An Architecture for On-the-fly File Integrity Checking

Mauro Borchardt, Carlos Maziero, and Edgard Jamhour

Graduate Program in Applied Computer Science
Pontifical Catholic University of Paraná
80.215-901 Curitiba – Brazil
Phone/fax +55 41 330 1669
{borchardt,maziero,jamhour}@ppgia.pucpr.br

Abstract. There are several ways for an intruder to obtain access to a remote computing system, such as exploiting program vulnerabilities, stealing passwords, and so. The intruder can modify system utilities in order to hide his/her presence and to guarantee an open backdoor to the system. Many techniques have been proposed to detect unauthorized file modifications, but they usually work off-line and thus detect file modifications only when the system is already compromised. This paper presents an architecture to deal with this kind of problem. Through the combined use of digital signature techniques and system call interceptions, it allows for transparent on-the-fly integrity check of files in Unix systems. Its evaluation in real-world situations validates the approach, by showing overheads under 10% for most situations.

1. Introduction

There are several ways for an intruder to obtain access to a remote computing system, such as exploiting program vulnerabilities, stealing passwords, hijacking network connections, and so on. After getting in the system, the intruder can exploit local vulnerabilities to get administrative privileges.

Once having the full control of the system, the intruder can modify some system utilities, in order to hide his/her presence, and to guarantee an open backdoor to that system. System utilities can be substituted by hacked versions that dissimulate the intruder's presence, hiding files, processes, network connections, and other system resources. The set of hacked utilities is known as a *rootkit*; sophisticated *rootkits* are hard to detect and may include changes in the operating system kernel itself [4].

Alongside with the efforts to minimize system bugs allowing root compromise, some work is done in detecting and/or preventing modifications of system files. Most known approaches work off-line, by periodically checking the system files' properties against previously stored values. One of the most known tools using this approach is *Tripwire* [11]. Although it can detect file violations, it is not able to prevent the use of the modified files, and the intruder may work for a while before being detected.

This paper presents an approach allowing to detect modified files and to prevent their usage as soon as possible. This is achieved through the joint usage of digital signatures and system call interceptions. The proposed architecture allows to transparently and quickly verify the integrity of any kind of file at its opening, including executables, libraries, scripts, and data files. Experimentation results validate the approach, by showing overheads under 10% for most situations.

The paper is structured as follows: section 2 introduces system call interception techniques; in section 3 the proposed architecture is detailed; in section 4 the performance results obtained with the prototype are presented and discussed; section 5 discusses the current status of the architecture and outlines future works; section 6 presents some related work, and section 7 concludes the paper.

2. System Calls Interception

To access operating system resources, such as files or sockets, processes make use of *system calls*, which are functions providing a controlled interface to the operating system kernel. The set of syscalls offered by the kernel constitutes its *API* (*Application Programming Interface*). The kernel API defines a clear separation between the specification and the implementation of the kernel services. This separation allows to transparently modifying the implementation of a given service: as long as the kernel API remains the same, all changes in underlying kernel services can be done with no need to modify the user-level code.

Extending basic kernel services can be done by modifying the kernel functions that implement the service to be extended, or by capturing the corresponding syscalls and transferring control to a code that implements the new feature [10,18]. Several architectures have been proposed to facilitate building and using kernel extensions, through plug-in structures and specific APIs. Some projects in this direction are *SPIN* [1], *Exokernel* [7], and *SLIC* [8].

There are several uses for kernel extensions, like process debugging, process migration between hosts, file systems extensions, and host-based intrusion detection systems, among others. A well-known example is the *Virtual File System* (VFS) [9], which allows for applications to transparently use multiple distinct media and file systems.

3. System Architecture

The goal of the system presented here is the dynamic file integrity checking. Before files are opened by processes, their integrity is checked using a digital signature[1] schema, which is activated by a syscall interception. All files to be monitored should have been associated a digital signature, which is verified by the operating system kernel just before opening them. If the signature generated from the actual file

[1] The *Digital Signature Standard* (DSS) [6] is the digital signature schema adopted in this work. The DSS aspect considered here concerns the ability to certify the integrity of a digital document, through an encrypted hash obtained from it. This encrypted hash is called a “digital signature” of the document.

contents matches the corresponding signature stored in the database, the file can be open (i.e. the original system call can proceed), otherwise the access is denied.

Starting from a list of files to monitor, a digital signature database should be built. Database entries are formed by the file paths and their respective digital signatures. At first, only static resources residing in the local file systems should be registered in the database: executable files, libraries and configuration files. The signature database should be built from the system in a known safe state. The best way to guarantee this is using a freshly installed system, avoiding the presence of intruder codes as *rootkits*, *viruses*, *backdoors*, and *trojans*.

Performance is a major concern in this approach. The signature checking procedure can be costly, especially on large files, and imposes a considerable performance penalty. To solve this problem, a *validation cache* is defined (section 3.3). It stores references to files already checked, to avoid repeating integrity checks on the same files. This cache has proven to be essential in this approach (section 4).

For the integrity checking mechanism to be transparent, the file system API should be respected. Also, the checking should be imposed on any file access, with no possibility to be circumvented. Thus, the system should be implemented under the kernel API. Considering the generic *POSIX* API, there are two system calls related to file opening that should be intercepted: `execve`, responsible for reading and starting an executable image from disk, and `open`, to open files for reading and writing.

The proposed architecture is presented in figure 1 and will be detailed in the following sections.

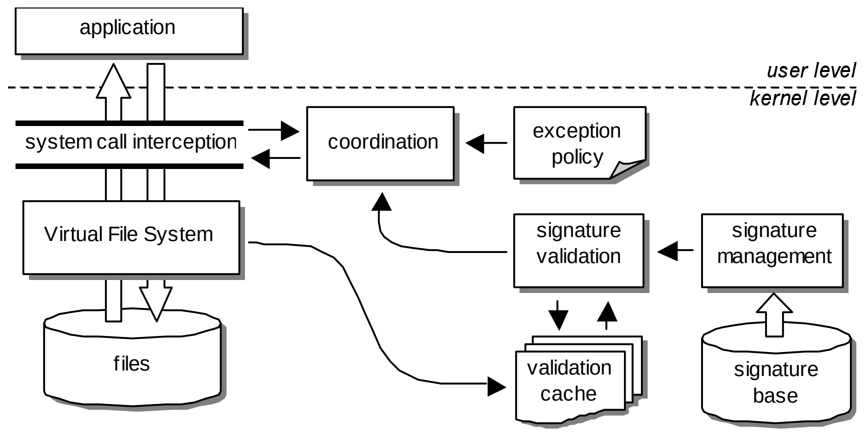

Fig. 1. The proposed architecture

3.1 System Call Interception Module

The **System Call Interception Module** intercepts the `open` syscall, gets the complete file name, and redirects the control flow to the coordination module, for signature checking. If the coordination module returns ok, the original system call is allowed to continue, otherwise the file access is denied. The `open` system call is also used to open directories and devices, but these are excluded from the integrity checking.

The `execve` syscall is responsible for opening an image file and preparing the kernel structures, in order to start the code execution. Although the file is also opened, this system call doesn't use the `open` syscall code, but makes use of other kernel functions. Thus, the `execve` syscall is also intercepted by this module. Depending on the operating system, other system calls should be intercepted as well.

3.2 Coordination Module

This module receives the name of the file to check and tries to open it using the same privilege level of the process that made the system call. A lookup in the validation cache is performed, using the pair [*i-node; device number*] got from the file as a key. If the file to be open has an entry in the cache, its signature was already validated before and the file contents did not change after that. This result can be returned immediately to the syscall interception module (section 3.1).

In case the file is not referenced in the validation cache, its signature should be verified. The **Signature Verification Module** is then activated, in order to check the integrity of the file using its previously stored digital signature. If the actual file signature agrees with the previously stored one, the file is valid, and a corresponding entry is added to the cache. Otherwise, an error is returned to the syscall interception module. If the file to be open has no associated signature, the **Exception Policy Module** will be activated, to decide whether to allow or to deny that access.

3.3 Validation Cache Module

As the signature verification is time-consuming, it is important to avoid repeating unnecessary verifications. This is achieved using a cache of validated files. The signature verification costs would turn unviable this proposal if there was not such a cache, as demonstrated by the results presented in the section 4. This module implements three operations on the cache: *insert*, *delete*, and *lookup* of files entries.

One main aspect in the cache management concerns entry deletions. An entry should be deleted from the cache if, and only if, the corresponding file is modified or removed. Some mechanisms from the VFS (Virtual File System) abstraction, present in most Unix systems, are used for detecting such events. In the implementation presented here, a hook was put in the VFS function `mark_inode_dirty`, which is called whenever a file is modified or removed.

3.4 Signature Validation Module

This module makes use of a digital signature algorithm to check the integrity of a file. For its application the module uses the digital signature of the file, obtained from the signature database, the hashing value of the current file content and the public key of the signature database. The access to this public key is a critical issue and will be discussed in section 5.

Before calculating the hash value for the file being open, its digital signature should be requested to the **Signature Management Module**. The hash value for the file is then calculated, using as input the following information: file contents, i-node number, device, size, owner, group, permissions, creation date, and last modification date. The simultaneous application of file attributes in the hash function simplifies the attribute verification procedure.

The digital signature standards adopted were SHA-1 [5] for the hash generation procedure, and FIPS 186-2 ECDSA [6] for the encryption procedure. The FIPS 186-2 ECDSA algorithm was chosen for its smaller execution time, smaller memory footprint, and smaller keys for the same protection level when compared to DSA [12].

3.5 Signature Management Module

This module was included in the architecture to provide transparency on the signature database location. Basically only two parameters are needed to uniquely identify a file: its complete path and the name of the host where it is stored. Using them, this module retrieves the signature value from the database and returns it to the signature validation module. The digital signature database can then be located anywhere, from a file in the local system to a remote LDAP server. Details regarding its implementation issues are discussed in section 5.

3.6 Exception Policy Module

It is not worthwhile to maintain up-to-date signatures for files whose contents are always changing, like log files and mailboxes. The **Exception Policy Module** was included in this proposal to implement policies to deal with these exceptions, allowing to define unsigned directories and files.

The coordination module activates it in order to verify whether a requested operation is allowed or not, according to policies defined by a rule set. The current implementation uses simple rules based on files, directories, users, groups, and operations (`open`/`execve`).

4. Implementation and Evaluation

The prototype implementation was done using a Linux kernel version 2.4.4; all the modules described here were implemented, some of them in a simplified way. The tests were run on a Pentium II 233MHz system, with 128 Mbytes of RAM and an IDE ultra-DMA33 hard disk (seek time around 13 ms).

Three distinct situations were chosen to evaluate the impact of the architecture: compilation, file compression, and user application. They have very different behaviors regarding resource usage (CPU, memory, I/O) and represent frequent situations. Due to the validation cache, subsequent accesses to a file are much faster than its first access. To measure this effect, each test was run in three situations: a) without the use of signature verification (the *reference system*), b) a first execution after system initialization (the validation cache is empty) and c) a second execution after system initialization (the validation cache has entries generated by the first execution). Each test was run several times, and the results obtained are stable.

To simplify the analysis of results, a "dummy" exception policy was adopted, allowing any file to be accessed, even if is not signed or if it's signature is not valid. For the tests, all files were signed, excepting temporary ones (in `/tmp` and `/var/log` directories).

4.1 Compilation

This test consisted on compiling the Linux kernel source (version 2.4.4, with standard parameters: `make bzImage modules`). The source tree occupies 135 Mbytes on disk and has 8875 files, but only 2051 of them were read in the compilation. The table 1 presents the results obtained.

Table 1. Compilation test results

Measurement	**Reference system**	**1st execution**		**2nd execution**	
#`open` operations	115494	115494		115494	
#`open` cache hit	—	113424	98.2%	114661	99.3%
#`open` cache miss	—	2070	1.8%	833	0.7%
#`execve` operations	2344	2344		2344	
#`execve` cache hit	—	2326	99.2%	2340	99.9%
#`execve` cache miss	—	18	0.8%	4	0.1%
user time (s)	663.12	664.80		662.24	
system time (s)	41.42	134.70		90.01	
total time (s)	704.54	799.50		752.25	
overhead (%)	—	13.47%		6.77%	

The meaning of the table fields are:

- **#`open`** (**#`execve`**): total number of syscalls invoked by the process(es).
- **#`open` cache hit** (**#`execve` cache hit**): number of syscalls in which the file was found in the cache.
- **#`open` cache miss** (**#`execve` cache miss**): number of syscalls in which the file was not found in the cache, forcing a complete signature verification.
- **User time** (s): time spent by the process(es) at user level.
- **System time** (s): time spent by the process(es) at kernel level.
- **Total time** (s): total time spent by the process(es).
- **Overhead** (%): the execution time compared to the reference system time.

4.2 File Compression

This test consisted on applying a `tar -czf` command on the Linux kernel source tree. This command applies both data compression (using the `gzip` tool) and agglutination of the files to generate an archive containing all the source files compressed. The obtained results are presented in table 2.

Table 2. Compression test results

Measurement	**Reference system**	**1st execution**		**2nd execution**	
#`open` operations	8876	8876		8876	
#`open` cache hit	—	12	0.14%	8876	100%
#`open` cache miss	—	8864	99.86%	0	0%
#`execve` operations	2	2		2	
#`execve` cache hit	—	0	0%	2	100%
#`execve` cache miss	—	2	100%	0	0%
user time (s)	55.28	55.75		55.27	
system time (s)	2.56	293.33		3.31	
total time (s)	57.84	349.08		58.58	
overhead (%)	—	503.5%		1.28%	

4.3 Execution

This experiment used the **`gv`** application, a popular PostScript file viewer. It consisted on opening a 40-pages PostScript file, quickly viewing all pages and closing it. The obtained results are shown on table 3.

Table 3. Execution test results

Measurement	**Reference system**	**1st execution**		**2nd execution**	
#`open` operations	117	117		117	
#`open` cache hit	—	46	39.3%	117	100%
#`open` cache miss	—	71	60.7%	0	0%
#`execve` operations	2	2		2	
#`execve` cache hit	—	0	0%	2	100%
#`execve` cache miss	—	2	100%	0	0%
user time (s)	20.56	20.83		20.59	
system time (s)	1.52	11.31		1.57	
total time (s)	22.08	32.14		22.16	
overhead (%)	—	45.56%		0,36%	

The results got from this test should not be considered as-is. Their goal is just to give a feeling about the performance impact of the system, from an ordinary user perspective. For a more precise numeric evaluation of this test case, one would need to measure the time spent by the graphical interface processes, the I/O devices, and the user latency on controlling the application.

4.4 Result Analysis

The signature verification procedure imposes a significant computing overhead, as shown by the differences between the first execution and the reference system execution. The first execution can be very time-consuming, because the validation cache is barely used. This is perceptible in the compression experiment: the first execution takes 503% more time than the reference system. This bad performance occurs because each file being compressed is accessed only once (cache miss ratio near to 100%). The validation cache influence is visible on the second execution of this experiment, which states an overhead of only 1.28% and cache hit ratios of 100%. This allows concluding that the overhead is around 1% if the cache is fully used.

On the other hand, the compilation experiment shows a much better picture for its first execution: only 13% overhead. This is due to the way compilation is done: header files are often included on several different files, generating lots of accesses on them. This can be seen on table 1: although 115494 `open` syscalls were performed, they generated only 2070 cache misses. The same can be concluded on the `execve` side: few tools (assembler, compiler, linker, etc) are extensively called, generating high cache hit ratios. Here, the second execution shows results not so good as in the compression experiment. This is due to the fact that the compilation modifies configuration scripts and object files that should be checked again.

The execution experiment shows that the performance impact for an average interactive user is acceptable. Although the first execution presents an overhead around 45%, the second execution overhead is near zero. In a multi-user environment, only the first user of an application will suffer from the first execution overhead. In this case, subsequent users will notice no overhead.

5. Current Status

Presently, the system is able to check file signatures and to allow/deny access to files depending on their signatures. This makes it useable in servers for common services like e-mail, web, proxy, and file sharing. Considerable work remains to be done in the signature generation and management aspects, before the system is ready for production multi-user systems. Some points for future work are outlined here.

As results show, the first access to a file is time-consuming. For large applications, this means an uncomfortable starting delay for its first user. For network service daemons, such response delay can leads a network client to time-out. This problem can be minimized by “proactively” populating the validation cache (by running a routine to check all files registered in the signature base, in moments of low system activity).

The system public key needs to be accessible to the kernel during the system initialization. It can be passed as a kernel boot parameter, made available from a read-only external media, or even be fetched from a remote key server. The public key should not be available for write access during system operation. If the public key is modifiable, the signature validation mechanism can be bypassed by tampering the signature base and re-signing all registered files.

Presuming that the public key is kept safe from tampering, the signature base file can be maintained on the local file system, protected from access by normal users. It does no need to be encrypted, as it is also signed and will be verified like other files. This is also valid for the exception rules file. In the case one of these control files is corrupted or removed, access to the signed files will be denied. As a consequence, the system will be useless, but preserving its integrity. A better solution would be to store these files in a remote server, to be fetch during system initialization.

User files are frequently modified. The file signature base would need to be frequently adjusted to reflect changes in those files, but this can be unfeasible if the signature base is a static and protected resource. One solution to this problem is to create exception rules for user directories, as stated in section 3.6. However, allowing file execution in user directories can open security breaches. A safer approach would be to allow trusted users to sign their executable files, using personal keys. This can be achieved by creating a local Public Key Infrastructure (PKI). The trusted users' keys should be certified by the system public key, and would be used to generate signatures for their files, which are stored in personal signature bases.

6. Related Work

The most known work in the domain on file integrity check is the *Tripwire* system [11], which inspired this work. It is a user-level application operating in batch mode (usually when the system load is low). Tripwire is strong in detecting file modifications, but is unable to prevent them. The *Bsign* project [17] is similar to Tripwire, but applies only to executable files in ELF format. It stores each file signature in the executable file itself (ELF files support "sections" in which particular data can be stored). The SOFFIC system [16] and the system presented in [2] for the NetBSD operating system have a similar approach to the system presented here, but at this moment there are no concrete results or analysis widely available about them.

The idea of signing files and checking their signatures on demand is not new and can be applied to different domains. The work presented in [15] uses this approach to validate web pages by the web server, as they are requested by remote clients. Their goal is to prevent delivering hacked pages to the Internet.

7. Conclusion

In this paper we present a software architecture to improve the security of a computing system using file integrity checking. It does not prevent a system to be intruded, but denies access to tampered files or to files dropped in protected areas. The proposed approach makes use of digital signature and system call interception techniques. Although it was implemented on an UNIX-like operating system, it can be adapted to other operating systems.

The performance results presented show that the proposed approach is worthwhile in real situations. Once the validation cache has enough information, the overhead imposed on the applications considered is under 10%.

The proposal is not complete, as it lacks features to improve user key management. Some possibilities to improve it were enumerated, but there are other approaches to consider. We are currently studying aspects related to user key management using a distributed PKI, and defining strategies in the case of network outage.

References

1. B. Bershad, C. Chambers, S. Eggers et al. SPIN: An Extensible Microkernel for Application-specific Operating System Services. In *Proc. 6th ACM SIGOPS Workshop on Matching Operating Systems to Application's Needs*, Warden, Germany, Sep 1994.
2. B. Lymn. *Verified Executables for NetBSD*. `http://members.optusnet.com.au/~blymn`, Nov 2002.
3. CERT Coordination Center. `http://www.cert.org`, Nov 2002.
4. ChkRootKit.org. `http://www.chkrootkit.org`, Nov 2002.
5. FIPS 180-1. *Secure Hash Standard*. NIST, Apr 1995.
6. FIPS 186-2. *Digital Signature Standard*. NIST, Jan 2000.
7. D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles*, 1995.
8. D. Ghormley, D. Petrou, S. Rodrigues, and T. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proc. USENIX 1998 Annual Technical Conference*, pages 15-19, Berkeley, USA, Jun 1998.
9. J. Heidemann and G. Popek. *A Layered Approach to File System Development*. University of California, Los Angeles, Technical Report, CSD-910007, 1991.
10. M. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proc. 14th ACM Symposium on Operating Systems Principles*, pages 80-93, Dec 1993.
11. G. H. Kim and E. H. Spafford. *The design and implementation of Tripwire: a file system integrity checker*. Tech. Report CSD-TR-93-071, Computer Science, Purdue Univ, 1993.
12. Menezes, P. Van Oorschot, S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1999.
13. Ronald L. Rivest. *The MD5 message-digest algorithm*. IETF RFC 1321, Apr 1991.
14. R. Rivest, A. Shamir, and L. Adleman. *A method for obtaining digital signatures and public key cryptosystems*. Communications of the ACM, 21(2): 120-126, 1978.
15. S. Sedaghat, J. Pieprzyk, and E. Vossough. *On-the-fly web content integrity checker boots user's confidence*. Communications of the ACM, 45(11):33-37, Nov 2002.
16. V. Serafim, A. Reguly. *SOFFIC – Secure On-the-Fly File Integrity Checker*. URL `http://www.inf.ufrgs.br/~gseg/projetos/soffic.shtml`, Nov 2002.
17. M. Singer. *The BSign Project*. `ftp://ftp.buici.com/pub/bsign`, Feb 2002.
18. C. Wright, C. Cowan, J. Morris, S. Smalley, G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proc. 11th USENIX Security Symposium*, San Francisco, CA, Aug 2002.