Malware Engineering

S. Katzenbeisser, C. Schallhart, H. Veith
Institut für Informatik, Technische Universität München
Boltzmannstrasse 3, D-85748 Garching bei München
{katzenbe,schallha,veith}@in.tum.de

Abstract: Starting from simple hand-crafted viruses, today's malware has evolved to constitute highly infectious computer diseases. The technical development of malware was mainly driven by the wish to improve and accelerate both attacks and proliferation. Although these programs have incurred significant hazard and financial losses, their mechanisms are relatively simple and are amenable to effective countermeasures—once, the first attack has been launched. From a software technology point of view, malicious software in fact is often very similar to network services with the main difference that security holes are exploited to enforce participation in the protocol.

In this position paper we outline the wide range of possible malware-specific engineering techniques which are not used in known viruses and worms, but are technically feasible and will therefore be realized in the foreseeable future—less likely by hackers than by organized illegal entities. The techniques we describe enable the malware to obfuscate its functionality, monitor and analyze its environment, and modify or extend itself in non-trivial ways. Consequently, future security policies and risk assessments have to account for these new classes of malware.

WE ARE THEIR FOOD. THOSE GERMS OF THE PAST
THAT BEST CONVERTED OUR BODIES INTO THEIR OWN PROPAGATION
ARE THE GERMS OF THE PRESENT. THOSE GERMS OF THE PRESENT
THAT BEST CONVERT OUR BODIES INTO THEIR OWN PROPAGATION
ARE THE GERMS OF THE FUTURE.
Paul W. Ewald [Ewa00]

1 Introduction

On March 19, 2004 at approximately 8:45pm PST, a new worm—later named *Witty*—was found in the wild. *Witty*, a program of only 700 bytes, targeted a buffer overflow in several Internet Security Systems (ISS) products. In just 45 minutes, *Witty* managed to infect 12.000 machines all over the world, which constitutes almost the entire vulnerable population [SM04, WE04]. Although the total number of infected machines was too low for sensational press coverage, *Witty* marks a paradigm shift in malware: *Witty* distinguished itself from previous viruses in that it carried a particularly destructive payload, was error-free and was launched in an organized manner from a set of compromised hosts. But most

importantly, *Witty* spread through a relatively small population in record time, only one day after the ISS vulnerability was published. As Bruce Schneier later wrote [Sch04]: "*Witty* represents a new chapter in malware. If it had used common Windows vulnerabilities to spread, it would have been the most damaging worm we have seen yet."

During the last years, the number of known viruses and worms has been growing almost exponentially. The number of infected computers has risen with each new generation of malware: whereas *CodeRed* (2001) infected "only" about 359.000 hosts, the versions of the *Sasser* (2004) worm together infected about 1 million machines. Recently the first Bluetooth worm has been discovered [Blu04]. This shows that worm authors got interested in targeting the increasing number of ubiquitous and mobile devices. We can expect this trend to continue in the near future.

While the first worms were apparently written for the cynical entertainment of the authors, we expect that explicitly criminal worms targeted at commercial fraud (e.g., *access for sale* [SS03]), will become an increasingly dangerous threat. In fact, worms arguably present a substantial threat to the world economy. Weaver and Paxson [WP04] estimate that a *worst-case* worm could cost the US economy $50 billion in direct economic damage, not counting indirect costs caused by power outages, transportation delays, general interruptions in the industrial supply chain, etc. In the hands of highly skilled malevolent experts, these economic vulnerabilities may turn virus and worm technology into a weapon.

This position paper expresses our conviction that the technological possibilities for systematically engineering malware have hardly been explored so far. In this paper we concentrate on computer worms; however the techniques can be adapted to other types of malware, too. Notwithstanding their destructiveness, today's worms are relatively simple pieces of hand-crafted software performing straightforward mechanisms which makes them amenable to effective countermeasures. In fact, from a software technology point of view, today's worms are structurally similar to system software which goes the direct way to provide an intended (malicious) functionality by utilizing security exploits.

Most of innovation in the past years concerned the attack mechanism, which makes attacks fast and vicious but constitutes a relatively simple-minded (though often effective) way of information warfare. In this paper, we will focus on natural technologies facilitating slower but possibly more dangerous attacks.

Extrapolating the status quo. Starting with the Morris Internet worm in 1988, we have seen a permanent evolution in the world of malware. About ten years later, the first virus (*ShareFun*, 1997) spread through e-mail. *Melissa* (1999) was the fist worm performing mass mailings, while *SirCam* (2001) even contained its own SMTP client. *Nimbda* (2001) used multiple attacks to spread and *Bibrog* (2003) targeted various existing peer-to-peer schemes [KE03].

Kienzle and Elder [KE03] claim one can observe a lack of innovation in e-mail worms recently. In fact, most recent e-mail worms are just minor variations of well-known techniques: worm authors typically replace one security exploit by another, enhance the distribution and target discovery mechanisms and modify the payload. This trend was also driven by the availability of virus toolkits which enable non-experts and script-kiddies to

create a huge zoo of related viruses. Still, most worms are naive, yet dangerous, pieces of software.

The lack of recent innovation for several months does not indicate the lack of danger, as shown by *Witty*. In fact, we believe that the saturation of the known technology marks a technical boundary which is going to provoke a new cycle of innovation, characterized by a switch from hacker programming to *malware engineering*. What does malware engineering set apart from hacking?

- **Hand-crafted vs. engineered.** Similar to the development of programming from Knuth's art to Dijkstra's science, we expect a corresponding development for malicious programming. There is a clear evolutional line from the first hand-crafted viruses over virus toolkits (designed to automatically generate new virus variants) towards well-engineered viruses and worms.

- **Savage vs. tactical.** Traditional viruses and worms attempted to spread and attack as quickly as possible, similar to the influenza virus in nature. This makes them easily detectable (e.g., just by monitoring the network load); however, at this point it may be too late to stop their spreading efficiently. More refined tactics may combine slower infection rates with intelligent, goal-oriented behavior.

- **Covert vs. ostensible.** The first worms essentially amounted to mobile code, making no attempts to obfuscate their content and existence. Polymorphic viruses present the first, but by no means the last, step towards advanced obfuscation techniques. Practical experiments [CJ04] have dramatically demonstrated that current virus detection software fails even for relatively simple syntactic modifications of malicious code. This situation clearly calls for the use of semantic methods, such as program analysis, for the sake of virus detection.

- **Dynamic vs. static.** The traditional understanding of software implies a predetermined functionality, with self-modifying code being a curiosity rather than an engineering principle. Well engineered viruses may employ randomized semantic program transformations and even accept a certain risk of code corruption in their replicas ("children").

- **Exploit oriented vs. investigative.** Many worms have been created in response to published vulnerabilities. An advanced worm may attempt to identify vulnerabilities on-the-fly (e.g., buffer overflows or open ports).

- **Ad hoc vs. well-tested.** One of the most surprising features of *Witty* was its correctness, in contrast to worms like *Sasser*. In the future, we expect to see more well-tested worms.

To wrap up this argument, we believe that future malware technology will use semantically non-trivial methods to hit their target.

The goal of this position paper is to focus on the issue of advanced malware technology. We will explore possible ways to realize the above mentioned lines of development. In

Section 2 we describe in detail various design principles. The new malware we describe will typically propagate more slowly. Section 3 deals in more detail with a suitable theoretic notion of code obfuscation.

Let us conclude the introduction with the following important note: it is evident that the ideas and methods presented in this paper are potentially dangerous and not intended for implementation. We do however strongly believe that these issues need to be raised and openly discussed within the research community early on. Only knowledge and analysis of future attack possibilities pave the way for feasible countermeasures. "Security by ignorance" is no better than "security by obscurity".

2 Perspectives of Malware Engineering

In computer science at large, the transition from hacking to software engineering is characterized by the increased utilization of theoretical results and engineering techniques [Sha90]. The main message of this paper is that we expect a similar trend in the context of malware engineering. In this section, we describe technical approaches that well-engineered malware may perform in the future. Although current malware programs only use these possibilities to a small extent, we can expect this to happen once all associated practical problems are solved. Viruses and worms relying on these approaches will be considerably more difficult to detect and remove, as virus detectors will have to resort to semantic analysis methods in the detection process.

In the rest of this section, we will survey the spectrum of future worms and suggest adequate terminology. We will classify certain features of future malware technology as *obfuscated*, *ductile*, *curious*, *concurrent* and *latent*.

2.1 Obfuscated Worms

Perhaps the most traditional technique to hinder virus and worm detection is the use of program obfuscation. Obfuscation techniques come in two flavors: they either operate directly on the machine code or they employ encryption techniques. In the first case, obfuscation techniques insert dummy instructions, unnecessary branches and loops or re-schedule independent program instructions (*polymorphic worms*). An overview of such program obfuscation techniques can be found in [CTL97]. In the second case, the worm spreads in encrypted form (*encrypted worms*). The worm mostly consists of a small stub implementing a decryption routine, which decrypts the core worm code. Each worm replica is produced by randomized encryption of the parent worm's code. Both techniques for program obfuscation are essentially syntactic, i.e., they do not modify the functionality of the worm.

Recent research has shown that good program obfuscation is difficult to achieve. Systems that follow the first paradigm risk easy detection unless the obfuscation engine uses novel methods not known to analysis tools. A sophisticated way of using program obfuscation

is to implement a randomized obfuscation engine in the worm itself, producing randomly obfuscated worm replicas (*metamorphic worms*, first applied by *Zmist* [Jor02]). This approach may foil virus detectors relying on syntactic detection mechanisms, but requires the obfuscation engine itself to be obfuscated. The second paradigm (encrypted spreading) has the main disadvantage that the decryption stub remains unchanged, thereby again opening a door for detection.

We expect to see more subtly obfuscated worms in the future. However, it remains an open question whether obfuscation techniques provide an efficient means for worms to escape detection. In particular, we do not currently know whether it is possible to produce *provably undetectable* viruses and worms through obfuscation, i.e., viruses for which it is possible to obtain a formal undetectability guarantee. Section 3 explores this issue in more detail.

An entirely different way of obfuscation is to utilize code that is already available at the infected host. Given the large number of different executables present at modern computers, it is not unrealistic to assume that future worms can reuse (i.e., directly call) code of different applications. This approach is not limited to passive use of the host code. A worm can as well change portions of the code to fits its purpose. Such a tactic does not only reduce the detectability of a worm, but also makes its removal much harder.

2.2 Ductile Worms

Ductile worms are able to modify its replicas in syntactically and/or semantically nontrivial ways.

A major step in this direction was marked by *Simile* (2002), which was remarkable for its metamorphic appearance, as it was able to re-write its own code from generation to generation. To produce a new replica, *Simile* first disassembles its code into a machine-independent intermediate language. This abstract representation is then modified, while preserving semantic equivalence. Finally, the abstract code is assembled to produce machine code. To hinder detection, redundant and unused code is added [PFS02].

Simile is a perfect example of a virus that comes very close to the concept of malware engineering: it relied on a carefully engineered obfuscation engine, was well-tested, applied efficient mutation techniques to limit its discovery and was highly dynamic.

However, ductile worms are not limited to syntactic modifications. In fact, well-engineered worms may also change the *semantics* of the code, for example by varying the timing-conditions or the payload. It is not even necessary to enforce that all replicas of a worm are functional. During semantic modifications, a worm can risk a certain extent of code corruption, if it is guaranteed that a significant portion of all replicas are functional (in particular, are able to produce functional replicas itself). Such a strategy is particularly effective, if a worm contains a small test engine, enabling it to test its replicas for correctness. To implement this idea, the worm uses a metamorphic engine to produce semantically modified replicas, which it tests using its embedded testbed. If the replica is deemed correct, the worm exposes this replica in the wild, otherwise it simply produces a new

replica. Even if such a correctness test does not give 100% accurate results, it still helps to reduce the number of defect worm replicas sent to newly infected hosts, thereby increasing its infectious potential. For practical purposes, such a testbed must be fast enough so that millions of mutations can be checked within reasonable time.

2.3 Curious Worms

Paraphrasing Bruce Schneier, the patch model for system security has dramatically failed, i.e., published vulnerabilities are immediately exploited by malicious code authors. Classic worms utilized previously published exploits. As argued above, *Witty* is a dramatic witness of the fact that such exploits can be incorporated into malware in extremely short time. On the other hand, the patch model makes sure that such exploit-based malware can be exterminated in a simple way. There is no principle reason, however, why a piece of *curious* malware should not be able to use program analysis techniques in order to actively search for vulnerabilities present in the host system. In the security literature, various methods for identifying vulnerabilities (in particular buffer overflows) by static program analysis have been described [GJC+03, XCE03, WFBA00], albeit usually on the level of code and not on the level of executables. Ironically, these methods can in principle be applied with malicious intentions. Certainly, such worms are not expected to spread very rapidly, but this is not necessarily an indicator of low quality (see Section 2.5).

The most obvious obstacle for such a malicious approach is the complexity of the analysis which has to be performed. Let us consider buffer overflows as an example. Identifying buffer overflows can be broken down into two tasks: identifying potentially vulnerable function entrypoints and searching for parameters which violate the stack integrity. While the second task can easily be automatized by exhaustive search, the first is more challenging as witnessed by the above cited literature. Unless one looks for very specific static signatures of vulnerable code pieces, we can hardly expect that a worm with small code size will achieve this goal. Nevertheless, malicious code for intrusive program analysis can be hidden in large software such as web browsers, which incur heavy network and CPU load anyway. In this scenario (which we refer to as WORM-at-home in analogy to the SETI-at-home project), infected browsers can even communicate vulnerabilities in a peer-to-peer fashion.

Another option to identify function entrypoints and expected values of parameters is to explore publicly available documentation, such as the man pages under UNIX.

2.4 Concurrent Worms

A natural approach to obfuscate the functionality and spreading of the malware is to transport the payload in several pieces, which need to interact on an infected host in order to achieve the intended damage. If these pieces are present in different software packages (or in different communicating devices, recall the Bluetooth worm [Blu04]), their mali-

cious interaction may occur very rarely and in a nonpredictable way. In combination with randomization, such malware may be as nasty to track and identify as concurrency errors, which are notoriously hard to detect.

Alternatively, a resource-heavy payload (including, for example, code to create large hit lists [SPW02]) can be distributed slowly in a very low-key unobtrusive approach. The activation code, consisting of a small number of bytes, is rapidly distributed at a later time. Such an approach allows to combine rapid spread with a coordinated attack plan.

2.5 Latent Worms

The authors of latent worms have a completely different motive than juvenile worm writers. Instead of aiming at infecting a huge number of hosts as quickly as possible (such worms were termed *Warhol* or *flash worms* by [SPW02]), the authors of latent worms try to maintain a stable population of infected hosts over an extended and continuous period of time. To this end, they employ worms that spread slowly using the techniques of the previous sections to remain undetected. These worms contain payloads which can be utilized for illegal purposes. A basic example for illegal activities enabled by latent worms is "access for sale" [SS03], where an attacker sells access to a number of infected hosts. As another example, latent worms can be utilized as launchpad for fast spreading worms, providing a large initial population of infected machines.

More generally, well-engineered malware will be latent for two reasons: first, the techniques described in the previous sections preclude worm detection. However, a fast spreading worm consumes too many resources (e.g., network and CPU load) to remain undetected. In particular, anomaly detection tools will indicate the presence of a malware, although removing the worm might be hard. Second, the effort required to engineer such advanced worms can only be undertaken by resourceful and organized illegal entities, which will seek a suitable "return of investment"—in most cases this goal can only be achieved over a longer period of time. On the other hand, latent worms *require* suitable engineering techniques to remain operational despite potentially being known to anti-virus systems during their lifetime; in addition, their existence may be uncovered by a single detection event.

3 Code Obfuscation for Worms

Recently the cryptographic community has become interested in the concept of secure code obfuscators [BGI+01, LPS04]. Informally, a code obfuscator is a probabilistic polynomial time program $\mathcal{O}$ that takes a program M as input and produces a functionally equivalent program $\hat{M}$ as output, yet $\hat{M}$ is "unintelligible" in some sense. The existence of secure obfuscators has various deep consequences for security in general—from the existence of homomorphic encryption functions over software watermarking towards software copyright protection.

In future malware, a cryptographically secure code obfuscator would be extremely dangerous. Suppose that a worm with code W contains an obfuscation engine that outputs functionally equivalent, but obfuscated code, $\hat{W}$, which it distributes along the traditional infection paths. Now, as $\hat{W}$ and W are functionally equivalent, $\hat{W}$ contains the obfuscation engine as well and is able to produce obfuscated replicas. If the obfuscation is perfect (in a sense to be described below), this would allow the construction of *provably undetectable worms*: even with access to a single copy of the worm W, it would be impossible for a virus analyzer to tell whether a specific program $\hat{W}$ is an instance of the known worm W. Such an obfuscated worm would clearly be a new chapter in malware.

How can we tell whether an obfuscation is secure? Barak et. al. [BGI$^+$01] introduced the notion of *black-box obfuscators*. They call an obfuscation secure, if *all* information that can be computed out of the code of an obfuscated program can already be computed out of input/output pairs of the program. In other words, knowledge of the code does not help in analyzing the program. Somewhat surprisingly, it is possible to show that under this security notion, obfuscation is *impossible to achieve*. Barak et. al. [BGI$^+$01] prove this by constructing a class of functions $\mathcal{F}$ that is unobfuscatable in the sense that there exists a property π of the functions $f \in \mathcal{F}$ that can be efficiently computed with access to an arbitrary code that computes f, whereas $\pi(f)$ can only be guessed with low probability without code access.

However, this limiting result is not necessarily the end of secure obfuscators for two reasons: first, the proof of Barak et. al. only shows that there exists a class of unobfuscatable programs. It might be the case that obfuscation still works for a relatively large class of practically relevant programs [LPS04]. Second, we believe that black-box obfuscators are not directly applicable to the field of computer worms. Indistinguishability in the sense of Barak et. al. hides every bit of information an analysis tool may be compute from a program in probabilistic polynomial time. In fact, this type of security definition is closely related to the concept of "learnability" in formal language theory (a function is called learnable, if it can be reconstructed by a polynomial number of evaluations). We conclude that black-box obfuscators are a theoretically important but too restrictive notion if we want to capture the kind of code obfuscation we have to deal with in practice.

Consequently, we believe that finding a suitable notion of obfuscation which suits both cryptographic principles and applications is a challenging and nontrivial question. We will now discuss several approaches towards this goal:

- **Indistinguishability.** We can call a worm code obfuscated, if it is not possible to decide (in randomized polynomial time) whether a specific program $\hat{W}$ is a possible replica of a given self-obfuscating worm W. That is, even with knowledge of W, a virus detector would be unable to efficiently test whether a specific program $\hat{W}$ is a possible variation of the worm W. Note that this problem is certainly decidable for randomized polynomial $\hat{W}$ and W by simulating all computation paths of W in exponential time.

- **False detection.** A different way of describing obfuscated worm code is through the false positives and false negatives of a virus detector. A false positive occurs, if a detector incorrectly classifies an innocuous program as a worm, whereas a false

negative occurs if the detector incorrectly classifies a worm as harmless. Although false negatives are potentially more problematic, the usefulness of a virus detector depends highly on *both* a low false positives and false negatives rate.

From this point of view, a worm can be called obfuscated, if *every* virus detector that correctly classifies the worm in question, has either a huge false positives or false negatives rate. In other words, even though the detector may correctly classify the worm together with its replicas as hostile, the detector overlooks other viruses/worms or classifies harmless binaries as malicious. Clearly, such a behavior would be unacceptable for practical purposes.

- **Detection of existence.** Another notion of obfuscation may rely on the observation that future obfuscated viruses (or worms) can be implanted in innocuous files on the infected host machine. In such a setting, we may call an obfuscation secure, if it is not possible to decide whether a known virus W (or one of its potential replicas $\hat{W}$) is embedded in a given binary X.

We pose it as an open question whether obfuscation techniques can be found that are indeed secure in one of the above mentioned security definitions.

4 Conclusion

In this paper, we have focussed on the risks of systematic malware engineering based on advanced methods from computer science, as opposed to the traditional design approach which is more ad hoc. We have argued that rapid distribution is not the only strategic option for malware. In particular, latent worms that spread slowly but towards a predetermined goal pose a significant threat to the Internet, even more so when combined with mature obfuscation and mutation techniques. We have also argued that the notion of black-box obfuscation brought forward recently by Barak et. al. is too restrictive, and have discussed potential alternatives which will be the topic of future work. Unfortunately, the impossibility results by Barak et. al. have evidently discouraged cryptographers from continuing work in this direction; we believe that in the light of malware engineering and detection a more fine-grained look at program obfuscation is worthwhile.

Acknowledgements. We thank Volker Baier, Piotr Esden-Tempski, Uwe Hermann, Daniel Reutter and Michael Tautschnig for their thoughts on next-generation computer worms.

References

[BGI+01] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)Possibility of Obfuscating Programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer Verlag, 2001.

[Blu04] SymbOS.Cabir. available at `securityresponse.symantec.com/avcenter/venc/data/epoc.cabir.html`, 2004.

[CJ04] M. Christodorescu and S. Jha. Testing Malware Detectors. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'04)*, 2004.

[CTL97] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, The University of Auckland, 1997.

[Ewa00] Paul Ewald. *Plague Time: How Stealth Infections Cause Cancer, Heart Disease, and Other Deadly Ailments*. Free Press, 2000.

[GJC+03] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer Overrun Detection using Linear Programming and Static Analysis. In *Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS'03)*, pages 354–354, 2003.

[Jor02] M. Jordan. Dealing with Metamorphism. available at `http://www3.ca.com/securityadvisor/newsinfo/collateral.aspx?cid=48051`, 2002.

[KE03] D. Kienzle and M. Elder. Recent Worms: A Survey and Trends. In *WORM'03—ACM Workshop on Rapid Malcode*, pages 1–10, 2003.

[LPS04] B. Lynn, M. Prabhakaran, and A. Sahai. Positive results and Techniques for Obfuscation. In *Advances in Cryptology—EUROCRYPT'04*, volume 3027 of *Lecture Notes in Computer Science*, pages 20–39. Springer Verlag, 2004.

[PFS02] Frédéric Perriot, Peter Ferrie, and Péter Ször. Striking Similarities—W32/Simile. Virus Bulletin, May 2002.

[Sch04] B. Schneier. The Witty worm: A new chapter in malware. Computerworld, available at `http://www.computerworld.com/securitytopics/security/virus/story/0,10801,93584,00.html`, 2004.

[Sha90] M. Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15–24, 1990.

[SM04] C. Shannon and D. Moore. The Spread of the Witty Worm. available at `http://www.caida.org/analysis/security/witty`, 2004.

[SPW02] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Securiy Symposium*, 2002.

[SS03] S. Schechter and M. Smith. Access for Sale: A new Class of Worm. In *WORM'03—ACM Workshop on Rapid Malcode*, pages 19–23, 2003.

[WE04] N. Weaver and D. Ellis. Reflections on Witty: Analyzing the Attacker. available at `http://www.icsi.berkeley.edu/~nweaver/login_witty.txt`, 2004.

[WFBA00] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security (NDSS'00)*, 2000.

[WP04] N. Weaver and V. Paxson. A Worst-Case Worm. In *3rd Annual Workshop on Economics and Information Security (WEIS'04)*, 2004.

[XCE03] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic path-sensitive analysis to detect memory access errors. In *9th European Software Engineering Conference and 11th ACM Symposium on the Foundations of Software Enginering (ESEC/FSE'03)*, 2003.