda, "Machine virtualization:efficient hypervisors, stealthy malware," 03 2013.
- [393] Microsoft, *SpinLock*, 03 2017. Last accessed on 2020-10-04.
- [394] Microsoft, "Virtualprotect function (memoryapi.h)," tech. rep., Microsoft, 12 2018.
- [395] L. Li and C. Wang, "Dynamic analysis and debugging of binary code for security applications," in *RV*, pp. 403–423, 09 2013.
-



- [396] B. Kiss, N. Kosmatov, D. Pariente, and A. Puccetti, “Combining static and dynamic analyses for vulnerability detection: Illustration on heartbleed,” in *Haifa Verification Conference* (P. N., ed.), vol. 9434, (Haifa, Israel), pp. 39–50, Springer Verlag, 11 2015.
- [397] M. Sebastiano, F. Lorenzo, G. Fabio, and D. Stefano, “Pindemonium: a dbi-based generic unpacker for windows executables,” in *Black Hat USA*, 2016.
- [398] N. Aaraj, A. Raghunathan, and N. Jha, “Dynamic binary instrumentation-based framework for malware defense,” in *DIMVA*, vol. 5137, 07 2008.
- [399] N. Aaraj, A. Raghunathan, and N. Jha, “Dynamic binary instrumentation-based framework for malware defense,” in *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 5137, 07 2008.
- [400] C. Lim, D. Sulistyan, S. MT, and K. Ramli, “Experiences in instrumented binary analysis for malware,” *Advanced Science Letters*, vol. 21, pp. 3333–3336, 10 2015.
- [401] D. C. D’Elia, E. Coppa, F. Palmaro, and L. Cavallaro, “On the dissection of evasive malware,” *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 2750–2765, 2020.
- [402] J. Kirsch, Z. Zhechev, B. Bierbaumer, and T. Kittel, *PwIN - Pwning Intel piN: Why DBI is Unsuitable for Security Applications*, pp. 363–382. Barcelona, Spain: 23rd European Symposium on Research in Computer Security, ESORICS 2018, 08 2018.
- [403] D. C. D’Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro, “Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed),” in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS ’19, (New York, NY, USA), pp. 15–27, Association for Computing Machinery, 07 2019.
- [404] Z. Zhechev, “Security evaluation of dynamic binary instrumentation engines,” Master’s thesis, University of Munich, 06 2018.
- [405] D. Bruening, Q. Zhao, and S. Amarasinghe, “Transparent dynamic instrumentation,” *SIGPLAN Not.*, vol. 47, pp. 133–144, 03 2012.
- [406] D. Kim, S. Kim, and J. Ryou, “Design and implementation of user-level dynamic binary instrumentation on arm architecture,” *The Journal of Supercomputing*, 07 2016.
- [407] V. Zhao, “Evaluation of dynamic binary instrumentation approaches: Dynamic binary translation vs. dynamic probe injection,” Master’s thesis, Williams College, 06 2018.
- [408] R. J. Rodriguez, J. Artal, and J. Merseguer, “Performance evaluation of dynamic binary instrumentation frameworks,” *Latin America Transactions, IEEE (Revista IEEE America Latina)*, vol. 12, pp. 1572–1580, 12 2014.
- [409] K. Julian and Z. Zhechko, “Pwning intel pin - reconsidering intel pin in context of security,” in *REcon*, REcon Montreal 2018, June 2018.
- [410] M. Polino, A. Continella, S. Mariani, S. D’Alessio, L. Fontana, F. Gritti, and S. Zanero, “Measuring and defeating anti-instrumentation-equipped malware,” in *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2017.
- [411] Intel, *Pin - Command Line Switches*, 05 2018.
- [412] A. Bougacha, “Detecting valgrind,” 09 2012.
- [413] R. Nahuel and F. Francisco, “Dynamic binary instrumentation frameworks: I know you’re there spying on me,” in *REcon 2012*, 06 2012.
- [414] H. Martin and J. Jakub, “Safemachine: malware needs love, too,” in *REcon 2012*, 09 2014.
-



- [415] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Powny, “You can run but you can’t read: Preventing disclosure exploits in executable code,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS ’14*, (New York, NY, USA), pp. 1342–1353, Association for Computing Machinery, 2014.
- [416] B. Karl, M. Jason, S. Kent, C. David, B. Drew, and S. Michael, “Thread local storage,” tech. rep., Microsoft, 05 2018.
- [417] W. Tyler, C. Saisang, and P. Andy, “Thread local storage (tls),” tech. rep., Microsoft, 09 2019.
- [418] S. Ke and L. Xiaoning, “Break out of the truman show: Active detection and escape of dynamic binary instrumentation,” in *Black Hat Singapore*, 03 2016.
- [419] R. Wa, G. Hunt, and D. Brubacher, “Detours: Binary interception of win32 functions,” *Proceedings of the 3rd Conference on USENIX Windows NT Symposium*, vol. 3, 02 1970.
- [420] Microsoft, “Pssetcreatethreadnotifyroutine function (ntddk.h),” tech. rep., Microsoft, 04 2018.
- [421] S. McLean, S. Kent, and S. Michael, “Handles and objects,” tech. rep., Microsoft, 05 2018.
- [422] H. Bui, “Hooking heaven’s gate - a wow64 hooking technique,” 05 2019.
- [423] L. Martignoni, R. Paleari, and D. Bruschi, “Conqueror: Tamper-proof code execution on legacy systems,” in *DIMVA*, pp. 21–40, 07 2010.
- [424] Microsoft, “What is .net?,” tech. rep., Microsoft, 02 2002.
- [425] J. Fiedor and T. Vojnar, “Anaconda: A framework for analysing multi-threaded c/c++ programs on the binary level,” in *RV* (S. Qadeer and S. Tasiran, eds.), vol. 7687 of *Lecture Notes in Computer Science*, pp. 35–41, Springer, 2012.
- [426] N. Chatterjee, S. Majumdar, S. Sahoo, and P. Das, “Debugging multi-threaded applications using pin-augmented gdb (pgdb),” 07 2015.
- [427] P. Ambavkar, “Debugging on linux,” *International organization of Scientific Research Journal of Engineering (IOSRJEN)February 2012*, p. 7, 02 2012.
- [428] Microsoft, “Debugging in visual studio,” 11 2016.
- [429] OllyDbg, “Ollydbg.”
- [430] R. community, “Radare2.”
- [431] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: chapter 23 - introduction to virtual machine extensions*. Intel, 2016.
- [432] K. Biswas and M. Islam, “Hardware virtualization support in intel, amd and ibm power processors,” *International Journal of Computer Science and Information Security*, vol. 4, 09 2009.
- [433] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3C*, 2016.
- [434] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A*, 2016.
- [435] vmware company, “vmware.”
- [436] D. Kuhn, C. Kim, and B. Lopuz, *VirtualBox for Oracle*, ch. 12, pp. 325–344. Apress, Berkeley, 01 2015.
- [437] Microsoft, *Hyper-V Technology Overview*, 11 2016. Last accessed on 2020-10-04.
- [438] Microsoft, *Introduction to Hyper-V on Windows 10*, 06 2018. Last accessed on 2020-10-04.
- [439] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, “Cloud container technologies: A state-of-the-art review,” *IEEE Transactions on Cloud Computing*, vol. 7, pp. 677–692, 05 2017.
-

- [440] J. Shetty, “A state-of-art review of docker container security issues and solutions,” *American International Journal of Research in Science, Technology, Engineering & Mathematics*, 01 2017.
- [441] P. Fawaz, S. Challita, Y. al dhuraibi, and P. Merle, “Model-driven management of docker containers,” in *9th International Conference on Cloud Computing (CLOUD)*, 07 2016.
- [442] B. Bashari Rad, H. Bhatti, and M. Ahmadi, “An introduction to docker and analysis of its performance,” *IJCSNS International Journal of Computer Science and Network Security*, vol. 173, p. 8, 03 2017.
- [443] S. Polyakov, A. Kryukov, and A. Demichev, “Docker container manager: A simple toolkit for isolated work with shared computational, storage, and network resources,” *Journal of Physics: Conference Series*, vol. 955, p. 012039, 01 2018.
- [444] S. Johnston and Co., “Docker.”
- [445] Microsoft, “Linux containers on windows 10,” tech. rep., Microsoft, 09 2019.
- [446] A. Raina, “Under the hood: Demystifying docker for mac ce edition,” *Collabnix - A Docker Captain's Blog*, 05 2018.
- [447] C. Ciborowski, “Getting started with linuxkit on mac os x with xhyve,” 04 2017.
- [448] P. Kumar, “docker-machine-driver-xhyve,” 02 2019.
- [449] C. Ltd., “Infrastructure for container projects.”
- [450] D. Rynd, “Day 5: The vm-exit handler, event injection, context modifications, and cpuid emulation,” *Research Blog Revere Engineering*, 07 2019.
- [451] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3C*, 2016.
- [452] K. Pearson, “Notes on regression and inheritance in the case of two parents,” in *Proceedings of the Royal Society of London*, pp. 240–242, 1895.
- [453] S. M. Stigler, “Francis Galton’s Account of the Invention of Correlation,” *Statistical Science*, vol. 4, no. 2, pp. 73 – 79, 1989.
- [454] S. Wright, “Correlation and causation,” *Journal of Agricultural Research*, vol. 20, 1921.
- [455] P. J. Huber, *Robust Statistics*, pp. 1248–1251. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.
- [456] J.-J. Dreesbeke, G. Saporta, and C. Thomas-Agnan, *Méthodes robustes en statistique*. Technip, 01 2015.
- [457] H. Theil, *A Rank-Invariant Method of Linear and Polynomial Regression Analysis*, pp. 345–381. Dordrecht: Springer Netherlands, 1992.
- [458] P. Sen, “Estimates of the regression coefficient based on kendall’s tau,” *Journal of the American Statistical Association*, vol. 63, pp. 1379–1389, 1968.
- [459] S. Gorard, “Revisiting a 90-year-old debate: the advantages of the mean deviation,” *British Journal of Educational Studies*, vol. 53, no. 4, pp. 417–430, 2005.
- [460] C. Grasland, “Initiation aux methodes statistiques en sciences sociales,” 1998.
- [461] Z. Jialong, G. Zhongshu, J. Jiyong, K. D., S. M., S. Xiaokui, and H. Heqing, “Scarecrow: Deactivating evasive malware via its own evasive logic,” *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 76–87, 2020.
- [462] V. Mathilde and D. Baptiste, “Fooling windows through superfetch,” in *Black hat USA 2020*, (Las Vegas - USA), 08 2020.
- [463] M. Venault and B. David, “Superfetch: the famous unknown spy,” *Journal of Computer Virology and Hacking Techniques*, pp. 1–14, 10 2020.
-

- [464] E. Filiol, “Strong cryptography armoured computer viruses forbidding code analysis: the bradley virus,” in *14th EICAR conference*, 05 2004.
- [465] E. Filiol, *Techniques virales avancées*. Collection IRIS, Springer Paris, 2007.
- [466] P. Beaucamps, “Advanced polymorphic techniques,” *International Journal of Computer and Information Engineering*, vol. 1, no. 10, pp. 3366–3377, 2007.
- [467] B. David, “New ways to manage secret for software protection,” in *Recon 2013*, (Montreal - Canada), 06 2013.
- [468] B. David, “New ways to manage secret for software protection ii,” in *Ground Zero Summit 2013*, (New Delhi - India), 11 2013.
- [469] J. Riordan and B. Schneier, “Environmental key generation towards clueless agents,” in *Mobile Agents and Security*, (Berlin, Heidelberg), pp. 15–24, Springer-Verlag, 1998.
- [470] S. J. Oh, B. Schiele, and M. Fritz, *Towards Reverse-Engineering Black-Box Neural Networks*, pp. 121–144. Explainable AI: Interpreting, Explaining and Visualizing Deep Learning, 09 2019.
- [471] W. Hua, Z. Zhang, and G. E. Suh, “Reverse engineering convolutional neural networks through side-channel information leaks,” in *Proceedings of the 55th Annual Design Automation Conference, DAC '18*, (New York, NY, USA), Association for Computing Machinery, 2018.
- [472] Y. Liu, D. Dachman-Soled, and A. Srivastava, “Mitigating reverse engineering attacks on deep neural networks,” *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 657–662, 2019.
- [473] T. Micro, “Keyloggers.”
- [474] N. Daehun, M. Aziz, and K. Jeonil, “Keylogging-resistant visual authentication protocols,” *IEEE Transactions on Mobile Computing*, vol. 13, no. 11, pp. 2566–2579, 2014.
- [475] Sophos, “Demystifying a keylogger - how they monitor what you type and what you can do about it?,” 12 2019.
- [476] K. Lab, “What is keystroke logging and keyloggers?.”
- [477] J. Wiener, “What is emotet malware and how can you protect your organization from it?,” 09 2017.
- [478] B. Schafer, “Boost in trickbot malware - new feature trickbooster spikes increase in malware,” 11 2019.
- [479] C. Paul, “Emotet and trickbot banking trojans acquire internet worm capabilities,” 08 2017.
- [480] C. . infrastructure security agency, “Alert (aa20-266a) - lokibot malware,” 10 2020.
- [481] Z. Xiaopeng and N. Chris, “New agent tesla variant spreading by phishing - fortiguard labs threat analysis report,” 04 2020.
- [482] O. . T. R. Team, “Hawkeye keylogger - reborn v8: An in-depth campaign analysis,” 07 2018.
- [483] V. L. Team, “Malware analysis spotlight: Formbook (september 2020),” 10 2020.
- [484] R. Jullian, “Formbook in-depth malware analysis,” in *Botconf 2018*, Stormshield, 12 2018.
- [485] Y. Abukar, M. Maarof, F. Hassan, and M. Abshir, “Survey of keylogger technologies,” *International Journal of Computer Science and Telecommunications*, vol. 5, pp. 25–31, 02 2014.
- [486] J. Fu, Y. Liang, C. Tan, and X. Xiong, “Detecting software keyloggers with dendritic cell algorithm,” *Communications and Mobile Computing, International Conference on*, vol. 1, pp. 111–115, 04 2010.
- [487] Intel, “8278 programmable keyboard interface,” tech. rep., Intel, 1978.
- [488] S. M. Corporation, “Keyboard and ps/2 mouse controller,” tech. rep., SMC, 10 1998.
-

- [489] A. Chapweske, “The ps/2 keyboard interface,” *Computer-Engineering.org*, 04 2003.
- [490] Brendan, “Ps/2,” *OSDev*, 03 2012.
- [491] G. Ravier, “”8042” ps/2 controller,” *OSDev*, 04 2019.
- [492] Wesleyac, “Ps/2 keyboard,” *OSDev*, 04 2020.
- [493] C. Peacock, “Usb in a nutshell, making sense of the usb standard,” *Beyond Logic*, 04 2018.
- [494] R. Chen, “Hardware engineers solve a usability problem with the ps/2 connector, but inadvertently create a new one,” tech. rep., Microsoft, 02 2021.
- [495] E. M. F. Labs, “The first modern model f mechanical keyboard,” 2014.
- [496] IBM, *Technical Reference: Personal Computer XT and IBM Portable Personal Computer*, 03 1986.
- [497] S. Nucifora, “Ibm pc model f keyboard - type 1 vs type 2,” *Vintage Computer*, 05 2018.
- [498] E. George P. and R. Mark E., “Circuit board for use in capacitive keyboard,” 08 1981.
- [499] Mike, “Operating systems development - keyboard,” 2009.
- [500] Intel, “Hmos single-component 8-bit microcontroller,” tech. rep., Intel Corporation, 07 1978.
- [501] Intel, “8048 family applications handbook,” tech. rep., Intel Corporation, 01 1978.
- [502] Intel, “Mcs-48th family of single chip microcomputers user’s manual,” tech. rep., Intel Corporation, 07 1978.
- [503] Y. K. Meng, “Why i use the ibm model m keyboard that is older than me?,” *yeokhengmeng.com*, 06 2018.
- [504] IBM, *PS/2 Reference Manuals*. Business Machine Corporation, 10 1990. [http://www.mcamafia.de/pdf/ibm\\_hitra01.pdf](http://www.mcamafia.de/pdf/ibm_hitra01.pdf).
- [505] I. B. M. Corporation, *IBM Personal Computer AT Technical Reference*. IBM, 03 1984.
- [506] IBM, *Personal System/2(R) - Hardware Interface Technical Reference*. Business Machine Corporation, 05 1988. [http://www.nj7p.org/Computers/IBM PC/work/PS2\\_HI.pdf](http://www.nj7p.org/Computers/IBM PC/work/PS2_HI.pdf).
- [507] A. Chapweske, “Ps/2 mouse/keyboard protocol,” *Computer-Engineering.org*, 1999.
- [508] A. Brouwer, “Keyboard scancodes,” 07 2009.
- [509] Thepowersgang, “A20 line,” 10 2017.
- [510] VETRA, “Application note v-107 ps/2 pc keyboard scan sets translation table,” 2001.
- [511] IBM, *Technical Reference: Personal Computer AT*, 09 1985.
- [512] IBM, *PC-XT Technical Reference*, 01 1983.
- [513] Microsoft, “Keyboard scan code specification,” *Microsoft Developer Network*, 03 2000.
- [514] IBM, *IBM 3270 Workstation Program Programming Guide*, 04 1987.
- [515] mrmixer, “Keyboard inputs - scancodes, raw input, text input, key names,” *Handmade Network*, 01 2018.
- [516] Microsoft, “Virtual-key codes,” tech. rep., Microsoft, 05 2018.
- [517] T. Hudek, “Ps/2 (i8042prt) drive,” tech. rep., Microsoft, 04 2017.
- [518] H. Ted and S. Tim, “Introduction to interrupt service routines,” tech. rep., Microsoft, 06 2017.
- [519] PCI-SIG, *PCI Express (R) Base Specification Revision 2.0*, 12 2006.
-

- [520] P. KADIONIK, “Le bus industriel pci,” Master’s thesis, enseirb, 2001.
  - [521] H. Ted, C. David, and S. Tim, “Registering an isr,” tech. rep., Microsoft, 06 2017.
  - [522] Microsoft, “Ioconnectinterruptex function (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [523] M. Baranauskas, “Interrupt descriptor table - idt,” *ired.team*, 01 2020.
  - [524] W. Alan, W. Yazhi, and L. Ryan, 03 2006.
  - [525] Microsoft, “Windows kernel opaque structures,” tech. rep., Microsoft, 10 2018.
  - [526] H. Ted and S. Tim, “Introduction to dpc objects,” tech. rep., Microsoft, 10 2018.
  - [527] Microsoft, “Keinsertqueuedpc function (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [528] H. Ted and C. David, “Spb device stacks,” tech. rep., Microsoft, 04 2017.
  - [529] Microsoft, “Keyboardclassservicecallback routine,” tech. rep., Microsoft, 08 2016.
  - [530] Microsoft, “Irp structure,” tech. rep., Microsoft, 04 2018.
  - [531] H. Ted and S. Tim, “Filter drivers,” tech. rep., Microsoft, 06 2017.
  - [532] H. Ted and S. Tim, “Types of wdm drivers,” tech. rep., Microsoft, 06 2017.
  - [533] B. S. W. Groups, “Core specification 5.2,” tech. rep., Bluetooth, 12 2019.
  - [534] U. Organisation, “Human interface devices (hid) information,” tech. rep., USB.org, 6 2001.
  - [535] C. I. M. NEC, “Universal serial bus specification 1.1,” tech. rep., USB.org, 09 1998.
  - [536] U. Organisation, “Universal serial bus 3.0 specification,” tech. rep., USB.org, 11 2008.
  - [537] U. Organisation, “Universal serial bus 3.0 specification,” tech. rep., USB.org, 07 2013.
  - [538] U. Organisation, “The usb 3.2 specification,” tech. rep., USB.org, 09 2017.
  - [539] U. Organisation, “Usb4 specification,” tech. rep., USB.org, 06 2020.
  - [540] U. Organisation, “Usb type-c(c) locking connector specification,” tech. rep., USB.org, 09 2016.
  - [541] Microsoft, “I/o request packets,” tech. rep., Microsoft, 04 2017.
  - [542] Microsoft, “Usb\_device\_descriptor structure,” tech. rep., Microsoft, 07 2018.
  - [543] Microsoft, “Usb\_configuration\_descriptor structure,” tech. rep., Microsoft, 07 2018.
  - [544] Microsoft, “Usb\_interface\_descriptor structure,” tech. rep., Microsoft, 07 2018.
  - [545] U. I. Forum, “Defined class codes,” 06 2016.
  - [546] S. S. C. . I. CORPORATION, “Universal serial bus common class specification,” tech. rep., USB.org, 12 1997.
  - [547] T. Hudek, “Usb configuration descriptors,” tech. rep., Microsoft, 04 2017.
  - [548] D. M. . T. Hudek, “Usb device layout,” tech. rep., Microsoft, 04 2017.
  - [549] M. E. Ltd, “Usb made simple,” 2006.
  - [550] M. E. Ltd, “Usb made simple,” 2006.
  - [551] Microsoft, “How to send a usb control transfer,” tech. rep., Microsoft, 04 2017.
  - [552] H. Ted, M. Duncan, and G. Eliot, “Overview of developing windows client drivers for usb devices,” tech. rep., Microsoft, 01 2019.
-

- [553] T. Hudek, “Usb device descriptors,” tech. rep., Microsoft, 04 2017.
  - [554] T. Hudek, “Usb configuration descriptors,” tech. rep., Microsoft, 04 2017.
  - [555] T. Hudek, “Usb interface association descriptor,” tech. rep., Microsoft, 04 2017.
  - [556] T. Hudek, “Usb string descriptors,” tech. rep., Microsoft, 04 2017.
  - [557] T. Hudek, “Usb descriptors,” tech. rep., Microsoft, 04 2017.
  - [558] T. Hudek, “Standard usb descriptors,” tech. rep., Microsoft, 04 2017.
  - [559] T. Hudek, “Usb request blocks (urbs),” tech. rep., Microsoft, 04 2017.
  - [560] H. Ted and M. Duncan, “Overview of developing windows applications for usb devices,” tech. rep., Microsoft, 04 2017.
  - [561] T. Hudek, “Winusb (winusb.sys),” tech. rep., Microsoft, 04 2017.
  - [562] T. Hudek, “Winusb architecture and modules,” tech. rep., Microsoft, 04 2017.
  - [563] H. Ted and M. Don, “How to access a usb device by using winusb functions,” tech. rep., Microsoft, 04 2017.
  - [564] H. Ted, M. Duncan, and G. Eliot, “Overview of microsoft-provided usb drivers,” tech. rep., Microsoft, 01 2019.
  - [565] Microsoft, “Usb device class drivers included in windows,” tech. rep., Microsoft, 04 2017.
  - [566] T. Hudek, “Overview of inf files,” tech. rep., Microsoft, 04 2017.
  - [567] M. Satran, “About inf files,” tech. rep., Microsoft, 05 2018.
  - [568] Microsoft, “System-defined device setup classes available to vendors,” tech. rep., Microsoft, 05 2018.
  - [569] H. Ted and B. Nathan, “Driver stacks,” tech. rep., Microsoft, 04 2017.
  - [570] T. Hudek, “Usb generic parent driver (usbccgp.sys),” tech. rep., Microsoft, 04 2017.
  - [571] T. Hudek, “Descriptors on usb composite devices,” tech. rep., Microsoft, 04 2017.
  - [572] H. Ted and M. Duncan, “How to select a configuration for a usb device,” tech. rep., Microsoft, 01 2019.
  - [573] H. Ted and B. Nathan, “Device nodes and device stacks,” tech. rep., Microsoft, 04 2017.
  - [574] H. Ted and S. Tim, “Introduction to device objects,” tech. rep., Microsoft, 06 2017.
  - [575] T. Hudek, “Enumeration of usb composite devices,” tech. rep., Microsoft, 04 2017.
  - [576] T. Hudek, “Enumeration of interfaces on usb composite devices,” tech. rep., Microsoft, 04 2017.
  - [577] T. Hudek, “Plug and play support,” tech. rep., Microsoft, 04 2017.
  - [578] H. Ted and M. Duncan, “Overview of enumeration of interface collections on usb composite devices,” tech. rep., Microsoft, 01 2019.
  - [579] H. Ted, L. Bill, and K. Martin, “Usb host-side drivers in windows,” tech. rep., Microsoft, 04 2017.
  - [580] H. Ted and S. Tim, “Types of wdm device objects,” tech. rep., Microsoft, 06 2017.
  - [581] H. Ted, C. David, Y. Kevin, and aahill, “Getting started with umdf,” tech. rep., Microsoft, 04 2017.
  - [582] W. Mao, “Windows driver frameworks,” tech. rep., Microsoft, 03 2015.
  - [583] H. Ted, C. David, M. Gene, L. Bill, and M. Don, “Choosing a driver model for developing a usb client driver,” tech. rep., Microsoft, 05 2018.
-



- 
- [584] H. Ted, C. David, L. Bill, M. Gene, and M. Duncan, “How to write your first usb client driver (umdf),” tech. rep., Microsoft, 06 2019.
- [585] H. Ted, C. David, L. Bill, M. Gene, M. Duncan, G. Eliot, and S. Nick, “Understanding the usb client driver code structure (umdf),” tech. rep., Microsoft, 06 2019.
- [586] H. Ted, C. David, M. Gene, M. Don, and M. Duncan, “How to write your first usb client driver (kmdf),” tech. rep., Microsoft, 06 2019.
- [587] H. Ted, C. David, M. Gene, M. Duncan, L. Bill, and aahill, “Understanding the usb client driver code structure (kmdf),” tech. rep., Microsoft, 06 2019.
- [588] Microsoft, “Modern standby sleepstudy,” tech. rep., Microsoft, 07 2020.
- [589] J. Axelson, *USB Complete The Developer’s Guide fourth edition*, vol. 1. Madison, WI 53704: Lakeview Research LLC, 2009.
- [590] Omarrr024, Waddlesplash, and No92, “Usb human interface devices,” tech. rep., osdev.org, 06 2017.
- [591] U. I. Forum, “Hid usage tables, version 1.12,” tech. rep., USB.org, 10 2004.
- [592] Microsoft, “Hid\_descriptor structure,” tech. rep., Microsoft, 04 2018.
- [593] USB-IF, “Hid descriptor tool,” 01 2001.
- [594] P. Hutterer, “Understanding hid report descriptors,” *who-t.blogspot.com*, 12 2018.
- [595] D. Shawn, B. Georgios, and S. Zhang, “Cve-2018-8169 hidparser elevation of privilege vulnerability,” *mitre.org*, 06 2018.
- [596] T. Hudek, “Hid architecture,” tech. rep., Microsoft, 04 2017.
- [597] H. Ted, L. Bill, and M. Duncan, “Hid transport overview,” tech. rep., Microsoft, 02 2020.
- [598] P. M. of Bluetooth SIG, “Human interface device (hid) profile - version 1.0,” tech. rep., Bluetooth SIG, 05 2003.
- [599] H. Ted and S. Tim, “Creating export drivers,” tech. rep., Microsoft, 10 2019.
- [600] Microsoft, “Hidregisterminidriver function,” tech. rep., Microsoft, 04 2018.
- [601] T. Hudek, “Binding minidriviers to the hid class,” tech. rep., Microsoft, 04 2017.
- [602] H. Ted, C. David, and S. Tim, “Introduction to wdm,” tech. rep., Microsoft, 06 2017.
- [603] Microsoft, “Hid\_minidriver\_registration structure,” tech. rep., Microsoft, 04 2018.
- [604] W. Oney, *Programming the Microsoft Windows Driver Model*, vol. 1. One Microsoft Way, Redmond, Washington 98052-6399: Microsoft Press, 2003.
- [605] Microsoft, “I/o request packets,” tech. rep., Microsoft, 04 2017.
- [606] H. Ted, C. Saisang, and S. Tim, “Irp major function codes,” tech. rep., Microsoft, 03 2020.
- [607] H. Ted and S. Tim, “Handling irps,” tech. rep., Microsoft, 06 2017.
- [608] Microsoft, “Minidriviers and the hid class driver,” tech. rep., Microsoft, 04 2020.
- [609] Microsoft, “Keyboard enhancements in windows 8,” 09 2012.
- [610] Microsoft, “Hidnotifypresence function,” tech. rep., Microsoft, 02 2019.
- [611] H. Ted and S. Tim, “Function drivers,” tech. rep., Microsoft, 06 2017.
- [612] H. Ted and L. Bill, “Hid application programming interface (api),” tech. rep., Microsoft, 02 2020.
-



- [613] Microsoft, “Human interface devices (hid),” tech. rep., Microsoft, 05 2018.
  - [614] H. Ted and S. Tim, “Creating ioctl requests in drivers,” tech. rep., Microsoft, 06 2017.
  - [615] T. Hudek, “Different ways of handling irps - cheat sheet,” tech. rep., Microsoft, 12 2017.
  - [616] Microsoft, “How to submit an urb,” tech. rep., Microsoft, 04 2017.
  - [617] H. Ted and R. Dimitriy, “Top-level collections opened by windows for system use,” tech. rep., Microsoft, 04 2017.
  - [618] T. Hudek, “Sensor hid class driver,” tech. rep., Microsoft, 04 2017.
  - [619] T. Hudek, “Airplane mode radio management,” tech. rep., Microsoft, 04 2017.
  - [620] T. Hudek, “Hid client drivers,” tech. rep., Microsoft, 04 2017.
  - [621] T. Hudek, “Creating a new device setup class,” tech. rep., Microsoft, 04 2017.
  - [622] R. Thapa, “Device object and driver stack,” *windowsbugcheck*, 05 2017.
  - [623] H. Ted, S. Tim, and C. David, “Example wdm device stack,” tech. rep., Microsoft, 06 2017.
  - [624] H. Ted and G. Eliot, “Top-level collections,” tech. rep., Microsoft, 04 2017.
  - [625] H. Ted and G. Eliot, “Link collections,” tech. rep., Microsoft, 04 2017.
  - [626] H. Ted and G. Eliot, “Hidp\_link\_collection\_node structure,” tech. rep., Microsoft, 04 2018.
  - [627] T. Hudek, “Button capability arrays,” tech. rep., Microsoft, 04 2017.
  - [628] T. Hudek, “Value capability arrays,” tech. rep., Microsoft, 04 2017.
  - [629] Microsoft, “Hidp\_caps structure,” tech. rep., Microsoft, 04 2018.
  - [630] T. Hudek, “Data indices,” tech. rep., Microsoft, 04 2017.
  - [631] T. Hudek, “Opening hid collections,” tech. rep., Microsoft, 04 2017.
  - [632] Microsoft, “Device installation functions,” tech. rep., Microsoft, 09 2017.
  - [633] T. Hudek, “Guid\_devinterface\_hid,” tech. rep., Microsoft, 10 2018.
  - [634] T. Hudek, “Guid\_devinterface\_keyboard,” tech. rep., Microsoft, 10 2018.
  - [635] Microsoft, “Setupdigetdeviceinterfacedetail function,” tech. rep., Microsoft, 05 2018.
  - [636] Microsoft, “Createfilea function,” tech. rep., Microsoft, 05 2018.
  - [637] T. Hudek, “Finding and opening a hid collection,” tech. rep., Microsoft, 04 2017.
  - [638] H. Ted and S. Tim, “Using pnp notification,” tech. rep., Microsoft, 06 2017.
  - [639] H. Ted and S. Tim, “Pnp notification overview,” tech. rep., Microsoft, 06 2017.
  - [640] H. Ted and S. Tim, “Registering for device interface change notification,” tech. rep., Microsoft, 06 2017.
  - [641] T. Hudek, “Preparsed data,” tech. rep., Microsoft, 04 2017.
  - [642] Microsoft, “Hidd\_getpreparseddata function,” tech. rep., Microsoft, 06 2019.
  - [643] Microsoft, “Ioctl\_hid\_get\_collection\_information ioctl,” tech. rep., Microsoft, 04 2018.
  - [644] Microsoft, “Hid\_collection\_information ioctl,” tech. rep., Microsoft, 04 2018.
  - [645] Microsoft, “Ioctl\_hid\_get\_collection\_descriptor ioctl,” tech. rep., Microsoft, 04 2018.
-

- [646] T. Hudek, “Obtaining prepared data,” tech. rep., Microsoft, 04 2017.
  - [647] Microsoft, “Hidp\_getcaps function,” tech. rep., Microsoft, 04 2018.
  - [648] T. Hudek, “Collection capability,” tech. rep., Microsoft, 04 2017.
  - [649] Microsoft, “Hidp\_button\_caps structure,” tech. rep., Microsoft, 04 2018.
  - [650] Microsoft, “Hidp\_getbuttoncaps function,” tech. rep., Microsoft, 04 2018.
  - [651] Microsoft, “Hidp\_getspecificbuttoncaps function,” tech. rep., Microsoft, 04 2018.
  - [652] Microsoft, “Hidp\_value\_caps structure,” tech. rep., Microsoft, 04 2018.
  - [653] Microsoft, “Hidp\_getvaluecaps function,” tech. rep., Microsoft, 04 2018.
  - [654] Microsoft, “Hidp\_getspecificvaluecaps function,” tech. rep., Microsoft, 04 2018.
  - [655] T. Hudek, “Hidd\_getinputreport function,” tech. rep., Microsoft, 06 2019.
  - [656] T. Hudek, “Hidd\_setfeature function,” tech. rep., Microsoft, 06 2019.
  - [657] T. Hudek, “Hidd\_setoutputreport function,” tech. rep., Microsoft, 06 2019.
  - [658] S. Laboratories, *HUMAN INTERFACE DEVICE TUTORIAL*.
  - [659] Microsoft, “Ioctl\_hid\_get\_input\_report ioctl,” tech. rep., Microsoft, 04 2018.
  - [660] Microsoft, “Ioctl\_hid\_set\_feature ioctl,” tech. rep., Microsoft, 04 2018.
  - [661] Microsoft, “Ioctl\_hid\_set\_output\_report ioctl,” tech. rep., Microsoft, 04 2018.
  - [662] Microsoft, “Hid\_xfer\_packet structure,” tech. rep., Microsoft, 04 2018.
  - [663] Microsoft, “Hidp\_initializeportforid function,” tech. rep., Microsoft, 04 2018.
  - [664] Microsoft, “Hidp\_setusages function,” tech. rep., Microsoft, 04 2018.
  - [665] Microsoft, “Hidp\_getlinkcollectionnodes function,” tech. rep., Microsoft, 04 2018.
  - [666] Microsoft, “Hidp\_link\_collection\_node structure,” tech. rep., Microsoft, 04 2018.
  - [667] T. Hudek, “Enforcing a secure read for a hid collection,” tech. rep., Microsoft, 04 2017.
  - [668] Microsoft, “Privilege constants (authorization),” tech. rep., Microsoft, 07 2020.
  - [669] msdn, “Usb hid to ps/2 scan code translation table,” 4 2004.
  - [670] Microsoft, “Human interface devices (hid),” tech. rep., Microsoft, 05 2018.
  - [671] Microsoft, “Wpp software tracing,” tech. rep., Microsoft, 04 2017.
  - [672] Microsoft, “Driver\_object structure,” tech. rep., Microsoft, 04 2018.
  - [673] Microsoft, “Driver\_add\_device callback function,” tech. rep., Microsoft, 04 2018.
  - [674] H. Ted and S. Tim, “Writing an adddevice routine,” tech. rep., Microsoft, 06 2017.
  - [675] Microsoft, “Iocreatedevice function,” tech. rep., Microsoft, 04 2018.
  - [676] H. Ted and S. Tim, “Creating a device object,” tech. rep., Microsoft, 06 2017.
  - [677] Microsoft, “Ioattachdevicetodevicestack function,” tech. rep., Microsoft, 04 2018.
  - [678] H. Ted and S. Tim, “Specifying device types,” tech. rep., Microsoft, 06 2017.
  - [679] Microsoft, “Interlockedexchangeadd function (winnt.h),” tech. rep., Microsoft, 12 2018.
-

- [680] H. Ted, C. David, and S. Tim, “Device extensions,” tech. rep., Microsoft, 06 2017.
  - [681] H. Ted and S. Tim, “Guidelines for writing dpc routines,” tech. rep., Microsoft, 06 2017.
  - [682] Microsoft, “Keinitializedpc function,” tech. rep., Microsoft, 04 2018.
  - [683] R. Mark E., I. Alex, and S. David A., “Understanding the windows i/o system,” *The Microsoft Press Store by Pearson*, 09 2012.
  - [684] Microsoft, “Kdeferred\_routine callback function,” tech. rep., Microsoft, 04 2018.
  - [685] Microsoft, “Posetpowerstate function,” tech. rep., Microsoft, 04 2018.
  - [686] H. Ted and S. Tim, “Implementing wmi,” tech. rep., Microsoft, 06 2017.
  - [687] Microsoft, “Ioreuseirp function,” tech. rep., Microsoft, 04 2018.
  - [688] Microsoft, “Iosetcompletionroutineex function,” tech. rep., Microsoft, 04 2018.
  - [689] Microsoft, “Io\_completion\_routine callback function,” tech. rep., Microsoft, 04 2018.
  - [690] H. Ted and S. Tim, “Using iocompletion routines,” tech. rep., Microsoft, 06 2017.
  - [691] H. Ted and S. Tim, “Passing irps down the driver stack,” tech. rep., Microsoft, 06 2017.
  - [692] Microsoft, “Writing preoperation and postoperation callback routines,” tech. rep., Microsoft, 04 2017.
  - [693] Microsoft, “Iofcalldriver function,” tech. rep., Microsoft, 04 2018.
  - [694] H. Ted and S. Tim, “Completing irps,” tech. rep., Microsoft, 06 2017.
  - [695] H. Ted and S. Tim, “Implementing an iocompletion routine,” tech. rep., Microsoft, 06 2017.
  - [696] Microsoft, “Hidp\_getusagesex function,” tech. rep., Microsoft, 04 2018.
  - [697] Microsoft, “Usb hid to ps/2 scan code translation table,” tech. rep., Microsoft Corporation, 04 2004.
  - [698] Microsoft, “Hidp\_usageandpagelistdifference function,” tech. rep., Microsoft, 04 2018.
  - [699] Microsoft, “Hidp\_usageandpagelistdifference,” tech. rep., Microsoft, 05 2004.
  - [700] H. Ted and S. Tim, “Using a customtimerdpc routine,” tech. rep., Microsoft, 06 2017.
  - [701] V. Hu, “Registry keyboard response,” 11 2017.
  - [702] R. Chen, “How does the keyboard autorepeat setting work?,” tech. rep., Microsoft, 12 2009.
  - [703] Microsoft, “Hidp\_keyboard\_direction (windows ce 5.0),” tech. rep., Microsoft, 09 2012.
  - [704] Microsoft, “Getasynckeystate function,” tech. rep., Microsoft, 12 2018.
  - [705] Microsoft, “Ioctl\_internal\_keyboard\_connect ioctl (kbdmou.h),” tech. rep., Microsoft, 04 2018.
  - [706] Microsoft, “Connect\_data structure (kbdmou.h),” tech. rep., Microsoft, 04 2018.
  - [707] Microsoft, “Pservice\_callback\_routine callback function,” tech. rep., Microsoft, 04 2018.
  - [708] Microsoft, “Irp\_mj\_pnp,” tech. rep., Microsoft, 12 2017.
  - [709] Microsoft, “Plug and play minor irps,” tech. rep., Microsoft, 12 2017.
  - [710] Microsoft, “Iobuilddeviceiocontrolrequest function,” tech. rep., Microsoft, 04 2018.
  - [711] Microsoft, “Hidp\_maxusagelistlength function,” tech. rep., Microsoft, 04 2018.
  - [712] H. Ted and R. Dimitriy, “Hid usages,” tech. rep., Microsoft, 04 2017.
-

- [713] Microsoft, “Irp\_mn\_query\_device\_relations,” tech. rep., Microsoft, 08 2017.
  - [714] M. Satran, “Device management,” tech. rep., Microsoft, 05 2018.
  - [715] H. Ted and R. Dimitriy, “Hidclass hardware ids for top-level collections,” tech. rep., Microsoft, 04 2017.
  - [716] H. Ted and C. David, “Standard usb identifiers,” tech. rep., Microsoft, 04 2017.
  - [717] Microsoft, “Windows kernel-mode object manager,” tech. rep., Microsoft, 10 2018.
  - [718] M. Russinovich, “Winobj v2.22,” tech. rep., Microsoft, 02 2011.
  - [719] H. Ted, M. Don, and G. Eliot, “Using debugger extensions,” tech. rep., Microsoft, 05 2017.
  - [720] S. McDowell, *Windows 2000 Kernel Debugging*. Microsoft Technologies Series, Prentice Hall, 2001.
  - [721] D. Vostokov and S. D. Services, *Accelerated Windows Debugging 3: Training Course Transcript and Windbg Practice Exercises, Second Edition*. Opentask, 2nd ed., 2018.
  - [722] Microsoft, “!devstack,” tech. rep., Microsoft, 05 2017.
  - [723] H. Ted, S. Tim, and C. David, “Device tree,” tech. rep., Microsoft, 06 2017.
  - [724] H. Ted and M. Don, “!devobj,” tech. rep., Microsoft, 05 2017.
  - [725] H. Ted, M. Don, M. Tim, and G. Eliot, “Hid extensions,” tech. rep., Microsoft, 11 2017.
  - [726] Microsoft, “Keyboard and mouse input,” tech. rep., Microsoft, 01 2019.
  - [727] S. Michael, J. Mike, K. John, B. Drew, and C. David, “Keyboard input,” tech. rep., Microsoft, 05 2018.
  - [728] S. Michael, J. Mike, S. McLean, B. Drew, and C. David, “Window messages,” tech. rep., Microsoft, 05 2018.
  - [729] S. Michael, J. Mike, K. John, B. Drew, and C. David, “About messages and message queues,” tech. rep., Microsoft, 05 2018.
  - [730] R. Chen, “Asynchronous input vs synchronous input, a quick introduction,” tech. rep., Microsoft, 06 2013.
  - [731] J. M. Richter, *Programming Applications for Microsoft Windows with Cdrom*. USA: Microsoft Press, 4th ed., 1999.
  - [732] K. Lake, “The warp wishlist,” *OS2BSS.com*, 12 2004.
  - [733] Microsoft, “Irp\_mj\_read,” tech. rep., Microsoft, 08 2017.
  - [734] Microsoft, “Irp\_mj\_read (kbdclass),” tech. rep., Microsoft, 08 2016.
  - [735] Microsoft, “IoCompleterquest function,” tech. rep., Microsoft, 04 2018.
  - [736] H. Ted and S. Tim, “Driver thread context,” tech. rep., Microsoft, 06 2017.
  - [737] Microsoft, “Keyboard\_input\_data structure (ntddkbd.h),” tech. rep., Microsoft, 05 2018.
  - [738] Microsoft, “Debugging with symbols,” tech. rep., Microsoft, 05 2018.
  - [739] S. Michael, B. Drew, J. Thomas, and C. David, “Scheduling priorities,” tech. rep., Microsoft, 05 2018.
  - [740] M. Jones, “How to: Set a thread name in native code,” tech. rep., Microsoft, 17 2018.
  - [741] Microsoft, “Zwsetinformationthread function,” tech. rep., Microsoft, 04 2018.
  - [742] Microsoft, “Unicode\_string structure,” tech. rep., Microsoft, 12 2018.
-

- [743] P. Yosifovich, M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More (7th Edition)*, vol. 1. USA: Microsoft Press, 7th ed., 2017.
- [744] M. J. (j00ru), “Windows csrss write up: the basics,” *J00ru vexillum Blog*, 07 2010.
- [745] Microsoft, “Createprocessa function,” tech. rep., Microsoft, 12 2018.
- [746] Microsoft, “Createremotethread function (processthreadsapi.h),” tech. rep., Microsoft, 12 2018.
- [747] A. Maillard, “Dans les coulisses de microsoft windows,” *ntoskrnl.org*, 08 2020.
- [748] R. Chen, “Who implemented the windows nt blue screen of death?,” tech. rep., Microsoft, 09 2017.
- [749] C. Marcho, “Application compatibility - session 0 isolation,” *Microsoft Techcommunity*, 04 2007.
- [750] C. Torre, “Session 0 changes and vista compatibility for services running as interactive with desktop,” 12 2006.
- [751] M. Nelson, “Debugging windows services,” *Dr Dobb’s*, 04 2014.
- [752] Microsoft, “Access control lists,” tech. rep., Microsoft, 05 2018.
- [753] Microsoft, “Loadlibrarya function,” tech. rep., Microsoft, 12 2018.
- [754] Microsoft, “Getprocaddress function,” tech. rep., Microsoft, 12 2018.
- [755] Microsoft, “Linker support for delay-loaded dlls,” tech. rep., Microsoft, 04 2016.
- [756] Microsoft, “Pssetcreateprocessnotifyroutine function,” tech. rep., Microsoft, 04 2018.
- [757] Microsoft, “Resumethread function,” tech. rep., Microsoft, 12 2018.
- [758] Morning, “Windows 10 and win32k.sys,” 11 2017.
- [759] Microsoft, “Windows display driver model (wddm) architecture,” tech. rep., Microsoft, 04 2017.
- [760] H. Ted, S. Tim, and C. David, “Windows kernel obsolete routines,” tech. rep., Microsoft, 10 2018.
- [761] Microsoft, “Interface\_type enumeration,” tech. rep., Microsoft, 04 2018.
- [762] Microsoft, “Kewaitformultipleobjects function,” tech. rep., Microsoft, 04 2018.
- [763] S. Michael, J. Mike, B. Drew, and C. David, “Desktops,” tech. rep., Microsoft, 05 2018.
- [764] S. Michael, J. Mike, and C. David, “Handling screen savers,” tech. rep., Microsoft, 05 2018.
- [765] S. Tkachenko, “Enable screen saver password protection in windows 10,” *Winaero*, 03 2017.
- [766] R. Chen, “There are really only two effectively distinct settings for the uac slider,” tech. rep., Microsoft, 08 2016.
- [767] Microsoft, “Windows integrity mechanism design,” tech. rep., Microsoft, 07 2007.
- [768] R. Chen, “Why does the elevation prompt have only the wallpaper as its background?,” tech. rep., Microsoft, 01 2019.
- [769] R. Chen, “How can i detect that the system is no longer showing a uac prompt?,” tech. rep., Microsoft, 04 2020.
- [770] H. Ted, C. David, and S. Tim, “Driver\_object structure (wdm.h),” tech. rep., Microsoft, 04 2018.
- [771] H. Ted, C. David, and S. Tim, “Introduction to driver objects,” tech. rep., Microsoft, 06 2017.
- [772] Microsoft, “Obreferenceobjectbyhandle function,” tech. rep., Microsoft, 04 2018.
-

- [773] Microsoft, “Zwreadfile function,” tech. rep., Microsoft, 04 2018.
- [774] Microsoft, “Irp\_mj\_read (ifs),” tech. rep., Microsoft, 11 2017.
- [775] n4r1B, “Dissecting the windows defender driver - wdfilter (part 1),” *n4r1b blog*, 01 2020.
- [776] Microsoft, “Test signing,” tech. rep., Microsoft, 04 2017.
- [777] S. McLean, C. David, B. Drew, and S. Michael, “Working set,” tech. rep., Microsoft, 05 2018.
- [778] Microsoft, “Kestackattachprocess function,” tech. rep., Microsoft, 04 2018.
- [779] Microsoft, “Zwallocatevirtualmemory function (ntifs.h),” tech. rep., Microsoft, 04 2018.
- [780] Microsoft, “Mmsecurevirtualmemory function,” tech. rep., Microsoft, 04 2018.
- [781] Microsoft, “Ioregisterplugplaynotification function,” tech. rep., Microsoft, 04 2018.
- [782] H. Ted and C. David, “Guid\_devinterface\_cdrom,” tech. rep., Microsoft, 10 2018.
- [783] Microsoft, “File\_object structure,” tech. rep., Microsoft, 04 2018.
- [784] Microsoft, “Io\_notification\_event\_category enumeration,” tech. rep., Microsoft, 08 2019.
- [785] Microsoft, “Iounregisterplugplaynotification function,” tech. rep., Microsoft, 04 2018.
- [786] H. Pulapaka, “One windows kernel,” *Microsoft Techcommunity*, 10 2018.
- [787] Microsoft, “Hidp\_getusages function,” tech. rep., Microsoft, 04 2018.
- [788] S. Michael, B. Drew, M. Dan, K. Vivek, K. John, and C. David, “Preventing hangs in windows applications,” tech. rep., Microsoft, 05 2018.
- [789] R. Chen, “The case of the hung explorer window,” tech. rep., Microsoft, 08 2016.
- [790] R. Chen, “The dangers of sleeping on a ui thread,” tech. rep., Microsoft, 02 2006.
- [791] R. Chen, “Pumping messages while waiting for a period of time,” tech. rep., Microsoft, 01 2006.
- [792] R. Chen, “Why isn’t the original window order always preserved when you undo a show desktop?,” tech. rep., Microsoft, 09 2004.
- [793] Microsoft, “Ishungappwindow function,” tech. rep., Microsoft, 12 2018.
- [794] Microsoft, “Stickykeys (windows ce 5.0),” tech. rep., Microsoft, 09 2012.
- [795] B. Karl, D. I. S. Mauricio, S. McLean, and C. David, “Accessibility in windows 10,” tech. rep., Microsoft, 09 2019.
- [796] J. Mike, M. Gene, B. Karl, H. Shawn, S. Michael, and C. David, “Accessibility overview,” tech. rep., Microsoft, 09 2020.
- [797] Microsoft, “2.3.1 ntstatus values,” tech. rep., Microsoft, 10 2020.
- [798] Citrix, “Hdx realtime media engine for microsoft skype for business,” 05 2020.
- [799] Microsoft, “Kesetevent function,” tech. rep., Microsoft, 04 2018.
- [800] Microsoft, “Keclearevent function,” tech. rep., Microsoft, 04 2018.
- [801] S. Michael, B. Drew, and D. Coulter, “Multimedia class scheduler service,” tech. rep., Microsoft, 05 2018.
- [802] Microsoft, “Precision touchpad tuning,” tech. rep., Microsoft, 05 2017.
- [803] Microsoft, “Precision touchpad implementation guide,” tech. rep., Microsoft, 05 2017.
-

- 
- [804] Microsoft, “Windows precision touchpad collection,” tech. rep., Microsoft, 05 2017.
- [805] Microsoft, “Zwqueryvaluekey function,” tech. rep., Microsoft, 04 2018.
- [806] Microsoft, “Kesettimer function,” tech. rep., Microsoft, 04 2018.
- [807] Microsoft, “Using the clipboard,” tech. rep., Microsoft, 05 2018.
- [808] Microsoft, “About the clipboard,” tech. rep., Microsoft, 05 2018.
- [809] Microsoft, “Clipboard messages,” tech. rep., Microsoft, 05 2018.
- [810] Microsoft, “Wm\_paintclipboard message,” tech. rep., Microsoft, 05 2018.
- [811] Microsoft, “Desktop app user interface,” tech. rep., Microsoft, 05 2018.
- [812] S. Michael, J. Mike, B. Drew, and C. David, “Hooks,” tech. rep., Microsoft, 05 2018.
- [813] Microsoft, “Foreground and background threads,” tech. rep., Microsoft, 03 2017.
- [814] S. Michael, J. Mike, B. Drew, and C. David, “Creating windows in threads,” tech. rep., Microsoft, 05 2018.
- [815] Microsoft, “Createwindowexa function,” tech. rep., Microsoft, 12 2018.
- [816] Microsoft, “Getmessage function,” tech. rep., Microsoft, 12 2018.
- [817] Microsoft, “Peekmessagea function,” tech. rep., Microsoft, 12 2018.
- [818] Microsoft, “Showwindow function,” tech. rep., Microsoft, 12 2018.
- [819] Microsoft, “Showwindowasync function,” tech. rep., Microsoft, 12 2018.
- [820] Microsoft, “Setfocus function,” tech. rep., Microsoft, 12 2018.
- [821] Microsoft, “Destroywindow function,” tech. rep., Microsoft, 12 2018.
- [822] Microsoft, “Setforegroundwindow function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [823] M. Satran, “About remote desktop services,” tech. rep., Microsoft, 05 2018.
- [824] Microsoft, “Getalttabinfoa function,” tech. rep., Microsoft, 05 2018.
- [825] Microsoft, “Windows,” tech. rep., Microsoft, 05 2018.
- [826] S. Michael, J. Mike, S. McLean, and C. David, “Creating a window,” tech. rep., Microsoft, 05 2018.
- [827] S. Michael, J. Mike, S. McLean, and C. David, “About window classes,” tech. rep., Microsoft, 05 2018.
- [828] Microsoft, “Wndclassa structure,” tech. rep., Microsoft, 12 2018.
- [829] Microsoft, “Unregisterclassa function,” tech. rep., Microsoft, 12 2018.
- [830] Microsoft, “Registerclassa function,” tech. rep., Microsoft, 12 2018.
- [831] R. Chen, “When something gets added to a queue, it takes time for it to come out the front of the queue,” tech. rep., Microsoft, 02 2014.
- [832] Microsoft, “Teb structure,” tech. rep., Microsoft, 12 2018.
- [833] R. Chen, “The posted message queue vs the input queue vs the message queue,” tech. rep., Microsoft, 03 2013.
- [834] R. Chen, “When you share an input queue, you have to wait your turn,” tech. rep., Microsoft, 06 2013.
-



- 
- [835] R. Chen, “Posted messages are processed ahead of input messages, even if they were posted later,” tech. rep., Microsoft, 03 2013.
- [836] Microsoft, “Msg structure,” tech. rep., Microsoft, 12 2018.
- [837] S. Michael, J. Mike, S. McLean, and C. David, “Writing the window procedure,” tech. rep., Microsoft, 05 2018.
- [838] Microsoft, “Translatemessage function,” tech. rep., Microsoft, 12 2018.
- [839] Microsoft, “Dispatchmessage function,” tech. rep., Microsoft, 12 2018.
- [840] Microsoft, “Wm\_char message,” tech. rep., Microsoft, 07 2020.
- [841] Microsoft, “Translateaccelerators function,” tech. rep., Microsoft, 12 2018.
- [842] Microsoft, “Postquitmessage function,” tech. rep., Microsoft, 12 2018.
- [843] S. Michael, J. Mike, and C. David, “Wm\_quit message,” tech. rep., Microsoft, 05 2018.
- [844] Microsoft, “Windowproc callback function,” tech. rep., Microsoft, 03 2018.
- [845] Microsoft, “Defwindowproca function,” tech. rep., Microsoft, 12 2018.
- [846] S. Michael, K. Jennifer, W. Maira, B. Drew, and C. David, “User interface principles,” tech. rep., Microsoft, 05 2018.
- [847] S. Michael, J. Mike, and C. David, “Using messages and message queues,” tech. rep., Microsoft, 05 2018.
- [848] S. Michael, B. Drew, J. Mike, B. Karl, K. John, R. Dimitriy, and C. David, “Wm\_keydown message,” tech. rep., Microsoft, 05 2018.
- [849] S. Michael, B. Drew, J. Mike, B. Karl, R. Dimitriy, and C. David, “Wm\_syskeydown message,” tech. rep., Microsoft, 05 2018.
- [850] S. Michael, B. Drew, and C. David, “Wm\_hotkey message,” tech. rep., Microsoft, 05 2018.
- [851] S. Michael, “Keyboard input notifications,” tech. rep., Microsoft, 05 2018.
- [852] S. Michael, B. Drew, J. Mike, B. Karl, K. John, R. Dimitriy, and C. David, “Wm\_syskeyup message,” tech. rep., Microsoft, 05 2018.
- [853] S. Michael, B. Drew, J. Mike, B. Karl, K. John, R. Dimitriy, and C. David, “Wm\_keyup message,” tech. rep., Microsoft, 05 2018.
- [854] Microsoft, “Registerhotkey function,” tech. rep., Microsoft, 12 2018.
- [855] M. Satran, “Keyboard,” tech. rep., Microsoft, 05 2018.
- [856] R. Chen, “Windows vista changed the alt+tab order slightly,” tech. rep., Microsoft, 07 2008.
- [857] R. Chen, “What is the alt+tab order?,” tech. rep., Microsoft, 10 2003.
- [858] R. Chen, “Which windows appear in the alt+tab list?,” tech. rep., Microsoft, 10 2007.
- [859] S. Michael, B. Drew, J. Mike, B. Karl, K. John, R. Dimitriy, and C. David, “Wm\_unichar message,” tech. rep., Microsoft, 05 2018.
- [860] S. Michael, J. Mike, and C. David, “Using keyboard input,” tech. rep., Microsoft, 05 2018.
- [861] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual - Volume 3B*. Intel Corporation, 08 2007.
- [862] Microsoft, “Getfocus function (winuser.h),” tech. rep., Microsoft, 12 2018.
-

- [863] S. Michael, B. Drew, J. Mike, and C. David, “Wm\_killfocus message,” tech. rep., Microsoft, 05 2018.
  - [864] S. Michael, B. Drew, and C. David, “Wm\_setfocus message,” tech. rep., Microsoft, 05 2018.
  - [865] R. Chen, “Sharing an input queue takes what used to be asynchronous and makes it synchronous, like focus changes,” tech. rep., Microsoft, 06 2013.
  - [866] Microsoft, “SetActiveWindow function (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [867] S. Michael, B. Drew, J. Mike, and C. David, “Wm\_activate message,” tech. rep., Microsoft, 05 2018.
  - [868] Microsoft, “GetActiveWindow function (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [869] S. Michael, K. John, B. Drew, C. David, H. Shawn, and R. Michael, “Window features,” tech. rep., Microsoft, 05 2018.
  - [870] Microsoft, “BringWindowToTop function (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [871] Microsoft, “SetWindowPos function (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [872] Microsoft, “DeferWindowPos function (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [873] R. Chen, “Eventually, nothing is special any more,” tech. rep., Microsoft, 10 2008.
  - [874] Microsoft, “AttachThreadInput function (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [875] R. Chen, “AttachThreadInput is like taking two threads and pooling their money into a joint bank account, where both parties need to be present in order to withdraw any money,” tech. rep., Microsoft, 06 2013.
  - [876] Microsoft, “SetParent function (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [877] R. Chen, “Is it legal to have a cross-process parent/child or owner/owned window relationship?,” tech. rep., Microsoft, 04 2013.
  - [878] S. Michael, B. Drew, and C. David, “Raw input,” tech. rep., Microsoft, 05 2018.
  - [879] H. Ted, L. Bill, and C. David, “Keyboard and mouse class drivers,” tech. rep., Microsoft, 04 2017.
  - [880] H. Ted, S. Tim, and C. David, “Specifying exclusive access to device objects,” tech. rep., Microsoft, 06 2017.
  - [881] H. Ted, K. Jason, G. Eliot, and C. David, “Inf addreg directive,” tech. rep., Microsoft, 04 2017.
  - [882] Microsoft, “WdmLibIoCreatedDeviceSecure function (wdmsec.h),” tech. rep., Microsoft, 04 2018.
  - [883] I. Matteo and RbMm, “Createfile over usb hid device fails with access denied (5) since windows 10 1809,” 12 2018.
  - [884] Microsoft, “IoRegisterDeviceInterface function (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [885] Microsoft, “IoSetDeviceInterfaceState function (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [886] Microsoft, “ZwCreateFile function (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [887] S. Michael, J. Mike, B. Drew, and C. David, “Access\_mask,” tech. rep., Microsoft, 05 2018.
  - [888] H. Ted, H. Matt Lori Whippler, and C. David, “Writing irp dispatch routines,” tech. rep., Microsoft, 04 2017.
  - [889] Microsoft, “Irp\_mj\_create),” tech. rep., Microsoft, 08 2017.
  - [890] H. Ted, S. Tim, C. David, and M. Jason, “Controlling device access (wdm),” tech. rep., Microsoft, 06 2017.
  - [891] S. Michael, R. Dimitriy, J. Mike, B. Drew, and C. David, “About raw input,” tech. rep., Microsoft, 05 2018.
-

- 
- [892] Microsoft, “Wm\_input message,” tech. rep., Microsoft, 04 2020.
- [893] Microsoft, “Rawinput structure (winuser.h),” tech. rep., Microsoft, 05 2018.
- [894] Microsoft, “Getrawinputdata function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [895] Microsoft, “Rawmouse structure (winuser.h),” tech. rep., Microsoft, 05 2018.
- [896] Microsoft, “Rawkeyboard structure (winuser.h),” tech. rep., Microsoft, 05 2018.
- [897] Microsoft, “Rawhid structure (winuser.h),” tech. rep., Microsoft, 05 2018.
- [898] Microsoft, “Getrawinputdeviceinfoa function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [899] Microsoft, “Getrawinputbuffer function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [900] S. Michael and J. Coyle, “Using raw input,” tech. rep., Microsoft, 05 2018.
- [901] Microsoft, “Rawinputdevice structure (winuser.h),” tech. rep., Microsoft, 05 2018.
- [902] Microsoft, “Getrawinputdevicelist function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [903] Microsoft, “Getrawinputdeviceinfoa function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [904] Microsoft, “Rawinputdevicelist structure (winuser.h),” tech. rep., Microsoft, 12 2018.
- [905] R. Chen, “How can i detect whether a keyboard is attached to the computer?,” tech. rep., Microsoft, 07 2015.
- [906] Microsoft, “Mapvirtualkeyexa function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [907] Microsoft, “Loadkeyboardlayouta function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [908] Microsoft, “Keyboard identifiers and input method editors for windows,” tech. rep., Microsoft, 05 2017.
- [909] R. Colin, C. Saisang, J. Mike, T. Next, and H. Gordon, “Creating a resource-only dll,” tech. rep., Microsoft, 01 2020.
- [910] S. Michael, B. Drew, and C. David, “Stringtable resource,” tech. rep., Microsoft, 05 2018.
- [911] Microsoft, “Default input profiles (input locales) in windows,” tech. rep., Microsoft, 05 2017.
- [912] Microsoft, “Getprocessdefaultlayout function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [913] Microsoft, “Getkeyboardlayoutlist function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [914] Microsoft, “Setprocessdefaultlayout function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [915] U. I. Forum, “Hid usage tables for universal serial bus (usb) - version 1.2,” tech. rep., USB.org, 07 2020.
- [916] H. Ted and C. David, “Overview of device and driver installation,” tech. rep., Microsoft, 10 2019.
- [917] H. Ted, C. David, G. Eliot, and Cymoki, “Roadmap for device and driver installation,” tech. rep., Microsoft, 04 2017.
- [918] Microsoft, “Getkeyboardlayoutnamea function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [919] Microsoft, “Getkeyboardlayout function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [920] S. Michael, K. John, B. Drew, and C. David, “Wm\_inputlangchange message,” tech. rep., Microsoft, 05 2018.
- [921] S. Michael, K. John, B. Drew, and C. David, “Wm\_inputlangchangerequest message,” tech. rep., Microsoft, 05 2018.
- [922] S. Michael, “About national language support,” tech. rep., Microsoft, 05 2018.
-

- [923] S. Michael and C. David, “Nls terminology,” tech. rep., Microsoft, 05 2018.
- [924] R. Chen, “A consequence of being the first to adopt a standard is that you may end up being the only one to adopt it: The sad story of korean jamo,” tech. rep., Microsoft, 10 2020.
- [925] K. Brown, “How to create a custom keyboard layout in windows,” *Dailydoseoftech*, 03 2019.
- [926] D. Price, “How to create a custom keyboard layout on windows,” *Make Use Of*, 08 2018.
- [927] B. Golden, “Keyboard layout samples,” tech. rep., Microsoft, 09 2019.
- [928] Microsoft, “Keyboard layout generator tool,” tech. rep., Microsoft, 06 2006.
- [929] M. S. Kaplan, “Knowing the layout doesn’t mean knowing how to lay it out,” 03 2011.
- [930] G. GALÉRON, “Création d’un clavier windows personnalisé,” 05 2019.
- [931] J. Kucera, “Keyboard layout info,” 03 2021.
- [932] S. McLean, S. Kent, C. David, B. Drew, and S. Michael, “Predefined keys,” tech. rep., Microsoft, 05 2018.
- [933] Microsoft, “SendMessage function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [934] Microsoft, “Postmessagea function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [935] S. Michael, J. Mike, B. Drew, and C. David, “SendMessage, postmessage, and related functions,” tech. rep., Microsoft, 05 2018.
- [936] R. Chen, “You can’t simulate keyboard input with postmessage,” tech. rep., Microsoft, 03 2005.
- [937] W. Steven, C. David, J. Mike, and S. Michael, “Using an input method editor in a game,” tech. rep., Microsoft, 05 2018.
- [938] M. Corporation, “Input method editor and text services framework accessibility in windows xp,” tech. rep., Microsoft, 06 2002.
- [939] B. Karl and S. Michael, “Input method manager,” tech. rep., Microsoft, 05 2018.
- [940] Microsoft, “Sendinput function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [941] Microsoft, “Input structure (winuser.h),” tech. rep., Microsoft, 05 2018.
- [942] R. Chen, “How do i forward an exported function to an ordinal in another dll?,” tech. rep., Microsoft, 11 2012.
- [943] R. Chen, “Exported functions that are really forwarders,” tech. rep., Microsoft, 07 2006.
- [944] H. Ted, Matt, and C. David, “Buffer handling,” tech. rep., Microsoft, 04 2017.
- [945] H. Ted, S. Tim, and C. David, “Accessing user-space memory,” tech. rep., Microsoft, 06 2017.
- [946] Microsoft, “User-mode interactions: Guidelines for kernel-mode drivers,” *Microsoft Developer Network*, 07 2006.
- [947] M. Satran, “Desktop window manager,” tech. rep., Microsoft, 12 2018.
- [948] GwynethM, S. Kent, W. Maira, and G. Andrew, “Windows keyboard layouts,” tech. rep., Microsoft, 01 2017.
- [949] S. Michael, B. Drew, J. Mike, R. Dimitriy, and C. David, “About mouse input,” tech. rep., Microsoft, 05 2018.
- [950] S. Sonia and H. Doron, “Virtual key codes,” 10 2013.
- [951] Microsoft, “Vkkeyscanexa function (winuser.h),” tech. rep., Microsoft, 05 2018.
-

- 
- [952] Microsoft, “Tounicodeex function (winuser.h),” tech. rep., Microsoft, 05 2018.
- [953] R. Chen, “Why does ctrl+scrolllock cancel dialogs?,” tech. rep., Microsoft, 02 2008.
- [954] Microsoft, “Windows data types,” tech. rep., Microsoft, 05 2018.
- [955] Microsoft, “Toasciix function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [956] S. Michael, B. Drew, J. Mike, K. Bridge, R. Dimitriy, and C. David, “Wm\_deadchar message,” tech. rep., Microsoft, 05 2018.
- [957] Microsoft, “Getlocaleinfo function (winls.h),” tech. rep., Microsoft, 12 2018.
- [958] Microsoft, “Widechartomultibyte function (stringapiset.h),” tech. rep., Microsoft, 12 2018.
- [959] Microsoft, “Multibytetowidechar function (stringapiset.h),” tech. rep., Microsoft, 12 2018.
- [960] R. Chen, “Disabling the prtsc key by blocking input,” tech. rep., Microsoft, 12 2013.
- [961] Microsoft, “Capturing an image,” tech. rep., Microsoft, 05 2018.
- [962] Microsoft, “Setwindowdisplayaffinity function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [963] R. Chen, “How do i make it more difficult for somebody to take a screenshot of my window?,” tech. rep., Microsoft, 06 2013.
- [964] S. Michael, J. Mike, and C. David, “About hot key controls,” tech. rep., Microsoft, 05 2018.
- [965] Microsoft, “Getkeystate function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [966] R. Chen, “What’s the difference between getkeystate and getasynckeystate?,” tech. rep., Microsoft, 11 2004.
- [967] Microsoft, “Getkeyboardstate function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [968] Microsoft, “Probeforwrite function (wdm.h),” tech. rep., Microsoft, 04 2018.
- [969] Microsoft, “Getsystemmetrics function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [970] Microsoft, “Setkeyboardstate function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [971] Microsoft, “Probeforread function (wdm.h),” tech. rep., Microsoft, 04 2018.
- [972] A. Hakobyan, “Toggling the num lock, caps lock, and scroll lock keys,” *Codeproject*, 05 2002.
- [973] Microsoft, “Getmsgproc callback function,” tech. rep., Microsoft, 03 2018.
- [974] Microsoft, “Callwndproc callback function,” tech. rep., Microsoft, 03 2018.
- [975] Microsoft, “Hookproc callback function (winuser.h),” tech. rep., Microsoft, 03 2018.
- [976] Microsoft, “Cbtproc callback function,” tech. rep., Microsoft, 03 2018.
- [977] S. Hickey, “Winevents,” tech. rep., Microsoft, 03 2018.
- [978] Microsoft, “Debugproc function,” tech. rep., Microsoft, 03 2018.
- [979] Microsoft, “Foregroundidleproc callback function,” tech. rep., Microsoft, 03 2018.
- [980] Microsoft, “Journalrecordproc callback function,” tech. rep., Microsoft, 03 2018.
- [981] Microsoft, “Journalplaybackproc callback function,” tech. rep., Microsoft, 03 2018.
- [982] Microsoft, “Keyboardproc callback function,” tech. rep., Microsoft, 03 2018.
- [983] Microsoft, “Lowlevelkeyboardproc callback function,” tech. rep., Microsoft, 03 2018.
-

- [984] Microsoft, “Mouseproc callback function,” tech. rep., Microsoft, 03 2018.
  - [985] Microsoft, “Lowlevelmouseproc callback function,” tech. rep., Microsoft, 03 2018.
  - [986] Microsoft, “Messageproc callback function,” tech. rep., Microsoft, 03 2018.
  - [987] Microsoft, “Shellproc callback function,” tech. rep., Microsoft, 03 2018.
  - [988] Microsoft, “Sysmsgproc callback function,” tech. rep., Microsoft, 03 2018.
  - [989] M. Botacin, A. Grégio, and P. De Geus, “Malware variants identification in practice,” in *SBSeg 2019*, 09 2019.
  - [990] J. Berdajs and Z. Bosnic, “Extending applications using an advanced approach to dll injection and api hooking,” *Software: Practice and Experience*, vol. 40, pp. 567 – 584, 06 2010.
  - [991] S. Zhang, M. Khambatti, and P. Dasgupta, “Processes migration through virtualization in a computing community,” *cactus.eas.asu.edu*, 10 2020.
  - [992] R. Chen, “What does the thread parameter to set`WindowsHookEx` actually mean?,” tech. rep., Microsoft, 09 2018.
  - [993] S. Michael, B. Drew, and C. David, “Using hooks,” tech. rep., Microsoft, 05 2018.
  - [994] Microsoft, “Protected processes,” *Microsoft Developer Network*, 11 2006.
  - [995] S. Michael, B. Drew, and C. David, “Protected media path,” tech. rep., Microsoft, 05 2018.
  - [996] J. Forshaw, “Injecting code into windows protected processes using com - part 1,” *Google Project Zero Blog*, 10 2018.
  - [997] S. Michael, H. Shawn, W. Tyler, J. Mike, B. Drew, and C. David, “Protecting anti-malware services,” tech. rep., Microsoft, 08 2018.
  - [998] Microsoft, “Unhookwindowshookex function (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [999] Microsoft, “Wm\_canceljournal message,” tech. rep., Microsoft, 05 2018.
  - [1000] Microsoft, “Freelibrary function (libloaderapi.h),” tech. rep., Microsoft, 12 2018.
  - [1001] S. Michael, B. Drew, J. Mike, S. McLean, K. John, G. Alex, and C. David, “Dllmain entry point,” tech. rep., Microsoft, 07 2020.
  - [1002] R. Chen, “Some reasons not to do anything scary in your dllmain,” tech. rep., Microsoft, 01 2004.
  - [1003] R. Chen, “Another reason not to do anything scary in your dllmain: Inadvertent deadlock,” tech. rep., Microsoft, 01 2004.
  - [1004] R. Chen, “Some reasons not to do anything scary in your dllmain, part 3,” tech. rep., Microsoft, 08 2014.
  - [1005] Microsoft, “Callnexthookex function (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [1006] R. Chen, “Why does setfocus fail without telling me why?,” tech. rep., Microsoft, 06 2019.
  - [1007] R. Chen, “Why does setwindowshookex take an hinstance parameter?,” tech. rep., Microsoft, 08 2006.
  - [1008] Microsoft, “Getmodulehandlea function (libloaderapi.h),” tech. rep., Microsoft, 12 2018.
  - [1009] Microsoft, “Getmodulefilenamew function (libloaderapi.h),” tech. rep., Microsoft, 12 2018.
  - [1010] P. DiLascia, “Atl virtual functions and vtables,” *MSDN Magazine*, 03 2000.
  - [1011] S. Michael, J. Mike, B. Drew, and C. David, “Desktop security and access rights,” tech. rep., Microsoft, 05 2018.
-

- 
- [1012] S. Michael, J. Mike, W. Jim, and C. David, “About atom tables,” tech. rep., Microsoft, 08 2020.
  - [1013] Microsoft, “Keattachprocess function (ntifs.h),” tech. rep., Microsoft, 04 2018.
  - [1014] Microsoft, “Kedetachprocess function (ntifs.h),” tech. rep., Microsoft, 04 2018.
  - [1015] R. Colin, C. Saisang, J. Mike, and H. Gordon, “/highentropyva (support 64-bit aslr),” tech. rep., Microsoft, 06 2018.
  - [1016] Microsoft, “Kesetprioritythread function (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [1017] R. Chen, “Not actually crossing the airtight hatchway: Applying per-user overrides,” tech. rep., Microsoft, 08 2019.
  - [1018] S. Michael, B. Drew, and C. David, “Asynchronous procedure calls,” tech. rep., Microsoft, 05 2018.
  - [1019] Microsoft, “Kbdllhookstruct structure (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [1020] S. Michael, B. Drew, W. Steven, J. Mike, and C. David, “Directx graphics and gaming,” tech. rep., Microsoft, 05 2018.
  - [1021] Microsoft, “Keyboard, mouse, and controller input in directx,” tech. rep., Microsoft, 08 2020.
  - [1022] Microsoft, “Microsoft.directx.directinput,” tech. rep., Microsoft, 11 2009.
  - [1023] Microsoft, “Directinput,” tech. rep., Microsoft, 09 2011.
  - [1024] C. Walbourn, “Directx tool kit for directx 11,” tech. rep., Microsoft, 06 2015.
  - [1025] S. Mason and R. Myopic, “Directx 8 and the keyboard,” 01 2002.
  - [1026] averagejoe, “Keylogger using directx,” 06 2017.
  - [1027] D. Cyril and raptor70, “La gestion des inputs avec directx : Directinput,” tech. rep., Developpez.com, 08 2008.
  - [1028] Microsoft, “Directinput8create,” tech. rep., Microsoft, 09 2011.
  - [1029] Microsoft, “Idirectinput8 interface,” tech. rep., Microsoft, 09 2011.
  - [1030] Microsoft, “Idirectinput8::createdevice method,” tech. rep., Microsoft, 09 2011.
  - [1031] Microsoft, “Using directinput,” tech. rep., Microsoft, 09 2011.
  - [1032] Microsoft, “Idirectinputdevice8 interface,” tech. rep., Microsoft, 09 2011.
  - [1033] Microsoft, “Idirectinputdevice8::setdataformat method,” tech. rep., Microsoft, 09 2011.
  - [1034] Microsoft, “Device data formats,” tech. rep., Microsoft, 09 2011.
  - [1035] Microsoft, “Cooperative levels,” tech. rep., Microsoft, 09 2011.
  - [1036] Microsoft, “Idirectinputdevice8::setcooperativelevel method,” tech. rep., Microsoft, 09 2011.
  - [1037] Microsoft, “Acquiring devices,” tech. rep., Microsoft, 09 2011.
  - [1038] Microsoft, “Polling and event notification,” tech. rep., Microsoft, 09 2011.
  - [1039] Microsoft, “Idirectinputdevice8::poll method,” tech. rep., Microsoft, 09 2011.
  - [1040] Microsoft, “Idirectinputdevice8::seteventnotification method,” tech. rep., Microsoft, 09 2011.
  - [1041] Microsoft, “Createeventa function (synchapi.h),” tech. rep., Microsoft, 12 2018.
  - [1042] Microsoft, “Directinput device data,” tech. rep., Microsoft, 09 2011.
-



- [1043] Microsoft, “Buffered and immediate data,” tech. rep., Microsoft, 09 2011.
  - [1044] Microsoft, “Keyboard data,” tech. rep., Microsoft, 09 2011.
  - [1045] Microsoft, “Immediate keyboard data,” tech. rep., Microsoft, 09 2011.
  - [1046] Microsoft, “Idirectinputdevice8::getdevicestate method,” tech. rep., Microsoft, 09 2011.
  - [1047] Microsoft, “Keyboard device enumeration,” tech. rep., Microsoft, 09 2011.
  - [1048] Microsoft, “Buffered keyboard data,” tech. rep., Microsoft, 09 2011.
  - [1049] Microsoft, “Interpreting keyboard data,” tech. rep., Microsoft, 09 2011.
  - [1050] Microsoft, “Device properties,” tech. rep., Microsoft, 09 2011.
  - [1051] Microsoft, “Idirectinputdevice8::setproperty method,” tech. rep., Microsoft, 09 2011.
  - [1052] Microsoft, “Diprodpword structure,” tech. rep., Microsoft, 09 2011.
  - [1053] Microsoft, “Dideviceobjectdata structure,” tech. rep., Microsoft, 09 2011.
  - [1054] Microsoft, “Idirectinputdevice8::getdevicedata method,” tech. rep., Microsoft, 09 2011.
  - [1055] S. Michael, B. Drew, K. Jennifer, K. Jim, W. Steven, J. Mike, S. McLean, and C. David, “Windows api index,” tech. rep., Microsoft, 05 2018.
  - [1056] P. Alcorn, “Windows xp source code leaked, posted to 4chan (update, it works),” *Tomshardware*, 10 2020.
  - [1057] P. Paganini, “Developer successfully compiled leaked source code for ms windows xp and windows server 2003 oss,” *Security Affairs*, 09 2020.
  - [1058] K. Panos, “Retrotechtacular: Cold war-era hardware keyloggers,” *Hackaday*, 11 2015.
  - [1059] S. Maneki and C. History, *Learning from the Enemy: The Gunman Project*. Center for Cryptologic History: Military Studies Press, 2012.
  - [1060] P. Kivolowitz, “The ‘security digest’ archives (tm) - a program to allow anyone to crack unix (4.1 and 2),” 11 1983.
  - [1061] B. Chacos, “Malicious keylogger malware found lurking in highly publicized gta v mod,” *PCWorld*, 05 2015.
  - [1062] TAHIN, “Keylogger in hp audio driver,” *spiceworks*, 05 2017.
  - [1063] T. Schroeder, “Keylogger in hewlett-packard audio driver,” *Modzero AG*, 05 2017.
  - [1064] B. Edmund, C. Earl, and T. Emmanuel, “Poisoning the well: Banking trojan targets google search results,” *Talos Intelligence*, 11 2017.
  - [1065] L. Ebach, “Analysis results of zeus.variant.panda,” *GData*, 06 2017.
  - [1066] O. Tom, “Keystroke logging (keylogging),” *Adventures in Security*, 05 2008.
  - [1067] S. Sagioglu and G. Canbek, “Keyloggers increasing threats to computer security and privacy,” *Technology and Society Magazine, IEEE*, vol. 28, pp. 10 – 17, 02 2009.
  - [1068] R. Creutzburg, “The strange world of keyloggers - an overview, part i,” *Electronic Imaging*, vol. 2017, pp. 139–148, 01 2017.
  - [1069] J. V. Monaco, “Sok: Keylogging side channels,” *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 211–228, 2018.
  - [1070] F. Mihailowitsch, “Detecting hardware keyloggers,” in *HITBSECCONG 2010 - Malaysia*, 10 2010.
-

- [1071] Keelog, “User’s guide keygrabber ps/2 with usb download accelerator,” tech. rep., Keelog, 10 2019.
- [1072] derek, “Open source hardware keylogger,” 04 2009.
- [1073] Jamby, “How to build your own usb keylogger,” *Instructables circuits*, 05 2011.
- [1074] Keelog, “User’s guide keygrabber usb inc. mac compatibility pack (mcp),” tech. rep., Keelog, 10 2019.
- [1075] Keelog, “User’s guide keygrabber forensic keylogger,” tech. rep., Keelog, 05 2019.
- [1076] Keelog, “Airdrive forensic keylogger pro - quick start,” tech. rep., Keelog, 12 2018.
- [1077] Keelog, “Wireless keylogger - do it yourself!,” tech. rep., Keelog, 01 2012.
- [1078] Keelog, “Keygrabber forensic keylogger cable pro - quick start,” tech. rep., Keelog, 07 2019.
- [1079] Keelog, “Keygrabber forensic keylogger module pro - quick start,” tech. rep., Keelog, 07 2019.
- [1080] Keelog, “Forensic keylogger keyboard - quick start,” tech. rep., Keelog, 07 2019.
- [1081] T. G. . (TG1), “Ieee 802.15 wpan,” 06 2002.
- [1082] R. S. Sreenivas and R. Anitha, “Detecting keyloggers based on traffic analysis with periodic behaviour,” *Network Security*, vol. 2011, no. 7, pp. 14 – 19, 2011.
- [1083] M. Moser and P. Schrödel, “27mhz wireless keyboard analysis report aka ”we know what you typed last summer”,” 2008.
- [1084] L. Zhuang, F. Zhou, and J. D. Tygar, “Keyboard acoustic emanations revisited,” *Association for Computing Machinery*, vol. 13, 11 2009.
- [1085] D. Asonov and R. Agrawal, “Keyboard acoustic emanations,” in *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pp. 3–11, 2004.
- [1086] NSA, “Nacsim 5000 tempest fundamentals.”
- [1087] M. Vuagnoux and S. Pasini, “Compromising electromagnetic emanations of wired and wireless keyboards,” *USENIX Security Symposium*, pp. 1–16, 01 2009.
- [1088] K. Chen, “Reversing and exploiting an apple firmware update,” *Black hat USA 2009*, 07 2009.
- [1089] K. Chen, “Reversing and exploiting an apple firmware update,” in *Black hat USA 2009*, (Las Vegas - USA), 07 2009.
- [1090] U. Frisk, “Attacking uefi runtime services and linux,” *Blog Frizk*, 01 2017.
- [1091] M. Kinney, “Edk ii uefi driver writer’s guide,” tech. rep., Tianocore, 04 2018.
- [1092] M. Kinney, “Edk ii module writer’s guide,” tech. rep., Tianocore, 09 2018.
- [1093] V. Zimmer, M. Rothman, and S. Marisetty, *Beyond BIOS: Developing with the Unified Extensible Firmware Interface, Third Edition*. Berlin, DEU: DeG Press, 3rd ed., 2017.
- [1094] Intx13, “System management mode,” 11 2015.
- [1095] D. Loic, L. Olivier, M. Benjamin, and G. Olivier, “Getting into the smram: Smm reloaded,” in *CanSecWest 2009*, (SGDN/DCSSI 51 boulevard de la Tour Maubourg 75007 Paris), Central Directorate for Information Systems Security, 2009.
- [1096] P. Chifflier, “Uefi et bootkits pci : le danger vient d’en bas,” *sstic 2013*, 06 2013.
- [1097] Y. Jiewen and Z. Vincent J., “Understanding uefi secure boot chain,” tech. rep., Tianocore, 06 2019.
- [1098] M. Pierre-Francois, “System management mode, when your hardware affects your os,” in *C0c0n 2020*, 09 2020.
-

- [1099] D. Hardware, “Smis are eeeevil (part 1),” *Microsoft Developer Network*, 08 2005.
- [1100] M. Kinney, “Edk ii minimum platform specification,” tech. rep., Tianocore, 05 2019.
- [1101] Cr4sh, “Building reliable smm backdoor for uefi based platforms,” *Blog CR4*, 07 2015.
- [1102] B. Erich and F. S. Foundation, “Gnu grub,” 1995.
- [1103] L. Mark, K. Igor, and P. Yury, “Mosaicregressor: Lurking in the shadows of uefi,” *Securelist*, 10 2020.
- [1104] W. Richard and R. Brian, “Uefi secure boot in modern computer security solutions,” *UEFI.org*, 08 2019.
- [1105] S. Kaczmarek, “Uefi and dreamboot,” in *Hack in the Box Security Conference*, (Kuala Lumpur, Malasia), 2013.
- [1106] D. Oleksiuk, “Boot backdoor for windows,” 04 2021.
- [1107] R. Andrew and S. Karen, “Bios integrity measurement guidelines,” *Computer Security Resource Center - NIST*, 11 2011.
- [1108] J. Butterworth and C. Kallenberg, “Problems with the static root of trust for measurement,” 2013.
- [1109] S. Romana, H. Pareek, and L. R., “Dynamic root of trust and challenges,” *International Journal of Security, Privacy and Trust Management*, vol. 5, pp. 01–06, 05 2016.
- [1110] S. Choinyambuu, “Root of trust for measurement mitigating - the lying endpoint problem of tnc,” Master’s thesis, HSR Hochschule fur technik Rapperswil, Oberseestrasse 10 CH-8640 Rapperswil, 06 2011.
- [1111] A. Marty Hernandez, K. Laura, H. Justin, V. Denise, S. Daniel, and G. Andres Mariano, “Windows defender system guard: How a hardware-based root of trust helps protect windows 10,” tech. rep., Microsoft, 03 2019.
- [1112] S. Nazmus, D. Alexander, and F. Chris, “Force firmware code to be measured and attested by secure launch on windows 10,” *Enterprise and OS Security - Microsoft*, 09 2020.
- [1113] M. Security, “System management mode deep dive: How smm isolation hardens the platform,” *Enterprise and OS Security - Microsoft*, 11 2020.
- [1114] N. Grebennikov, “Keyloggers: Implementing keyloggers in windows. part two,” *Securelist*, 06 2011.
- [1115] E. TINAZTEPE, “The adventures of a keystroke - an in-depth look into keyloggers on windows,” 10 2014.
- [1116] H. Ted, G. Eliot, R. Dimitriy, C. David, S. Nick, and aahill, “Keyboard and mouse hid client drivers,” tech. rep., Microsoft, 04 2017.
- [1117] B. Jack, “Remote windows kernel exploitation - step into the ring 0,” in *Black Hat USA 05*, 2005.
- [1118] Z. Wang, X. Jiang, W. Cui, and P. Ning, “Countering kernel rootkits with lightweight hook protection,” in *2009 ACM Conference on Computer and Communications Security*, (Chicago, Illinois, USA), pp. 545–554, CCS 2009, 11 2009.
- [1119] S. Mohd, Z. Syed, and M. Maarof, “In memory detection of windows api call hooking technique,” *2015 International Conference on Computer, Communications, and Control Technology (I4CT)*, pp. 294–298, 08 2015.
- [1120] Y. Pavel, I. Alex, R. Mark E., and S. David A., *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more (7th Edition)*, vol. 1. Microsoft Press, 05 2017.
- [1121] Microsoft, “Windows 10 device guard and credential guard demystified,” tech. rep., Microsoft, 02 2016.
- [1122] A. Chevalier, “Virtualization based security - part 1: The boot process,” *Blog Amossys*, 02 2017.
- [1123] H. Ted, C. David, and Cymoki, “Driver signing,” tech. rep., Microsoft, 04 2017.
-

- 
- [1124] D. Grayson, “Practical windows code and driver signing - code and driver signing for microsoft windows 10, 8.1, 8, 7, vista, and xp,” *davidegrayson*, 05 2018.
- [1125] N. S. Agency, “Cve-2020-0601 - windows cryptoapi spoofing vulnerability,” 01 2020.
- [1126] F. Raynal, “Petite faq sur la faille windows cve-2020-0601 et la nsa a l’usage des paranoiaques,” *Medium*, 01 2020.
- [1127] R. Chen, “Not actually crossing the airtight hatchway: Applying per-user overrides,” tech. rep., Microsoft, 08 2019.
- [1128] R. Chen, “It rather involved being on the other side of this airtight hatchway: Planting files onto a custom path,” tech. rep., Microsoft, 04 2020.
- [1129] R. Chen, “It rather involved being on the other side of this airtight hatchway,” tech. rep., Microsoft, 05 2006.
- [1130] Microsoft, “Microsoft security servicing criteria for windows,” tech. rep., Microsoft, 01 2020.
- [1131] C. PICHAUD, “Rootkit part 1 - keylogger,” *devoteam*, 06 2019.
- [1132] M. Russinovich, “Ctrl2cap v2.0,” tech. rep., Microsoft, 11 2006.
- [1133] H. Ted, S. Tim, and C. David, “Buffer descriptions for i/o control codes,” tech. rep., Microsoft, 06 2017.
- [1134] H. Ted and C. David, “Registry trees and keys for devices and drivers,” tech. rep., Microsoft, 04 2017.
- [1135] H. Ted, F. Nabil, C. David, and K. Varadharajan, “Using a universal inf file,” tech. rep., Microsoft, 04 2020.
- [1136] H. Ted, S. Kenichi, N. Jason K., G. Eliot, and C. David, “Hklm\system\currentcontrolset\services registry tree,” tech. rep., Microsoft, 04 2017.
- [1137] H. Ted, C. David, and N. Jason K., “Introduction to registry keys for drivers,” tech. rep., Microsoft, 04 2017.
- [1138] T. Hudek, “How windows selects a driver for a device,” tech. rep., Microsoft, 03 2020.
- [1139] H. Ted, S. Kenichi, N. Jason K., G. Eliot, and C. David, “Hklm\system\currentcontrolset\control registry tree,” tech. rep., Microsoft, 04 2017.
- [1140] H. Ted and C. David, “Overview of device interface classes,” tech. rep., Microsoft, 04 2017.
- [1141] Microsoft, “Overview of device setup classes,” tech. rep., Microsoft, 04 2017.
- [1142] Microsoft, “Class and classguid entries for inf version section,” tech. rep., Microsoft, 08 2020.
- [1143] H. Ted and C. David, “Installing a filter driver,” tech. rep., Microsoft, 04 2017.
- [1144] Selerner, Zlockard, J. Mike, G. Eliot, and C. David, “Device filter driver ordering,” tech. rep., Microsoft, 04 2019.
- [1145] H. Ted, M. Gene, and M. Duncan, “Usb device registry entries,” tech. rep., Microsoft, 04 2017.
- [1146] H. Ted and C. David, “Ps/2 (i8042prt) driver,” tech. rep., Microsoft, 04 2017.
- [1147] Microsoft, “IoctlInternal\_i8042\_hook\_keyboard ioctl (ntdd8042.h),” tech. rep., Microsoft, 04 2018.
- [1148] Microsoft, “Internal\_i8042\_hook\_keyboard structure (ntdd8042.h),” tech. rep., Microsoft, 04 2018.
- [1149] Microsoft, “ntdd8042.h header,” tech. rep., Microsoft, 05 2018.
- [1150] Microsoft, “Pi8042\_keyboard\_initialization\_routine callback function (ntdd8042.h),” tech. rep., Microsoft, 04 2018.
-

- [1151] Microsoft, “Pi8042\_synch\_read\_port callback function (ntdd8042.h),” tech. rep., Microsoft, 04 2018.
  - [1152] Microsoft, “Pi8042\_synch\_write\_port callback function (ntdd8042.h),” tech. rep., Microsoft, 04 2018.
  - [1153] Microsoft, “Pi8042\_keyboard\_isr callback function (ntdd8042.h),” tech. rep., Microsoft, 04 2018.
  - [1154] Microsoft, “Kbfilter\_isrhook routine,” tech. rep., Microsoft, 08 2016.
  - [1155] Microsoft, “Windows driver framework,” tech. rep., Microsoft, 05 2018.
  - [1156] H. Ted and C. David, “Using wdf to develop a driver,” tech. rep., Microsoft, 04 2017.
  - [1157] H. Ted, G. Eliot, and C. David, “Wdm concepts for wdf drivers,” tech. rep., Microsoft, 04 2017.
  - [1158] H. Ted, aahill, C. David, and S. Ammar, “User-mode driver framework frequently asked questions,” tech. rep., Microsoft, 04 2017.
  - [1159] S. Michael, J. Mike, and C. David, “Impersonation,” tech. rep., Microsoft, 05 2018.
  - [1160] H. Ted and C. David, “Differences between wdm and wdf,” tech. rep., Microsoft, 04 2017.
  - [1161] H. Ted and C. David, “Which drivers can be ported and where,” tech. rep., Microsoft, 04 2017.
  - [1162] Microsoft, “Kbfilter\_servicecallback routine,” tech. rep., Microsoft, 04 2016.
  - [1163] A. Piekarska, “Keylogger - simple key logger kmfd driver,” 12 2018.
  - [1164] H. Ted, S. Tim, and C. David, “System worker threads,” tech. rep., Microsoft, 06 2017.
  - [1165] R. Colin, S. Kent, T. Next, J. Mike, H. Gordon, and C. Saisang, “Exporting from a dll,” tech. rep., Microsoft, 11 2016.
  - [1166] R. Wells, “How to make a windows keylogger by yourself,” *DevOps Zone*, 06 2020.
  - [1167] Y. AYDIN, “Keylogger,” 03 2020.
  - [1168] A. Randhawa, “Keylogger,” 08 2018.
  - [1169] E. of Ra, “Windows keylogger part 1: Attack on user land,” *Eye of Ra blog*, 06 2017.
  - [1170] H. Ted and M. Duncan, “Directinput overview,” tech. rep., Microsoft, 04 2017.
  - [1171] A. Fortuna, “How a keylogger works: a simple powershell example,” *Blog Andrea Fortuna*, 06 2019.
  - [1172] D. Lukan, “Using createremotethread for dll injection on windows,” *Infosec institute*, 05 2013.
  - [1173] Microsoft, “Openprocess function (processthreadsapi.h),” tech. rep., Microsoft, 12 2018.
  - [1174] Microsoft, “Virtualallocex function (memoryapi.h),” tech. rep., Microsoft, 12 2018.
  - [1175] Microsoft, “Readprocessmemory function (memoryapi.h),” tech. rep., Microsoft, 12 2018.
  - [1176] Microsoft, “Writeprocessmemory function (memoryapi.h),” tech. rep., Microsoft, 12 2018.
  - [1177] M. Don and H. Ted, “Debug privilege,” tech. rep., Microsoft, 05 2017.
  - [1178] D. B. Galen Hunt, “Detours: Binary interception of win32 functions,” *Proceedings of the 3rd USENIX Windows NT Symposium*, pp. 135–143, 07 1999.
  - [1179] M. Baig and W. Mahmood, “A robust technique of anti key-logging using key-logging mechanism,” in *2007 Inaugural IEEE-IES Digital EcoSystems and Technologies Conference*, pp. 314 – 318, 03 2007.
  - [1180] M. Dadkhah, M. Jazi, A.-M. Ciobotaru, and E. Barati, “An introduction to undetectable keyloggers with experimental testing,” *International Journal of Computer Communications and Networks*, vol. 4, pp. 1–5, 09 2014.
-

- [1181] J. Fu, Y. Liang, C. Tan, and X. Xiong, “Detecting software keyloggers with dendritic cell algorithm,” *2010 International Conference on Communications and Mobile Computing*, vol. 1, pp. 111–115, 2010.
- [1182] M. Corregedor and S. H. Solms, “Implementing rootkits to address operating system vulnerabilities,” *2011 Information Security for South Africa*, pp. 1–8, 2011.
- [1183] A. Baliga, L. Iftode, and X. L. Chen, “Automated containment of rootkits attacks,” *Comput. Secur.*, vol. 27, pp. 323–334, 2008.
- [1184] J. Fu, H. Yang, Y. Liang, and C. Tan, “Enhancing keylogger detection performance of the dendritic cell algorithm by an enticement strategy,” *J. Comput.*, vol. 9, pp. 1347–1354, 2014.
- [1185] H. Yin, Z. Liang, and D. Song, “Hookfinder: Identifying and understanding malware hooking behaviors,” in *NDSS*, 01 2008.
- [1186] S. Z. Mohd Shaïd and M. A. Maarof, “In memory detection of windows api call hooking technique,” in *2015 International Conference on Computer, Communications, and Control Technology (I4CT)*, pp. 294–298, 2015.
- [1187] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: Capturing system-wide information flow for malware detection and analysis,” in *Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS ’07*, (New York, NY, USA), pp. 116–127, Association for Computing Machinery, 2007.
- [1188] H. Yin, P. Poosankam, S. Hanna, and D. Song, “Hookscout: Proactive binary-centric hook detection,” *DIMVA 2010*, vol. 6201, pp. 1–20, 07 2010.
- [1189] Y. Al-Hammadi and U. Aickelin, “Detecting bots based on keylogging activities,” *SSRN Electronic Journal*, 01 2008.
- [1190] M. Aslam, M. Baig, and M. Arshad, “Anti-hook shield against the software key loggers,” 01 2004.
- [1191] T. Müller, H. Spath, R. Mäckl, and F. Freiling, “Stark: Tamperproof authentication to resist keylogging,” in *Financial Cryptography*, vol. 7859, pp. 295–312, 04 2013.
- [1192] S. Ortolani, C. Giuffrida, and B. Crispo, “Bait your hook: A novel detection technique for keyloggers,” in *RAID*, vol. 6307, pp. 198–217, 09 2010.
- [1193] V. Shah, “Analysis and implementation of decipherments of keylogger keywords 53 x indian journal of applied research,” *Indian Journal of Applied Research*, vol. 5, p. 3, 01 2015.
- [1194] S. Michael, J. Mike, B. Drew, and C. David, “Windows management instrumentation,” tech. rep., Microsoft, 05 2018.
- [1195] S. Michael, B. Drew, W. Steven, K. John, and C. David, “Win32\_process class,” tech. rep., Microsoft, 07 2020.
- [1196] J. Rrushi, *Multi-range Decoy I/O Defense of Electrical Substations Against Industrial Control System Malware*, ch. Resilience of Cyber-Physical Systems, pp. 151–175. Springer, 01 2019.
- [1197] Microsoft, “keybd\_event function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [1198] djpnewton, “vmulti,” 11 2010.
- [1199] Microsoft, “Setupdienumdeviceinterfaces function,” tech. rep., Microsoft, 05 2018.
- [1200] H. Ted and C. David, “Creating the filter device object,” tech. rep., Microsoft, 04 2017.
- [1201] H. Ted and C. David, “Creating device objects in a filter driver,” tech. rep., Microsoft, 04 2017.
- [1202] Microsoft, “Iogetrequestorprocessid function (ntifs.h),” tech. rep., Microsoft, 04 2018.
-



- [1203] S. Michael, B. Drew, and C. David, “Synchronization and overlapped input and output,” tech. rep., Microsoft, 05 2018.
- [1204] Microsoft, “How can i tell if the screen saver is active?,” tech. rep., Microsoft, 08 2005.
- [1205] R. Chen, “Why does windows wait longer than my screen saver idle timeout before starting the screen saver?,” tech. rep., Microsoft, 08 2009.
- [1206] R. Chen, “How can i log users off after a period of inactivity, rather than merely locking the workstation? is there a ”logoff” screen saver?,” tech. rep., Microsoft, 07 2019.
- [1207] S. Michael and C. David, “Scrnsave.exe,” tech. rep., Microsoft, 05 2018.
- [1208] Microsoft, “Getmodulefilenameexa function (psapi.h),” tech. rep., Microsoft, 12 2018.
- [1209] KicKEr, “Module 1 : objectif keylogger,” 2002.
- [1210] P. Cunha, “Golang-windows-keylogger,” 03 2017.
- [1211] A. Bhardwaj and S. Goundar, “Keyloggers: silent cyber security weapons,” *Network Security*, vol. 2020, pp. 14–19, 02 2020.
- [1212] M. Trojahn, “Biometric authentication through a virtual keyboard for smartphones,” *International Journal of Computer Science and Information Technology*, vol. 4, pp. 1–12, 10 2012.
- [1213] M. Mehrubeoglu and V. Nguyen, “Real-time eye tracking for password authentication,” in *2018 IEEE International Conference on Consumer Electronics (ICCE)*, pp. 1–4, 2018.
- [1214] H. Mohsin and H. Bahjat, “Anti-screenshot keyboard for web-based application using cloaking,” in *Proceedings of the 8th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT’18), Vol.1* (M. S. Bouhleb and S. Rovetta, eds.), (Cham), pp. 473–478, Springer International Publishing, 2020.
- [1215] N. Adhikary, R. Shrivastava, A. Kumar, S. Verma, M. Bag, and V. Singh, “Battering keyloggers and screen recording software by fabricating passwords,” *International Journal of Computer Network and Information Security*, vol. 4, 01 2012.
- [1216] R. Chen, “How can i prevent the user from using snip and sketch to take screen shots?,” tech. rep., Microsoft, 05 2020.
- [1217] C. Bacara, V. Lefils, J. Iguchi-Cartigny, G. Grimaud, and J.-P. Wary, “Virtual keyboard logging countermeasures using human vision properties,” in *Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on*, (New York, NY, United States), 08 2015.
- [1218] S. Gong, J. Lin, and Y. Sun, “Design and implementation of anti-screenshot virtual keyboard applied in online banking,” in *Proceedings of the 2010 International Conference on E-Business and E-Government, ICEE ’10*, (USA), pp. 1320–1322, IEEE Computer Society, 2010.
- [1219] Microsoft, “Wm\_paint message,” tech. rep., Microsoft, 05 2018.
- [1220] Microsoft, “Updatewindow function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [1221] D. Nyang, A. Mohaisen, and J. Kang, “Keylogging-resistant visual authentication protocols,” *IEEE Transactions on Mobile Computing*, vol. 13, no. 11, pp. 2566–2579, 2014.
- [1222] Q. Yan, J. Han, Y. Li, J. Zhou, and R. Deng, “Designing leakage-resilient password entry on touchscreen mobile devices,” in *ASIA CCS ’13*, pp. 37–48, 05 2013.
- [1223] Y. Huang, Z. Huang, H. Zhao, and X. Lai, “A new one-time password method,” *IERI Procedia*, vol. 4, pp. 32–37, 12 2013.
-



- [1224] Shally, G. Aujla, and S. Aujla, “A review of one time password mobile verification,” *International Journal of Computer Science Engineering and Information Technology Research (IJCSEITR)*, vol. 4, pp. 113–118, 06 2014.
- [1225] M. Mannan and P. V. Oorschot, “Leveraging personal devices for stronger password authentication from untrusted computers,” *J. Comput. Secur.*, vol. 19, pp. 703–750, 2011.
- [1226] A. A. Cain, S. Werner, and J. D. Still, “Graphical authentication resistance to over-the-shoulder-attacks,” in *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA ’17, (New York, NY, USA), pp. 2416–2422, Association for Computing Machinery, 2017.
- [1227] J. Still and A. Cain, *Over-the-Shoulder Attack Resistant Graphical Authentication Schemes Impact on Working Memory*, pp. 79–86. Washington D.C., USA: Proceedings of the AHFE 2019 International Conference on Human Factors in Cybersecurity, 07 2019.
- [1228] A. T. M., A. Omer S. A., and E. Abeer E. W., “Random multiple layouts: Keylogger prevention technique,” *2016 Conference of Basic Sciences and Engineering Studies (SGCAC)*, pp. 1–5, 2016.
- [1229] T. N. Witte, “Mouse underlaying: Global key and mouse listener based on an almost invisible window with local listeners and sophisticated focus,” *EAI Endorsed Transactions on Security and Safety*, vol. 5, 10 2018.
- [1230] G. C. Grammatikos, “How to set the keyboard layout through group policy (gpo),” *Microsoft TechNet Articles*, 05 2020.
- [1231] M. E. Dalkiliç and G. Dalkiliç, “On the cryptographic patterns and frequencies in turkish language,” in *Proceedings of the Second International Conference on Advances in Information Systems*, ADVIS ’02, (Berlin, Heidelberg), pp. 144–153, Springer-Verlag, 10 2002.
- [1232] I. Suzuki, Y. Mikami, A. Ohsato, and Y. Chubachi, “A language and character set determination method based on n-gram statistics,” *ACM Trans. Asian Lang. Inf. Process.*, vol. 1, pp. 269–278, 09 2002.
- [1233] S. P. Goring, J. R. Rabaiotti, and A. J. Jones, “Anti-keylogging measures for secure internet login: An example of the law of unintended consequences,” *Computers & Security*, vol. 26, pp. 421–426, 2007.
- [1234] Y. Cheng, X. Ding, and R. H. Deng, “Driverguard: Virtualization-based fine-grained protection on i/o flows,” *ACM Trans. Inf. Syst. Secur.*, vol. 16, 09 2013.
- [1235] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 164–177, 10 2003.
- [1236] Intel, *Intel® Virtualization Technology for Directed I/O - Architecture Specification*, 06 2019.
- [1237] Y. Jang, S. Lee, and T. Kim, “Breaking kernel address space layout randomization with intel tsx,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’16, (New York, NY, USA), pp. 380–392, Association for Computing Machinery, 2016.
- [1238] Microsoft, “Reduce attack surfaces with attack surface reduction rules,” tech. rep., Microsoft, 10 2020.
- [1239] Microsoft, “Overview of attack surface reduction,” tech. rep., Microsoft, 10 2020.
- [1240] Microsoft, “Virtualization-based security (vbs),” tech. rep., Microsoft, 09 2017.
- [1241] A. Marty Hernandez, V. Denise, R. Thomas, jreeds, S. Daniel, and R. Angela, “Microsoft defender application guard overview,” tech. rep., Microsoft, 09 2020.
- [1242] A. Marty Hernandez, V. Denise, jreeds, and S. Zankharia, “Application guard testing scenarios,” tech. rep., Microsoft, 09 2020.
- [1243] G. Barry, H. Ted, S. Florian, G. Eliot, M. Don, and C. David, “Hypervisor-protected code integrity (hvci),” tech. rep., Microsoft, 05 2020.
-

- [1244] Microsoft, “Enable virtualization-based protection of code integrity,” tech. rep., Microsoft, 04 2019.
- [1245] Microsoft, “Manage windows defender credential guard,” tech. rep., Microsoft, 10 2020.
- [1246] Microsoft, “Hardware-based isolation in windows 10,” tech. rep., Microsoft, 07 2020.
- [1247] S. Michael, B. Karl, and S. Artur, “Isolated user mode (ium) processes,” tech. rep., Microsoft, 05 2018.
- [1248] K. David, Z. Adam, and G. Rafael, “Introducing windows defender system guard runtime attestation,” *Microsoft Security Blog*, 04 2018.
- [1249] A. Chevalier, “Virtualization based security - part 1: The boot process,” *Amossys Security Blog*, 02 2017.
- [1250] A. Chevalier, “Virtualization based security - part 2: kernel communications,” *Amossys Security Blog*, 02 2017.
- [1251] Microsoft, “Introduction to hyper-v on windows 10,” tech. rep., Microsoft, 06 2018.
- [1252] Microsoft, “Create a virtual machine with hyper-v,” tech. rep., Microsoft, 07 2018.
- [1253] Microsoft, “Architecture hyper-v,” tech. rep., Microsoft, 04 2020.
- [1254] Microsoft, “Hyper-v architecture,” tech. rep., Microsoft, 10 2017.
- [1255] Microsoft, “Secure boot,” tech. rep., Microsoft, 07 2019.
- [1256] Biswapriyo, Veovis, and M. Dave, “Why can’t virtualbox or vmware run with hyper-v enabled on windows 10,” 05 2017.
- [1257] J. Hannah, Justin, and S. John, “Windows hypervisor platform api definitions,” tech. rep., Microsoft, 05 2019.
- [1258] J. Hannah, C. Sarah, A. Sandra, S. Nick, and W. Craig, “Windows hypervisor platform,” tech. rep., Microsoft, 12 2017.
- [1259] A. Ionescu, “Simpleator,” 12 2018.
- [1260] L. Craig and al., “What is the windows subsystem for linux?,” tech. rep., Microsoft, 07 2020.
- [1261] C. Loewen, “Announcing wsl 2,” *Microsoft Windows Command Line Blog*, 06 2019.
- [1262] Malekal, “Les anti-keylogger pour se protéger des keyloggers,” 03 2018.
- [1263] R. Chen, “How to set focus in a dialog box,” tech. rep., Microsoft, 08 2004.
- [1264] R. Chen, “Pressing a registered hotkey gives you the foreground activation love,” tech. rep., Microsoft, 02 2009.
- [1265] S. Michael, J. Mike, S. McLean, B. Drew, and C. David, “Notifications and the notification area,” tech. rep., Microsoft, 05 2018.
- [1266] H. Ted, S. Tim, and C. David, “Nt device names,” tech. rep., Microsoft, 06 2017.
- [1267] R. Chen, “How can i make sure my program is launched only from my helper program and no other parent?,” tech. rep., Microsoft, 02 2014.
- [1268] R. Chen, “A single-instance program is its own denial of service,” tech. rep., Microsoft, 06 2006.
- [1269] R. Chen, “Appinit\_dlls should be renamed deadlock\_or\_crash\_randomly\_dlls,” tech. rep., Microsoft, 12 2007.
- [1270] R. Chen, “I bet somebody got a really nice bonus for that feature,” tech. rep., Microsoft, 11 2006.
- [1271] R. Chen, “Le chatelier’s principle in action: Administrative overrides,” tech. rep., Microsoft, 04 2014.
-

- 
- [1272] S. Michael, B. Drew, J. Mike, and C. David, “Dynamic-link library best practices,” tech. rep., Microsoft, 05 2018.
- [1273] hfiref0x, “Trend micro’s rootkit buster - blast from the past? nope.,” 04 2021.
- [1274] Microsoft, “Deviceiocontrol function (ioapiset.h),” tech. rep., Microsoft, 12 2018.
- [1275] Microsoft, “Windows 7 compatibility - user interface privilege isolation (uipi),” 2009.
- [1276] Microsoft, “Changewindowmessagefilter function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [1277] Microsoft, “Setwineventhook function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [1278] S. Michael and B. Drew, “Event constants,” tech. rep., Microsoft, 05 2018.
- [1279] H. Ted, Matt, and H. Lori Whippler, “Filter manager concepts,” tech. rep., Microsoft, 02 2020.
- [1280] R. Chen, “The arms race between programs and users,” tech. rep., Microsoft, 02 2004.
- [1281] K. Jamwal and L. Sharma, “Clickjacking attack: Hijacking user’s click,” *International Journal of Advanced Networking Applications*, vol. 10, pp. 3735–3740, 01 2018.
- [1282] U. U. Rehman, W. Khan, N. A. Saqib, and M. Kaleem, “On detection and prevention of clickjacking attack for osns,” in *2013 11th International Conference on Frontiers of Information Technology*, pp. 160–165, 12 2013.
- [1283] L. Wu, B. Brandt, X. Du, and B. Ji, “Analysis of clickjacking attacks and an effective defense scheme for android devices,” in *2016 IEEE Conference on Communications and Network Security (CNS)*, pp. 55–63, 10 2016.
- [1284] GuardedID, “Guardedid whitepaper,” Corporate Headquarters StrikeForce Technologies, Inc. 1090 King Georges Post Road 108 Edison, NJ 08837, USA, 01 2017.
- [1285] H. Ted, S. Tim, and C. David, “Filtering registry calls,” tech. rep., Microsoft, 06 2017.
- [1286] Microsoft, “Cmregistercallbackex function (wdm.h),” tech. rep., Microsoft, 04 2018.
- [1287] Microsoft, “Psetloadimagenotifyroutineex function (ntddk.h),” tech. rep., Microsoft, 04 2018.
- [1288] Microsoft, “Pload\_image\_notify\_routine callback function (ntddk.h),” tech. rep., Microsoft, 04 2018.
- [1289] M. Jang, H. Kim, and Y. Yun, “Detection of dll inserted by windows malicious code,” *Convergence Information Technology, International Conference*, vol. 0, pp. 1059–1064, 11 2007.
- [1290] R. Chen, “How can i write an unkillable program, redux,” tech. rep., Microsoft, 05 2016.
- [1291] R. Chen, “If one program blocks shutdown, then all programs block shutdown,” tech. rep., Microsoft, 04 2020.
- [1292] R. Chen, “Solving the problem rather than answering the question: Why does somebody want to write an unkillable process?,” tech. rep., Microsoft, 06 2013.
- [1293] D. M. Chess and S. R. White, “An undetectable computer virus,” in *In Proceedings of Virus Bulletin Conference*, 2000.
- [1294] F. Cohen, “A note on the role of deception in information protection,” *Computers & Security*, vol. 17, no. 6, pp. 483 – 506, 1998.
- [1295] F. Cohen, “The deception toolkit,” 1998.
- [1296] R. Chen, “Documentation creates contract, which is why you need to be very careful what you document,” tech. rep., Microsoft, 05 2015.
- [1297] R. Chen, “Hardware backwards compatibility,” tech. rep., Microsoft, 08 2003.
-

- [1298] R. Chen, “Why is there no programmatic access to the start menu pin list?,” tech. rep., Microsoft, 09 2003.
  - [1299] R. Chen, “When programs grovel into undocumented structures...,” tech. rep., Microsoft, 12 2003.
  - [1300] R. Chen, “Why not just block the apps that rely on undocumented behavior?,” tech. rep., Microsoft, 12 2003.
  - [1301] Microsoft, “Mitigate threats by using windows 10 security features,” tech. rep., Microsoft, 10 2007.
  - [1302] S. McLean, S. Kent, C. David, ishimada, B. Drew, J. Mike, and S. Michael, “The taskbar,” tech. rep., Microsoft, 05 2018.
  - [1303] R. Chen, “Microspeak: Walls and ladders,” tech. rep., Microsoft, 01 2012.
  - [1304] A. Paul and D. Baptiste, “How to secure the keyboard chain,” in *Defcon 23*, (Las Vegas, USA), Defcon groups, 09 2015.
  - [1305] B. Golden, “Keyboard input wdf filter driver (kbfldr),” tech. rep., Microsoft, 09 2019.
  - [1306] Microsoft, “Driver\_initialize callback function (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [1307] H. Ted, C. David, C. Saisang, and N. Vaishnav, “Driverentry for wdf drivers routine,” tech. rep., Microsoft, 10 2018.
  - [1308] H. Ted, S. Tim, and C. David, “Writing a driverentry routine,” tech. rep., Microsoft, 06 2017.
  - [1309] H. Ted, S. Tim, M. Duncan, and C. David, “Driverentry’s required responsibilities,” tech. rep., Microsoft, 06 2017.
  - [1310] H. Ted, C. David, S. Tim, and G. Eliot, “Introduction to standard driver routines,” tech. rep., Microsoft, 06 2017.
  - [1311] H. Ted, S. Tim, and C. David, “Writing an unload routine,” tech. rep., Microsoft, 06 2017.
  - [1312] H. Ted, S. Tim, and C. David, “Non-pnp driver’s unload routine,” tech. rep., Microsoft, 06 2017.
  - [1313] H. Ted, S. Tim, and C. David, “Pnp driver’s unload routine,” tech. rep., Microsoft, 06 2017.
  - [1314] H. Ted, S. Tim, and C. David, “Adddevice routines in function or filter drivers,” tech. rep., Microsoft, 06 2017.
  - [1315] H. Ted and S. Tim, “Dispatch routine functionality,” tech. rep., Microsoft, 10 2018.
  - [1316] H. Ted, M. Duncan, S. Tim, and C. David, “Required dispatch routines,” tech. rep., Microsoft, 12 2018.
  - [1317] H. Ted, S. Tim, M. Duncan, and C. David, “Optional dispatch routines,” tech. rep., Microsoft, 06 2017.
  - [1318] Microsoft, “Driver\_dispatch callback function (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [1319] Microsoft, “Device\_object structure (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [1320] H. Ted, C. David, K. Heesung, S. Tim, and C. Jon, “Windows kernel macros,” tech. rep., Microsoft, 10 2018.
  - [1321] Microsoft, “Ioalldrver macro (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [1322] H. Ted, S. Tim, and C. David, “Passing irps down the driver stack,” tech. rep., Microsoft, 06 2017.
  - [1323] H. Ted, Matt, and C. David, “Example: Passing the irp down without setting a completion routine,” tech. rep., Microsoft, 04 2017.
  - [1324] Microsoft, “Ioctl\_keyboard\_set\_indicators ioctl (ntddkbd.h),” tech. rep., Microsoft, 12 2018.
  - [1325] J. Cutrer, “Windows 10: Numlock on startup registry key,” 11 2017.
-

- [1326] Microsoft, “IoctlInternalLi8042\_keyboard\_start\_information ioctl (ntdd8042.h),” tech. rep., Microsoft, 04 2018.
- [1327] Microsoft, “InternalLi8042\_start\_information structure (ntdd8042.h),” tech. rep., Microsoft, 04 2018.
- [1328] H. Ted, “Os driver installation,” tech. rep., Microsoft, 04 2017.
- [1329] O. S. R. Inc., “Fun with filters - win2k/wdm device filter drivers,” *The NT Insider*, vol. 7, 02 2000.
- [1330] S. Michael, B. Drew, and C. David, “Registry value types,” tech. rep., Microsoft, 05 2018.
- [1331] OSR, “Devicetree,” 03 2011.
- [1332] D. Coppersmith, H. Krawczyk, and Y. Mansour, “The shrinking generator,” in *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO ’93, (Berlin, Heidelberg), pp. 22–39, Springer-Verlag, 1994.
- [1333] P. Caballero-Gil, A. Fuster-Sabater, and M. Eugenia Pazo-Robles, “New attack strategy for the shrinking generator.,” *Journal of Research and Practice in Information Technology*, vol. 41, pp. 171–180, 01 2008.
- [1334] J. D. Golic, “Towards fast correlation attacks on irregularly clocked shift registers,” in *EUROCRYPT*, 1995.
- [1335] D. Brumley and D. Boneh, “Remote timing attacks are practical,” in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM’03, (USA), p. 1, USENIX Association, 2003.
- [1336] P. C. Kocher, “Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems,” in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO ’96, (Berlin, Heidelberg), pp. 104–113, Springer-Verlag, 1996.
- [1337] D. Bernstein, T. Chou, and P. Schwabe, “Mcbits: Fast constant-time code-based cryptography,” in *Proceedings of the 15th international conference on Cryptographic Hardware and Embedded Systems*, vol. 8086, pp. 250–272, 08 2013.
- [1338] T. Pornin, “Why constant-time crypto?,” 12 2016.
- [1339] O. Reparaz, J. Balasch, and I. Verbauwhede, “Dude, is my code constant time?,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE ’17, (Leuven, BEL), pp. 1701–1706, European Design and Automation Association, 2017.
- [1340] S. Michael, B. Drew, and C. David, “Device input and output control,” tech. rep., Microsoft, 05 2018.
- [1341] H. Ted, S. Tim, M. Duncan, and C. David, “Introduction to i/o control codes,” tech. rep., Microsoft, 06 2017.
- [1342] H. Ted, S. Tim, and C. David, “Security issues for i/o control codes,” tech. rep., Microsoft, 06 2017.
- [1343] T. Opferman, “Driver development part 2: Introduction to implementing ioctls,” *Code Project*, 03 2005.
- [1344] E. Asselin, “Userland/kernel communication - deviceiocontrol method,” 2010.
- [1345] D. Marshall, H. Ted, and C. David, “Driver security guidance,” tech. rep., Microsoft, 02 2017.
- [1346] D. Marshall, H. Ted, and C. David, “Windows security model for driver developers,” tech. rep., Microsoft, 02 2018.
- [1347] D. Marshall, G. Eliot, H. Ted, C. David, and T. Next, “Driver security checklist,” tech. rep., Microsoft, 03 2020.
- [1348] D. Marshall, H. Ted, and C. David, “Threat modeling for drivers,” tech. rep., Microsoft, 06 2018.
- [1349] H. Ted, C. David, S. Tim, and M. Don, “Named device objects,” tech. rep., Microsoft, 09 2017.
- [1350] H. Ted, S. Tim, and C. David, “Controlling device namespace access,” tech. rep., Microsoft, 06 2017.
-

- [1351] H. Ted, S. Tim, and C. David, “Specifying device characteristics,” tech. rep., Microsoft, 06 2017.
  - [1352] O. Staff, “Making device objects accessible ... and safe,” *OSR Blog, NT-Insider*, 08 2020.
  - [1353] Microsoft, “Readfile function (fileapi.h),” tech. rep., Microsoft, 12 2018.
  - [1354] Microsoft, “Writefile function (fileapi.h),” tech. rep., Microsoft, 12 2018.
  - [1355] S. McLean, C. David, B. Drew, J. Mike, and S. Michael, “How to sign an app package using signtool,” tech. rep., Microsoft, 05 2018.
  - [1356] Microsoft, “Signtool,” tech. rep., Microsoft, 10 2020.
  - [1357] H. Ted and C. David, “Digital signatures,” tech. rep., Microsoft, 04 2017.
  - [1358] Microsoft, “Setthreadpriority function (processthreadsapi.h),” tech. rep., Microsoft, 12 2018.
  - [1359] S. Michael, J. Mike, turingcompl33t, and C. David, “Cng features,” tech. rep., Microsoft, 05 2018.
  - [1360] S. Michael, B. Drew, and J. Mike, “Cng algorithm identifiers,” tech. rep., Microsoft, 05 2018.
  - [1361] M. Dworkin, “Recommendation for block cipher modes of operation: Methods and techniques,” *Computer Security Resource Center - NIST*, 12 2001.
  - [1362] M. John P, “Intel digital random number generator (drng) software implementation guide,” *Intel Development Topics & Technologies*, 05 2014.
  - [1363] S. Michael, B. Drew, turingcompl33t, and C. David, “Cprocess security and access rights,” tech. rep., Microsoft, 05 2018.
  - [1364] A. Ionescu, “Why protected processes are a bad idea,” *Alex Ionescu’s Blog*, 04 2007.
  - [1365] Microsoft, “Appendix c: The windows integrity mechanism and windows kernel mode code integrity,” tech. rep., Microsoft, 07 2007.
  - [1366] Kaspersky, “About protected process light (ppl) technology for windows,” 07 2018.
  - [1367] A. Ionescu, “The evolution of protected processes part 1: Pass-the-hash mitigations in windows 8.1,” *Alex Ionescu’s Blog*, 11 2013.
  - [1368] A. Ionescu, “The evolution of protected processes part 2: Exploit/jailbreak mitigations, unkillable processes and protected services,” *Alex Ionescu’s Blog*, 12 2013.
  - [1369] A. Ionescu, “Protected processes part 3: Windows pki internals (signing levels, scenarios, root keys, ekus & runtime signers),” *Alex Ionescu’s Blog*, 12 2013.
  - [1370] Y. Shafir, “Critical, protected, dut processes in windows 10,” *Windows Internals Blog*, 08 2020.
  - [1371] Microsoft, “Obregistercallbacks function (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [1372] Microsoft, “Ob\_callback\_registration structure (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [1373] H. Ted, Matt, H. Lori Whippler, K. Heesung, and C. David, “Load order groups and altitudes for filter drivers,” tech. rep., Microsoft, 04 2017.
  - [1374] H. Ted, H. Lori Whippler, Matt, P. Joseph, S. Nick, R. Jesse, G. Eliot, and aahill, “Allocated filter altitudes,” tech. rep., Microsoft, 03 2020.
  - [1375] H. Ted, H. Lori Whippler, and Matt, “Filter altitude request,” tech. rep., Microsoft, 08 2020.
  - [1376] Microsoft, “Pob\_pre\_operation\_callback callback function (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [1377] Microsoft, “Ob\_pre\_operation\_information structure (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [1378] Microsoft, “Ob\_pre\_create\_handle\_information structure (wdm.h),” tech. rep., Microsoft, 04 2018.
-



- 
- [1379] Microsoft, “Pob\_post\_operation\_callback callback function (wdm.h),” tech. rep., Microsoft, 04 2018.
- [1380] Microsoft, “Ob\_post\_operation\_information structure (wdm.h),” tech. rep., Microsoft, 04 2018.
- [1381] Microsoft, “Duplicatehandle function (handleapi.h),” tech. rep., Microsoft, 12 2018.
- [1382] Microsoft, “Zwduplicateobject function (ntifs.h),” tech. rep., Microsoft, 04 2018.
- [1383] H. Ted, C. David, jasonpepperly, and S. Tim, “Windows kernel-mode process and thread manager,” tech. rep., Microsoft, 10 2018.
- [1384] Microsoft, “Pssetloadimagenotifyroutine function (ntddk.h),” tech. rep., Microsoft, 04 2018.
- [1385] Microsoft, “Pload\_image\_notify\_routine callback function (ntddk.h),” tech. rep., Microsoft, 04 2018.
- [1386] A. Stewar, “Dll side-loading: A thorn in the side of the anti-virus industry,” *FireEye SECURITY REIMAGINED*, 2014.
- [1387] T. Kwon and Z. Su, “Automatic detection of unsafe component loadings,” in *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010* (P. Tonella and A. Orso, eds.), pp. 107–118, ACM, 2010.
- [1388] A. Stewart, “Dll side-loading: Another blind-spot for anti-virus,” 04 2014.
- [1389] H. Ted, S. Tim, and C. David, “Managing kernel objects,” tech. rep., Microsoft, 06 2017.
- [1390] H. Ted, S. Tim, and C. David, “Defining a callback object,” tech. rep., Microsoft, 06 2017.
- [1391] Microsoft, “Excreatecallback function (wdm.h),” tech. rep., Microsoft, 04 2018.
- [1392] S. Michael and L. Xiaoyin, “Key storage and retrieval,” tech. rep., Microsoft, 05 2018.
- [1393] G. Andres Mariano, H. Justin, S. Daniel, and B. Andrea, “Trusted platform module,” tech. rep., Microsoft, 09 2018.
- [1394] A. Marty Hernandez, G. Ed, P. Liza, S. Nick, H. Justin, M. Duncan, S. Daniel, G. Andres Mariano, and B. Andrea, “Trusted platform module technology overview,” tech. rep., Microsoft, 11 2018.
- [1395] S. Nick, H. Justin, M. Duncan, S. Daniel, G. Andres Mariano, and B. Andrea, “How windows 10 uses the trusted platform module,” tech. rep., Microsoft, 10 2017.
- [1396] F. O. for Information Security, *Work Package 5: Trusted Platform Module and Unified Extensible Firmware Interface “Secure Boot”*, ch. 5. Post Box 20 03 63D - 53133 Bonn: Federal Office for Information Security, 2019.
- [1397] A. Marty Hernandez, V. d. V. Sander, S. Nick, H. Justin, M. Duncan, S. Daniel, G. Andres Mariano, and B. Andrea, “Tpm fundamentals,” tech. rep., Microsoft, 08 2017.
- [1398] W. Johannes and D. Kurt, “A hijacker’s guide to communication interfaces of the trusted platform module,” *Computers & Mathematics with Applications*, vol. 65, no. 5, pp. 748–761, 2013.
- [1399] S. Han, W. Shin, J.-H. Park, and H. Kim, “A bad dream: Subverting trusted platform module while you are sleeping,” in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 1229–1246, USENIX Association, 08 2018.
- [1400] D. Moghimi, B. Sunar, T. Eisenbarth, and N. Heninger, “TPM-FAIL: TPM meets Timing and Lattice Attacks,” in *29th USENIX Security Symposium (USENIX Security 20)*, (Boston, MA), USENIX Association, Aug. 2020.
- [1401] Microsoft, “Vulnerability in tpm could allow security feature bypass,” 10 2017.
- [1402] D. Andzakovic, “Extracting bitlocker keys from a tpm,” *Pulse Security*, 03 2019.
-



- [1403] D. Weston, “Meet the microsoft pluton processor - the security chip designed for the future of windows pcs,” *Microsoft Security*, 11 2020.
  - [1404] L. H. Newman, “Microsoft is making a secure pc chip with intel and amd’s help,” *Wired*, 11 2020.
  - [1405] D. Han, “Introduction to page files,” tech. rep., Microsoft, 08 2020.
  - [1406] S. McLean, C. David, B. Drew, and S. Michael, “Working with pages,” tech. rep., Microsoft, 05 2018.
  - [1407] D. Han, “Virtuallock function (memoryapi.h),” tech. rep., Microsoft, 12 2018.
  - [1408] Microsoft, “Virtualunlock function (memoryapi.h),” tech. rep., Microsoft, 12 2018.
  - [1409] S. McLean, C. David, B. Drew, and S. Michael, “Memory pools,” tech. rep., Microsoft, 05 2018.
  - [1410] T. Hudek, D. Coulter, N. Schonning, and T. Sherer, “Distinguishing fast startup from wake-from-hibernation,” tech. rep., Microsoft, 10 2018.
  - [1411] H. Ted, C. David, and S. Tim, “System sleeping states,” tech. rep., Microsoft, 07 2020.
  - [1412] H. Ted and S. Tim, “System working state s0,” tech. rep., Microsoft, 06 2017.
  - [1413] H. Ted and S. Tim, “System shutdown state s5,” tech. rep., Microsoft, 06 2017.
  - [1414] Microsoft, “System power states,” tech. rep., Microsoft, 05 2018.
  - [1415] C. David, Chcurlet, and S. Katie, “Modern standby vs s3,” tech. rep., Microsoft, 04 2020.
  - [1416] S. Katie, Chcurlet, G. Eliot, H. Elena, P. Sarah, and B. Joshua, “What is modern standby,” tech. rep., Microsoft, 04 2020.
  - [1417] W. D. D. D. Team, P. Sarah, and B. Joshua, “Device-specific power management for modern standby,” tech. rep., Microsoft, 05 2017.
  - [1418] Chcurlet, C. David, S. Katie, G. Eliot, T. Hudek, H. Elena, Jebourli, and B. Joshua, “Wake sources,” tech. rep., Microsoft, 12 2020.
  - [1419] Microsoft, “System wake-up events,” tech. rep., Microsoft, 05 2018.
  - [1420] Chcurlet and H. Elena, “Modern standby states,” tech. rep., Microsoft, 12 2019.
  - [1421] J. Metz, “Windows hibernation file format,” 11 2015.
  - [1422] Microsoft, “Bitlocker overview,” tech. rep., Microsoft, 08 2016.
  - [1423] Microsoft, “Bitlocker,” tech. rep., Microsoft, 01 2018.
  - [1424] B. Aurélien, “Bitlocker,” in *SSTIC 2011*, 06 2011.
  - [1425] Microsoft, “Poregisterpowersettingcallback function (ntifs.h),” tech. rep., Microsoft, 04 2018.
  - [1426] Microsoft, “Rtlsecurezeromemory function (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [1427] R. Chen, “What’s the point of securezeromemory?,” tech. rep., Microsoft, 05 2013.
  - [1428] S. McLean, S. Kent, C. David, J. Mike, , and S. Michael, “Registering for power events,” tech. rep., Microsoft, 05 2018.
  - [1429] Microsoft, “Registerpowersettingnotification function (winuser.h),” tech. rep., Microsoft, 12 2018.
  - [1430] S. McLean, S. Kent, B. Drew, and S. Michael, “Wm\_powerbroadcast message,” tech. rep., Microsoft, 05 2018.
  - [1431] S. McLean, S. Kent, and S. Michael, “System power management events,” tech. rep., Microsoft, 05 2018.
-

- 
- [1432] S. McLean, S. Kent, C. David, B. Drew, J. Mike, , and S. Michael, “Power setting guids,” tech. rep., Microsoft, 05 2018.
- [1433] S. McLean, S. Kent, B. Drew, and S. Michael, “Pbt\_apmsuspend event,” tech. rep., Microsoft, 05 2018.
- [1434] S. McLean, S. Kent, and S. Michael, “System sleep criteria,” tech. rep., Microsoft, 05 2018.
- [1435] Microsoft, “Securezeromemory function,” tech. rep., Microsoft, 02 2018.
- [1436] S. McLean, S. Kent, B. Drew, and S. Michael, “Pbt\_apmresumesuspend event,” tech. rep., Microsoft, 05 2018.
- [1437] Microsoft, “Getlasterror function (errhandlingapi.h),” tech. rep., Microsoft, 12 2018.
- [1438] R. Chen, “How do i create a topmost window that is never covered by other topmost windows?,” tech. rep., Microsoft, 03 2011.
- [1439] R. Chen, “What if two programs did this?,” tech. rep., Microsoft, 06 2005.
- [1440] Microsoft, “Wineventproc callback function (winuser.h),” tech. rep., Microsoft, 12 2018.
- [1441] S. Bratus, N. D’Cunha, E. Sparks, and S. W. Smith, “Toctou, traps, and trusted computing,” in *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications*, vol. 4968 of *Trust ’08*, (Berlin, Heidelberg), pp. 14–32, Springer-Verlag, 03 2008.
- [1442] R. Chen, “It rather involved being on the other side of this airtight hatchway: Elevation from administrator to system,” tech. rep., Microsoft, 09 2015.
- [1443] R. Chen, “Solutions that don’t actually solve anything,” tech. rep., Microsoft, 05 2006.
- [1444] R. Chen, “Beware the image file execution options key,” tech. rep., Microsoft, 12 2005.
- [1445] D. Simpson, “Boot sequence flowchart,” tech. rep., Microsoft, 11 2018.
- [1446] W. Claus and S. Arne, “Windows 7: The boot process explained,” *Microsoft TechNet Articles*, 05 2012.
- [1447] M. Ravirala, “Windows boot environment,” in *UEFI Plugfest September 2007*, Microsoft, 09 2007.
- [1448] M. Don, C. David, and G. Eliot, “Overview of boot options in windows,” tech. rep., Microsoft, 04 2019.
- [1449] J. Gerend, “diskpart,” tech. rep., Microsoft, 10 2020.
- [1450] Microsoft, “mountvol,” tech. rep., Microsoft, 10 2017.
- [1451] R. Chen, “Why is a registry file called a ”hive“?,” tech. rep., Microsoft, 08 2003.
- [1452] G. Barry and Matt, “Boot and uefi,” tech. rep., Microsoft, 04 2017.
- [1453] M. Kinney, “5.3.13 setvirtualaddressmap(),” tech. rep., Tianocore, 04 2018.
- [1454] P. Rudolph, “Intel trusted execution technology,” 07 2019.
- [1455] Intel, “Intel(r) trusted execution technology (intel(r) txt), software development guide, measured launch environment developer’s guide,” tech. rep., Intel Corporation, 01 2021.
- [1456] H. Ted and C. David, “Specifying driver load order,” tech. rep., Microsoft, 04 2017.
- [1457] H. Ted, C. David, and Selerner, “Inf addservice directive,” tech. rep., Microsoft, 04 2017.
- [1458] H. Lori Whippler, C. David, S. Mimi, and B. Nathan, “Setting the start type value,” tech. rep., Microsoft, 04 2017.
- [1459] H. Lori Whippler, C. David, and Matt, “File system filter load order,” tech. rep., Microsoft, 08 2020.
-

- [1460] H. Ted, C. David, N. Jason K., G. Eliot, and S. Kenichi, “Hklm\system\currentcontrolset\enum registry tree,” tech. rep., Microsoft, 04 2017.
  - [1461] Microsoft, “Ex\_callback\_function callback function (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [1462] Microsoft, “Reg\_notify\_class enumeration (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [1463] Microsoft, “List\_entry structure (ntdef.h),” tech. rep., Microsoft, 12 2018.
  - [1464] H. Ted, C. David, and S. Tim, “Singly and doubly linked lists,” tech. rep., Microsoft, 06 2017.
  - [1465] H. Ted and S. Tim, “No-execute (nx) nonpaged pool,” tech. rep., Microsoft, 10 2018.
  - [1466] H. Ted, D. Marshall, G. Eliot, and C. David, “Evaluate hvci driver compatibility,” tech. rep., Microsoft, 05 2020.
  - [1467] W. D. D. D. Team, H. Ted, G. Eliot, C. David, W. Maira, M. Don, and P. Sarah, “Hypervisor-protected code integrity (hvci),” tech. rep., Microsoft, 05 2020.
  - [1468] H. Ted, C. David, G. Eliot, and B. Nick, “Overview of early launch antimalware,” tech. rep., Microsoft, 04 2017.
  - [1469] R. Quinn, S. McLean, K. John, C. David, B. Drew, and S. Michael, “Early launch antimalware,” tech. rep., Microsoft, 05 2018.
  - [1470] W. Beth, A. Marty Hernandez, H. Dani, G. Andres Mariano, Eavena, S. Adamn, B. Andrean, M. Duncann, and S. Daniel, “Microsoft virus initiative,” tech. rep., Microsoft, 11 2020.
  - [1471] H. Ted, C. David, S. Andrei-George, and B. Nick, “Elam prerequisites,” tech. rep., Microsoft, 04 2017.
  - [1472] H. Ted, C. David, and K. Meera, “Whql release signature,” tech. rep., Microsoft, 04 2017.
  - [1473] H. Ted, C. David, and Aahill, “Elam driver submission process,” tech. rep., Microsoft, 04 2017.
  - [1474] B. Golden, “Early launch anti-malware driver,” tech. rep., Microsoft, 03 2015.
  - [1475] H. Ted, C. David, and D. Holan, “Inf signatureattributes section,” tech. rep., Microsoft, 04 2017.
  - [1476] Microsoft, “Bdcb\_classification enumeration (ntddk.h),” tech. rep., Microsoft, 04 2018.
  - [1477] H. Ted, C. David, Aahill, S. Kenichi, and B. Nick, “Elam driver requirements,” tech. rep., Microsoft, 04 2017.
  - [1478] Microsoft, “Ioregisterbootdrivercallback function (ntddk.h),” tech. rep., Microsoft, 04 2018.
  - [1479] Microsoft, “Boot\_driver\_callback\_function callback function (ntddk.h),” tech. rep., Microsoft, 10 2018.
  - [1480] Microsoft, “Bdcb\_image\_information structure (ntddk.h),” tech. rep., Microsoft, 04 2018.
  - [1481] Microsoft, “Bdcb\_callback\_type enumeration (ntddk.h),” tech. rep., Microsoft, 04 2018.
  - [1482] Microsoft, “Iounregisterbootdrivercallback function (ntddk.h),” tech. rep., Microsoft, 04 2018.
  - [1483] M. Satran, “Driver\_unload callback function (wdm.h),” tech. rep., Microsoft, 04 2018.
  - [1484] M. Satran, “Cng cryptographic primitive functions,” tech. rep., Microsoft, 05 2018.
  - [1485] R. Chen, “I wrote the original blue screen of death, sort of,” tech. rep., Microsoft, 09 2014.
  - [1486] G. Eliot, C. David, and B. Nathan, “Interpreting a bug check code,” tech. rep., Microsoft, 04 2017.
  - [1487] D.-G. Brendan, “Parsing windows minidumps,” *Push the Red Button - Moyix blog*, 05 2008.
  - [1488] H. Deland and X. Simonx, “Overview of memory dump file options for windows,” tech. rep., Microsoft, 09 2020.
-

- 
- [1489] H. Deland, M. Dan, M. Gary, B. Tina, H. Imran, and V. Denise, “Generate a kernel or complete crash dump,” tech. rep., Microsoft, 08 2019.
  - [1490] D. Marshall, “Forcing a system crash from the keyboard,” tech. rep., Microsoft, 07 2019.
  - [1491] B. Karl and S. Michael, “Minidump files,” tech. rep., Microsoft, 05 2018.
  - [1492] Microsoft, “Minidump\_header structure (minidumpapiset.h),” tech. rep., Microsoft, 05 2018.
  - [1493] B. Karl, C. David, fran ki, B. Drew, J. Mike, and S. Michael, “Windows error reporting,” tech. rep., Microsoft, 05 2018.
  - [1494] B. Karl, C. David, S. Andrei-George, J. Mike, and S. Michael, “Collecting user-mode dumps,” tech. rep., Microsoft, 05 2018.
  - [1495] Microsoft, “Minidumpwritedump function (minidumpapiset.h),” tech. rep., Microsoft, 12 2018.
  - [1496] W. Steven, C. David, B. Drew, J. Mike, H. Bart, and S. Michael, “Crash dump analysis,” tech. rep., Microsoft, 05 2018.
  - [1497] M. Don and M. Tim, “.dump (create dump file),” tech. rep., Microsoft, 07 2020.
  - [1498] H. Ted, C. David, K. Martin, P. Theano, M. Don, and S. Tim, “Writing a bug check reason callback routine,” tech. rep., Microsoft, 05 2019.
  - [1499] Microsoft, “Kbugcheck\_reason\_callback\_routine callback function (wdm.h),” tech. rep., Microsoft, 05 2019.
  - [1500] Microsoft, “Kregisterbugcheckreasoncallback function (wdm.h),” tech. rep., Microsoft, 05 2018.
  - [1501] Microsoft, “Kbugcheck\_callback\_reason enumeration (wdm.h),” tech. rep., Microsoft, 05 2019.
  - [1502] M. Don and C. David, “Reading bug check callback data,” tech. rep., Microsoft, 06 2019.
  - [1503] Microsoft, “Pssetcreateprocessnotifyroutineex function (ntddk.h),” tech. rep., Microsoft, 04 2018.
  - [1504] Microsoft, “Pcreate\_process\_notify\_routine\_ex callback function (ntddk.h),” tech. rep., Microsoft, 04 2018.
  - [1505] Microsoft, “Psgetprocessid function (ntddk.h),” tech. rep., Microsoft, 04 2018.
  - [1506] H. Ted, C. David, and S. Tim, “Windows kernel-mode hal library,” tech. rep., Microsoft, 10 2018.
  - [1507] L. Bill, M. Duncan, and O. Johannes, “Transport minidriver overview,” tech. rep., Microsoft, 04 2017.
  - [1508] Microsoft, “Download the windows driver kit (wdk),” tech. rep., Microsoft, 08 2020.
  - [1509] C. David, G. Eliot, H. Ted, H. Greg, and B. Joshua, “Windows hello,” tech. rep., Microsoft, 05 2017.
  - [1510] Microsoft and al., “Windows hello for business,” tech. rep., Microsoft, 05 2020.
  - [1511] M. Satran, “Windows biometric framework,” tech. rep., Microsoft, 05 2018.
  - [1512] C. David, G. Eliot, H. Ted, B. Joshua, and mapalko, “Windows hello: Steps to submit a fingerprint driver,” tech. rep., Microsoft, 07 2017.
  - [1513] N. Grebennikov, “Keyloggers: How they work and how to detect them (part 1),” *Securelist*, 03 2007.
  - [1514] R. Chen, “It rather involved being on the other side of this airtight hatchway: Booting into another operating system,” tech. rep., Microsoft, 03 2020.
  - [1515] M. Brinkmann, “Firefox will block dll injections,” 01 2019.
  - [1516] M. Don, C. David, and B. Nathan, “Tools for verifying drivers,” tech. rep., Microsoft, 06 2020.
  - [1517] M. Don, C. David, M. Gary, G. Eliot, aahill, and B. Nathan, “Driver verifier,” tech. rep., Microsoft, 04 2017.
-

- [1518] Microsoft, “Windows hardware lab kit,” tech. rep., Microsoft, 11 2018.
- [1519] B. DAVID, “(en) how to fool antivirus software?,” in *Nuit du hack XV*, 06 2017.
- [1520] G. Faivre, “Nuit du hack xv,” 06 2017.
- [1521] J. Leitch, “Process hollowing,” 09 2011.
- [1522] L. Tal and K. Eugene, “Lost in transactions: Process doppelganging,” in *Black Hat Europe 2017*, (London, United Kingdom), 12 2017.
- [1523] J. Shaw, “Process herpaderping,” 07 2020.
- [1524] G. Landau, “What you need to know about process ghosting, a new executable image tampering attack,” 06 2021.
- [1525] A. Hosseini, “Ten process injection techniques: A technical survey of common and trending process injection techniques,” 07 2017.
- [1526] B. Jan and B. Zoran, “Extending applications using an advanced approach to dll injection and api hooking,” *Software; Practice and Experience*, vol. 40, pp. 567 – 584, 06 2010.
- [1527] R. Colin, S. Kent, J. Mike, B. Mike, H. Gordon, and C. Saisang, “Linker support for delay-loaded dlls,” tech. rep., Microsoft, 01 2021.
- [1528] R. Colin, S. Kent, J. Mike, B. Mike, H. Gordon, and C. Saisang, “/delayload (delay load import),” tech. rep., Microsoft, 11 2016.
- [1529] M. Pietrek, “An in-depth look into the win32 portable executable file format,” *MSDN Magazine*, 02 2002.
- [1530] R. Colin, S. Kent, J. Mike, B. Mike, H. Gordon, and C. Saisang, “Understand the delay load helper function,” tech. rep., Microsoft, 01 2021.
- [1531] M. Baranauskas, “Import adress table (iat) hooking,” *ired.team*, 06 2018.
- [1532] S. McLean, S. Gurkirat, B. Drew, C. David, J. Mike, and S. Michael, “Dynamic-link library search order,” tech. rep., Microsoft, 09 2020.
- [1533] V. Denise and S. Daniel, “Protect important folders with controlled folder access,” tech. rep., Microsoft, 02 2021.
- [1534] S. Aurangzeb, M. Aleem, M. Iqbal, and A. Islam, “Ransomware: A survey and trends,” *Journal of Information Assurance and Security (ESCI - Thomson Reuters Indexed)*, vol. 12, pp. 48–58, 06 2017.
- [1535] A. Abdulrahman Abba, A. Muhammad, and K. K. Mohammed, “Ransomware strategies,” 11 2020.
- [1536] A. Gazet, “Comparative analysis of various ransomware virii,” *Journal in Computer Virology*, vol. 6, pp. 77–90, 2008.
- [1537] N. Shah and M. Farik, “Ransomware-threats, vulnerabilities and recommendations,” *International Journal of Scientific & Technology Research*, vol. 6, pp. 307–309, 06 2017.
- [1538] S. Popoola, U. Iyekekpola, S. Ojewande, F. Sweetwilliams, S. John, and P. A. Atayero, “Ransomware: Current trend, challenges, and research directions,” in *World Congress on Engineering and Computer Science 2017*, vol. 1, (San Francisco, USA), p. 7, WCECS 2017, 10 2017.
- [1539] W. H. Lori, D. Dmitri, and Matt, “Installing a filter driver,” tech. rep., Microsoft, 08 2020.
- [1540] W. H. Lori, C. David, K. Andrew, S. Sameer, and Matt, “File systems driver design guide,” tech. rep., Microsoft, 09 2020.
- [1541] W. H. Lori and Matt, “Controlling filter manager operation,” tech. rep., Microsoft, 04 2017.
-

- 
- [1542] W. H. Lori, C. David, aahill, and Matt, “Writing preoperation callback routines,” tech. rep., Microsoft, 04 2017.
- [1543] W. H. Lori, C. David, and Matt, “Writing postoperation callback routines,” tech. rep., Microsoft, 04 2017.
- [1544] W. H. Lori, C. David, and Matt, “Writing preoperation and postoperation callback routines,” tech. rep., Microsoft, 04 2017.
- [1545] W. H. Lori, C. David, and Matt, “Modifying the parameters for an i/o operation,” tech. rep., Microsoft, 04 2017.
- [1546] W. H. Lori and Matt, “About minifilter contexts,” tech. rep., Microsoft, 04 2017.
- [1547] T. Ganacharya, “Stopping ransomware where it counts: Protecting your data with controlled folder access,” *Windows Defender Security Intelligence*, 10 2017.
- [1548] N. Minh Hai, “A statistical approach for packer identification,” *Vietnam Journal of Science and Technology*, vol. 54, p. 129, 03 2018.
- [1549] BitDefender, “Anti-virus technology whitepaper,” 2007.
- [1550] E. Masabo, K. Kaawaase, J. Sansa-Otim, J. Ngubiri, and D. Hanyurwimfura, “A state of the art survey on polymorphic malware analysis and detection techniques,” *Hum. Cent. Comput. Inf. Sci.* 8, 08 2018.
- [1551] Z. Arthur and W. James E., “Pre-fetching of pages prior to a hard page fault sequence,” 03 1999.
- [1552] C. Tilbury, “What is new in windows application execution?,” *SANS.ORG*, 01 2015.
- [1553] I. Alex, R. Mark E., and S. David A., *Windows Internals, Part 2: System architecture, processes, threads, memory management, and more (6th Edition)*, vol. 2. Microsoft Press, 12 2014.
- [1554] S. Narasimha and N. Dylan, “Digital forensic analysis on prefetch files,” *International Journal of information security science*, 2015.
- [1555] J. Metz, “Superfetch databases,” 04 2014.
- [1556] R. Blog, “Windows superfetch file format - partial specification,” 10 2011.
- [1557] hiddenillusion Blog, “Go prefetch yourself,” 05 2016.
- [1558] R. Chen, “A question about avoiding page faults the first time newly-allocated memory is accessed,” tech. rep., Microsoft, 05 2017.
- [1559] C. Marcho, “The basics of page faults,” tech. rep., Microsoft, 06 2008.
- [1560] M. Don, C. David, B. Kelly, T. Ryen, and G. Eliot, “Symbol path for windows debuggers,” tech. rep., Microsoft, 10 2019.
- [1561] Microsoft, “Rtlcompressbuffer function (ntifs.h),” tech. rep., Microsoft, 04 2018.
- [1562] Mikben, S. Kent, and S. Michael, “File caching,” tech. rep., Microsoft, 05 2018.
- [1563] M. Schofield, K. Sharkey, D. Coulter, M. Jacobs, and M. Satran, “Localsystem account,” tech. rep., Microsoft, 05 2018.
- [1564] G. Eliot, C. David, P. Ken, and B. Joshua, “Windows secure boot key creation and management guidance,” tech. rep., Microsoft, 05 2017.
- [1565] J. de Haas, “20 ways past secure boot,” in *HITB Security Conference*, (Malaysia), Riscure Security Lab, 10 2013.
- [1566] R. Michael, Z. Vincent, and L. Tim, *Harnessing the UEFI Shell: Moving the Platform Beyond DOS, Second Edition*. De Gruyter, Incorporated, 2017.
-



- [1567] M. Pierre-François, I. Armand, S. Solène, and D. Baptiste, “Uefi, the heart of the system,” *MISC*, 08 2019.
- [1568] M. Pierre-François, I. Armand, S. Solène, and D. Baptiste, “Uefi programing,” *MISC*, 01 2020.
- [1569] M. Pierre-François, I. Armand, S. Solène, and D. Baptiste, “Uefi analysis with windbg,” *MISC*, 03 2020.
- [1570] M. Pierre-François, I. Armand, and S. Solène, “L’uefi, l’univers avant le big bang,” *SecuriteOff*, 03 2019.
- [1571] R. Dingledine, N. Mathewson, and P. Syverson, “Tor: The second-generation onion router,” *Paul Syverson*, vol. 13, 06 2004.
- [1572] D. Maxence, F. Eric, and D. Baptiste, “Investigation and surveillance on the darknet - an architecture to reconcile legal aspects with technology,” in *Proceedings of the 18th European Conference on Cyber Warfare and Security*, (University of Coimbra, Portugal), ECCWS 2019, 07 2019.
- [1573] W. Hasan, “A survey of current research on captcha,” *International Journal of Computer Science & Engineering Survey*, vol. 7, pp. 1–21, 06 2016.
- [1574] S. Sivakorn, J. Polakis, and A. D. Keromytis, “I’m not a human : Breaking the google recaptcha,” in *Black Hat Asia 16*, 2016.
- [1575] Techopedia, “Spider trap,” 11 2017.
- [1576] V. V. Steven, K. Ondrej, and Z. Vojtech, “Crawler traps: How to identify and avoid them,” 08 2020.
- [1577] D. Maxence, D. Baptiste, and F. Eric, “Detection of crawler traps: Formalization and implementation defeating protection on internet and on the tor network,” in *Proceedings of the 6th International Conference on Information Systems Security and Privacy - ForSE*, pp. 775–783, INSTICC, SciTePress, 2020.
- [1578] B. David, M. Delong, and E. Filiol, “Detection of crawler traps: formalization and implementation - defeating protection on internet and on the tor network,” *Journal of Computer Virology and Hacking Techniques*, 04 2021.
- [1579] B. Nosek, G. Alter, G. Banks, D. Borsboom, S. Bowman, S. Breckler, S. Buck, C. Chambers, G. Chin, G. Christensen, M. Contestabile, A. Dafoe, E. Eich, J. Freese, R. Glennerster, D. Goroff, D. Green, B. Hesse, M. Humphreys, and T. Yarkoni, “Promoting an open research culture,” *Science (New York, N.Y.)*, vol. 348, pp. 1422–5, 06 2015.
- [1580] D. Carraway, “Sugarplum – spam poison,” 04 2003. Accessed: 2019-11-21.
- [1581] J. Stickano, “Stickano/tarantula: Spider trap,” 10 2018. Accessed: 2019-11-21.
- [1582] S. Kullback, *Information Theory and Statistics*. New York: Wiley, 1959.
- [1583] D. J. C. MacKay, *Information Theory, Inference & Learning Algorithms*. New York, NY, USA: Cambridge University Press, 2002.
- [1584] J. A. T. Thomas M. Cover, *Elements of Information Theory*. Wiley-Interscience Publication, 1991.
- [1585] D. Whelsh, *Codes and cryptography*. Oxford University Press, 1988.
- [1586] C. E. Shannon, “A mathematical theory of communication,” *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.
- [1587] C. SHANNON, “Prediction and entropy of printed english,” *The Bell System Technical Journal*, vol. 30, pp. 50–64, 09 1950.
- [1588] Y. Cai and L.-H. Lim, “Distances between probability distributions of different dimensions,” *ArXiv*, vol. abs/2011.00629, 2020.
- [1589] J. Chung, P. Kannappan, C. Ng, and P. Sahoo, “Measures of distance between probability distributions,” *Journal of Mathematical Analysis and Applications*, vol. 138, no. 1, pp. 280–292, 1989.
-



- [1590] M. Markatou, D. Karlis, and Y. Ding, “Distance-based statistical inference,” *Annual Review of Statistics and Its Application*, vol. 8, pp. 1–27, 09 2020.
- [1591] J. Lin, “Divergence measures based on the shannon entropy,” *IEEE Trans. Information Theory*, vol. 37, pp. 145–151, 1991.
- [1592] K. Matusita, “Decision rules, based on the distance, for problems of fit, two samples, and estimation,” *The Annals of Mathematical Statistics*, vol. 26, 12 1955.
- [1593] A. Bhattacharyya, “On a measure of divergence between two multinomial populations,” *Sankhya: The Indian Journal of Statistics*, vol. 7, pp. 401–406, Juli 1946.
- [1594] D. Comaniciu, V. Ramesh, and P. Meer, “Kernel-based object tracking,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, pp. 564–577, May 2003.
- [1595] W. Feller, *An Introduction to Probability Theory and Its Applications*, vol. 1-2. Wiley, 01 1968.
- [1596] M. Kendall and A. Stuart, *The advanced theory of statistics*, vol. 1: Distribution theory. New York, NY: Macmillan, 4th ed., 1977.
- [1597] G. Saporta, *Probabilités, analyse des données et statistique, 2e édition révisée et augmentée*. Technip, 2006.
- [1598] A. Pollastri and F. Tornaghi, “Some properties of the arctangent distribution,” 01 2004.
- [1599] Z. M., “L’impiego della funzione arcotangente incompleta nello studio della distribuzione asintotica dello scarto standardizzato assoluto massimo di una trinomia,” *Statistica*, vol. 2, no. XXXIX, pp. 269–286, 1979.
- [1600] M. Stone, “Cross-validators choice and assessment of statistical predictions,” *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 36, pp. 111–133, jan 1974.
- [1601] S. R. Fisher, “272: The nature of probability,” *Centennial Review*, vol. 2, pp. 261–274, 1958.
- [1602] H. Fischer, *A history of the central limit theorem. From classical to modern probability theory*. Springer-Verlag New York, 01 2011.
- [1603] G. Pólya, “über den zentralen grenzwertsatz der wahrscheinlichkeitsrechnung und das momentenproblem,” *Mathematische Zeitschrift*, vol. 8, pp. 171–181, 1920.
- [1604] N. Popov, “Changes to git commit workflow,” 03 2021.
- [1605] P. Andrey, K. Omer, and M. Shachar, “Jfrog detects malicious pypi packages stealing credit cards and injecting code,” in *jfrog.com*, 07 2021.
- [1606] D. Goodin, “Software downloaded 30,000 times from pypi ransacked developers’ machines,” *arstechnica*, 07 2021.
- [1607] G. Kroah-Hartman, “Re: [patch] sunrpc: Add a check for gss\_release\_msg,” 04 2021.
- [1608] G. Kroah-Hartman, “[patch 000/190] reversion of all of the umn.edu commits,” 04 2021.
- [1609] W. Qiushi and L. Kangjie, “On the feasibility of stealthily introducing vulnerabilities in open-source software via hypocrite commits,” *Proc. Oakland, page to appear*, 2021.
- [1610] W. Qiushi and L. Kangjie, “Clarifications on the ”hypocrite commit” work (faq),” 12 2020.
- [1611] G. Kroah-Hartman, “[git pull] char/misc driver fixes for 5.13-rc3,” 05 2021.
- [1612] H. Mats and T. Loren, “Statement from cs&e on linux kernel research,” 04 2021.
- [1613] W. Qiushi and L. Kangjie, “An open letter to the linux community,” 04 2021.
- [1614] G. Kroah-Hartman, “Re: An open letter to the linux community,” 04 2021.
-

THIS PAGE INTENTIONALLY LEFT BLANK

---

### Résumé :

L'objectif de cette thèse est de présenter des travaux de recherche afin de protéger les systèmes informatiques. L'approche proposée utilisée ici vise à adopter le point de vu de l'attaquant pour chercher des failles dans les systèmes afin de les corriger par la suite. Entre une anticipation des nouvelles menaces et la correction de celles existantes, les travaux portés ici offrent une vision duale, c'est-à-dire autant offensive que défensive, dans le rapport à la menace cyber.

Les principaux résultats de cette thèse offrent des systèmes plus sûrs pour la génération du code, la détection des malware et la neutralisation de la menace posée par les keyloggers (enregistreur de frappes). De plus, elle apporte une documentation inédite sur les mécanismes internes à Windows 10.

**Mots clés :** sécurité informatique, programmation bas niveau, rétro-conception, malware, vision offensive, keylogger

### Abstract :

The objective of this PhD is to present research work to protect computer systems. The proposed approach used in this work aims at taking the point of view of the attacker to look for vulnerabilities in the systems in order to correct them afterwards. Between the anticipation of new threats and the correction of existing ones, the work presented here offers a dual vision, i.e. both offensive and defensive, in the relationship to the cyber threat.

The main results of this thesis offer safer systems for code generation, malware detection and neutralization of the threat raised by keyloggers (keystroke loggers). In addition, it provides unprecedented documentation on the internal mechanisms of Windows 10.

**Keywords :** cyber security, kernel programming, reverse engineering, malware, offensive, keylogger.