DigSig: Run-time Authentication of Binaries at Kernel Level

Axelle Apvrille – Trusted Logic
David Gordon – Ericsson
Serge Hallyn – IBM LTC
Makan Pourzandi – Ericsson
Vincent Roy – Ericsson

ABSTRACT

This paper presents a Linux kernel module, DigSig, which helps system administrators control Executable and Linkable Format (ELF) binary execution and library loading based on the presence of a valid digital signature. By preventing attackers from replacing libraries and sensitive, privileged system daemons with malicious code, DigSig increases the difficulty of hiding illicit activities such as access to compromised systems.

DigSig provides system administrators with an efficient tool which mitigates the risk of running malicious code at run time. This tool adds extra functionality previously unavailable for the Linux operating system: kernel level RSA signature verification with caching and revocation of signatures.

Introduction

In the past years, the economical impact of malware-like viruses and worms has regularly increased. Even though the target platform of many malware is Windows, with the increasing popularity of Linux as a desktop platform and its wide use as public server, the risk of seeing viruses or Trojans developed for this platform is rapidly growing.

These malware can be installed on the system through different sources. On desktop systems, a major source of malware lies in careless users who introduce viruses, worms, Trojans, or other nuisances through email attachments or internet downloads of Trojaned software.

On server side, very often, vulnerabilities like buffer overflows in public services are exploited by the attacker to install rootkits to replace system binaries and libraries with Trojaned versions. These rootkits are then used to assure a continued access to a compromised machine and mask attacker's illicit activity. For instance, the *Remote Shell Trojan* (RST) infects all ELF binaries of the /bin directory, offering a backdoor process with a command shell at the privilege level of the invoking user.

Even though there are actually different origins to the spread of these malware, the final result is often the same, an unauthorized binary running on the system.

To mitigate this risk, system administrators commonly deploy restrictive solutions such as firewalls, virus scanners, or intrusion detection tools. Although those tools do have positive impact on system security, they have proven to be insufficient; a firewall for instance is usually incapable of detecting covert channels. Papers such as [1] have already raised the alarm, and [2] even compares firewalls to the French Maginot line.[1] Virus scanners and intrusion detection systems also show several limits, such as their incapacity to detect totally new viruses or intrusions. Indeed, their detection engine most usually relies on an extensive signature database, sometimes enhanced with an heuristic algorithm to detect known viruses/intrusions and their close cousins. This results in a time gap between the first spread of new viruses and their characterization through signatures. This gap can be used by attackers to penetrate the internal network of the company. In theory, only few systems based on users' normal behavior or misuse detection model are capable of detecting brand new viruses; however, they are more at research stage yet [5].

Supporting the concept of defense in depth, this paper consequently proposes a new layer of defense to existing mechanisms, named *DigSig*. Used in addition to firewalls, virus scanners, or IDS, this paper highlights how DigSig can significantly increase security. DigSig does not prevent malicious applications to be installed on the system, but prevents their execution, which is when they actually become dangerous.

The paper is organized as follows. First, we describe how DigSig enforces digital signature verification at ELF file loading time. Second, we explain how system administrators would typically deploy DigSig on one or several Linux hosts. Then, we focus on the security aspects of this kernel module and in

[1] For readers not familiar with the history of the second world war, the German army went around the main French defense line, called Maginot and defeated the French army in 40 days!

what way it counters attacks. We analyze the performance impacts of DigSig. Finally, we mention some related work, including some complementary systems DigSig might be added to.

DigSig Kernel Module

DigSig is implemented as a Linux kernel module, which checks that loaded binaries and libraries contain a valid digital signature. In the case of an invalid signature for the binary or for any of its shared libraries, the execution is aborted.

When an ELF file is to be loaded into an executable memory region, DigSig searches the file for a signature section. If no such section is available, loading is refused. Otherwise, DigSig hashes the contents of all text and data segments, and compares the result to the contents of the signature section decrypted with a system's public key. If values do not match, this means the file was not signed with the corresponding private key, or that it was modified after signing. In such a situation, loading is refused.

An attacker who now attempts to replace *ps, ls, sshd, libc*, or any other common rootkit target with Trojaned versions will find these files cannot be executed.

DigSig adds a new section into the ELF binary; therefore, it works only with binaries with ELF format. As ELF is the predominate format in Linux systems, compiling a kernel with only ELF support should not cause problems. In this article, whenever mentioning binaries, only the case of ELF formatted binaries is considered. At this time, DigSig also does not cover the case of scripts. This is outside the scope of this paper, but will be addressed in the future.

The following subsections detail the implementation of DigSig.

Signing the Binary

Before verifying the signature of a binary, the binary needs to be signed and the signature stored in order to retrieve it. Executables and libraries are initially signed offline, using the Debian userspace package *BSign* [3]. BSign embeds an RSA signature of all text and data segments into a new ELF segment called the "signature" segment (see Figure 1). Then, once signed, the RSA private key is safely stored offline and removed from the system, while the corresponding public key is loaded in the DigSig kernel module.

Verifying the Signature of the Binary

At execution time, the DigSig kernel module verifies the signature of the binary/library. This requires the following functionality at kernel level:

- **Support for hash functions**: The first step of each digital signature consists of hashing the data to sign. This is provided by the *CryptoAPI*, which is part of Linux 2.6.x main stream kernels.
- **Public key cryptography**: This is in order to verify the signature.
- **Executable file loading mediation**: This allows us to verify a signature before running the binary, and optionally refuse execution.

The RSA algorithm is used for verifying the signature of the binary. As there is currently no native implementation of RSA at kernel level, we had to import our own implementation into the kernel. Yet, in cryptography, re-inventing the wheel often turns out to be extremely dangerous, so it was decided **to use the well-tested, GPL'ed GPG implementation of RSA** and port the necessary parts for use in kernel space. In order to avoid bloating in the kernel, only 10% of the original code of GPG has been imported. This code is currently isolated in a specific directory (digsig/gnupg) of DigSig.

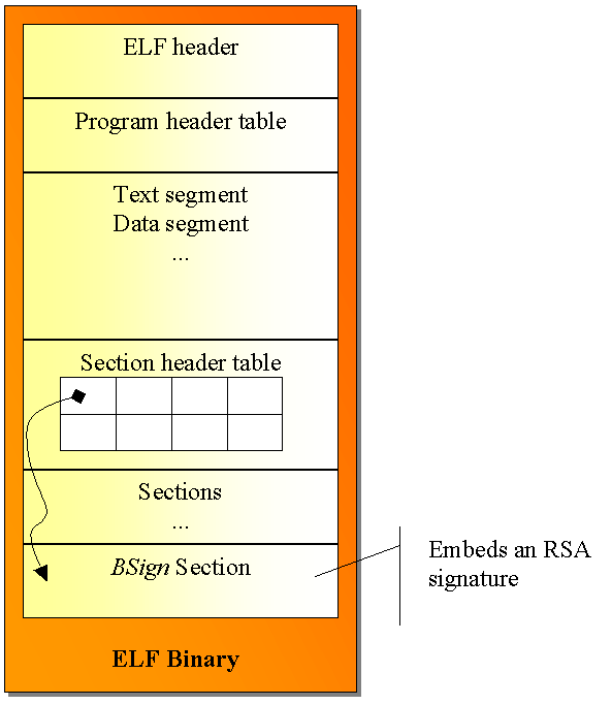

Figure 1: Sign signature section as added in an ELF binary.

As for mediating the loading of executables, DigSig uses the Loadable Security Modules (LSM) architecture [6], which has now become an established part of the kernel. It allows a module to define hooks to annotate kernel objects with security data and mediate access decisions for such objects. One such hook, security_file_mmap, is called whenever mmap is called to map a file to a memory region. This is done by sys_execve to load binaries, as well as by dlopen to load shared libraries. The mmap hook is consequently a convenient location for DigSig to mediate executable mapping of ELF binaries [7].

Caching and Revocation Lists

In order to increase performance of signature verification, DigSig caches a list of binaries whose signatures have already been verified. The first time

an ELF binary or library is loaded, its signature is verified. If the signature is correct, then the successful signature verification is cached. In subsequent loads, DigSig only checks the presence of this signature validation in the cache. This results in a significant improvement in performance, as detailed later. If the file is later opened for writing, the security_inode_permission LSM hook will be triggered, to which DigSig will respond by removing its signature from the cache. The size of the signature verification cache can be specified at module load time, but defaults to 512 signature verifications.

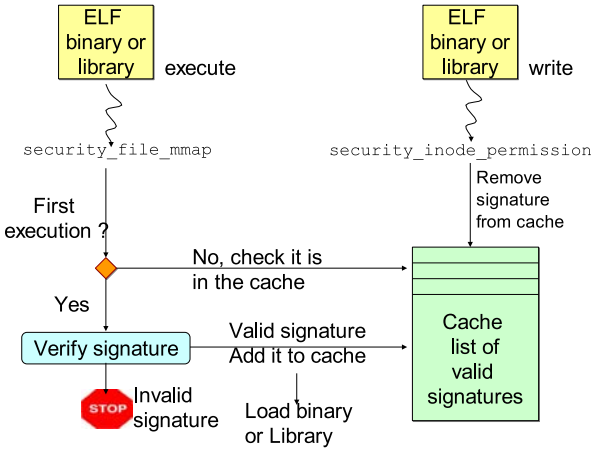

Figure 2: DigSig's caching mechanism.

The introduction of a signature validation cache might seem like a step toward a binary whitelist. In particular, it might seem a simpler solution to eliminate digital signatures altogether, and keep only a whitelist of acceptable binaries and libraries. Files on the whitelist would be executable but not writable, and files not on the whitelist would not be executable. This approach would eliminate the need to verify cryptographic signatures on each file on the first load. However, the DigSig approach has several advantages. First, files with cached signature validations that are not being executed can still be written to. Their signature validation will merely be removed from the cache. Second, the signature validation cache need not be updated to install new libraries or executables. Therefore, software can be installed and upgraded on a live system, and a DigSig system in many ways acts as a normal Linux system. Third, the signature validation cache cannot be edited from user-space, preventing attacks against the cache that an editable whitelist might be subject to. Fourth, a DigSig system can use a relatively small cache, as its sole purpose is performance improvement. A pure whitelist system would need to keep entries for every binary and library.

DigSig also implements a signature **revocation list**, initialized at startup and checked before each signature verification. When programs that were previously signed with the correct private key are later found to contain vulnerabilities, their signature may be revoked by adding them to the revocation list. This is particularly convenient because it eliminates the need to generate a new key pair and resign all binaries and libraries on the system just for a few revocations. The revocation list is communicated to DigSig using the sysfs filesystem, by writing to the /sys/digsig/digsig_revoke file.

Therefore, DigSig allows the binaries to be easily updated over time. This simplifies the evolution of the system over time which is a major requirement for many systems where the binaries can be changed or added over the lifetime of the system.

Deploying DigSig

DigSig requires neither kernel patching nor recompilation of the binaries. Therefore, there are no major changes in the system necessary to deploy DigSig.

DigSig installation requires three initial steps:

1. Generate an asymmetric key pair (for instance using GPG).
2. Sign all trusted binaries and libraries (with BSign).
3. Modify the system startup procedure to load DigSig.

From a deployment point of view, the first step raises the issue of key storage. Obviously the private key should be kept confidential. To this end, Bsign suggests it should be kept on a removable physical support such as a floppy disk or a flash memory key or a read-only CDROM. It is also a good idea to keep a hash of the public key, for instance using sha1sum, to check that an attack hasn't replaced the real public key with another one. This approach is particularly convenient in centralized networks where users connect from remote terminals onto a single application server: DigSig only needs to be deployed on the server to secure execution of all users. However, in networks of individual PCs or laptops, a compromise needs to be made between having one key per host (good security, but perhaps a burden for administrators) and having a single shared key for all (security issue: if one host is compromised, all are). Such issues are not specific to DigSig, but general to PKI.

As for the second step, signing all binaries and libraries can easily be done through the following command:

```
bsign -s -v -I -i / -e /proc \
        -e /dev -e /boot -e
/usr/X11R6/lib/modules
```

It takes about an hour to sign *all* binaries of a Fedora Core 1 installation using Bsign 0.4.5 on a Pentium 4 2.2 GHz with 512 MB of RAM. Note that the revocation list system reduces the need for re-signing the whole system. This fact is particularly important for systems that wish to offer uninterrupted service.

The third step only requires minor modification of the operating system's startup files. Automatic loading of the DigSig kernel module can be achieved by adding DigSig to /etc/modules, and appropriate options in /etc/modutils.

Security Considerations

In this section, a short security analysis of DigSig is conducted to help system administrators understand under what circumstances DigSig does or does not increase security. This assists system administrators in deciding the best approach in deployment.

DigSig operates as a kernel module. It thus requires root privileges for loading and unloading the module, and assumes the secrecy of the DigSig private key, the integrity of its public key, root access to the system, and the Linux kernel itself are not compromised. For the rest of the study, these requirements are taken for granted; however, please note this is not always the case (see the section on related work).

It is important to understand DigSig has not been designed for *vendors* but rather for *system administrators*. System administrators have total control over what should, or shouldn't, execute on the machines they administrate. There is no way a vendor can hope to lock up a given machine to a given software unless with the system administrator's consent. In brief, DigSig targets more prevention against attackers than DRM or software version management. Its two major goals are the following:

1. If a binary has been signed, no one can modify the binary without the modification being detected.
2. Nobody can execute or load an ELF binary or library unless it has been signed.

However, note DigSig cannot protect a system from vulnerabilities within legitimately installed and signed software. Let's see how DigSig achieves security. First, we detail how DigSig prevents modification of ELF binaries. Second, we examine the case of libraries. Finally, we analyze the security of the signature caching and revocation mechanisms.

As described earlier, when a file with a cached signature verification is opened for writing, the signature verification is removed from the cache. However, this does not protect files that are still being executed. Fortunately, the second protection comes from the Linux kernel itself: the kernel forbids executing a file that is opened for writing and reciprocally. This is accomplished by calling deny_write_access(file) kernel hook, and even the superuser is subject to such restrictions.

Unfortunately, the same defense is not extended to shared libraries. Worse, the deny_write_access function is not exported to kernel modules. DigSig must therefore implement its own protection for libraries, which it does very similarly to the kernel. It blocks any attempt to mmap a library with executable permission if that library is already open for writing. If the mmap succeeds, then DigSig increments a usage counter for the inode. So long as the usage counter shows the file to be in use as a library by some process, no one, including the superuser, may open the file for writing.

Under some circumstances, these defenses may still not be sufficient. In particular, deny_write_access (file) and the DigSig shared library writer lock work by marking the VFS inode. They are therefore restricted to a single machine. An NFS mounted file being executed on one client, for instance, could be modified on the server or on any other client. To reduce this threat, DigSig does not cache signature verifications for NFS mounted files. However, this does not protect NFS mounted files while they are in use.

As for the signature caching mechanism, one might fear it introduces a possible attack point. However, since the cache is stored in kernel memory, user space programs cannot directly insert fraudulent signature validations. Signatures may be added to the cache only through a non-exported function dsi_cache_signature, which is only called in one place, when a signature has in fact been validated during dsi_file_mmap. While a user-space application cannot directly inject fraudulent signature validations, a Trojan kernel module could of course do so by directly manipulating memory. However, a Trojan kernel module could also stop DigSig altogether [10], so this is not a valid argument against signature caching.

Finally, it is important to note the signature revocations open the possibility of denial of service. It is vital that an attacker not be able to add *valid* signatures to the revocation list. To ensure this, DigSig restricts access to the communication interface (/sys/digsig/digsig_revoke) to root, so that only root can provide revocation lists to DigSig. As a further precaution, revocation lists can only be appended to before DigSig begins enforcing; that is, before a public key is provided. Care should therefore be taken to ensure that DigSig is enabled before the system is exposed to threats, such as before a network connection is enabled if the network is the primary means of attack. Additionally, the integrity of the collection of revocations must be guarded. For instance, they can be stored on a read-only media (such as a cdrom), or simply signed by GPG.

Performance

Figure 3 presents DigSig's overhead according to the execution time. In the following, all measures are done with a key size of 1024 bits for the RSA algorithm and use of SHA-1. The overhead induced by DigSig grows linearly with the size of executables. However, the gradient is very small: approximately only 0.0016 microsecond per byte.

This is not believed to be critical because unlike Windows operating systems, Unix systems only have few very large executables. As a matter of fact, a typical

Debian Woody workstation only shows 1.8% of executables and libraries above 512 KB.

As DigSig's signature verification is performed once, at the beginning of load time, it is important to note that its induced overhead naturally decreases for long-lived applications. Yet, on Unix systems, administrators and users keep on executing small commands such as *ls*, *cp* and *cd*. In such cases, the cost of signature verification is amortized by DigSig's signature cache (see the second section).

```
Kernel without DigSig
    real    0m0.004s
    user    0m0.000s
    sys     0m0.001s
DigSig without caching
    real    0m0.041s
    user    0m0.000s
    sys     0m0.038s
DigSig with caching
    real    0m0.004s
    user    0m0.000s
    sys     0m0.002s
```

Figure 4: Time required for "/bin/ls -Al".

The efficiency of the caching system is demonstrated by Figure 4. This figure displays the average execution time, in seconds, when running a typical texttt-tls -Al command 100 times, using the Unix time command. The benchmark was run on a Linux 2.6.6 kernel with a Pentium 4 2.2 Ghz, 512 MB of RAM. As signature validation occurs in execve, DigSig's overhead is expected to show up during system time (sys). The benchmark results clearly highlight the improvement: there is now hardly any impact when DigSig is used.

Finally, to provide a better insight into the actual impact of DigSig on real workloads, three kernel compiles were timed on a non-DigSig system, and three on a digsig system. The tests were performed using a 2.6.7 kernel on a Pentium 4 2.4 GHz with 512 MB of RAM. The kernel being compiled was a 2.6.4 kernel, and the same .config was used for each compile. Each compile was preceded by a "make clean". Results are shown at Figure 5. The first execution time, both with and without DigSig, appears to reflect extra time needed to load the kernel source data files from disk.

```
Kernel without DigSig
    real            sys
    19m21.890s      1m27.992s
    19m  9.276s     1m26.584s
    19m  9.464s     1m26.191s
    19m  7.717s     1m25.799s
Kernel with DigSig
    real            sys
    19m19.957s      1m28.541s
    19m  7.485s     1m26.832s
    19m  7.883s     1m26.549s
    19m  6.494s     1m26.618s
```

Figure 5: Time required for 2.6.4 kernel "make".

Related Work

This section presents a few related tools that more or less have the same goals as DigSig, but also supplementary work that can be used together with DigSig.

As previously stated, on a security point of view, DigSig assumes the root account has not been compromised. In circumstances where this is unacceptable, there are ways to circumvent this requirement.

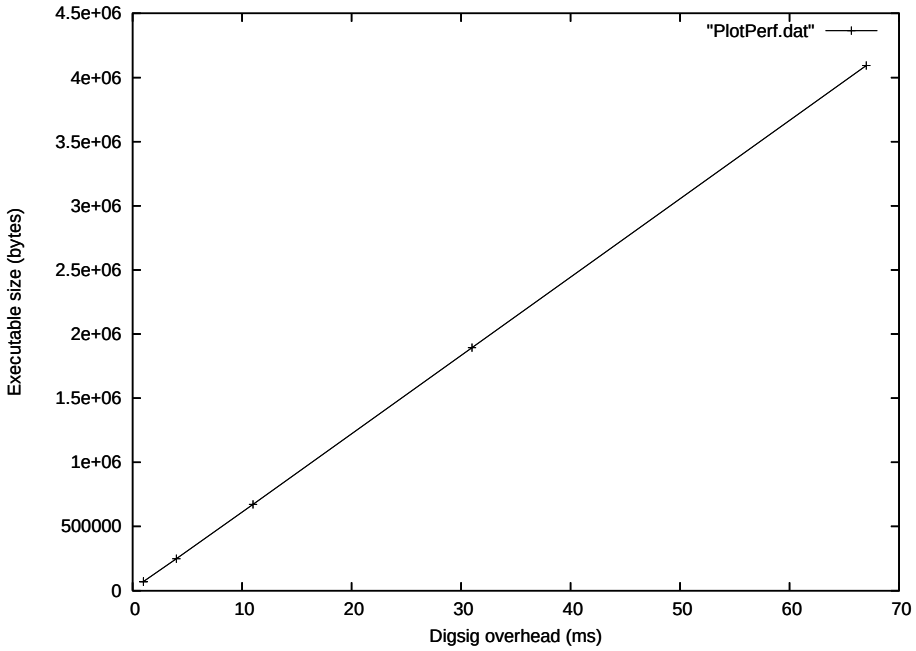

Figure 3: DigSig overhead (ms) for the first load (without caching) per executable size (bytes).

One solution, well known in the Linux domain, relies on SELinux [9]. This security enhanced Linux proposes an implementation of Mandatory Access Control policies, where access control on objects is set according to their sensitivity and not necessarily by their owner. An immediate consequence to this model is that root becomes much less powerful. On ''normal'' Unix operating systems, root is a super user with super powers. With SELinux, root does not necessarily have access to all objects; this limits risks in case of root compromise. Permissions are specified at a very fine grained level and include access to files, shared memory, POSIX capabilities, and sockets, among others. SELinux could be used in addition to DigSig, to provide the needed secrecy and integrity guarantees of DigSig keys and revocation lists, even in the case of a root compromise.

Another alternative relies on Trusted Computing solutions, such as the specifications provided by the Trusted Computing Group (TCG) [11]. In TCG, normal PC architecture is enhanced with a small security hardware called the *Trusted Platform Module*. In particular, the TPM offers protected storage of data, irremediably binding data to *Platform Configuration Registers* (PCRs). The secret stored on the TPM may only be retrieved by the TPM owner if the configuration of the PC (held in PCRs) hasn't changed. This offers two levels of protection in case of root compromise. First, the TPM owner and root are not necessarily the same person. Second, if root account has been compromised, most of the time this is due to a malicious application (a rootkit) has been installed. Fortunately, installing a rootkit impacts the PC's configuration, so the TPM forbids retrieval of secrets it stores. Sample implementations of trusted computing on Linux can be found in [12, 13, 14].

SELinux and Trusted Computing largely encompass DigSig's goals. However, they offer practical opportunities to supplement DigSig in stricter security sensitive situations. On the other hand, disadvantages of such solutions rely on their complexity, and on the need for specific hardware in the case of trusted computing. On the contrary, DigSig's small size makes it easy to install, configure or re-use.

DigSig may also be compared to similar tools such as PaX [15] and ExecShield [16]. It is important to note that PaX and ExecShield do not exactly have the same goal as DigSig; they attempt to prevent software exploits from being used to execute arbitrary code. This is done by placing strict limits on mmaped regions, and by using address space randomization. In brief, PaX and Exec protect the system against *exploits* of malicious code, while DigSig prevents malicious code from *being executed*.

Tripwire is in some respects similar to DigSig. It maintains a signature database of all files on a machine, and notifies the administrator when some of them are modified (possibly replaced by Trojaned versions for instance). However there are two major differences from DigSig. First, Tripwire works at user level, not within the kernel. Second, it does not provide on-the-fly verification of file signatures. For instance, there is no way to trigger signature verification when a binary is executed. So, Tripwire could more accurately be compared to an off-line file integrity verification tool.

Closer to home, there is a Linux kernel patch written by Greg Kroah-Hartman, from the IBM Linux Technology Center. It is a proof of concept implementing digital signatures in kernel modules. Although it does not check binaries and has no use for caching, it is complementary to DigSig in that the latter does not check Linux kernel modules. The patch modifies a file called module.c, which is responsible for kernel module handling. Unfortunately, LSM does not provide any hooks here. Overall, being a proof of concept, the patch does not benefit from any form of benchmarking or flexibility.

Availability

DigSig is available from SourceForge at http://disec.sourceforge.net. It is available under the GNU Public License version 2. BSign is available with the Debian project, from http://packages.debian.org/unstable/admin/bsign.html.

Conclusion

In this paper, a new tool, named DigSig, is presented. DigSig answers the needs of system administrators in terms of run-time security of Linux operating systems. It focuses on preventing execution of malicious code (ELF binaries or libraries) by checking an embedded RSA signature for each file. The implementation is based on LSM hooks and optimized with a signature caching and revocation mechanism.

On a deployment point of view, the paper has shown that DigSig introduces only little additional installation and management effort. In particular, the initial setup of the system which consists insigning all valid binaries and libraries can be launched once and for all by a single command. Future sporadic changes on the system do not require this step and are handled by DigSig's signature revocation mechanism. From a security point of view, the paper has also demonstrated DigSig is believed to be safe, under reasonable assumptions for most security environments.

DigSig has also been benchmarked, and the results indeed show a very small overhead at load time (a few nanoseconds per byte of executable's size) and even less with the signature caching mechanism. It is therefore reasonable to conclude DigSig should not impact machine's performance from an end-user point of view.

Finally, the paper has presented some other related work. Some, such as SELinux and TCG, may be used in addition of DigSig to supplement it. Others,

such as ExecShield, Tripwire, present similarities and differences that make them suitable for other situations. To our knowledge, at this time, DigSig is the only GPL'ed run-time executable signature verification integrated to the Linux kernel.

In the future, we mainly hope to extend our work to protect Linux systems against malicious shell scripts.

Acknowledgements

The authors would like to thank the LISA reviewers for helpful comments about the extended abstract. They also thank the LISA typesetter, Rob Kolstad, for helpful information as well as for taking the formatting concerns off our hands.

Authors

Axelle Apvrille (axelle.apvrille@trusted-logic.fr) is a senior computer security engineer, currently working for Trusted Logic, in Sophia Antipolis, France.She received her computer science engineering degree in 1996 at ENSEIRB, Bordeaux, France, and then specialized in computer security working at MSI S.A. and in research laboratories of StorageTek and Ericsson Canada. She holds several patents and papers in magazines and international conferences.Her main interests are cryptography, security protocols and embedded security.

David Gordon has a bachelor's degree from the university of Sherbrooke. His interests include security and next-generation networks.

Serge Hallyn graduated from Hope College with a B.S., and the College of William and Mary with M.S. and Ph.D. in computer science. He currently works with the security team at the IBM LTC in Austin, TX. He can be reached at serue@us.ibm.com.

Makan Pourzandi (makan.pourzandi@ericsson.ca) works for Ericsson Research Canada in the Open Systems Research Department. His research domains are security, cluster computing, and component-based methods for distributed programming. He received his doctoral degree on parallel computing in 1995 from the University of Lyon, France.

Vincent Roy is a student in electrical engineering at the University of Sherbrooke. He worked at Ericsson during summer 2004. He can be contacted at gaspoucho@yahoo.com .

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of IBM.

Bibliography

[1] Loscocco, P., S. Smalley., P. Muckelbauer, R. Taylor, S. Turner, and J. Farrell, "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments," *Proceedings of the 21st National Information Systems Security Conference (NISSC)*, October 6-9, 1998.

[2] Frankston, W., *Firewalls: The New Maginot Line*, http://www.frankston.com/public/Essays/Firewalls.asp , February 1998.

[3] BSign Debian package, http://packages.debian.org/stable/admin/bsign .

[4] Apvrille, A., M. Pourzandi, D. Gordon, and V. Roy, "Stop Malicious Code Execution at Kernel Level," *Linux World Magazine*, Vol. 2, No. 1, January 2004.

[5] Sinclair, C., L. Pierce, and S. P. Matzner, "An Application of Machine Learning to Network Intrusion Detection," *Proc. of 15th Annual Computer Security Applications Conference (ACSAC)*, Phoenix, Arizona, 1999.

[6] *Linux Security-Module (LSM) Framework*, http://lsm.immunix.org .

[7] *Tools Interface Standards and Manuals, ELF: Executable and Linkable Format*, http://www.x86.org/intel.doc/tools.htm .

[8] *The Distributed Security Infrastructure Project (DSI)*, http://disec.sourceforge.net .

[9] NSA, *Security Enhanced Linux*, http://www.nsa.gov/selinux/index.cfm .

[10] Truff, "Infecting Loadable Kernel Modules," *Phrack Magazine*, Vol. 11, No. 61, File 10/15, August 2003.

[11] The Trusted Computing Group, http://www.trustedcomputinggroup.org .

[12] Marchesini, J., S. Smith, O. Wild, and R. MacDonald, *Experimenting with TCPA/TCG Hardware, Or How I Learned to Stop Worrying and Love the Bear*, Technical Report TR2003-476, December 15th 2003.

[13] Sailer, R., Y. Zhang, T. Jaeger, and L. van Doorn, *Design and Implementation of a TCG-based Integrity Measurement Architecture*, IBM Research Report RC23064, January 16th, 2004.

[14] Safford, D., J. Kravitz, and L. van Doorn, "Take Control of TCPA," *Linux Journal*, Issue 112, http://www.linuxjournal.com/article.php?sid=6633 , August 2003.

[15] Busser, Peter, "Memory Protecting with PaX and the Stack Smashing Protector," *Linux Magazine*, Issue 40, March 2004.

[16] Exec-Shield, http://people.redhat.com/mingo/exec-shield .

[17] Kroah-Hartman, Greg, "Signed Kernel Modules," *Linux Journal*, Issue 117, http://www.linuxjournal.com/article.php?sid=7130 , January 2004.