

A Format-Independent Architecture for Run-Time Integrity Checking of Executable Code *

Luigi Catuogno and Ivan Visconti

Dipartimento di Informatica ed Applicazioni
Università di Salerno
Via S. Allende,
84081 Baronissi (SA), Italy.
{luicat, visconti}@dia.unisa.it

Abstract. A robust architecture against network intrusions plays a main role for information security and service reliability. An intruder that obtains an unauthorized access to a remote system could read restricted information or hide this access for future and eventually more dangerous actions. Temporary intrusions can become permanent (i.e., resistant to reboots) if malicious code is installed in a system not adequately protected. In this paper we propose an infrastructure for the run-time integrity checking of executable code. Our approach is general as the specification of our infrastructure includes support for every file format. Moreover we also present our implementation that supports run-time integrity checking for ELF and shell script files. Experimental results show that our solution is a practical and effective protection for workstations connected to the Internet offering services to local and remote users.

1 Introduction

External intrusion is one of the most serious threats to the security of a system that is connected to a network. Typically, an attacker exploits a bug of a network daemon to obtain superuser privileges in the form of a shell session owned by the root. Once this has been accomplished, the attacker has complete control of the system and access to all the data stored on the machine. Obviously, if for some reason the machine is rebooted, the attacker has to start again. Moreover, the legitimate administrator of the system could detect the ongoing intrusion and kill the shell session and terminate the intrusion. Once the attack has been detected, the system administrator can install a new version of the network daemon that does not include the bug that allowed the intrusion. Thus, the attacker cannot repeat the same attack on the machine but he has to find another weak daemon or exploit a weakness of the new version of the same daemon

* Supported by a grant from the Università di Salerno and by Young Researchers grants from the CNR.

(which, unfortunately, most of the times is easy to do). We call this form of attack the *weak intrusion attack*. A more serious threat comes from an attacker that, once root privileges have been gained, tries to *colonize* the system; that is, tries to keep control of the machine across reboots. We call this kind of attack the *strong intrusion attack*. One way of doing this is to install new malicious code and/or modifying existing executables. This has two main goals. First of all, every time the legitimate system administrator executes the modified executables the attacker regains control of the machine. Second, the attacker modifies tools used by the administrator in order to hide the ongoing activity. For example, the `ifconfig` command could be replaced by its malicious version which hides the promiscuous mode of a network card used to sniff passwords and a fake `ls` command could hide new executable files installed in a directory. Using these mechanisms an attack could become resistant to reboots and attack detection strategies. In this work we address the problem of *strong intrusion attacks* and present a security architecture that prevents the execution of malicious code on the system so that only trusted code is executed. Thus, as a consequence, we have that an attacker cannot use the techniques described above to permanently keep control of a machine. We stress that our work does not address the general problem of buggy software daemons (e.g., buffer overflow bugs) or, in general, the problem of system intrusion. Our architecture does not guarantee that an attacker cannot obtain root privileges but only that, once the attacker has been successful, the administrator can still detect the attack and eradicate it. In other words, we reduce the strong form of intrusion to the weak form of intrusion.

In Section 2, we present the most used techniques used by intruders for their malicious purposes. In Section 3, we describe the most interesting proposed architectures (only some of them are actually really implemented) to protect servers. This paper describes the construction of a secure infrastructure against permanent attacks and in Section 4 we propose our approach against such kind of intrusions. We have implemented the proposed infrastructure and in Section 5 we show details of our implementation, while in Section 5.3 we illustrate the management of a cache that has an important role to make our solution practical. Finally in Section 6 we conclude the paper re-emphasizing the features of our approach and motivating what are the future developments related to our work.

2 Tools for intrusion

In this section we describe the main techniques used by intruders in order to gain and maintain the control of a machine. We focus on the power of each technique so that we can measure the benefits of our architecture versus other proposals described in Section 3.

2.1 Rootkits and code modification

A rootkit is a subset of common system commands and daemons that have been modified in order to perform malicious operations, hide their effects, and possibly set up some "back-doors". Once an intruder obtains the access to a victim

machine, he installs the kit. Since malicious executable files take the place of the widely and most used commands (e.g., UNIX commands: `ls`, `ps`, etc.), execution of malicious code is performed by legitimate users of the system including the superuser. We illustrate the following scenario as an example. An intruder installs a network sniffer[16] (like `tcpdump`) that periodically logs the network traffic. In order to prevent the detection of such malicious activity, the intruder replaces the commands `ls` and `ps` with their respective evil counterparts that avoid the visualization of sniffer's log files, and hide sniffer's process listing. Moreover, the intruder can also install a malicious version of the `ifconfig` command, in order to hide that network interfaces have been set in *promiscuous mode* by the sniffer.

It is also technically possible to inoculate "parasite code" into executables of some binary format (without re-compiling them), taking advantage of certain properties of their memory image. In such a way, an intruder could even modify applications that he cannot replace. However, this technique is cumbersome, inefficient and it strongly depends on the operating system, the hardware architecture, the binary format and the memory layout of the target executable. An example of such attack can be found in [3].

Moreover many UNIX distributions are provided with several scripts (written in several interpreted languages like Perl, python and so on) that take care automatically about setup and configuration procedures of the operating system and of many packages. Malicious modifications of these scripts could invalidate possible software re-installations or upgrades avoiding the removal of malicious code or misconfiguring the system, thus guaranteeing back doors and vulnerabilities for future intrusions.

It becomes easy to understand that this attacks cannot easily be detected and they can be so invasive that recoveries often require the re-installation of the operating system from scratch.

2.2 Installation of untrusted software

Another threat to system integrity is the download and installation of untrusted software. Many packages are currently distributed over the network already in binary format with no integrity check information, thus there is no guarantee about the honesty of the application beyond our trust on the download source. In this cases, it is very difficult to realize what the installed application really does at any time.

2.3 Code Injection (Buffer-Overflows)

Buffer-overflow [6] is probably one of the most serious software vulnerabilities. Some applications (for example: daemons that provide network services) do not take much care (or do not take care at all) about the bounds of data areas during their execution. Thus it could happen that providing to that applications an amount of input data greater than the one assumed by their designer, the exceeding part of that data could overwrite areas of the process image that are contiguous to the I/O buffer. In this cases, process data, behaviour and even

execution flow can be altered. Attackers can take advantage from this weakness inserting malicious code into the process image, and then starting its execution. These attacks (and related defences) have been studied following different approaches. We invite to see [5] and [13] for details. As we said above prevention of Buffer-Overflow attacks is out of the scope of this work.

2.4 Run-time Kernel Corruption

Several UNIX-like systems support the capability of loading, on demand, some sections of the kernel in memory at run-time. These sections are named *Loadable Kernel Modules(LKM)*. Usually, modules provide new features to the system as filesystems, device drivers and so on. Unfortunately, there is no way to prevent that an intruder, who as gained root privileges, pushes malicious code into the kernel using a loadable module. As it is shown in [7], a LKM could modify each field of kernel structures, even the system call table. Thus, for example, the intruder could redirect some process system calls to its own table and, in such a way, he modifies the behaviour of processes (even the verification tools) without modifying them.

Moreover, devices representing the memory of the system (e.g., the Linux's `/dev/kmem`), allow processes in user-space (although they are executed with root privileges) to perform read/write operations on the memory and can be used to write malicious code (e.g., stealing secret data, encrypting keys etc.) directly in the memory. Some examples of this attacks can be found in [4, 20].

We point out that no verification strategy can be successfully used if the kernel is not trusted. To perform that, it is necessary that:

1. The Kernel boots in a secure state(see Section 4.1).
2. The Loadable Kernel Modules support is disabled.
3. Write operations with `/dev/kmem`-like devices are not allowed.

3 Related works

In this section we analyze the most used solutions against malicious code, and we motivate the context in which they are useful and their drawbacks.

3.1 Tripwire

Tripwire[11, 12] is one of most widely employed tools to prevent unauthorized modification of files. In practice, Tripwire's approach consists in storing in a secure database a digest of each file actually present in the file system. Periodically, an agent computes the digests of each file in the filesystem and checks it against the digest stored in the secure database. If a mismatch is found an alert message is sent to the administrator. The main weakness of this approach is that malicious code activities are allowed between two executions of the agent. If instead the agent checks the file system very often, then the performance of the system could be heavily affected.

3.2 Java Code Signing

Java classes and files can be grouped in archives (JAR) and the author of the code can append a digital signature to link the code to his identity[21], in a PKI [18] context. When an archive is downloaded the code contained in it is not executed if access control policies of the client application (in general a browser) that downloads it forbid the execution. In general tool is available for the management of access control policies and thus adding a public key (in general embedded into a digital certificate[18]) to a set of trusted entities whose code execution is allowed.

3.3 Adding signatures to binary formats

W. Arbaugh *et al.*[1] proposed a mechanism to sign and verify ELF binaries[9, 10] for the Linux operating system (for an overview of the ELF format see Appendix 7.) The approach of [1] is very close to ours and in the rest of this section we present the architecture of [1] and stress some points which we consider weakness of their approach. Signatures of binaries are computed using the MD5 hash function and the RSA digital signatures scheme[22]. Signatures are added to files by storing them in a new ELF segment that covers each ELF segment involved in the execution. When an executable is loaded, the ELF format manager extracts the signature from the file and verifies each referenced segment. If the verification fails then the execution is not allowed.

The kernel verifies the signature of text segments and provides a new system call: `verify` that should be called by user-space interpreters (`ld.so`, `/bin/sh`, etc.) in order to verify dynamically loaded objects or script files. This choice, however, implies that every interpreter should be modified to request signature verification. On the other hand (as pointed out in [1]), this approach increases the number of points where the signature verification takes place, and does not cover at all those scripts that are given to interpreters as input files (this problem is not solved by our solution either).

The architecture presented in [1] features a cache in order to avoid the verification of signatures at each execution. Each file-cache entry contains the *pathname* of the cached file and the result of the last verification. When the system executes a cached file, it checks the related cache entry. If that entry is still valid (i.e., the file has not been modified since the last verification), the kernel will not verify the file signature again. On the other hand, the kernel traces each `open` invocation and, if the subject of the call is a cached file then the corresponding entry is invalidated. Unfortunately, since it is not possible to trace file accesses on remote filesystems, only files on local volumes can be cached.

Before introducing our approach, we wish to summarize some of the aspects of the work of Arbaugh *et al.* [1]:

- The kernel directly verifies only a part of ELF binaries (the part which is loaded by the `exec` system call).
- Executable scripts, and dynamic segments of ELF binaries, are verified in user space by the respective interpreters.

- In order to verify its executable files, each interpreter has to invoke the `verify` system call. In other words, all interpreters (including `ld.so` and the Perl interpreter, for example) have to be modified in order to verify and run signed executable files.

4 Our Approach

In Section 3, we have described the most used strategies to prevent execution of malicious code on a system, in particular we have illustrated their features and drawbacks.

In our approach we follow the lead of [1] as we believe that validating executable code at run time is an efficient strategy against malicious code, provided it is applied to all executable formats supported by the system.

We emphasize two important assumptions on top of which, we build our infrastructure and that will be further discussed in Section 4.1:

- At the end of the bootstrap, the system is assumed to be in a *secure state*. Thus, we are guaranteed that whenever unauthorized code had been previously injected in the system, it cannot be longer executed.
- Executable files (of any format) have to be signed before they are downloaded and installed. We assume that packages are downloaded from a given set of trusted repositories. In such a way, our system will never execute applications that have not been signed by a trusted entity.

Moreover, our work follows the guidelines listed below:

1. Integrity of executable files has to be verified at run-time.
2. Signed executable should be completely compliant with non-verifying handler (i.e., it must be possible to execute a signed script even on a system that does not feature signature verification.)
3. Integrity verification is under the sole responsibility of the kernel. In such a way, we keep the verification phase inside a *trusted zone*, and moreover, we shall no longer need to modify user-space interpreters in order to allow them to verify the scripts.
4. Private keys should not be in memory, when the system connected to the network and thus potentially vulnerable to attacks.
5. The cost of administration has to be minimal.

4.1 Assumptions of our approach

Secure boot. Our work is based on the assumption that the kernel is trusted. The AEGIS [2] project proposes the design of a secure bootstrap system with a high assurance bootstrap process in which the integrity of the kernel loaded at boot time is guaranteed. In [8] an improvement to AEGIS called sAEGIS has been proposed to protect users from malicious administrators supporting a large set of operating systems. We assume that our infrastructure lies on a secure boot system like AEGIS, and such system takes care to collect (at the bootstrap time) all public keys required to verify all signed executable files.

Signing executable files. We require that before the distribution of a package all executable files have been signed. The entity that distributes the package uses its private key to sign all files that contain executable code and each signature is appended to the corresponding file. This approach perfectly fits the software distribution scenario. Consider a major Linux distribution (SuSE, RedHat, Debian, Mandrake,...) with its pair of public/private RSA keys. When a new release of the distribution is available, all executable files are signed using the private key and then the signed executable files along with the *public key* are released. Once the Linux distribution is installed the signed executable files are present in the system. Each time the execution of a command is requested the signature is checked using the distribution public key.

Since different formats of executable files are structured and parsed in different ways, different techniques for the signature should be specified for each format, in order to allow seamless interoperability of signed and unsigned executables. Thus, we provide a signing tool and a verification procedure for each executable format. For example, in order to allow script to be signed, we provide the utility `scriptsign` and the API function `verify` for the kernel handler scripts. We point out that, as we will see in Section 5, our `verify`, unlike the one of [1] has not to be called by the script (e.g., shell, Perl) interpreter which, consequently, needs not to be modified.

The signature of executable files must be performed by code distributors. The appropriate private key is provided to the signing tools and each file containing executable code must be signed using the appropriate tool for its format.

4.2 Verifying executable code

When a file is executed, the kernel loads it in memory and reads the *magic number* from the file. This number specifies the format of the invoked executable. For each supported format, the kernel has an appropriate handler (e.g., we have the ELF handler, the COFF handler, etc.). Using the magic number, the kernel looks for the appropriate handler, and (if present) executes it.

The handler verifies the integrity of the current file calling the appropriate `verify` procedure. If dynamic parts (e.g., shared objects) are present, they are verified before they are merged in the process image.

We give a look, as an example, to what happens when a signed script is executed. On the basis of the script's *magic number* (the sequence `#!` at beginning of the file), the kernel executes the script-format handler that extracts the interpreter pathname and then runs it providing, as a command-line argument, the invoked script file. The interpreter might be, for example, an ELF binary, so it will be verified by its own handler. In our approach, the verification of the script file is performed by the script handler that accomplishes it by invoking its `verify` procedure. In Fig. 1 we describe the verification steps for scripts performed by our scheme with respect to the Arbaugh's one. Notice that in our scheme the verification step is performed by our handler and then the normal execution proceeds while in the Arbaugh's scheme the verification process is performed by a modified shell script interpreter.

Arbaugh's scheme	our scheme
user runs the bar shell script	
the kernel loader searches for the script handler	
	script handler verifies bar
the kernel loads, verifies and runs:	
(a modified) /bin/sh	the standard /bin/sh
/bin/sh loads and verifies bar	/bin/sh loads bar
/bin/sh executes bar	

Fig. 1. Script file verification

Execution of dynamic libraries is accomplished in the same way: the manager extracts pathnames of each dynamic library used by the application, then it verifies each library. If all verifications succeed, the process execution is allowed (this can appear an expensive task but we remove this drawback by using a caching mechanism).

In Fig. 3 we compare the verification steps for ELF binaries performed by our scheme with the one of Arbaugh. Notice that in our scheme the verification step for both executables and libraries is performed by our handler and then the normal execution proceeds while in the Arbaugh's scheme the verification process is performed by both the kernel that verifies the executable and a modified `ld.so` that verifies the referenced dynamic libraries.

As we can see, in our architecture, the signature verification is always performed by the kernel, so, interpreters needs no modifications.

4.3 Key management

Key management is an important component of our architecture. We do not need to handle any secret key as executables are assumed to be signed off-line by software distributors. It is crucial, however that the list of trusted public key be not manipulated by the intruder.

To allow maximum flexibility, we have abstracted the functionality of the key management into a different kernel module. The communication between the handler and the key management module takes place via an in-kernel API (see Figure 2). In this way different key management schemes can be implemented without affecting the handler.

Furthermore, we provide a set of utilities that allow users to provide/retire keys to/from the key agent, and to sign some kind of executable files.

The key management supports multiple public keys, in order to allow users to verify executable files that have been signed by different software distributors. To do so, each key pair is provided with a *key-id* that identifies it. Key-id is included with the signature and is used by the format handler to ask to the key management for the proper public key.

4.4 Benefits

The architecture presented in this paper can be an useful tool for system security. We point out that its adoption provides for several benefits:

- The system never runs corrupted executables or executables that are not signed (under the assumptions stated at beginning of this section).
- Our architecture has a low impact on the usual system administration duties. Once they are signed, packages can be managed as usual.
- Our architecture is quite transparent from the user’s point of view. In other words, executables verification is completely hidden to users that can do their work as usual.
- Although our implementation currently supports only two of the most widely employed formats (ELF and scripts), we stress again that our system can support any executable file format.
- The verified file caching covers both local and remote files.
- The key management section is completely independent from cryptographic algorithms and from the type of repository that will be used to store keys.

5 Implementation issues

Now we discuss some relevant aspects of our implementation for the Linux operating system. The goal of our experimental work was to provide an implementation for the case of ELF binaries and scripts as a proof of concept. Obviously our implementation can easily accommodate other executable formats (e.g., COFF, a.out). The implementation work can be divided in two parts. First, we modified the kernel handlers for the two formats in order to add the verification capabilities. Second, we developed the utilities (`elfsign` and `scriptsign`) to add signatures to the executables. Although our prototypes use the RSAREF[19] library, we stated that signature algorithms and schemes should be modular, in order to allow the system administrator to choose its favorite system, according to his security and performance considerations.

5.1 Signing and verifying ELF files

Our technique to sign ELF files is quite similar to the one proposed in [1]. We are interested only to the format of an ELF file from an execution point of view

Operation	Subject	Purpose
<i>PUT_KEY</i>	in: <i>key-id</i> , in: <i>public-key</i>	provides the given public key to the agent
<i>GET_KEY</i>	in: <i>key-id</i> , out: <i>public-key</i>	asks to the agent for the specified public key
<i>RESET_KEY</i>	in: <i>key_id</i>	set the given key status to <i>expired</i> and forces the agent to re-read the given key from repository.
<i>RESET_LIST</i>	none	set all keys status to <i>expired</i> .

Fig. 2. Operators provided by the key management.

and thus we only consider the segments of an ELF file (the format of ELF files is described in Appendix A.).

When an ELF file has to be signed, the `elfsign` utility (that has been provided with the proper key pair), computes a hash (MD5) of each file segment, and then computes a digital signature of their concatenation. The identifier of the public key (its MD5 representation) and the computed signature are stored in new ELF sections: the `identifier` section and the `signature` section.

More in detail, we provide a tool for the generation of an RSA digital signature that is appended to an ELF file. The steps performed by the tool are:

1. It computes a digest (using the MD5 function) of each segment contained in the file and then it computes the signature of the digest (still using MD5) of the previous computed digests.
2. The Section Header Table of the ELF file is extended with 2 entries, the identifier entry pointing to the identifier section that contains the hash of the public key and the signature entry pointing to the signature section that contains the digital signature ¹ computed above.

Our signing tool is able to append a digital signature to each ELF file, thus dynamic libraries represented by shared object files (i.e., with extension `.so`) can be signed as well.

As we stated in Section 4, in order to preserve the generality of our system, we chose not to modify the `ld.so` interpreter. Then, our ELF handler extracts pathnames of all dynamic segments and verifies them before the execution of the interpreter (see Fig. 3.)

Arbaugh's scheme	our scheme
user runs the foo ELF binary	
the kernel loader searches for the ELF handler	
ELF handler verifies foo	
ELF handler loads, verifies and runs (a modified) ld.so	ELF handler extracts pathnames of dynamic parts of foo (including ld.so) and verifies them
ld.so extracts pathnames of dynamic parts of foo and verifies them	
foo is executed	

Fig. 3. Elf file verification

¹ The signature is encoded in its primitive binary format, however in the final implementation we will use the portable PKCS7 [15] format.

5.2 Signing Script files

A script file is a text file that contains instructions executable by an interpreter. In the implementation of our prototype we provide a tool to extend a script file with a digital signature codified in PEM [14] format ² (a base64 representation useful to leave scripts as text files). A digest (using MD5) of the file is computed and then signed using the private key. The signature is encoded in PEM and then it is appended to the end of the file bounded by two rows that specify the begin and the end of the signature. The code appended to the original file is not executed by the script interpreter because we add it as a comment for script interpreters.

5.3 Caching verified executable files

The execution of all the verification steps could slowdown the performance of the system. A general strategy for preserving performances in such security setting is the management of a cache for files that have been already verified.

To protect the cache from intruders, we propose a pool of entries each appearing as a device in the filesystem. Each device is actually stored in the kernel memory and thus can only be modified by the kernel.

In order to integrate such mechanism in the infrastructure discussed above we have designed the following strategy:

- when the execution process of an executable file begins the cache is checked;
- if the file is already in the cache then the execution process proceeds using the cached file;
- else the file is inserted in the cache and then the execution process proceeds using the cached file.

This strategy is well combined with the approach discussed in the previous sections in which an extendible infrastructure is illustrated together with the required specifications necessary to add the support of new formats. In fact our cache strategy is independent from the format of the executable files, when a file is executed it is eventually put in cache and then the cached file is executed. When a corresponding handler is found the corresponding verifying code is executed as we have discussed in the previous sections.

5.4 Experimental results

The proposed architecture has been implemented providing tools for the generation of private and public keys, for the signature of ELF and script files, for listing the shared objects referenced by ELF files and for pushing public keys in the device. Modules for the kernel 2.4.17 have been implemented for the run-time

² At the moment the simple structure used in the RSAREF library is codified, however we will use the standard PKCS7 [15] formats in the final implementation of our work.

integrity checking of ELF binaries and shell script files, for the key management and for the cache management.

Our software uses the RSAREF library [19] that provides the required implementations of cryptographic primitives. We have chosen for our implementation only RSA keys because signature verification is faster as we can choose low public exponents.

Experimental results show that run-time integrity checking of executable code can be performed with reasonable performances and that our architecture and implementation are thus an effective solution against malicious code.

Table in Figure 4 shows results of a very preliminary test.

Executable	Not signed	Signed(512 bit key)	Signed (1024 bit key)
/bin/rpm	.026	.079	.083
/bin/gawk	.008	.017	.022
/bin/tar	.006	.018	.023
/usr/bin/emacs	.014	.124	.129
/usr/bin/gdb	.013	.077	.082

Fig. 4. Performances of Signed ELF executable files. Execution times are in seconds and have been measured on Pentium III 500 MHz with 128 Mb RAM and kernel 2.4.17.

6 Conclusions

We have presented a solution for run-time integrity checking of executable code. Our architecture can be used for any type of executable file format provided that a tool for the signature and a kernel handler are available. A format independent cache management mechanism has been designed and implemented to improve performances. We provide a proof of concept implementation for ELF and script files.

Our approach is suitable for workstations that provide services to end users and not for developing applications. In fact the simple compilation of a program generates an executable file that can become trusted only if the author signs it and its public key is in the list of trusted public keys.

Experimental results show that our solution is very efficient but other issues have to be addressed in order to protect a workstation from attacks based on malicious code.

Attacks to our infrastructure are possible by changing the kernel during its execution, however such attacks are not as easy as the installation of a root-kit and can only have a temporary success.

Next we point out two drawbacks of our approach and leave them as open questions. First of all, we observe that scripts that are invoked by passing the name of the file as a command line argument to the interpreter are not verified at all. Our architecture only guarantees that the interpreter is verified. One solution,

as suggested by [1], is to modify all interpreters thus losing the universality of the approach. Moreover we believe that is not an easy task to modify complex objects like the Perl interpreter and thus a better solution to this problem is needed. A second weakness of our architecture regards dynamic libraries. In our architecture, dynamic libraries that are specified at linking time (and thus are referenced in the executable) are verified. On the other hand, the case of dynamic libraries loaded at run time that are not referenced in the executable is more complex. In the Linux operating system there are two ways for an application to load a dynamic library at run-time. The first way is to invoke the `uselib` system call, which in turns invokes the handler for the specific format (i.e., the handler for ELF) and thus the library can be verified. As second way involves the use of the library function `dlopen`. This function uses the `ld.so` interpreter to load the library which is then directly memory-mapped. In this way, our checking of signatures is bypassed and malicious code could be executed. We are not sure that the use of `dlopen` is the right way of loading dynamic libraries as most of the work is done at user level and thus escape the checking of the kernel as opposed to `uselib` that makes sure that the loading is performed by the kernel. At the same, we observe that the `dlopen` mechanism is used by several applications and thus a solution to this problem is needed. One easy way would be to patch `ld.so` but we would like to see a more general approach.

7 Acknowledgments

We thank Marco Cesati, Louis Granboulan and the anonymous reviewers for their remarks about our work.

References

1. W. A. Arbaugh, G. Ballintijn, L. van Doorn: Signed Executables for Linux. Tech. Report CS-TR-4259. University of Maryland, June 4, 2001
2. W. Arbaugh, D. Farber, J. Smith: A Secure and Reliable Bootstrap Architecture. Proceedings of 1997 IEEE Symposium on Security and Privacy, pp. 65–71. May 1997.
3. S. Cesare: Unix ELF parasites and virus. Unpublished technical report. <http://www.big.net.au/silvio/elf-pv.txt>
4. S. Cesare: Runtime Kernel KMEM Patching. Unpublished technical report. <http://www.big.net.au/silvio/runtime-kernel-kmem-patching.txt>
5. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. Proc. 7th USENIX Security Conference, pp. 63-78. San Antonio Texas, Jan. 1998
6. C. Cowan, P. Wagle, C. Pu, S. Beattie, J. Walpole: Buffer Overflows: Attacks and Defenses for the Vulnerability of the decade. DARPA Information Survivability Conference an Expo (DISCEX). Hilton Head Island SC, Jan. 2000
7. Halfife: Bypassing Integrity Checking Systems. Phrack, issue 51. September 1997.

8. N. Itoi, W. A. Arbaugh, S. J. Pollak, D. M. Reeves: Personal Secure Booting. Proceedings of Australian Conference on Information Security and Privacy, pp. 130–144. Sydney, July 11-13, 2001
9. Tool Interface Standards Committee: Tool Interface Standards (TIS) Portable Formats Specification version 1.1. <http://developer.intel.com/vtune/tis.htm>, October 1993
10. H. Lu: ELF: From the programmer perspective. <http://citeseer.nj.nec.com/lu95elf.html>. May 17, 1995
11. G. H. Kim, E. H. Spafford: The design and Implementation of Tripwire: a System Integrity Checker. Proceedings of Conference on Computer and Communications Security, pages 18-29. Fairfax (Virginia), 2-4 November 1994
12. G. H. Kim, E. H. Spafford: Experiences with Tripwire: Using integrity checkers for intrusion detection. In Systems Administration, Networking and Security Conference III. USENIX, April 1994.
13. C. Ko, T. Fraser, L. Badger, D. Klipatrick: Detecting and Countering System Intrusions Using Software Wrappers. Proceedings of the 9th USENIX Security Symposium. Denver, Colorado, August 14-17, 2000.
14. J. Linn: Privacy Enhancement for Internet Electronic Mail. PKIX Working Group, RFC1421, February, 1993.
15. RSA Laboratories: PKCS7 Cryptographic Message Syntax Standard. <ftp://www.rsasecurity.com>, November 1, 1993
16. S. McCanne, V. Jacobson: The BSD Packet Filter: a new architecture for user-level packet capture. Proceedings of the 1993 winter USENIX conference, pp. 259-269. San Diego CA, 1993.
17. Sun Microsystems Corporation: Java Code Signing. <http://java.sun.com/security/codesign>, 1996
18. R. Housley, W. Ford, W. Polk, and D. Solo: Internet X509 Public Key Infrastructure: Certificate and CRL Profile. Network Working Group, RFC 3280, April, 2002
19. RSA Laboratories: RSAREF: A Cryptographic Toolkit for Privacy-Enhanced Mail. <http://www.aus.rsa.com>, 1994
20. SD: Linux on-the-fly kernel patching without LKM. Phrack issue 58, December 2001
21. Sun Microsystems: Java™ Security Evolution and Concepts. Technical Articles. <http://developer.java.sun.com/>
22. D. Stinson: Cryptography: Theory and Practice. CRC Press.

Appendix A: The ELF Binary Format

The Executable and Linking Format [9,10] was developed to provide a binary interface that is operating system independent. Three types of ELF files have been identified:

1. *relocatable files* that can be linked to have an executable or a shared object file (these are the `.o` files);
2. *shared object files* that can be used by the dynamic linker to create a process image (these are the `.so` files);
3. *executable files* that hold code and data suitable for the execution.

From the execution point of view, an ELF file starts with the ELF header, which is followed by the program header table, the segments and the optional section header table (see Fig. 5.)

The ELF header describes the organization of the file and its fields specify the offsets of the *program header table* and of *section header table*, the size of an entry of each header table and the number of entries in each header table.

The program header table has the information needed to locate the *segments* of the file that contains data required to create a process image. The program header has an entry for each segment that is specified, among other information: location and type. The segment type specifies the purpose of the segment and how it can be used by the loader. For the aim of this brief overview, we are interested in: PT_LOAD, PT_DYNAMIC and PT_INTERP segments. *LOAD* segments are mapped in the process memory image. They include text of executable files, data and so on. *DYNAMIC* segments include references to other ELF files that will be loaded into the process image. These entries, are usually (but not exclusively) related to dynamic libraries. The *INTERP* segment (that appears only once), contains the reference of an *interpreter*, a program that allows ELF executable files to load shared object dynamically at run-time. The system retrieves the interpreter and maps it in the process image. The interpreter provides the process of an environment, and takes care to load dynamic segments. Since the interpreter is loaded into a new process image, it is executed in user space.

The section header table instead has the information required to locate the *sections* of the file which are used for linking. The section header table has an entry for each section that specifies the offset, the size of the section and the type of section. The program header table is required for executables and shared object files and is optional for relocatable files. On the other hand, the section header table is required for relocatable objects and is optional for executable files and shared objects. Moreover, from inspecting executable files generated using the `gcc` compiler, we noticed that executable files always have a section header which is always found at the end of the file.

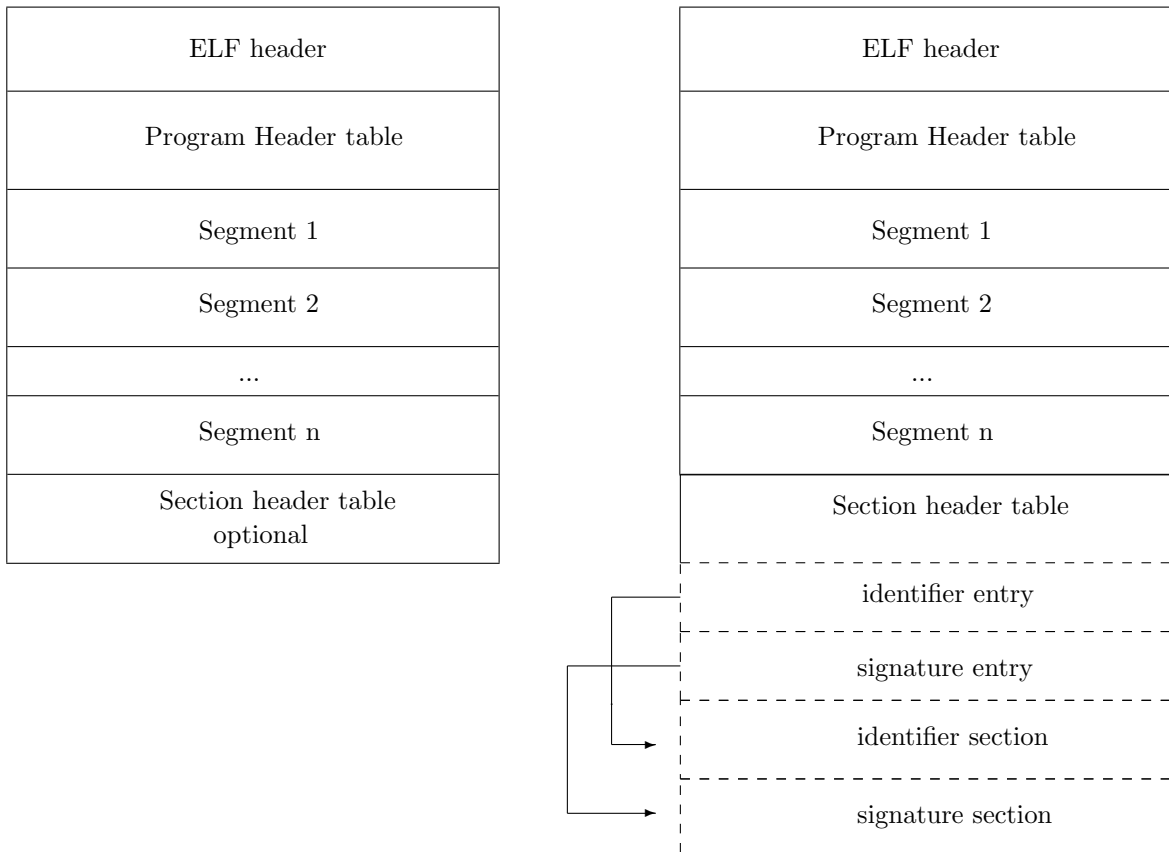


Fig. 5. Format of ELF files.