Computers & Security (2004) **23**, 52—62

Computers & Security

www.elsevier.com/locate/cose

Restraining and repairing file system damage through file integrity control

Vinícius da Silveira Serafim*, Raul Fernando Weber

PPGC—Instituto de Informática, Universidade Federal do Rio Grande do Sul, Av. Bento Gonçalves, 9500, Porto Alegre, RS, Brazil

Received 1 April 2003; revised 17 September 2003; accepted 24 September 2003

KEYWORDS
File integrity;
Survivable systems;
Intrusion tolerance;
Cryptographic hash;
Linux Kernel

Abstract Today, many security researches focus on survivable or intrusion-tolerant systems, which are not conceived to be invulnerable, but rather to be able to withstand the impact of an attack and continue providing critical services despite ongoing attacks. We join this fast-growing research community and use this article to present a solution of file integrity control which is able to detect unauthorized file system modifications as fast as possible and repair them in order to keep the system's integrity, availability and confidentiality.
© 2004 Elsevier Ltd. All rights reserved.

Introduction

According to Bishop (2003), "computer security is based on three properties: confidentiality, availability and integrity. Confidentiality is the concealment of information resources, availability means the ability to use the desired resource, and finally integrity refers to the trustworthiness of data or resources, and it is usually phrased in terms of preventing improper or unauthorized change." A lot of effort has been placed in the search of *highly* secure systems, which is clearly known as quite a complex goal to be fully achieved. Nevertheless, practice shows that many security managers and network administrators still follow this philosophy, which results in the formation of systems falsely considered secure. Fortunately, this philosophy has been progressively replaced by another one that fits reality much better: systems will eventually be intruded, so what should we do in order to warrant that critical services be provided even when intruded?

The concepts of survivable systems and intrusion tolerance stem out of this new vision. Formally, according to Pal et al. (2001), "attack survival means the ability to provide some level of service despite an ongoing attack by tolerating its impact." In order to tolerate the impact of intrusions, a system must have mechanisms able to hamper the advance of an attacker towards gaining privileges, and attempt to restrain damage to a single part of the system while still running its most critical functions.

As a result of maturity in this field, in which research began about six years ago, a growing

* Corresponding author.
E-mail addresses: serafim@inf.ufrgs.br (V. da Silveira Serafim), weber@inf.ufrgs.br (R.F. Weber).

number of contributions have been presented to the community. Such contributions comprise attempts to formalize concepts, strategies to develop survivable systems and even tools that can be employed to put the concepts and strategies to practice.

This paper presents our research, whose contribution is the conception and implementation of a tool able to aid substantially the making of an attack-tolerant system through file integrity control. Controlling file integrity is certainly not a new idea, let alone using cryptographic hash functions (Menezes et al., 1996; Schneier, 1996) to make the process easier. There are already various available solutions such as AIDE,[1] L6,[2] Integrity[3] and Tripwire (Kim and Spafford, 1993). The later is probably the most popular one. Even though it is not new, this minor point is of great importance in the context of survival systems and has not been adequately explored. This could be because the existing (or simply proposed) solutions are considered enough and, if not so, maybe the importance of file integrity control has been downplayed in favor of other mechanisms.

Therefore, it is not file integrity control itself that is new, but rather the way the process is carried out, featuring highly important aspects such as detection, attack restraint and self-healing. The absence of these features is often pointed out as the main weakness of the existing solutions, as Pal et al. (2001) and Bernaschi et al. (2002) make clear in their articles. The latter is explicit: "...research in this area is focused mainly on the detection of intrusions after attacks have been successfully completed, and not on their prevention."

It is clear that this present solution alone does not make a system intrusion-tolerant, but we are absolutely sure that it may be at least a very valuable resource to build survivable systems. It should be noted that the term system used here means a set of resources such as hosts and routers which are combined in order to make a certain organizational mission possible. Moreover, seeking greater integration with other ongoing studies in the field, we use the same notions of security domains documented in Pal et al. (2001). It can be composed of a single host or a local network, and may also include other equipment, such as routers and switches.

Each security domain should not trust a neighboring domain, e.g., the administrator of one domain must be authenticated every time it accesses another security domain. The purpose is to create difficulties to the action of an attacker by slowing down the speed at which it can gain access privileges on the system.

External attackers do not have direct access to the internal subsystems; rather, mechanisms such as firewalls restrict them to specific services, without full access to subsystems. So, their first targets are usually the subsystems on the system border and within easy reach. At first, attacks are necessarily remote, i.e., attackers still do not have any way to explore the existing local vulnerabilities, but only to find flaws that can be explored remotely and allow them to proceed with the attack.

Having gained some level of privilege in the security domain which the subsystem is a part of, attackers then move on to a second phase, in which they seek to gain the privileges needed to finish their attack, in case they still do not have them. At this stage, attackers are usually able to install some tools in the intruded subsystem, such as:

- exploits for local flaws that cannot be explored remotely, in order to gain more privileges;
- rootkits and backdoors in order to hide their files and processes, and to ensure their return even after flaws are possibly corrected;
- scanners, exploits for remote flaws and sniffers in order to attack neighboring subsystems.

Note that, as the previous item puts it, the intruded subsystem becomes a base for the direct attack to internal subsystems the attacker could not reach before (Fig. 1). Here, once again it is the presence of security domains that reinforces the obstacles an attacker must cross in order to damage the entire system.

The goal, then, is to slow down an attacker's action as long as possible in order to allow other mechanisms, which seek to ensure the survival of critical parts of the system, to have a chance to react by adapting the environment of an application

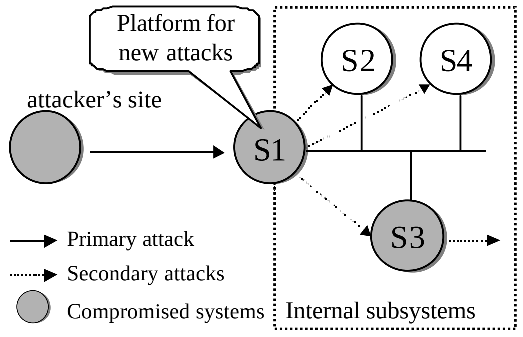

Figure 1 Attack progression.

[1] http://www.cs.tut.fi/~rammer/aide.html.
[2] http://www.pgci.ca/l6.html.
[3] http://integrit.sourceforge.net.

or even the application itself so that it runs despite an ongoing attack. One of the possible approaches to achieve this goal is to block the use of violated or malicious files (e.g. rootkits).

In this article we suggest the use of file integrity checking mechanisms to identify these kinds of files before they are used. The remainder of the article is organized as follows: section 'Why and how to control file integrity?' discusses the importance of file integrity control and presents some concepts and the critical points of the process; section 'Secure on-the-fly file integrity checker (SOFFIC)' presents details of the conception of the tool; section 'Implementation' has some data concerning the implementation and the first performance tests. The last section presents the conclusions of this present study.

Why and how to control file integrity?

All the codes of any operating system and all of its applications, as well as configuration files, data bases and activity logs, are always kept in some non-volatile memory, which is organized according to a file system. It does not take a lot of effort to notice the clear interdependence between the behavior of an application and its data (code, configuration files, etc.) stored in disks. By altering the latter, the whole system can be affected, including access controls implemented by the operating system and high level applications.

Not only file content alteration may lead the system to an insecure, even non-functional state, but also the insertion of new files, e.g., Trojan horses (backdoors, rootkits), tools that explore local vulnerabilities, attack tools (sniffers, DDoS zombies, scanners), etc.

Even though one of the main pillars of system security is the security of its file system, none of the current operating systems implements access control based on the integrity of the files to be used, being limited to controlling access based on the identification of users and their rights.

Having briefly explained the role of the file system in the security of such system, perhaps an unnecessary step for readers who are more familiar with the area, we now focus on the process of file integrity control itself, attempting to show the reader some of its most critical points, and then providing the means needed to understand the premises of our tool fully.

To control the integrity of a given object means to be able to warrant that it has exactly the same characteristics at a certain time T_1 that it had at a previous time T_0, when the object was considered valid. From that, the most obvious solution to control file integrity is to make and store a copy of the files at moment T_0, then later, at T_1, compare them to the files present in the system. This is called object reconciliation. However efficient, such procedure has some negative points that can make its application more difficult:

- verifying integrity by comparing each byte of each file would cost quite some time, thus making it not practical to use this tool too frequently under penalty of harming system performance; and
- the use of compression algorithms aiming to save storage space would require even more time to run the verification process.

At this point of the study, cryptographic hash algorithms[4] (Menezes et al., 1996) become elements that are easily used to store the characteristics of a number of files at time T_0 in a very fast and compressed manner. It is not our goal here to describe hash algorithms in detail, so we will only mention their main features which are essential to a solution of file integrity control. According to Menezes et al. (1996), they are:

- the calculation of a cryptographic hash does not use complex operations, being fast and easy to implement;
- the resulting hash H will always be the same size regardless of amount and content of the mass of information (the most usual sizes are 128 bits and 160 bits, respectively, for algorithms MD5 and SHA-1);
- cryptographic hash algorithms have a property known as *resistance to collisions*, i.e., it is computationally difficult to obtain two masses of information that have the same hash H value.

To verify integrity at T_1, hash is calculated again (H') for every file in the system, and then compared to that value H generated at T_0.

This method allows for a gain in speed and storage space; however, the possibility of repairing violated system files is lost since it is not possible to recover the original data of a file from its hash H. The solution to this problem is not new and it is called backup.

The startup process, run at T_0, to use a file integrity control mechanism can be briefly seen in Fig. 2. The complete process for file integrity control (Fig. 3) is much more complex than the startup

[4] Also called *message digest*.

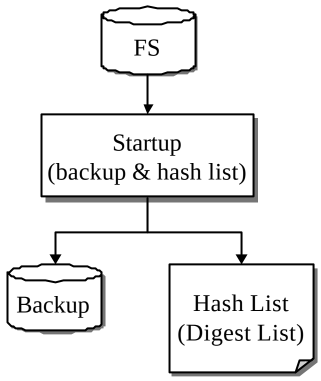

Figure 2 Startup process for file integrity control.

process. In fact, even this can be rather complex given certain situations.

Time T_0, when the backup and hash list are generated, is crucial to ensure successful integrity control because this is when the parameters to verify and recover violated files are set—the basis of the entire process. If the system is already corrupted at time T_0, the whole process becomes useless, since any changes performed by an attacker would be consolidated and accepted as valid.

The most appropriate T_0 for this first step is right after installation of the system because, if performed in an appropriate environment, no malicious agent will have the opportunity to corrupt any part of the system at this critical moment for file integrity control. An appropriate environment is one where:

- there is control of the physical access to the equipment where installation is taking place;
- there is not any possibility of remote access to the system, whether by authorized users or not;
- all programs to be installed have had their origin and integrity verified;
- after installation, the system is scanned by programs to detect malicious code, such as antivirus programs.

In the very frequent cases where the system is already in operation, the difficulty to obtain high reliability in its integrity is considerably increased. In such cases, one must consider the security and monitoring levels implemented in the system since its installation, as well as its security incidents history and how they have been held.

It must be clear that security is not necessary only when creating a backup and a corresponding hash list, but also when these two elements are stored. This is so because if attackers have a chance to cause changes in these elements, they may prevent verification and restoration processes, and make updates so as to validate their own changes to the file system.

The rest of the process (Fig. 3) is triggered regularly, when file integrity is verified. Once one or more file changes are detected, they must be classified as either authorized or unauthorized (potentially malicious). Such classification strongly depends on the knowledge of the system administrator about its configurations and characteristics. Only after this classification is it possible to decide correctly about the following steps, which may result in triggering an incident response process, leading to file restoration and system updates as well as backup and hash list updates.

The time interval between two integrity checks is the time an attacker has to act undetected. As a consequence, this interval and security are inversely proportional, i.e., the longer the interval, the weaker the security provided.

Therefore, when the goal is to reach a high security level, one tends to reduce the time interval to the minimum necessary. Unfortunately, such practice threatens the performance of the system as a whole, and may delay the services provided to users, perhaps even causing the integrity control mechanism to be turned off. Hence, performance is an important factor to be considered when modeling and implementing any file integrity control mechanism.

As the verification interval increases, the loss of performance becomes more acceptable,

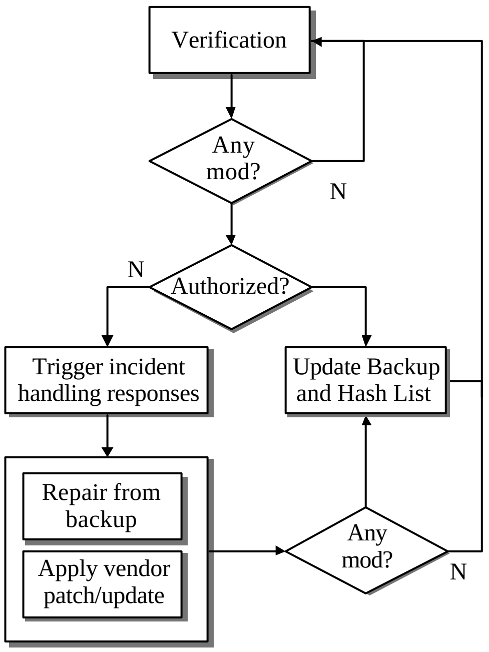
Verification
Any mod?
N
N
Authorized?
Trigger incident handling responses
Update Backup and Hash List
Repair from backup
Apply vendor patch/update
Any mod?
N

Figure 3 Complete file integrity control process.

i.e., when time intervals between verifications are greater, any performance loss is temporary and should not cause damage to the routine operations of the system. However, by giving attackers more time to act undetected, the system security can be compromised.

The loss of performance and security provided are not determined only by the time interval between verifications, but also by the manner in which the mechanism is implemented and the number of monitored files.

The solution presented in section 'Implementation', instead of minimizing the time between verifications, uses another strategy in order to ensure high level of security regarding file integrity, though strongly reducing its impact on system performance.

Secure on-the-fly file integrity checker (SOFFIC)

The first concept of SOFFIC was based on three main goals:

- not to allow *the use* of files that have been altered or included in the system without authorization;
- to impact the least possible on system performance, thus making the solution applicable; and
- additional cryptography-based mechanisms must be used to ensure security of the solution itself.

When the first implementation and subsequent deeper studies were carried out, another objective of great importance was defined:

- possibility of automatically repairing the system after a violation is detected (self-healing).

It is important to notice this because it is not practical to monitor the integrity of *all* files constantly, the strategy chosen is to avoid the altered or unduly inserted files to be used. Thus, even after gaining the privileges required for installation, attackers cannot run rootkits, worms, viruses, DDoS zombies or even their tools to attack adjacent systems. The result is that attacks are prevented—if not totally, at least partially to a point of substantially hampering the action of attackers.

An attacker might explore this functionality of SOFFIC to cause a denial of service by altering files considered critical. As a consequence of this undue change, SOFFIC would deny access to these files until the manager restored them. This situation is avoided by complying with the fourth goal above, i.e., by giving SOFFIC the ability to repair violated files automatically. So, when detecting a violated file, SOFFIC first attempts to restore this file from a backup and then actually blocks access to it in case restoration fails.

Architecture

Even though it might slightly overlap with section 'Implementation', which deals specifically with the implementation of SOFFIC, it is necessary to number its components and their corresponding descriptions in order to understand the conceived model fully. These components can be seen in Fig. 4. Some of them are showed twice only to represent situations where they are on disk or loaded in memory.

The next figure shows, from top down, the file system and the following components:

- Configuration File: file that contains SOFFIC's configuration options, such as the IP number of the ISBB Server, location of HL and TFL, etc.;
- Hash List (HL), or Digest List: it has the hash values for each file that SOFFIC controls;
- Trusted Files List (TFL): a list with the names of files that can be accessed without any verification; and
- Integrated Secure Backup Base (ISBB): backup of system files available to SOFFIC to restore violated files.

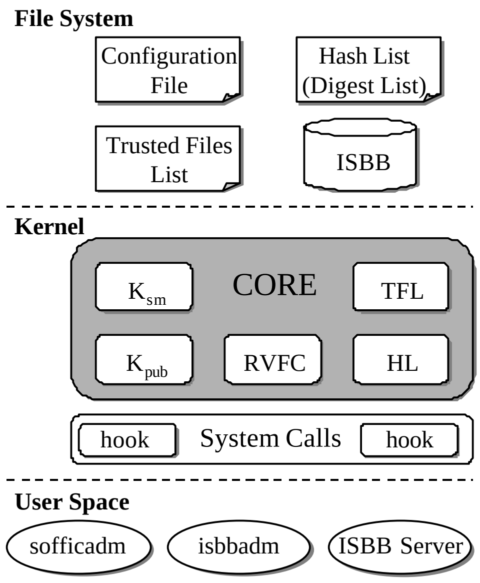

Figure 4 SOFFIC components.

Still moving downwards in Fig. 4, the components inserted in the system kernel are:

- Core: the nucleus of SOFFIC is the most important component, containing all routines to verify integrity, digital signatures, etc.;
- HL and TFL: only a copy of the contents in the corresponding files as described previously;
- Recently Verified Files Cache (RVFC): this cache avoids repetitive verification of often used files;
- Symmetric Key (K_{sm}): a symmetric key used to make some contents of SOFFIC secret;
- Public Key (K_{pub}): SOFFIC manager's public key, used to verify signatures. The manager keeps private key K_{pri}; and
- Hooks: slight changes in system calls that redirect the execution flow to SOFFIC.

The last part of Fig. 4 shows the components that run on user space:

- sofficadm: a tool used to generate and update the Hash and Trusted Files Lists, and other managerial tasks related to SOFFIC;
- isbbadm: tool used to generate and update ISBB, and other managerial tasks related to it; and
- ISBB Server: server that recovers files from the backup base as requested by SOFFIC, which is fundamental for self-healing.

Functionality

After this brief explanation of the components, it is possible to take a closer look at the operation of SOFFIC. Assuming that HL, TFL, and optionally ISBB have been generated, the starting point of our explanation can be a process *P* that requests the system to run a file (Fig. 5).

Upon that request, the operating system applies the Standard Access Control Mechanisms (SACM) and decides whether to grant access. In case it is granted, a hook on the system call redirects the execution flow onto SOFFIC, allowing it to make a final decision about the access requested by process *P*. On its turn, SOFFIC first looks for an entry in RVFC referring the requested file. If found, which means the file has been verified recently, access is immediately granted; otherwise, the HL is consulted (Fig. 6).

Once an entry is found in the HL, file integrity is then verified by calculating its hash *H′* and comparing it to hash *H* stored in the HL. If they are equal, file integrity is verified, then an entry is included in RVFC and the file can be accessed.

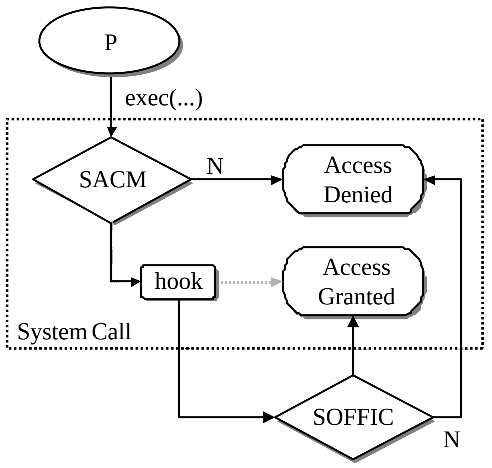

Figure 5 Modified system call.

Otherwise, the file being not valid, SOFFIC tries to restore it from the ISBB.[5]

The last resource for SOFFIC, when there are no entries either in RVFC or HL for the requested file, is the TFL. The TFL was conceived to deal with exceptions such as the /tmp directory and its files, i.e., it can be used to decide which files and directories can be accessed for reading or writing[6] without any verification. If the requested file is in the TFL, access is granted, but no entry is included in RVFC.

Even though the TFL is necessary, it must be noted that the way it operates makes it a critical component to the security of SOFFIC when implemented and configured.

Finally, if no entry for the file *P* requested is found either in RVFC, HL or TFL, access is denied.

SOFFIC's policy is to deny access to any file that has not been previously inserted in any of its lists, whether the HL or the TFL.

Security considerations

Up to now, only the first two goals of SOFFIC have been considered: not to allow access to violated files and to attempt to minimize the impact on system performance. This was dealt with by using RVFC and hash algorithms to verify integrity. In order to reach the third goal, it is necessary to analyze each component of SOFFIC from a security point of view. The following items present a summary of this analysis and show the security requirements for each component.

[5] The operation of the ISBB is described separately later in this section.

[6] Execution is excluded here.

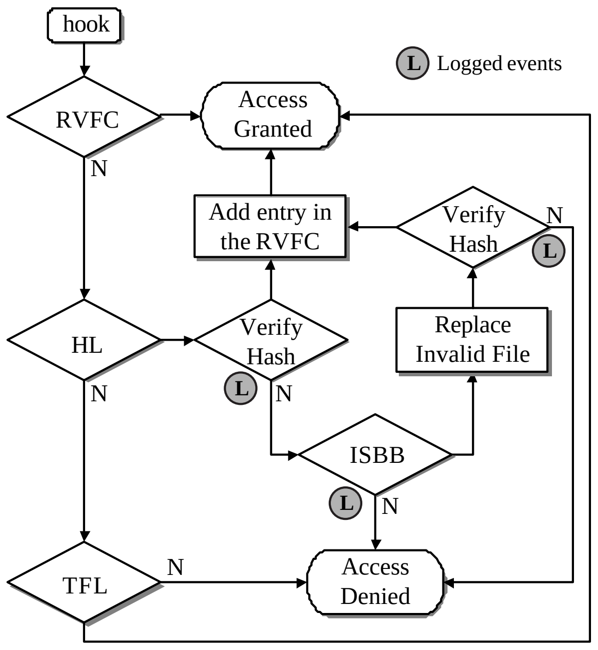

Figure 6 SOFFIC functional overview.

The functional aspects of ISBB as well as the security requirements of the components directly related to it are described separately in order to help understand the solution better.

Configuration file

Obviously, alterations in the configuration file can affect the operation of SOFFIC in a way the manager does not wish. Besides this, some of the information it stores may help an attacker's action, such as the IP number of the ISBB Server.

Because of this, the configuration file is digitally signed by the SOFFIC manager and encrypted with symmetric key K_{sm}.

Hash list (in the file system)

The HL alone is secure against undue reading, i.e., the secrecy of its content is not a fundamental requirement for the security of SOFFIC. However, secrecy can be obtained by encrypting the file with symmetric key K_{sm}.

The only essential protection is against undue writing because, as seen in section 'Why and how to control file integrity?' above, attackers with privileges to manipulate the HL can consolidate unauthorized file changes by updating the HL themselves. This protection is given by a digital signature of the SOFFIC manager at the time HL is generated.

It is important to point out the fact that because the HL is signed, all files whose hashes are contained in it are signed too, since their hashes represent the content of each one in a unique way.

Trusted files list (in the file system)

As shown before, the TFL is a very sensitive component from a security standpoint. By reading the contents of this list, an attacker may learn which files and directories are free from verification. As well as undue reading, undue writing may also significantly harm the security SOFFIC provides.

Therefore, the protection this component requires and the mechanisms applied are identical to those of the configuration file: digital signature of SOFFIC manager and encryption with symmetric key K_{sm}.

Core on disk

Core is a part of the system kernel, therefore the object to be protected is the image of the kernel itself. If an attacker manages to change the kernel image, SOFFIC or any other functionality of the system may be affected.

In order to obtain a minimum security level for the kernel image, it is included in the HL and its integrity is verified as soon as SOFFIC starts running when the system is loaded. SOFFIC does not provide protection in cases where the kernel image is completely replaced, so other mechanisms should be used to ensure boot time security.

Core in memory

In this state, the protection of Core faces the same problems of Core on disk. SOFFIC itself obstructs an attacker's attempt to change the memory-stored kernel because the tools to do so must be run locally and SOFFIC only allows the use of valid files; in other words, the attack could only be carried out with the system tools.

Symmetric key

Symmetric key K_{sm} is necessary for SOFFIC to access practically all of its files at boot time. So, to avoid this it remains in the system, whether on disk or in memory, the manager must supply it at boot time and then SOFFIC uses it and eliminates it from memory.

Asymmetric key pair

Private key K_{pri} is not kept in the system, but rather on off-line media under control of the SOFFIC manager. Still, it is also protected by a symmetric key known only and exclusively by the manager. This key must not be the same as K_{sm}.

Public key K_{pub} does not need secrecy, but protection against undue writing, because attackers could replace the manager's K_{pub} key with their own public key and then generate valid signatures before SOFFIC.

When key K_{pub} is included in Core, it is protected by the same mechanisms that protect Core itself, whether on disk or in memory. If K_{pub} is in a file pointed by the configuration file, it is then encrypted with key K_{sm}.

Recently verified files cache

While advancing the performance of SOFFIC, RVFC is also a critical point for security. One of the possible attacks against SOFFIC is the alteration and use of a file that has an entry in RVFC. In order to protect SOFFIC from this kind of attack, file changes are monitored and every time a file with an entry in RVFC is modified, the entry is immediately eliminated.

Like HL, RVFC can be freely read without harming security. Being an internal component of Core, RVFC is protected by the same means as Core in memory.

Sofficadm

Because sofficadm is the interface through which the manager configures and maintains SOFFIC, it is clear that it must be protected against undue modifications.

In order to avoid this kind of attack, an entry to sofficadm is added to the HL. Since it is necessary only to maintain SOFFIC, sofficadm can be kept on off-line media like private key K_{pri}.

The requirements described in any of the previous items must be complied with so that SOFFIC may have its own security ensured.

Integrated secure backup base

Focusing again on the ISBB, shown very briefly in Fig. 6, and our fourth goal, below is its detailed operation.

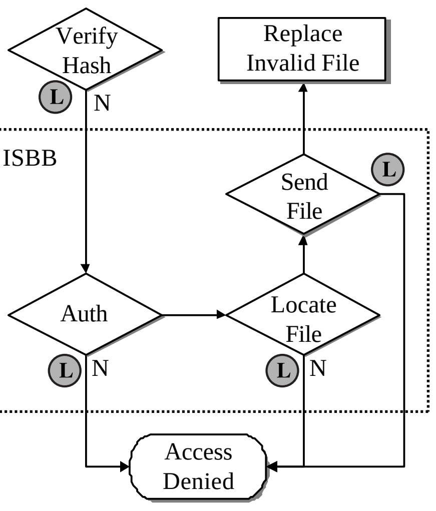

Figure 7 ISBB functional overview.

Having compared hashes and verified that a given file has been violated, SOFFIC attempts to restore the file from the ISBB, if it is available. It is important to remember that ISBB comprises the backup base itself and the ISBB Server that tends to SOFFIC requests. Before requesting a file, it seeks authentication before the ISBB Server. Upon success, the request is sent and the ISBB Server locates the file in its backup base (Fig. 7). Once found, the file is sent to SOFFIC, which then replaces the violated file and verifies its hash again (Fig. 6).

ISBB is the fundamental piece that enables SOFFIC to meet its fourth goal: self-healing. It is therefore necessary that its own security and that of the files it keeps is also warranted. This is the main reason ISBB is independent from SOFFIC—it is possible to keep an ISBB Server in a separate machine with even more limited access than the others. There would be no sense in keeping ISBB on the same machine as SOFFIC because the files ISBB kept would be under the same protection as the other system files.

Still, the same ISBB may serve more than one single instance of SOFFIC (Fig. 8).

To make it possible, it suffices to have a structure similar to the one presented in Fig. 9. With this structure, ISBB can keep the files of two different systems in its base, e.g., subsystem 1 and subsystem 2, in a simple and efficient way.

The basic security requirements for ISBB are three:

- individual authentication of each instance of SOFFIC before the ISBB Server: this prevents a given subsystem from accessing files it does not own. This feature reinforces the separation of security domains defined in section 'Secure on-the-fly file integrity checker (SOFFIC)'. Authentication is also necessary to prevent attackers from obtaining copies directly from an ISBB Server;
- authentication of ISBB Server before each instance of SOFFIC: the purpose of this is only to ensure that SOFFIC is receiving the file from

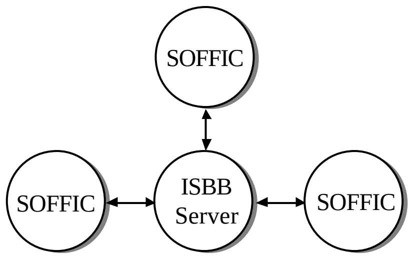

Figure 8 One ISBB Server and many SOFFICs.

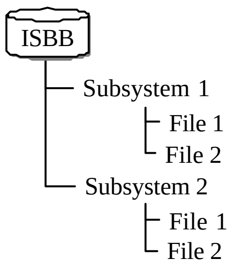

Figure 9 ISBB structure.

a valid source. Here, no additional authentication mechanism is necessary because SOFFIC itself verifies the received file hash, i.e., even without authentication from the ISBB Server, SOFFIC can be sure files are authentic, or not; and
- secrecy of communication between SOFFIC and ISBB Server in both ways: this is to prevent an attacker from getting a copy of the files through a sniffer.

The executable file of the ISBB Server itself can be protected by a local instance of SOFFIC with no self-healing ability, or basing it on a secondary ISBB Server.

As for the isbbadmin, which is only used to generate and maintain the ISBB, it may be kept on off-line media or included in the HL, like sofficadm.

SOFFIC capabilities

Operating as described and supported by the security requirements of its components, SOFFIC is able to:

- keep an attacker, even after a successful first stage of attack, from operating a Trojan horse, rootkit or DDoS zombies;
- keep worms and viruses from damaging the system or even spreading;
- keep an attacker from using tools it installed in the intruded subsystem to attack neighboring systems;
- repair violated files automatically, thus providing self-healing.

In addition, because it uses cryptographic hash functions, there is no practical possibility of SOFFIC presenting false negatives, i.e., attesting that a file is unchanged when it is in fact violated. Neither the contrary, false positives, as long as the manager keeps the HL and the ISBB adequately updated.

When comparing SOFFIC to other solutions like L6 and Tripwire, the most important features are:

(a) SOFFIC is able to detect the violation of a file right before it is used in the system, reducing significantly the resulting risks of a security break-in;
(b) SOFFIC is able to automatically restore the violated file from an on-line backup base, thus providing a self-healing process.

Another very important aspect is that neither SOFFIC nor any other known solution is capable of classifying file modifications as authorized or not. Therefore, they will work properly, detecting just unauthorized modifications, only if the system administrator keeps the hash list updated. Otherwise, any file changes are handled as unauthorized.

Implementation

The purpose of this article is not to describe in full detail the whole conception of the solution along with its implementation.[7] Therefore, we will only highlight a few interesting points.

From its first prototypes, SOFFIC has been implemented on Linux, and that has been so for four reasons:

- Linux kernel source code is open: this makes the study of the kernel and the implementation of the solution much easier;
- GNU license: this ensures freedom of distribution of the implemented solution;
- It is easy to port the solution to other Unix platforms; and
- Other works of our research group are also based on Linux.

All cryptography functions used on user space are obtained with OpenSSL library.[8] Regarding the kernel, all hash and symmetric key functions are obtained from the application of patch CriptoAPI.[9] As for the function of digital signature verification RSA, it was implemented separately as a patch we developed for the kernel.

The implementation is completely independent from the file system used, since SOFFIC is in the Virtual File System (Fig. 10). Therefore, one can always refer to a file through its device/inode pair, regardless of whether such structure exists in the file system (Beck et al., 1998; Bishop, 2003).

As a result, the HL is sorted according to the device/inode pair and then the binary research method is used, which is widely known as fast.

[7] Latest public release available at: http://www.inf.ufrgs.br/~gseg/.
[8] http://www.openssl.org.
[9] http://www.kerneli.org.

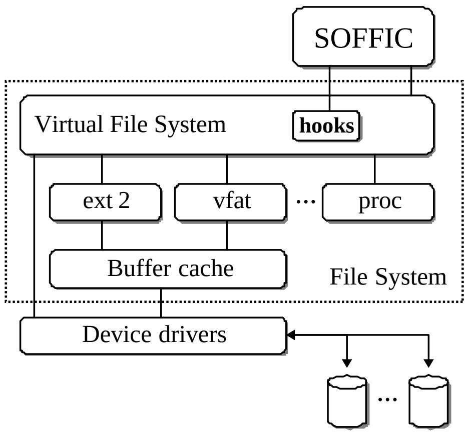

Figure 10 SOFFIC and the Linux Kernel (Aivazian, Linux kernel internals).

Along with the fact that the HL is all in memory, there is an important factor to reduce the impact on system performance.

Because RVFC is another essential component for the performance of SOFFIC, it was implemented with 256 entries and each entry position is defined according to a hash table.[10] The search with a hash table is even faster than binary research, resulting in great speed and flexibility for the RVFC.

The performance tests run on the first versions of SOFFIC resulted as expected, i.e., without using RVFC the access time to the file system grew substantially, increasing by 60—80%. As the hash calculation requires one extra reading, this effect was foreseeable. The use of RVFC lowered the impact on the file system drastically, with increases around 6—10%. Tests were run on a Pentium III 1 GHz, with an HL of approximately 11,000 entries.

Aware that these tests were rather limited, we are now developing another work aimed to measure the impact of SOFFIC on different environments adequately.

In order to reinforce the level of security of SOFFIC, we have been performing several practical experiences with SchlumbergerSema CryptoFlex smartcard. This smartcard has its own processor and several built-in cryptography functions, besides its own file system with robust access controls. We aim not only to ensure greater security, but also to ease the management of symmetric and asymmetric keys as well as the authentication of the manager before SOFFIC.

From the management point of view, the characteristics of the environment in which SOFFIC will be used determine how simple it is to deploy SOFFIC. The more stable the environment, regarding files creation and modification (essentially programs and configuration files), the more easily can SOFFIC be used. Some environments considered quite stable and thus perfect for SOFFIC are firewalls, HTTP/FTP servers, mail servers and database servers. In these cases the hash and trusted files lists need to be generated only once, and there is a minimal cost for updating them. Some worst cases for SOFFIC are developing stations and prototype environments, where the lists must be updated quite frequently.

The most recent version of SOFFIC publicly available is 0.2p2, and it can be found at http://www.inf.ufrgs.br/~gseg/.

[10] It is not the same as a cryptographic hash.

Conclusion

The integrity control mechanisms currently available are not appropriate to the goals focused in the construction of survivable systems and this may be the reason their importance in this scenario has been reduced.

Seeking to elaborate a mechanism to fill in the existing gap, we have developed the model of a tool with features designed for the application in survivable systems. Among these is the ability to delay an increase in privileges to a system's attacker through controlling and repairing the damages (self-healing) caused by the attack, thus ensuring the continuity of the critical operations of the system.

The implementation presented, though in its initial stages, already shows the practical feasibility of the mechanism, and it can be immediately applied to protect files to be run. The tool should soon be fully implemented and then much more detailed performance tests can be run.

Finally, we hope this study moves on and is able to cause important discussions for the development of this topic in survival systems and to aggregate value through collaborative work with other researchers.

Acknowledgements

To all members of Grupo de Segurança da UFRGS (UFRGS Security Group) for the suggestions and healthy criticism, the latter being the most. To Philippe Biondi for showing me that the kernel source code is the best documentation of the kernel source code. To André Fávero who volunteered to be engaged in the development of SOFFIC. To

Álvaro Reguly for the help with kernel version 2.4.18. To CNPq for allowing this project to take its first step from March 2000 to February 2002 with its program of scholarships for Master's courses in Computer Science.

References

T. Aivazian. Linux kernel internals. Available from: http://www.linuxdoc.org.

Beck M, Bohme H, Dziadzka M, Kunitz U, Magnus R, Verworner D. Linux kernel internals. Addison Wesley Longman; 1998.

Bernaschi M, Gabrielli E, Mancini LV. Remus: a security-enhanced operating system. ACM Trans Inform Syst Secur (TISSEC). Feb. 2002;5(1).

Bishop M. Computer security: art and science. Addison Wesley; 2003.

Kim GH, Spafford EH. The design and implementation of tripwire: a file system integrity checker. Technical Report CSD-TR-93-071. Purdue University; 1993.

Menezes AJ, van Oorschot PC, Vanstone SA. Handbook of applied cryptography. CRC Press; 1996.

Pal P, Webber F, Schantz R. Survival by defense-enabling. Proceedings of the 2001 Workshop on New Security Paradigms. Sept. 2001.

Schneier B. Applied cryptography. Protocols, Algorithms, and Source Code in C. 2nd ed. John Wiley & Sons; 1996.

Vinícius da Silveira Serafim graduated from Universidade de Passo Fundo (UPF), Brazil, in 1999 with a Bachelor's degree in Computer Science. He received his Master's degree in Computer Science from Universidade Federal do Rio Grande do Sul (UFRGS), Brazil, in 2002 and is currently a Doctorate student in computer security at UFRGS. He is doing research on computer security issues since 1997.

Raul Fernando Weber is Professor at the Computer Science Institute in the Universidade Federal do Rio Grande do Sul. He graduated in 1976 in Electrical Engineering (UFRGS), has a Master's degree in Computer Science from UFRGS (1980) and a Doctor's degree in Computer Science from Karlsruhe University, Germany (1986). His research areas are computer architecture and, since 1991, computer security.