Chapter 25

IN-KERNEL CRYPTOGRAPHIC EXECUTABLE VERIFICATION

Yusuf Motara and Barry Irwin

Abstract This paper discusses the problems posed by Trojan horses and unauthorized code, and reviews existing solutions for dealing with them. A technique involving the in-kernel verification of executables is proposed. Its advantages include simplicity, transparency, ease of use and minimal setup time. In addition, the technique has several applications, including assisting with honeypot implementations, incident response and forensic investigations.

Keywords: Trojan horse, signed binaries, executable verification, cryptography

1. Introduction

Computer users around the world are continuously bombarded with viruses, worms and exploits, many of which leave unwanted executable content on their machines. Current solutions to the problem include patching, using anti-virus programs or securing systems with firewalls, all of which are preventative measures [10]. No standard technique exists to stop a program from executing if it manages to bypass security controls. This paper describes a technique involving the in-kernel verification of executables, which detects unauthorized code and (optionally) prevents it from executing.

The following section discusses the problems posed by Trojan horses and unauthorized code, and reviews existing solutions for dealing with them. Next, the technique involving in-kernel verification of executables is described in detail. Finally, applications of the technique, including honeypot implementations, incident response and forensic investigations, are highlighted.

2. Background

This section discusses Trojan horses and unauthorized code, and evaluates techniques for guarding against them involving integrity checking, pre-execution validation and comprehensive security solutions.

2.1 Trojan Horses

A Trojan horse is defined as "a malicious, security-breaking program that is disguised as something benign" [5]. This paper considers a broader definition of a Trojan horse to include programs that are neither malicious nor security-breaking. Examples include "adware" that piggy-backs on more legitimate programs and anti-piracy programs that execute whenever certain applications are started. Thus, a Trojan horse is taken to mean any code on a system that is not explicitly allowed or expected to be run by the user.

A machine can become the host to a Trojan horse in several ways.

- A user may visit a website that exploits a browser or operating system flaw on the user's machine. An example is the Bofra vulnerability [14], which only requires a user to click on a link before executable content is downloaded to the user's machine.

- A use may download a program out of curiosity, leading to a compromise, even if the program is supposedly uninstalled later on.

- A user may open and execute an email attachment either unintentionally or because it comes from a trusted source. An example is the Sober worm [16] that uses social engineering to spread.

- A user may execute the preview, auto-accept or auto-get features of a program that lead to code execution and a Trojan horse being deposited on the user's system.

- An intrusion may leave Trojan horse replicas of important system utilities. This type of Trojan horse, called a rootkit, installs binaries that mask the compromise and leave a back-door into the system. Rootkits are freely available for a number of platforms.

- An untutored user might blindly follow instructions on a webpage that leave a Trojan horse on the system. Many users are deceived by phishing, which is the practice of using a site that appears legitimate to convince users to perform certain actions. For example, a "bank website" might request a user to download a "security update" for his computer.

Avoiding infection requires that users be certain about the source of a file and its contents, and that the system be secured and fully patched. Even these precautions may be insufficient if the user is fooled (e.g., by phishing) into compromising his own system. Alternatively, an attacker can take advantage of the time delay between an exploit being created and a patch being released.

2.2 Unauthorized Code

Unauthorized code is code that should not be running on a system. Examples include user-created binaries and games on a mail server. The difference between unauthorized code and a Trojan horse is that the former is installed on the system with the knowledge of the user while the latter is not.

Note that many privilege escalation attacks require that binaries be run by untrusted users with access to normal user privileges. An example is the privilege escalation attack that takes advantage of the `sudo` utility's failure to clean the environment [7]. This is an example of unauthorized code that is malicious and is also likely to be run by script kiddies on an otherwise secure system. Since there is almost inevitably a delay between the release of an exploit and the creation (and application) of a vendor-supplied patch or workaround, the intervening time leaves systems vulnerable to compromise. Of course, once a system is compromised, hiding the compromise using a rootkit is trivial.

2.3 Integrity Checking

Traditionally, guarding against Trojan horses and unauthorized code has been accomplished using a solution such as Tripwire [13], which takes "snapshots" of a system. These snapshots include relevant information about files that makes tampering easy to detect. After a suspected compromise, or simply as part of a daily security regimen, the integrity of files on the system is checked against the snapshot. Integrity checkers have varying levels of sophistication, with some (e.g., `mtree` [6]) checking file integrity as a side-effect of their main functionality.

Conventional integrity checkers suffer from the fact that a time lapse exists between the compromise and the check. This may be exploited by attackers who can launch further attacks using executables placed on the compromised machine. Therefore, integrity checkers are only a partial solution to the problems noted in the discussion of Trojan horses and unauthorized code (Sections 2.1 and 2.2).

It is also important to note that should a compromise occur, it may be difficult to verify the integrity of the database used for comparing file characteristics. For this reason, the database should be stored either offline or on a different (and secure) system.

2.4 Pre-Execution Validation

Validating the integrity of binaries using digital signatures or simple hashes just before execution eliminates the time lapse problem suffered by integrity checkers. The following tools may be used to perform pre-execution validation.

2.4.1 CryptoMark. CryptoMark digitally hashes and signs an Executable and Linking Format (ELF) binary program, storing the result within a SHT_NOTE section [3]. It computes an MD5 hash of the loadable file segments, and checks the hash and signature via a kernel module whenever the executable is run. Userspace tools are used to perform and manage the signing of a file.

CryptoMark may be run in a number of configurations. One common configuration requires all binaries to be signed; unsigned binaries or those with incorrect signatures are not allowed to run. Another configuration requires all binaries that run as the superuser to be signed. This allows users to compile and run their own programs, but denies them the ability to compile and run binaries that run as the superuser, even if they have managed to gain superuser access.

2.4.2 WLF. WLF [4] verifies the integrity of ELF binaries in-kernel, whereas other techniques (e.g., [2]) verify binaries using modified interpreters. Key management is stressed by WLF, which makes provision for using signed binaries from different sources by embedding a `KeyID` field in the file signature. WLF can verify a large variety of files using a plug-in architecture.

2.4.3 TrojanXproof. TrojanXproof [15] verifies the integrity of ELF binaries using a secured database rather than signing the executables themselves. It takes the form of kernel patches for the FreeBSD and OpenBSD operating systems, and verifies shared libraries and ELF executables. TrojanXproof does not cryptographically secure files using digital signatures. Instead, it simply creates MD5 hashes of files and relies on the security of the database.

2.4.4 DigSig. DigSig [1] uses a kernel module to check the signatures generated by BSign [11], a tool for signing ELF binaries. All binaries must be signed correctly in order to run. DigSig caches signatures (also suggested in [2]), which makes the repeated use of commands

Table 1. Evaluation of existing solutions.

Name	Ease of Use	Executable Checking	Pre-Exec. Validation	Active Developmt.	Transprnt. Checking
Tripwire	X	X		X	
CryptoMark	X		X		X
WLF			X		X
TrojanXproof			X	X	X
DigSig	X		X	X	X
SELinux		X		X	X

much faster than it would be otherwise. In general, DigSig has most mature implementation of runtime validation.

2.4.5 Status of Tools. Of the tools described above, only DigSig appears to be under active development. CryptoMark, a project of Immunix [8], has disappeared from the vendor's website. Work on WLF has ceased, and the code that does exist is quite fragile. TrojanXproof is unsupported in most cases.

2.5 Comprehensive Security Solutions

Unauthorized binaries and Trojan horses can be defeated using comprehensive security solutions such as SELinux [9], which restrict executable access to known valid binaries and ensure fine-grained access control. Comprehensive security solutions, however, are complex to configure. For example, SELinux policies require an understanding of role-based access control, type-based enforcement, domains, access vectors and more. This makes them difficult for the average user to understand and use, resulting in an increased likelihood of misconfiguration.

2.6 Evaluation of Solutions

Table 1 provides an evaluation of existing solutions. "Ease of use" is determined by the amount of effort the solution takes to set up and maintain. "Transparent checking" refers to how transparent the checks are to the user. The other categories are self-explanatory.

3. Pre-Execution Validation Strategies

This section discusses strategies for pre-execution validation of files that can secure machines against Trojan horses and unauthorized code.

3.1 Basic Strategy

It is important that a system used to verify whether or not tampering has occurred is itself tamper-proof. In other words, it should not be possible for an attacker who has gained administrator privileges to subvert the system. The following two techniques are used for this purpose.

- **Digital Signatures:** Binaries are digitally signed using a private key that may be stored offline or protected with a password. Only properly-signed binaries are allowed to run. Without access to the private key, an attacker cannot sign his own binaries, preventing him from running unauthorized executables on the system.

- **In-Kernel Verification:** Signatures of binaries are verified in-kernel. A system kernel is one of the hardest components to compromise. It cannot (currently) be replaced without a reboot, so an attacker would have to reboot the machine – something that a system administrator is likely to notice. Also, the kernel image may be specific to the machine and, therefore, difficult to replicate.

  Creating a custom kernel that contains untrusted code is a non-trivial task. However, this situation may change with the introduction of system calls such as `kexec` in the Linux MM-series kernel that allows a running kernel to be replaced without rebooting the system. Such a "feature" should never find its way into a secure system. To address this issue, machines could boot off CDROMs or other secure read-only media that contain the kernel and base system, or they could use a kernel made available over a network. Alternatively, a kernel checksum could be placed on another machine, or the kernel could be cryptographically signed, and these could be checked during booting. These measures make compromising the kernel difficult, if not impossible.

Although we use the term "signing binaries" in this paper, note that that both binaries and libraries must be digitally signed. Signing ideally occurs by sending the binary to a trusted party who examines it byte-for-byte against a known good copy, and signs it if it matches. This process can be automated quite easily, using a web service or remote procedure call (RPC) variant that signs binaries.

To simplify the discussion, we consider the case of having one "correct" signature rather than a number of possible signatures. The techniques described below are easily adapted to handling more than one signature.

Signature revocation must be taken into account in all designs. Revocation occurs if an application is found to be exploitable or otherwise unsuitable for use on a system. In the absence of revocation, an attacker who has saved an earlier signed version of the exploitable file could use it to replace the current version, opening up a security hole in the system. It should not be possible to remove signatures from a list of revoked signatures, nor should it be possible to add signatures to the list improperly. The former may lead to a compromise; the latter to denial of service as legitimate programs would be prevented from executing.

3.2 In-Binary Signatures

An ELF object file has a number of sections, some which are loaded into memory at runtime and some which are not. Sections that hold vendor-specific information are of type SHT_NOTE, and are not loaded into memory to form the executable image at runtime [12]. An ELF binary may have its signature stored in a specially-crafted SHT_NOTE section.

The advantage of this approach is that a binary and its signature are linked through one file; there is no need to have separate storage for the signature, which simplifies verification. Since SHT_NOTE sections are not loaded to form an executable image, the behavior of the executable is the same as if the section did not exist at all. This means that signed binaries are just as portable as unsigned binaries.

Signature revocation is problematic as there is no good way to keep track of a list of revoked signatures. The revoked signatures cannot be kept within the file as this would not get around the problem of file replacement. Also, the need to maintain a separate database that must be kept with the files removes the benefit of having a single storage site for signatures. In addition, the database grows as signatures are revoked; checking every signature against an ever-increasing "revoked" list does not scale well. Another drawback is that only ELF binaries may be checked; Python scripts, for example, do not have SHT_NOTE sections.

3.3 External Signatures

Keeping signatures separate from executable files means that two files – the binary itself and the signature database – must be opened whenever a binary (executable or library) is run. The database is secured by signing it with a private key; the file hashes within the database need not be encrypted individually. Signing is implemented by sending the executable and the database to a third party. The third party verifies the file, unlocks the database, adds the new signature or replaces the old one, and returns only the database.

One advantage of this approach is that any executable, not just ELF files, can be checked. Meta-information about the file (e.g., user/group ownership) can be stored and checked along with the signature for added security. Revocation is not an issue as replacing a file with a previous version is the same as having a file with an invalid signature; therefore, no special revocation check needs to be done. To ensure that the database is not replaced by an earlier version, it is necessary to maintain a version number in-kernel and in the database, or to have the database located on secure read-only media (e.g., locked flash memory). The database itself can be secured in a number of ways, as it is only one file rather than hundreds of binaries.

A disadvantage of this approach is that opening and checking two files upon every execution is slower than simply dealing with one file.

3.4 Caching

Validating an executable file before execution leads to an inevitable delay at program startup. While the delay may not be noticeable for a large executable, e.g., a word-processing program or a web browser, it can be quite significant for small system utilities such as `ls` or `grep`. The problem can be ameliorated by caching signatures, especially in the case of external signatures.

A signature cache should be invalidated after writing to a file, removing a file from a directory hierarchy, and rebooting. In addition, networked files to which writes cannot be reliably detected should never have their signatures cached. This approach to caching is taken by DigSig, which has contributed to significant increases in speed [1]. In contrast, CryptoMark does no caching [3]; it was removed from public distribution in 2004 due to sluggish performance.

3.5 Limitations

Performance limitations are significant in the case of networked file systems. As indicated in Section 3.4, it is difficult to maintain good performance without caching. Cached signatures are valid only when a kernel can record accesses to a file system. Since this is not possible for networked file systems, signature caching cannot be employed.

This approach is intended to increase system security; it is not a comprehensive security solution, e.g., SELinux. Nevertheless, the approach has several advantages, including transparency, ease of use and minimal setup time, all of which contribute to ease of deployment across machines intended for very different purposes. However, the trade-off is that customization and the benefits of a comprehensive, flexible security policy are lost.

Of course, all security is lost if the kernel or the third party who holds the private key are compromised. As discussed in Section 3.1, several techniques exist for securing the kernel. Securing the third party is outside the scope of this work.

4. Digital Forensics

Of course, denying an executable permission to run is simply one response to an invalid or nonexistent signature. Several other options are possible – from logging the behavior of the executable to notifying the system administrator. The proposed technique can be used in a honeypot implementation to log executable files that do not have signatures, make backup copies of them and maintain records of their behavior. It can also be used in conjunction with a network traffic logger to record the behavior of certain exploits "in the wild" for later analysis, for input to an intrusion detection system (IDS) or as evidence of malicious activity.

In the case of a suspected intrusion, system tools can be limited to a known good set by disallowing the execution of all unsigned binaries. In this state the system can be checked for a compromise. While other methods exist for locking down a system, the benefits of this approach are that it allows read-write access to the system and ensures the integrity of the tools used for analysis. The system can be locked down as soon as suspicious behavior is noted by an intrusion detection system. If a production web server is compromised at night or on a weekend, the lockdown strategy ensures that the server continues to operate while the threat of further compromise is mitigated and unsigned executables are stored in a safe location for further analysis.

5. Conclusions

The focus of this work is protecting systems from Trojan horses and unauthorized code. However, the approach has several other applications, including assisting with honeypot implementations, incident response and forensic investigations. Other variations, such as only allowing signed binaries to be executed in a superuser context, are also possible.

The approach is intended to increase system security; it is not a comprehensive security solution, e.g., SELinux. Nevertheless, the approach has several advantages, including simplicity, transparency, ease of use and minimal setup time.

References

[1] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi and V. Roy, The DigSig Project, *LinuxWorld Magazine*, vol 2(1), December 22, 2003.

[2] W. Arbaugh, G. Ballintijn and L. van Doorn, Signed executables for Linux, Technical Report CS-TR-4259, University of Maryland, College Park, Maryland, 2001.

[3] S. Beattie, A. Black, C. Cowan, C. Pu and L. P. Yang, CryptoMark: Locking the stable door ahead of the Trojan horse, Technical Report, WireX Communications Inc., Portland, Oregon, 2000.

[4] L. Catuogno and I. Visconti, A format-independent architecture for run-time integrity checking of executable code, in *Security in Communication Networks, Lecture Notes in Computer Science, Volume 2576*, S. Cimato, C. Galdi and G. Persiano (Eds.), Springer, Berlin-Heidelberg, pp. 219-233, 2003.

[5] FOLDOC, Trojan horse, *FOLDOC: The Free On-Line Dictionary of Computing* (www.foldoc.org/foldoc/foldoc.cgi?query=Trojan+Horse&action=Search).

[6] FreeBSD, `mtree(8)`, *FreeBSD 5.3 System Manager's Manual*, January 11, 2004.

[7] L. Helmer, Sudo environment cleaning privilege escalation vulnerability (secunia.com/advisories/13199).

[8] Immunix Inc. (www.immunix.org).

[9] National Security Agency, Security-Enhanced Linux (www.nsa.gov/selinux).

[10] B. Paul, *Evaluation of Security Risks Associated with Networked Information Systems*, Master's Thesis, School of Business Administration, Royal Melbourne Institute of Technology, Melbourne, Australia, 2001.

[11] M. Singer, `bsign(1)`, The Debian Project (packages.debian.org/testing/admin/bsign), 2001.

[12] Tool Interface Standards Committee, Executable and Linkable Format (ELF), Technical Report, Unix System Laboratories, Summit, New Jersey, 2001.

[13] Tripwire Inc., Tripwire for servers datasheet, Technical Report, Tripwire, Inc., Portland, Oregon (www.tripwire.com/files/literature/product_info/Tripwire_for_Servers.pdf), 2005.

[14] B. Wever and ned, Microsoft Internet Explorer malformed IFRAME remote buffer overflow vulnerability (securityresponse.symantec.com/avcenter/security/Content/11515.html).

[15] M. Williams, Anti-Trojan and Trojan detection with in-kernel digital signature testing of executables, Technical Report, NetXSecure NZ Limited, Canterbury, New Zealand, 2002.

[16] C. Wueest, W32.Sober.I@mm (sarc.com/avcenter/venc/data/w32.sober.i@mm.html).