Protection of Components based on a Smart-Card Enhanced Security Module

Joaquín García-Alfaro[1], Sergio Castillo[1], Jordi Castellà-Roca[2], Guillermo Navarro[1], and Joan Borrell[1]

DEIC-UAB, 08193 Bellaterra (Catalonia), Spain
Email: {jgarcia,scastillo,gnavarro,jborrell}@deic.uab.es

DEiM-ETSE-URV, 43007 Tarragona (Catalonia), Spain
Email: jordi.castella@urv.net

Abstract. We present in this paper the use of a security mechanism to handle the protection of network security components, such as *Firewalls* and *Intrusion Detection Systems*. Our approach consists of a kernel-based access control method which intercepts and cancels forbidden system calls launched by a potential remote attacker. This way, even if the attacker gains administration permissions, she will not achieve her purpose. To solve the administration constraints of our approach, we use a smart-card based authentication mechanism for ensuring the administrator's identity. Through the use of a cryptographic protocol, the protection mechanism verifies administrator's actions before holding her the indispensable privileges to manipulate a component. Otherwise, the access control enforcement will come to its normal operation. We also show in this paper an overview of the implementation of this mechanism on a research prototype, developed for GNU/Linux systems, over the *Linux Security Modules* (LSM) framework.

1 Introduction

The protection of network security components, such as *Firewalls* and *Intrusion Detection Systems*, is a serious and important problem which must be solved. Otherwise, whenever a remote adversary manages to compromise the security of these components, she may obtain the control of the system itself. Contrary to many other elements of a network, security components are almost always working with special privileges to properly execute their tasks [6]. This situation is very likely to lead remote attackers to acquire these privileges in an unauthorized manner. For instance, the existence of programming errors within the code of these components, the illicit manipulation of their related resources (such as processes, filesystem, and so on), or even the increase of privileges though operating system's errors, are just a few examples regarding means in which a remote adversary can bypass traditional security policy controls.

In [4, 5] we presented an enhanced protection module integrated into the kernel of an attack prevention system intended to intercept and cancel forbidden system calls launched by a remote attacker. Specifically, the mechanism presented in [4, 5] prevents a privilege escalation attack on the prevention system itself – through an enhanced access control scheme which handles the protection of the system's elements. This strategy introduces, however, some administration constraints, since the administrators are not able to throw system calls which may suppose a threat to the protected system. To solve these constraints, we present in this paper an extended version of our approach which includes a smart-card based authentication mechanism, which acts as a reinforcement of the kernel-based access control. The objective of this complementary mechanism is twofold. First, it holds to the administrator the indispensable privileges to carry management and configuration activities just when she verifies her identity through a two-factor authentication mechanism. Second, it allows us to avoid those attacks focused on getting the rights of the administrative entity, such as dictionary-based attacks or buffer overflows.

The rest of this paper is organized as follows. Section 2 summarizes some related works. Section 3 shows an overview of our protection strategy. Section 4 takes a closer look at the development of the proposed mechanism. Section 5 presents our smart-card based authentication protocol intended to solve the administration constraints introduced by the protection mechanism. An evaluation concerning the efficiency of our proposal is then presented in Section 6. Finally, Section 7 closes the paper with a list of conclusions.

2 Related Work

There are two main approaches to safely execute processes with special privileges on modern operating systems. A first approach, as the one presented in this paper, is to apply a kernel-based access control to the outcoming system calls. A second approach is the creation of restricted environments, in which the processes will be executed and controlled outside the trusted system space.

Regarding the first approach, the proposals closest to ours are the protection mechanisms presented in [9] and [11] for the creation of enhanced access control mechanisms integrated in the kernel of the GNU/Linux operating system. The main goal behind these two proposals is to reinforce the complete system by controlling the system calls and ensuring which process or user does the system call and against what it will be done. The ability to control the access to the resources allows to protect the security components and to avoid that nobody (including an attacker with administrator privileges) can disable them.

Nevertheless, both approaches differ from ours in a number of ways. First, and to our best knowledge, neither [9] nor [11] do not address the management of administration constraints, as our proposal does through the two-factor authentication mechanism we present in Section 5. Second, our approach, entirely based on the *Linux Security Modules* (LSM) framework [13], guarantees the compatibility with previous applications and kernel modules without the necessity of modifications. However, both [9] and [11] require the rewriting of some features of the original Linux kernel to properly work. This situation may force to recompile existing code and/or modules in order to obtain the new security features. Although it exists a LSM-based prototype for the approach presented in [9], it does not seem to be actively maintained for the current Linux-2.6 kernel series.

Regarding the second approach, we find in [8] a protection mechanism for the creation of restricted environments within Unix setups. The authors in [8] present the use of a special system call to restrict the access to a specific area of the file system. This specific area is intended just for the processes that are executed under each restricted environment. Then, this system call properly changes the root directory to the given path. This way, the process remains in a safe space from where it is not possible to escape – even if the component is compromised, the whole system will remain safe since the illicit activities are caught within the replicated file system. This proposal requires, however, a replicated file system tree for each environment. Hence, the administrator in charge of the system must reproduce the original file system tree to include, for example, shared libraries or configuration files, and copy them to the new path. Other disadvantage of this proposal is that it does not guarantee the correct execution flow of a process, i.e., the behavior of a process can be modified by using, for example, a buffer overflow. Hence, the attacker can overwrite the configuration or logs files of such a process by simply using an arbitrary code execution attack – since these files remain in the same environment of the protected security component process.

Extended versions of the previous model, as the one presented in [7], may also offer support for access control to resources and guarantee the integrity of the security component's resources. Nonetheless, these extended proposals do not protect from vulnerabilities placed outside the trusted environment. A simple bug in a privileged service, or even the use of stolen passwords, may lead the attacker from the external environment to attack the component and its resources.

3 Our proposal

As introduced in Section 1, our main motivation is the protection of network security components, such as *Firewalls* and *Intrusion Detection Systems*, which, if successfully attacked, are very likely to lead an intruder to get the control of the whole system. This problem leads to the necessity for introducing a protection mechanism on the different elements of each component, keeping with their protection and mitigating – or even eliminating – any attempt to attack or compromise the component's elements and their operations. This way, even if an attacker compromises the security of the component, she would not be able to achieve her purpose.

We consider the protection of the elements carried by the kernel of the operating system as a proper solution for such a protection. First, the protection at kernel level avoids that potentially dangerous system calls (e.g., *killing* a process) could be produced from one element against another one. This protection is achieved by incorporating an access control mechanism into the kernel system calls. This way, one may allow or deny a system call based on several criteria – such as the identifier of the process making the call, some parameters of the given call, etc. The kernel's access control allows to eliminate the notion of trust associated to privileged users, delegating the authorization for the execution of a given system call to the internal access control mechanisms. In addition, and contrary to other approaches, it provides a unified solution, avoiding the implementation of different specific mechanisms for each component. This mechanism allows us, moreover, to enforce the compartimentalization principle [12]. This principle is based in the segmentation of a system, so several elements can be protected independently one from another. This ensures that even if one of the elements is compromised, the rest of them can operate in a trusted way.

In our case, several elements from each component are executed as processes. By specifying the proper permission based on the process ID, we can limit the interaction between these elements of the component. If an intruder takes control of a process associated to a given component (through a buffer overflow, for example), she will be limited to make the system call for this given process.

It is not always possible, however, to achieve a complete independence between the elements. There is a need to determine which system calls may be considered as a threat when launched against an element from the component. This requires a meticulous study of each one of the system calls provided by the kernel, and how they can be misused. On the other hand, we have to define the access control rules for each one of these system calls. For our approach, we propose the following three protection levels to classify the system calls: (1) critical process protection; (2) communication mechanisms protection; and (3) protection of files associated to the elements.

The first level of protection (critical processes) comprises actions that can cancel the proper execution of the processes associated to a component, either by interaction over them by signals, or the manipulation of the memory space. Some examples are: execution of a new application already in memory, manipulation of the address space and process traces, and so on. The second level (communication mechanisms protection) includes the protection of all those processes that allows an attacker to modify, generate or eliminate any kind of messages exchanged between component's elements. Finally, the third level of protection (protection of files associated to the elements) takes into account all those actions that can maliciously address the set of files used by the elements of the component, such as executable, or configuration files.

4 Prototype Implementation

In this section we outline the current implementation of SMARTCOP (which stands for *Smart Card Enhanced Linux Security Module for Component Protection*). In accordance with the protection scheme proposed in Section 3, it consists of a kernel-based access control mechanism, and its development has been done over the *Linux Security Modules* (LSM) framework for *GNU/Linux* systems [13]. The LSM framework does not consist of a single specific access control mechanism; instead it provides a generic framework, which can accommodate several approaches. It supplies several hooks (i.e., interception points) across the kernel that can be used to implement different access control strategies. Such hooks are: *Task hooks, Program Loading Hooks, File systems Hooks* and *Network hooks*.

These LSM hooks, can be used to provide protection at the three levels pointed out above. Furthermore, LSM adds a set of benefits to our implementation. First, it introduces a minimum load to the system when comparing it to kernels without LSM, and does not interfere with the detection and reaction processes (cf. Section 6). Second, the access control mechanism can be composed in the system as a module, without having to recompile the kernel. And third, it provides a high degree of flexibility and portability to our implementation when compared to other proposals for the Linux kernel, such as [9] and [11], where the implementation requires the modification of some features of the original Linux-2.6 kernel series.

The LSM interface provides an abstraction, which allows the modules to mediate between the users and the internal objects from the operating system kernel. To this effect, before accessing the internal object, the hook calls the function provided by the module and which will be responsible to allow or deny the access. This can be seen in Figure 1. There, a module registers the function to make a check over the *inodes* of the file system. At the same time, LSM allows to keep the *discretionary access control* (DAC) provided by the kernel Linux, by standing between the discretionary control and the object itself. This way, if a user does not have permissions in relation to a given file, the DAC of the operating system will not allow the access and no call to the function registered by the LSM will be made. This architecture reduces the load of the system when compared to an access control check centralized in the operating system call interface, which always gets used for all the system calls.

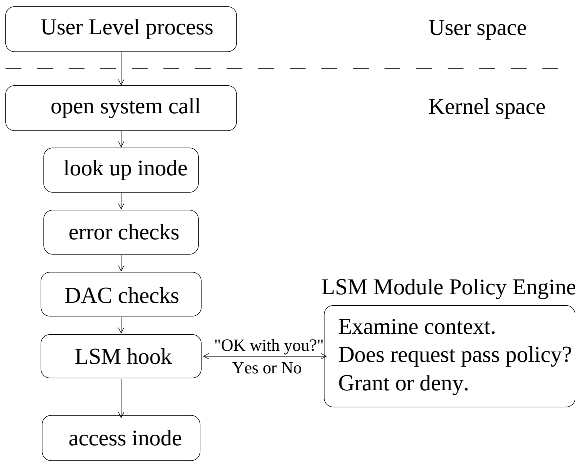

Fig. 1. Linux Security Modules (LSM) Hooks.

The component's elements will be allowed to make operations only permitted to the system administrator – such as packet filtering, process or application cancellation, etc. This implies that the system processes associated to each element will be executed by the administrator – i.e., root user in Unix systems. On the contrary, if we associate the processes to a non privileged user, the discretionary access control of Linux will not allow the execution of some specific calls.

The internal access control mechanisms at the kernel is based in the process identifier (PID) that makes the system call, which will be associated to a specific element. Each function registered by a LSM module, determines which component is making the call from the PID of the associated process. It then, applies the access control constraints taking also into account the parameters of the system call. Thus, for example, a given element can access its own configuration files but not configuration files from other elements.

An important issue in the implementation is the administration of the access control mechanisms and the management of each one of the elements. As pointed out in previous sections, the administrators should not be able to throw a system call, which may suppose a threat to the component. This prevents an intruder doing any harm to the component even if she could scale her privileges to the administrator ones. This contrasts with the administration of the component, since, if an administrator can not interact with the elements of the component, she will not be able to carry out any management or configuration process and activities.

To solve this hazard, we propose a smart-card based authentication mechanism. Specifically, we use the functionality of a smart-card for ensuring the administrator's identity. Through the use of an authentication protocol, the LSM module verifies administrator's actions before holding her the indispensable privileges to manipulate the component. Otherwise, the access control enforcement will come to its normal operation. In the following section, a detailed description of such a mechanism is given.

5 Smart-Card Based Authentication Mechanism

Traditional user authentication, also known as single-factor authentication, relies on user's knowledge of some secret – for instance, a password or a PIN. Then, using this knowledge as the only requirement, the user may proof his identity. Nevertheless, single-factor authentication is not secure enough – the existence of password attacks, man-in-the middle techniques, etc., is a proof of that. A two-factor authentication mechanism, on the other hand, solves most of these problems. Two-factor authentication mechanisms require to prove both the knowledge of some secret and the possession of some characteristic. This characteristic must be unique, and not easily replicable (e.g., a smart-card).

Therefore, to better assure the administrator's identity in our protection scheme, we propose the use of a two-factor authentication mechanism based on the cryptographic functions of a smart-card. This mechanism is intended for authenticating the administrator to the LSM modules and holds with the following requirements:

- The actions must be authorized by the use of a smart-card;
- The smart-card only authorizes one action iff the PIN is correct;
- The LSM module only authorizes the action iff the smart-card response is valid, i.e., the cryptographic operation is correct.

According with these requirements, just when the smart-card is connected to the system, and the authentication protocol's result is satisfactory, the administrator is able to hold the indispensable privileges to manipulate the node. On the contrary, when the device is retired or the authentication protocol fails, the access control enforcement, presented in Section 4, comes to its normal operation.

5.1 Protocol Description

We give in the following a detailed description of the cryptographic protocol that leads our smart-card based authentication mechanism. Let us recall that the cryptographic engine of such a smart-card is capable of performing several cryptographic functions, such as symmetric key generation, symmetric cryptographic algorithms execution, etc.

Protocol 1

1. *The system administrator opens a new console and she requests an action X. It is assumed that X must be authorized by using the smart-card;*
2. *The module receives the request from the console and does the following steps:*
   (a) *Open a connection to the smart-card reader device;*
   (b) *Print a message in the console, asking for the smart-card insertion to the smart-card reader device;*
   (c) *While the smart-card has not been inserted do;*
      i. *Detect the insertion of the smart-card;*
   (d) *Print a message in the console asking for the operation PIN;*
3. *The system administrator types the operation PIN in the keyboard;*
4. *The module does the following steps:*
   (a) *Obtain the operation PIN;*
   (b) *Obtain a NONCE value at random;*
   (c) *Compute the Message Authentication Code (MAC) of NONCE with the shared key K, $\mu_1 = MAC(K, NONCE)$;*
   (d) *Execute the Procedure 1 inside the smart-card using the operation PIN and the NONCE, and obtain a response μ_2;*
   (e) *Print a message in the console to remove the smart-card from the smart-card reader device;*
   (f) *While the smart-card has not been removed do;*
      i. *Detect the removing of the smart-card;*
   (g) *if μ_2 is* ERROR *the LSM module does not authorize the action X;*
   (h) *else do:*
      i. *if $\mu_1 \neq \mu_2$ the module does not authorize the action X;*
      ii. *if $\mu_1 = \mu_2$ the module authorizes the action X;*

As we can see in Protocol 1, an *operation PIN* and one *administration password* are used in our protocol. The operation PIN is at least six digits long. We use the operation PIN in order to authorize the actions. On the other hand, the administration password is used to change the operation PIN and other management tasks. The system administrator has three consecutive chances to enter the operation PIN. In the third chance if the smart-card receives an incorrect operation PIN it blocks itself. The smart-card only can be unblocked with the administration password. Again, there are three chances to enter the correct administration password. If the smart-card is blocked with the administration password the smart-card becomes useless.

The security parameters of the LSM module are properly initialized when it is installed. The system administrator inserts a smart-card in the reader device and the cardlet application is downloaded to the smart-card. Once the applet has been download and registered, the system administrator introduces the administration password and the operation PIN. The LSM module then sends the shared key K – it stores the shared K in a secure file, so the file can be read exclusively by the LSM module.

Then, the smart-card and the LSM module share a secret key K. In Step 1 of such a protocol, the system administrator requests an action to the LSM module which, in turn, blocks the communication channel between the smart-card reader and the LSM module (cf. Step 2a). The data sent between the LSM module and the smart-card can not be sniffed because the channel is blocked. The protocol avoids the smart-card remains in the smart-card reader when is not necessary. In Step 2c, the LSM module waits until the smart-card insertion, and in Step 4f the LSM module does not proceed since the smart-card has been removed.

In Step 3 the operation PIN travels in a secure way from the keyboard because the LSM module has blocked the channel between the keyboard and the module itself. Then, the LSM module sends a NONCE obtained at random and the PIN in Step 4d. The smart-card returns a Message Authentication Code (*MAC*) of the NONCE computed with the shared key K. In the last Step, i.e., Step 4h, the LSM module verifies whether the MAC has been properly computed.

Let us finally show the following procedure (cf. Procedure 1), which is executed within the smart-card to validate the operation PIN. If the operation PIN is correct, it computes the MAC of NONCE with the shared key K.

Procedure 1 $[PIN, NONCE]$

```
1. Validate the operation PIN;
2. If the operation PIN is correct do:
    (a) Compute the Message Authentication Code (MAC) of NONCE with the shared
        key $K$, $\mu_2 = MAC(K, NONCE)$;
    (b) return $\mu_2$;
3. If the operation PIN is no correct return ERROR;
```

5.2 Security Considerations

To ensure the proper execution of both Protocol 1 and Procedure 1 (cf. Section 5.1), we must consider the protection of the entities and the channels involved in such a process, avoiding attacks like impersonation or channels data manipulation. The lack of ability to avoid these attacks and their impact makes our proposed protection mechanism usefulness. Regarding the different entities that take part in the protocol, we suggest in this section the following considerations.

First, the possible console attacks could be directed to the binary executable file and the console process in execution time. If this happens, an overwrite of the executable console's file using malicious code could lead an attacker to take the control of the authentication process, giving her the possibility to complete the protocol and get the control of the system – and even to steal the smart-card's PIN. To eliminate this attack, the LSM module guarantees that the binary file of the console can not be overwritten by anybody (even the administrator), remaining the permissions as read-only.

Second, the binary executable of the administration console is compiled in a static fashion. This allows us to reduce the complexity of the protection's console process, since we do not need to consider extra tasks introduced by the loading of shared libraries and its associated files. At the same time, it enables us to centralize and reduce the failure points that could be used by an intruder to tamper the console's process. Thus, and to protect the process associated to the console, the LSM module controls that each system call launched by some process can not be dangerous for the correct execution flow of the console process, such as keyboard key capture, cancellation, or debugging process system calls.

Let us recall that the communication channels can not be manipulated by any opponent. To achieve this purpose, the LSM mediates between the system calls related with the communication channels and the entities that take part within the protocol (the LSM module, the smart-card, and the console process).

To conclude, and as pointed out in [2], the LSM module does not need to be directly protected since we can assume the kernel environment as a trusted area – since it is mandatory for the kernel security model of our prototype's operating system.

6 Evaluation

This section describes the performance evaluation of SMARTCOP for GNU/Linux systems based on the *Linux Security Modules* (LSM) framework.

We show the outcome of several tests steered towards measuring the penalty introduced by the installation of SMARTCOP as a LSM module, over the normal operation of the system. We do not take into account in this evaluation, the performance penalty during administrative tasks. That is, operations carried out by the system administrator making use of the authentication scheme presented in the previous section.

The set of tests is based on the use of the Strace [1] tool and the LMbench [10] package. Strace is a debugging tool, which allows us to trace the system calls made after the execution of a given process. This can be used to analyze and evaluate the time taken by these calls. On the other hand, LMbench is used to perform *microbenchmarks*, which are used to take more precise measures of the time taken for file access, memory access, etc.

The evaluation was carried out on a single machine with an Intel-Pentium M 1.4 GHz, with 512 MB of RAM memory and an IDE hard disc of 5400 rpm, running a Debian GNU/Linux operating system and ext3 file system. The objective of these tests is to compare the performance of the system using a Linux 2.6.15 kernel without LSM support against the performance of the same system and kernel with LSM support and the SMARTCOP module loaded.

Test Type	**2.6.15**	**2.6.15 + smartcop**	**% Overhead with smartcop**
null call	0.255	0.255	0%
kill	231.10	241.65	4.6%
stat	1.99	2.03	2%
open/close	2.96	3.02	1.9%
select TCP	18.63	18.86	1.2%
sig inst	0.9	0.9	0%
sig handl	1.85	1.88	0.1%
fork proc	95.61	96.52	0.9%
exec proc	100.50	103.86	3.3%
sh proc	2227	2302	3.3%

Process tests, time in μseconds

Test Type	**2.6.15**	**2.6.15 + smartcop**	**% Overhead with smartcop**
pipe	1342	1338	0.2%
AF Unix	1334	1320	1%
TCP	1088	1078	0.9%
file read	1330	1308	1.6%
mmap read	1480	1425	3.8%
mem bcopy	5278	5277	0.01%
mem bzero	4548	4548	0%
mem read	25600	25590	0.03%
mem write	24888	24869	0.07%

Local communication bandwidth in MB/s

Test Type	**2.6.15**	**2.6.15 + smartcop**	**% Overhead with smartcop**
0K file create	193	193	0%
0K file delete	489	489	0%
10K file create	175	176	0.5%
10K file delete	658	668	1.5%
mmap latency	2348	2348	0%
par mem	1.26	1.26	0%
page fault	0.974	0.981	0.8%

File and VM sytem latencies, time in μseconds

Fig. 2. Performance evaluation of SMARTCOP.

The results of the tests are shown in Figure 2. They are organized in three tables depending on the three protection levels stated in Section 3. As it can be appreciated in the results, the penalty introduced by SMARTCOP has a minimum impact on the performance of a standard GNU/Linux 2.6.15 system.

The first table (*Process tests*) shows the latency in microseconds for a set of operations related to the execution of processes and system calls such as process creation through *fork()*, *fork()+exec()* and *sh()*, process cancellation through *kill()*, descriptor waiting through *select()*, opening and closing files through *open()*/*close()*, signal installation and management, etc.

This first category of tests shows that more than the 50% of the tests indicate a performance penalty below 2%. For example, the process creation with *fork()* is scarcely penalized with a 0.9%. The same can be noticed for process creation with *fork()+exec()* and *sh()*, which have an approximate penalty of 3.3%. On the other hand, the higher performance penalty is presented by the process cancellation through the system call *kill()* with a 4.6%. This higher penalty is produced by the access control verifications of SAMRTCOP at kernel level, during the identification checks of the process, system call parameters, etc.

The second set of tests shown in the second table of Figure 2, present the bandwidth of operations related to communication issues such as reading, writing and copy of memory sections through *read()* and *mmap()*, Inter Process Communications (IPC) using TCP, pipes and sockets of the Unix address family (*AF Unix sockets*), etc. Again, the results show a minimum penalty in the performance. In this case the greater penalty (3.8% approx.) is found in the reading and summing of a file via the memory mapping *mmap()* interface.

Finally, the set of tests from the third table (Figure 2) shows the latency found in operations related to file and memory manipulation. The performance penalty of the system is also minimum. The greater penalty being introduced by the file elimination due to the verifications performed by SMARTCOP during the associated system calls.

7 Conclusion

In this paper we have presented an access control mechanism specially suited for the protection of network security components, such as *Firewalls* and *Intrusion Detection Systems*. Whenever one of these components, or one of its elements, is compromised by an attacker, it may lead her to obtain the full control of the network. The protection of these components is not easy, specially when dealing with distributed setups, made up of different elements distributed over a complex network. Like for example, the attack prevention platform presented in [3].

The solution we provide in this paper proposes the protection of the components by making use of the LSM system in the Linux kernel over GNU/Linux systems. The mechanism we have developed, called SMARTCOP (*Smart Card Enhanced Linux Security Module for Component Protection*), works by providing and enforcing access control rules at system calls, and is based on a protection module integrated into the operating system's kernel, providing a high degree of modularity and independence between elements.

The use of LSM allows our protection system to be used in new components and elements, by just considering its environment and its interactions (regarding access control). It reinforces the modularity of the system and provides an easy and generic way to introduce new elements without having to consider each component separately. Thus, we consider that our proposal provides a high degree of scalability.

The introduction of new components provides a minimum performance penalty, because the LSM framework and the access control scheme do not introduce an excessive computational complexity. We have measured the penalty introduced by the use of SMARTCOP against the usual performance of the system. The results show the minimum performance impact of SMARTCOP. To reinforce the protection mechanism itself, SMARTCOP provides a complementary authentication method, based on smart-cards. This additional enhancement is based both on a secret (smart-card PIN) and a physical token (the smart-card itself). This way, we can prevent some logical attacks (e.g., password forgery) against the protection mechanism proposed in this paper.

For all these reasons, we can finally conclude that the enhanced access control provided by SMARTCOP, and integrated inside the operating system's kernel, offers a good degree of transparency to the administrator in charge, and it does not interfere directly with user space's processes.

Acknowledgments

J. García-Alfaro, S. Castillo, G. Navarro, and J. Borrell are partially founded by the Spanish Government project *TIC2003-02041*, and the Catalan Government grant *2003-FI126*. J. Castellà-Roca is partly supported by the Spanish Government project SEG-2004-04352-C04-01.

References

1. W. Akkerman. Strace, `http://liacs.nl/~wichert/strace/`
2. M. Borchardt, C. Maziero, and E. Jamhour. An architecture for on-the-fly file integrity checking. In *Latin American Symposium on Dependable Computing*, 117-126, Brazil, 2003.
3. J. García, F. Autrel, J. Borrell, S. Castillo, F. Cuppens, and G. Navarro. Decentralized publish/subscribe system to prevent coordinated attacks via alert correlation. In *6th Int. Conf. on Information and Communications Security*, 223–235, Spain, 2004.
4. J. García, S. Castillo, G. Navarro, and J. Borrell. ACAPS: An Access Control Mechanism to Protect the Components of an Attack Prevention System. In *Journal of Computer Science and Network Security*, 5(11):87-94, November 2005.
5. J. García, S. Castillo, G. Navarro, and J. Borrell. Mechanisms for Attack Protection on a Prevention Framework. In *39th Annual IEEE International Carnahan Conference on Security Technology*, 137–140, Spain, October 2005.
6. D. Geer. Just How Secure Are Security Products? *IEEE Computer*, 37(6):14–16, June 2004.
7. A. Herzog and N. Shahmehri. Using the Java Sandbox for Resource Control. In *7th Nordic Workshop on Secure IT Systems (NORDSEC 2002)*, Linköpings universitet, Linköping, Sweden, 2002.
8. P. Hope. Using Jails in FreeBSD for Fun and Profit. *Login; The Magazine of Usenix & Sage*, 27(3):48–55, June 2002.
9. P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *11th FREENIX Track: 2001 USENIX Annual Technical Conference*, USA, 2001.
10. L. McVoy. LMbench, Portable Tools for Performance Analysis. In *1996 USENIX Annual Technical Conference*, USA, 1996.
11. A. Ott. The Role Compatibility Security Model. In *7th Nordic Workshop on Secure IT Systems*, Sweden, November 2002.
12. J. Viega, and G. McGraw. *Building Secure Software - How to Avoid Security Problems the Right Way*. Addison-Wesley, September 2002.
13. C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *11th USENIX Security Symposium*, USA, August 2002.