ATTACKING SIGNED BINARIES

Marco Slaviero, Jaco Kroon, Martin S Olivier

ICSA Research Group
Department of Computer Science
University of Pretoria
Pretoria
0002

{mslaviero,jkroon,molivier}@cs.up.ac.za

ABSTRACT

The digital verification of binaries at the kernel level has been proposed as a method to prevent trojaned programs and unauthorised execution. However, the nature of attacks which various signed binary schemes seek to prevent vary quite considerably. Further, unrealistic assumptions are often made as to the security of the environment in which the verification takes place.

In this paper, the authors explore one such kernel-level verification tool, DigSig, and show how the security assumptions that DigSig makes are too broad. Various attacks which succeed given a reduced set of assumptions are then demonstrated. A number of recommendations are made, which alleviate most attacks described without requiring a vastly more complex system.

KEY WORDS

digital signature, digsig, pre-execution validation, run-time verification,

1 INTRODUCTION

Attacks on network-connected machines have grown as the total number of network links has increased. The nature of attacks vary considerably, but in a large proportion of these attacks the attacker attempts, at some stage, to execute code on the victim machine.

In addition, the type of code executed can also vary. An attacker may inject code into an already running process, or existing programs might be executed, or new programs compiled and then executed. Each action changes the state of the victim machine.

Separate from these malicious concerns is the responsibility of a system

administrator to safeguard the smooth running of the systems under her control. To this end, the administrator would like to ensure that any unknown programs are not run by the legitimate users of the system.

A solution to prevent the use of malicious and/or unknown programs can be found by verifying a set of programs, and then allowing only those programs to be executed[1]. To protect the integrity of the 'safe' programs, each is digitally signed in some manner. When the programs are executed, a check is performed by the operating system to verify the signature. If the signature is good, the process is started, otherwise it is not allowed to execute.

A number of schemes and ideas have been proposed which implement signed binaries [2, 3, 4], plus a few abandoned efforts. In this paper the authors will examine one framework, DigSig [5], and show how a very specific set of security criteria must be satisfied before the signed binary strategy is effective, and can be reasonably trusted. Where appropriate, similarities and differences between other signed binary solutions are highlited.

The remainder of the paper is as follows: Section 2 covers the factual background necessary to understand DigSig's operation, as well as the assumptions which provide the foundations for binary signing. In Section 3 we look at various methods with which signed binaries can be compromised. The lessons learnt from this investigation may be found in Section 4, along with suggestions for improving security. Finally, the paper is concluded in Section 5.

2 BACKGROUND AND ASSUMPTIONS

The need for binary protection emanates from the desire to have multiple layers through which attackers must break, in order to compromise machines. However, there is no common viewpoint as to what attacks this particular layer should protect against, and what features are required. The operating environment in which the attacker is assumed to work varies amongst the different signed binary solutions, as will be seen.

Three proposals are briefly described here, but a more extensive examination of these and other signed binary schemes may be found in [4].

1. DigSig provides signed ELF [6] objects under Linux. It supports signature caching and revocation [2].

[1]Of course, such an approach does not prevent against code insertion attacks against existing processes. Such protection may be found in the grsecurity project [1].

2. WLF [7] supports multiple executable formats, however its support for libraries is limited to those libraries loaded at commencement of execution. Specifically, libraries loaded via the `dlopen(3)` call are *not* checked [3].

3. Motara has proposed a number of strategies which include third-party signing, separating signatures from the binary and noted the usefulness of signed binaries in digital forensics [4]. We assign the moniker IKCVE to his work, but that is our own term and not to be used outside of this paper without his blessing.

DigSig is perhaps the most active signed binary initiative [4]. It consists of a Linux kernel module which is loaded with a public key. Executables and libraries in the ELF format are signed with the `bsign` tool [8] in userspace, by means of a private key. The signatures are RSA-style signatures of the text and data segments of the binaries.

When the binary is loaded, the kernel hashes the text and data segments of the binary, and compares the hash to the decrypted signature. If they match, execution proceeds, otherwise it fails. DigSig also contains a facility for revoking old or insecure binaries.

There are two basic invalid code executions which DigSig (and all signed binary mechanisms) seeks to prevent. The first is that of replaced or trojaned binaries, and the second are unauthorised user programs. The administrator will therefore sign all binaries needed for smooth operation. Under most Linux distributions, binaries are owned by the `root` user. Thus in order to replace a binary, an attacker needs to either elevate his privileges to the `root` account somehow, or execute some other attack[2]. No special permissions are normally required by users to execute their own programs (such programs might be compiled on the same machine, or simply transferred from another location).

The assumptions imposed by the signed binary schemes differs quite considerably. For example, DigSig assumes that the private key used to sign the binaries remains secret, that the public key is not tampered with, and that the `root` account and the kernel itself remain uncompromised. The designers of WLF presuppose that the kernel is always trusted, which implies that the kernel boots securely, that kernel modules are unsupported and that kernel memory cannot be written to. Finally, IKCVE allows for the case whereby an attacker gains administrator access (in Unix terms, 'gets `root`') but assumes that compromising the running kernel requires a reboot.

[2]These 'other' attacks would most likely include physical access to the machine.

There is also an implicit assumption that the system remains secure, given the previous assumptions. However this precludes the use of standard tools to circumvent the systems.

Given the wide range of assumptions, it is difficult to discern exactly what *malicious* attacks integrity verifiers are capable of preventing. Since DigSig in particular assumes that the administrator account is not compromised, then it is impossible for an attacker to replace system binaries which are owned by the administrator, so this attack is not prevented.

Others, such as WLF assume that the attacker may gain administrative privileges but cannot tamper with the kernel. Such assumptions are more realistic than an uncompromised `root` account.

If the goal is to only prevent unauthorised user programs, then simpler measures exist. For example, the `/home` directory could be mounted with the `noexec` option, which prevents any binaries in the user's home directory from being executed, without the overhead of extra checks.

3 ATTACK VECTORS

In the attack scenarios, the `root` account is used. The reason for this is that the authors believe the requirement that `root` remains 'safe' is unreasonable. Once an attacker has access to the machine via a normal user account, elevating privileges to `root` is often a matter of time[3]. The attacks can be divided into four categories:

1. Attacking the kernel
2. Attacking the integrity verifier
3. Attacking the system
4. Combined attacks

In this examination, the authors installed DigSig on a Gentoo Linux machine running kernel 2.6.11 with DigSig version 1.4.1 and bsign version 0.4.5. The installation of DigSig and bsign occurred as per their respective documentation. Public/private keypairs were created with gpg, and all system binaries were signed.

We then proceeded with the above-mentioned attacks.

[3]The length of time will vary, across different environments.

3.1 Attacking the kernel

The simplest attack, given `root`, is to simply unload the kernel module:

```
# rmmod digsig_verif
```

This attack is possible since DigSig makes no attempt to prevent unloading.

A possible solution might be to simply not sign the `rmmod` program, however this falls short of prevention. A 22-byte shell code was produced that executes the system call which unloads the DigSig module. Such code could be injected into whatever vulnerability exists on the target machine. Clearly, the DigSig module needs to protect itself against unloading.

If kernel memory is writable (via a `/dev/kmem` type mechanism) then a number of attacks become possible. The internal DigSig calls could be modified to simply accept all binaries as successfully verified. Another approach might be to overwrite the public key held in kernel memory. In doing so, the attacker would sign her own binaries with her private key, and substitute the legitimate public key with that of the attacker. If the administrator tried to replace the trojaned programs with the correct programs, they would not execute since the attacker had not signed them.

A proof-of-concept was constructed whereby the small piece of kernel memory holding the public key structure was overwritten with random data. This caused the DigSig module to fail all verifications since its public key was corrupted, and the system was immediately unusable because no new processes could be executed. The next step in developing a more useful attack tool would entail supporting replacement of the public key, but the current denial-of-service is highly effective in using the system to thwart itself.

It might be argued that `/dev/kmem` attack tools would never be signed by the administrator, and thus could never be run in the first place. However, it will be shown later how arbitrary programs can be run and therefore the attacker could compile and execute any program, including her attack tool.

Additionally, it should be pointed out that writing to kernel memory can also be accomplished by loading a kernel module, and so malicious modules must be protected against.

3.2 Attacking the integrity verifier

If it is taken for granted that kernel memory cannot be touched, either because `/dev/kmem` writing or module loading is banned, weaknesses still exist in DigSig which can be exploited. Specifically, DigSig caches signatures so as

Step	Action
1	Edit smb.conf, add new share point `//DISJOINT/vuln` to point to `/digsig`, restart Samba.
2	`# mount //DISJOINT/vuln /mnt/programs`
3	`# cp /bin/ps /mnt/programs`
4	`# /mnt/programs/ps`
5	`# cp /tmp/trojan /digsig/ps`
6	`# /mnt/programs/ps`

Table 1: Circumventing DigSig with Samba

to boost performance. These cached signatures are invalidated upon write operations to the file for which a signature is cached.

The cache invalidation scheme does not function correctly for networked file systems, as noted in the DigSig documentation. In particular, the Network File System (NFS) is considered insecure and therefore no signatures are cached for binaries which reside on NFS mounts. The rationale is clear; cached signatures are invalidated only on write operations to their respective files. However any particular file which sits on an NFS mount is also accessible from its host file system. Writes from the host file system cannot be detected by the machine accessing the file via NFS, so the file can be changed without the cached signature being removed.

DigSig explicitly does not cache signatures for files on NFS mounts, but a number of other network file systems exist which DigSig fails to recognise as insecure. The authors chose to use Samba [9], which provides file and print sharing over TCP/IP networks. It is available on most Linux distributions and many administrators make use of it, so the chances that the Samba binaries are present and signed on the target remain high. A number of other network file systems which DigSig does not check for also exist[4]. The steps which can be followed to trick DigSig into running an arbitrary program are given in Table 1, which describes how a machine with the name DISJOINT was compromised.

When the legitimate binary, `ps`, is executed in step 4, its signature is cached. Step 5 involves replacing the legitimate binary with an arbitrary trojan of the same name. In the final step, execution of the trojan via the Samba mount is allowed because of the cached signature. Note that no limitation is placed on the location of the file system mounted by Samba. It could conceivably sit on another machine altogether. Alternatively, the

[4]The interested reader is directed to the Linux kernel source for more file systems which function over the network [10].

attacker could work with an already mounted Samba share.

A second attack against DigSig's caching mechanism may be possible using removable media, however experiments in this direction were not successful. The authors theorise the attack as follows: copy a legitimate binary onto some form of removable media which supports manual ejecting, such as a floppy disk. Execute the binary from that location (and therefore generate a cached signature). Manually eject the media, and, on another machine, overwrite the binary with a trojan. Reinsert the media into the target and execute the trojan.

The authors' attempts to perform this attack were hindered by the changing of FAT32 data when overwriting the binary on the secondary machine. However, with careful manipulation of the FAT it is believed that an avenue of attack exists.

3.3 Attacking the system

The simplest attack in this category involves replacing the public key and rebooting the machine. Such an action is easy to perform on a machine without adequate protection against unauthorised rebooting. The effect of the attack is simply a denial-of-service, since no binaries will be allowed to execute upon the reboot.

If the file system is protecting access to the public key, then it might also be possible to obtain raw access to the disk, and change the key in that fashion.

A more complex attack might involve creating a separate file system in free space on the disk, and duplicating the system minus DigSig there. When the relevant configuration files are changed, a reboot would see the machine running exactly as before, but without any DigSig protection.

3.4 Combined attacks

This last classification encompasses any union of the three previous classes. For example, an attacker might use the Samba attack to execute an attack tool which overwrites the public key in kernel memory.

4 LESSONS LEARNT

It is clear that DigSig by itself is not adequate protection against illegal binaries[5]. The fundamental problem is that it cannot verify the *effect* of

[5]While it may be easy to simply recommend a total security solution such as SELinux [11], that is often overkill. Thus, the authors attempt to secure DigSig with-

Attack target	Recommended protection
Kernel	Use the `seclvl` module.
Integrity verifer	Patch DigSig to prevent unloading, and whitelist filesystems
System	Utilise a secure booting platform, such as AEGIS

Table 2: Recommendations

programs, which is a stronger indication of nefarious activity. This being said, when used in conjunction with software that provides a much more controlled `root` environment, DigSig becomes harder to circumvent. One possible method for achieving this is to use the `seclvl` module under Linux. This kernel module enforces a read-only `/dev/kmem`, prevents raw I/O and disables further module loading or unloading [12].

The vulnerabilities pointed out in DigSig, namely no unloading protection and the problem with signature caches and network file systems, must also be repaired. Protecting against unauthorised module unloading can take the form of disabling the unload facility entirely by disallowing forced module unloading and not defining a `module_exit()` function, or requiring a password as used in `seclvl` [12].

The caching problem is more difficult to solve. By introducing caching, security is weakened. Therefore if security is paramount, the authors recommend disabling signature caching. In the event that an administrator is willing to live with a slight reduction in safety, DigSig should be re-written to only cache on file systems known to be local, rather than blacklisting each network file system as it comes along. This approach is more conservative, but prevents Samba-style attacks.

Lastly, attacking the system cannot be prevented by DigSig, since it assumes a sane state when its module is loaded. The AEGIS secure booting platform and its variant [13, 14] can provide guarantees as to the correctness of the system across restarts.

These recommendations are concisely displayed in Table 2.

5 CONCLUSION

The analysis of DigSig provided in this paper reveals that numerous attacks exist, which could lead to the total compromise of the target. Attacks were divided into four groups, depending on the component targeted. They were the kernel, the integrity verifier, the system, and combinations of the previous

out resorting to such overbearing recommendations.

three attacks.

In addition to the noted theoretical attacks, a proof-of-concept was developed to alter the public key structure held in kernel space, which produced a denial-of-service since no further binaries could be verified. It was also shown how DigSig fails to identify all network file systems, leading to another attack vector.

Concern over the justification for the caching mechanism was noted, and recommendations made.

Finally, current work includes integrating these recommendations into the DigSig code base, which will be submitted to the project authors shortly.

References

[1] grsecurity. http://www.grsecurity.net/.

[2] A Apvrille, D Gordon, S Hallyn, M Pourzandi, and V Roy. DigSig: Runtime Authentication of Binaries at Kernel Level. In *Proceedings of the 18th Large Installation System Administration Conference (LISA'04)*, November 2004.

[3] L Catuogno and I Visconti. An Architecture for Kernel-Level Verification of Executables at Run Time. *The Computer Journal*, 47(5), September 2004.

[4] Y Motara and B Irwin. In-Kernel Cryptographic Executable Verification. In *First IFIP WG 11.9 International Conference on Digital Forensics*, February 2005.

[5] DSI homepage. http://disec.sourceforge.net/.

[6] TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification, May 1995. Version 1.2.

[7] Run-Time Integrity Check of Executables. http://libeccio.dia.unisa.it/wlf/.

[8] Debian bsign. http://packages.debian.org/unstable/admin/bsign.html.

[9] Samba - Opening Windows to a Wider World. http://www.samba.org.

[10] The Linux Kernel Archives. http://www.kernel.org/.

[11] Security-Enhanced Linux. http://www.nsa.gov/selinux/.

[12] M A Halcrow. Using the BSD Secure Levels LSM. *Sys Admin*, September 2004.

[13] W A Arbaugh, D J Farber, and J M Smith. A secure and reliable bootstrap architecture. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, May 1997.

[14] N Itoi, W A Arbaugh, S J Pollack, and D M Reeves. Personal Secure Booting. In *ACISP '01: Proceedings of the 6th Australasian Conference on Information Security and Privacy*, July 2001.