# Self-Signed Executables: Restricting Replacement of Program Binaries by Malware*

Glenn Wurster P.C. van Oorschot
School of Computer Science
Carleton University, Canada

## Abstract

We propose using digital signatures to protect binaries already on the system from modifications by malware. While applicable to any file which is not intended to be modified by an end user, we concentrate on protecting programs and libraries present on the system before infection. Our protection does not rely on a central trusted authority or PKI, and can be incrementally deployed. While presented in the context of the Linux environment, our approach applies to other operating systems such as Windows.

## 1 Introduction and Overview

A modern computer system is composed of many interacting applications installed at various times and written by different vendors or designers. Each application normally consists of a number of program binaries along with associated libraries. These files containing executable code may be used by other applications subsequently installed on the system. Normally, the installation of a subsequent application will not overwrite previously installed binaries. A new application may use libraries previously installed by other software, but normally does not replace or modify these libraries.

Malware installed on a system does not necessarily follow the same practise of leaving system programs and libraries unaffected. It may replace common system binaries (e.g. `ls`, `ps`, `netstat`, and `ifconfig` in the Unix environment). This has been long recognized and leveraged in file system integrity checkers. One goal of these checkers is to verify that system applications and libraries remain unchanged. In this paper, we propose a method for protecting libraries and programs against modification while still allowing software updates, without the need to update data structures commonly associated with integrity checkers. The design avoids complex scanning mechanisms and alerts introduced by software installs and updates. Current integrity checkers such as Tripwire [11, 12] make no distinction between approved and malicious file modifications, resulting in a large number of alerts during any install or upgrade.

Our approach is to associate with each library file and executable a digital signature of the file which is checked by the kernel when performing an update on the binary. As explained below, the file is in essence self-signed. Our approach relies on support from the kernel to prevent modifications to installed binaries. We therefore assume that the kernel is secure against attack by malware. While this assumption is currently not accurate in most current systems,[1] we believe that this will change in the future. Malware exploitation of the kernel is a growing trend [3]. We expect that this will result in a concentrated effort by the security community to protect the kernel (in fact several proposals already exist [17, 21, 14, 7, 18]). One technique already commercially deployed in Microsoft Windows Vista 64bit is kernel module code signing [6], designed to prevent code not signed from getting into the kernel. Similar solutions exist on Linux [13]. As the threat of kernel malware increases, we believe additional such techniques will be proposed. These will allow our proposal, as discussed below, to operate securely.

We concentrate on a small piece of the malware problem, giving a simple solution to maintaining a flexible base of software which can be trusted even after system (but not kernel) compromise. By this, we mean that we allow for the possibility of, for example, malware exploitation of a buffer overflow to gain root-level privilege running as an application, but as

*Version: July 31, 2007

[1] This assumption is nonetheless true for some systems as we mention shortly.

explained above, *not* the ability to alter kernel code. Our solution provides a foothold for reclaiming machines at the application level, allowing for additional protection mechanisms. Our mechanism alone does not guarantee that previously installed security software will always be run. Malware can still terminate security applications (if the kernel allows it) or install separate copies of core binaries in alternate directories and cause them to be used through environment variables (as we discuss in Section 4). Additional copies of binaries installed by malware, however, can be avoided during forensic analysis by running applications from the known trusted base directly (e.g. by invoking the kernel directly to start a new program), avoiding such environment variables.

While alternative schemes have been proposed to deal with the problem of protecting applications against modification by malware, current solutions are either highly restrictive (e.g. requiring every application to be signed by a central authority [9]), protect only certain files [22, 15], or are not easily properly used [12]. Other solutions attempt to solve a much larger problem at the expense of additional complexity [2, 10, 7]. In contrast, we believe a smaller first step has a greater chance of being widely deployed, and accepted. While our proposal is similar to that of Pennington et al. [16], we focus on prevention rather than detection (although our proposal can also be implemented on a file server).

## 2 Implementation

Our method of protecting against modification of binary files involves a simple but well-planned use of digital signatures, perhaps best described as providing a type of self-signed file. The digital signature consists of a cryptographic hash of the binary signed with a private key; we do not attempt to tie the signature to an entity. The public key is embedded in the file being signed by the developer who compiles the application, along with the digital signature.[2] We use the digital signature along with kernel modifications to enforce one simple protection rule: *A library file or executable on disk can only be replaced by a library or executable containing a digital signature verifiable using a public key in the installed binary.* We propose supporting a few standardized digital signature algorithms (the particular choice is specified alongside the digital signature). If the new binary can not be verified, the kernel refuses to replace the original. Deployment is incremental – unsigned binaries can be replaced without restriction (which is how most systems currently operate), but once a binary is signed it must be replaced by a signed binary. Our method is quite different than SDSI/SPKI [8]; we trust keys only in a very limited setting (replacing a binary which was signed with the same key). We do not use names or certificates.

Executable files always follow a specified structure to allow greater interoperability. Most Unix distributions (including Linux) use the binary format file ELF (Executable and Linkable Format). On Windows, the PE (Portable Executable) format is used. Both of these formats allow additional data to be included in the file along with executable code. We propose adding a new element to the file, the digital signature of the file. When the kernel receives a request to overwrite a file on the system, it first checks the current file on the system. If the current file is signed, the kernel enforces the protection rule. It verifies that the new file can be verified by a public key contained in the currently installed version of the file. If the signature verifies, then the replacement is allowed to happen, otherwise it is denied. If there is no previous version of a file (as determined by the filename) in the target directory at the time of the attempted install, the kernel allows the file to be installed. While we concentrate exclusively on binary code in this note, our method could be expanded to include other file formats.

The digital signature is embedded into the file as one of the last steps of the compilation process. Using standard tools, this process can be simplified and automated [4]. For programs distributed as source, a common or universal key would *not* be distributed (the person compiling the application would use their own key with the signature tools).

Our scheme, while similar to *Windows File Protection (WFP)* [15], bears several key differences.

1. WFP appears to run at application level. It is intended to protect against a non-malicious user [5].
2. WFP only protects certain pre-defined system files created or signed by Microsoft, not all applications on the system. Our method does not depend on a central authority, making it accessible to all developers.
3. The list of files protected by WFP is stored in

[2]The portion of the file holding the digital signature itself is not included in the range of the signature, to prevent a recursive definition. The public key, however, is integrity-protected by the signature.

a separate file (`sfcfiles.dll`) maintained by Microsoft. Our protection method does not attempt to maintain a central list of files.

# 3 Benefits

Benefits of the proposal include the following.

**1. No Central Key Repository.** Because the signature on a to-be-installed binary file is verified using the public key embedded in the previous version of the same file (by filename) installed on the system, there is no need to centrally register a key or involve any central repository.[3] An application author can create a signing key-pair and begin using it immediately. Furthermore, if desired, different keys can be used for each file on a system, limiting the threat from a key compromise (the attacker incurs a per-executable cost for replacing protected binaries). Because there is no dependence on a trusted authority, development of new software remains unrestricted. We make no effort to restrict the software which can be installed on a system, as long as that software does not modify already-installed binaries. Other digital signature schemes proposed in the past have relied on a trusted central authority [1, 20].

**2. Trusted Software Base Even After Compromise.** The operating system and core applications are normally installed before malware has a chance to infect a machine. We exploit this temporal property. Typical malware, because it is installed after the operating system (including core libraries and programs), is not capable of changing any of the operating system files. The operating system files, therefore, can be guaranteed to be unmodified. If a virus checker is installed before any malware, the virus checker can also be automatically protected using the same mechanism.[4] This makes it harder for malware to hide on a compromised system. Core software on a compromised system can be trusted, allowing much greater control over an infected system without requiring a reboot to clean media.

**3. Low Overhead.** Signatures are only checked during installation of binary files. During normal system operation, libraries do not change; hence the mechanism results in negligible overhead.

[3]This includes any form of central certification authority or public key infrastructure (PKI).

[4]As noted in Section 1, malware may still terminate a running virus checker process if allowed by the kernel. Our method, however, prevents stealthy replacement by a trojaned virus checker which offers the illusion of protection.

**4. Incremental Deployability with Incremental Benefit.** Kernels which do not yet support the replacement verification scheme treat signed binary files as normal binary files, and are unaffected by the existence of a digital signature. Similarly, binaries without digital signatures are allowed on a kernel which supports the proposal (in contrast to most proposed code signing schemes [20]). Either the kernel or libraries can be modified first without an adverse affect on non-supporting systems.

# 4 Issues

Here we consider additional issues.

**1. Denial of Service.** Malware can prevent files from being installed on the system by installing its own version before the legitimate application (in the same directory with the same filename) is installed. This is not a problem with core system libraries as they are always installed first. The issue occurs with applications installed after the malware. We presently see no easy way to prevent against this attack without relying on additional infrastructure. The attack, however, will be noticed on attempting to install the legitimate software.

**2. Key Updates.** Over time, inevitably signing keys will be lost or compromised. Both situations can be accommodated by allowing, as an option, the embedding of multiple verification keys in a binary file. If one key is lost, the other key(s) can be used to sign a subsequent version of the file (which can also contain new keys). We do not specify any conditions on who or what controls the private keys corresponding to these additional verification public keys, but many options exist including community trusted organizations or trusted friends who function as backups. While we specify no specific infrastructure for key revocation, pro-actively installing a new version of a file which does not allow future versions signed with the previous key (i.e. which excludes the old verification public key(s) from those embedded in the new version) prevents a compromised key from being a threat indefinitely (the same technique can be used to prevent downgrading binaries). Because each file can be signed with a different key, the effect of a compromised key can be limited.

**3. File Deletion/Movement.** Care must be taken to ensure that malware can not delete (or move a signed file to a different directory) and then install a new file as a way of avoiding the digital signature check. For this reason, deleting or moving a file be-

comes more complicated. We believe both can be dealt with by retaining kernel knowledge of the verification keys associated with a file which has been deleted or moved (e.g., by keeping a stub version of the file containing only the verification keys) – future files must still obey the verification key constraints. Because legitimate applications rarely install completely different binaries in the exact same location, we believe this may be an acceptable resolution for deletion. Furthermore, application binaries are rarely moved once installed.

**4. Unverified Modifications.** Occasionally, a system administrator or software developer may have a valid reason for replacing a file by one which is signed by a different verification key than included in the current version. We foresee unverified modifications as being rare but necessary. We therefore mention a (admittedly cumbersome) method for updating files on disk despite differing keys, ensuring that malware is not equally capable of using this method to install itself. To replace a file with one not signed by an allowed key (or a file containing no signature at all), reboot the system into a clean environment (e.g. from a bootable CD-ROM which is not infected by malware). Once running a kernel which does not enforce digital signatures, the file system can be modified at will by the administrator. The technique for allowing unverified modifications can also be used to recover from a denial of service (discussed above).

**5. Raw Disk Writes** For binaries to remain secure on disk, the kernel must restrict raw disk writes. Raw disk writes are mainly useful during system initialization (e.g. partitioning, formatting) and disaster recovery (e.g. `fsck` or `scandisk`). As disabling raw disk writes on a system also prevents other attacks [19], we view limiting writes as an acceptable solution.

**6. Aliases.** Our mechanism only protects the binaries on disk. If malware can prevent the correct binary on disk from being invoked, then it retains a measure of control over previously installed programs. As an example, running the `ps` command from the prompt without a pre-pended path (i.e., fully qualified filename) will cause the first copy of `ps` found to be run (even though it may not be the `/bin/ps` binary). While our scheme is designed primarily to protect binaries against modification, these binaries are of no use if they are not used. We must ensure therefore on an infected system that the legitimate binary can be easily run instead of one found at a location of the attacker's choice. Additional copies of binaries installed by malware, however, can be avoided during forensic analysis by running applications from the trusted base directly, avoiding the environment. There are a number of methods for accomplishing this, including calling the kernel directly (e.g. using the `execve` system call) to run a program. Because much of the aliasing functionality is implemented by libraries likely to be protected by our scheme, the implementation could also be adjusted to avoid pitfalls during forensic analysis (e.g., restricting `PATH` during forensic analysis to include only core directories or not allowing aliasing on critical programs). Additional difficulties arise from operating system features such as the ability to operate within a *chroot* environment or mount/unmount filesystems. It is not clear what the best approach is to solving *chroot* and mounting problems.

# 5 Concluding Remarks

If widely deployed, it is likely that the method will be challenged by malware. Therefore, an important question to ask is: What is the next step of malware, assuming knowledge of our method of file protection? Currently, as mentioned, our trust in the kernel leaves the proposal open to attack in most systems (but as noted, we expect this to change in the future). Another issue is aliasing, as discussed above.

Our protection method does nothing to prevent malware from being installed on the system, but rather restricts the files that the malware can modify. By having a core trusted application space remain after infection, other advantages present themselves. Anti-virus and host based intrusion detection systems installed can attempt to detect malware without having to worry about being subverted themselves.

One important question we leave open is the ability to kill applications that are running on the system. Current Unix kernels allow any program running with superuser privileges to kill any other program on the system. In the event of malware infection, malware could kill any trusted program which attempts to run, preventing it from completing. One possible solution is to have the kernel confirm with the program to be killed whether the kill signal should be honoured (assuming the kernel is trusted, this method could not be subverted). This would allow programs such as anti-virus checkers to remain running on an infected system. A drawback of this is that it results in some programs on a system which can not be killed.

While simple, we believe our scheme provides sig-

nificant progress in being able to trust installed applications on an infected system. It protects applications from being modified directly by malware, allowing programs present on a system before infection to remain unmodified. It furthermore does not restrict user choice; new applications can be installed at will.

We have concentrated in this note on solving a small piece of the malware problem, proposing a simple solution to maintaining a flexible base of software which can be trusted after system compromise. While many activities of malware are not prevented by our proposal, it preserves a trusted base, allowing additional research to focus on the detection of malware while benefiting from file integrity protection. Our trusted base is immediately beneficial for current anti-virus and host-based intrusion detection systems. In the event of system compromise by malware, our proposal allows partial forensic analysis using the trusted base without incurring significant downtime – contrary to traditional forensic analysis.

**Acknowledgements.** The first author acknowledges NSERC for funding his PGS D scholarship. The second author acknowledges NSERC for an NSERC Discovery Grant and his Canada Research Chair in Network and Software Security. Both authors also acknowledge MITACS. We thank Lionel Litty and anonymous referees for their comments on a preliminary draft.

# References

[1] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, and V. Roy. Digsig: Run-time authentication of binaries at kernel level. In *Proc. LISA '04: Eighteenth Systems Administration Conference*, pages 59–66, 2004.

[2] A. Baliga, X. Chen, and L. Iftode. Paladin: Automated detection and containment of rootkit attacks. Technical Report DCS-TR-593, Rutgers University Department of Computer Science, January 2006.

[3] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *Proc. 2007 IEEE Symposium on Security and Privacy*, pages 246–251, May 2007.

[4] bsign. Web site. `http://packages.debian.org/stable/admin/bsign.html`.

[5] J. Collake. Hacking Windows file protection. Web Page, May 2007. `http://www.bitsum.com/aboutwfp.asp`.

[6] M. Conover. Analysis of the Windows Vista security model. Technical report, Symantec Corp., 2006. `http://www.symantec.com/avcenter/reference/Windows_Vista_Security_Model_Analysis.pdf`.

[7] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. 2003 Network and Distributed Systems Security Symposium*, pages 191–206. Internet Society, February 2003.

[8] J. Y. Halpern and R. van der Meyden. A logical reconstruction of SPKI. *Journal of Computer Security*, 11(4), January 2004.

[9] A. Huang. *Hacking the Xbox*. No Starch Press, Inc., 2003.

[10] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 91–104. ACM Press, 2005.

[11] G. H. Kim and E. H. Spafford. Experiences with Tripwire: Using integrity checkers for intrusion detection. Technical Report CSD-TR-93-071, Purdue University, 1993.

[12] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[13] G. Kroah-Hartman. Signed kernel modules. *Linux Journal*, 117:48–53, January 2004.

[14] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proc. 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 91–100, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Microsoft. Description of the Windows file protection feature. Web Page, May 2007. `http://support.microsoft.com/kb/222193`.

[16] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proc. 12th USENIX Security Symposium*, pages 137–151, August 2003.

[17] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proc. 13th USENIX Security Symposium*, pages 179–194, August 2004.

[18] N. L. Petroni Jr., T. Fraser, A. Walters, and W. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proc. 15th USENIX Security Symposium*, pages 289–304, August 2006.

[19] J. Rutkowska. Subverting Vista kernel for fun and profit. Blackhat Presentation, August 2006. `http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf`.

[20] L. van Doorn, G. Ballintign, and W. A. Arbaugh. Signed executables for linux. Technical Report CS-TR-4259, University of Maryland, 2001.

[21] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 368–377, June 2005.

[22] X. Zhao, K. Borders, and A. Prakash. Towards protecting sensitive files in a compromised system. In *Proc. Third IEEE International Security in Storage Workshop (SISW'05)*, pages 21–28, 2005.