Stop Malicious Code Execution at Kernel-Level

A. Apvrille, M. Pourzandi, D. Gordon, V. Roy
Open Systems Lab, Ericsson Research Canada,
8400 Décarie Blvd, Town of Mount-Royal, (QC) Canada H4P 2N2.
{Makan.Pourzandi, Axelle.Apvrille, Vincent.Roy}@Ericsson.ca
{davidgordonca}@yahoo.ca

In this article, we present a Linux kernel module capable of verifying digital signatures of ELF binaries before running them. This kernel module is available under the GPL license at http://sourceforge.net/projects/disec/, and has been successfully tested for kernel 2.5.66 and above.

1 Why Check the Signature of Your Binaries Before Running Them ?

The problem with blindly running executables is that you are never sure they actually do what you think they are supposed to do (and nothing more...): if viruses spread so much on Microsoft Windows systems, it is mainly because users are frantic to execute whatever they receive, especially if the title is appealing... The LoveLetter virus, with over 2.5 million machines infected, is a famous illustration of this. Yet, Linux is unfortunately not immune to malicious code either [1]. By executing unknown and untrusted code, users are exposed to a wide range of Unix worms, viruses, trojans, backdoors etc. To prevent this, a possible solution is to digitally sign binaries you trust, and have the system check their digital signature before running them: if the signature cannot be verified, the binary is declared corrupt and operating system will not let it run.

2 Related Work

There has already been several initiatives in this domain, such as Tripwire, BSign, Cryptomark, and IBM's Signed Executables [2, 3, 4, 5] but we believe the DigSig project is the first to be both easily accessible to all (available on Sourceforge, under the GPL license) and to operate at kernel level (see Tab. 1).

3 The DigSig Solution

In order to avoid re-inventing the wheel, we based our solution on the existing open source project BSign: a Debian userspace binary signing package. BSign signs the

	Real-time signature verification	File type	Level	Availability
Tripwire	No	All	User	Commercial & GPL
Cryptomark	Yes	Binaries	Kernel	CryptoMark's web: "Not ready for public release"
Signed Executables	Yes	Binaries & scripts	Kernel	Not GPL
DigSig	Yes	Binaries	Kernel	GPL

Table 1: Comparison between file signing tools.

binaries and embeds the signature in the binary itself. Then, at kernel level, DigSig verifies these signatures at execution time and denies execution if signature is invalid.

Typically, in our approach, binaries are not signed by vendors, but we rather hand over control of the system to the local administrator. He/she is responsible to sign all binaries he/she trusts with his/her private key. Then, those binaries are verified with the corresponding public key. This means you can still use your favorite (signed) binaries: no change in habits. Basically, DigSig only guarantees two things: (1) if you signed a binary, nobody else than you can modify that binary without being detected, and (2) nobody can run a binary which is not signed or badly signed. Of course, you should be careful not to sign untrusted code: if malicious code is signed, all security benefits are lost.

4 How do I use DigSig ?

DigSig is fairly simple to use. First, you need to sign all binaries you trust with BSign (version 0.4.5 or more). Then you need to load DigSig with the public key which corresponds to the private key used to sign the binaries.

In the following we show step by step how to sign the executable "ps":

```
$ cp ‘which ps‘ ps-test
$ bsign -s ps-test // Sign the binary
$ bsign -V ps-test // Verify the validity of the signature
```

Then, you need to install the DigSig kernel module. To do so, a recent kernel version is required (2.5.66 or more), compiled with security options enabled (CONFIG_SECURITY=y). To compile DigSig, assuming your kernel source directory is /usr/src/linux-2.5.66, you do:

```
$ cd digsig
$ make -C /usr/src/linux-2.5.66 SUBDIRS=$PWD modules
$ cd digsig/tools && make
```

This builds the DigSig kernel module (digsig_verif.ko), and you are probably already half-way through the command to load it, but wait ! If you are not cautious about the following point, you might secure your machine so hard you'll basically freeze it.

As a matter of fact, once DigSig is loaded, verification of binary signatures is activated. At that time, binaries will be able to run only if their signature is successfully verified. In all other cases (invalid signature, corrupted file, no signature...), execution of the binary will be denied. Consequently, if you forget to sign an essential binary such as /sbin/reboot, or /sbin/rmmod, you'll be most embarrassed to reboot the system if you have to... Therefore, for testing purposes, we recommend you initially run DigSig in debug mode. To do this, compile DigSig with the DSI_DIGSIG_DEBUG and DSI_DEBUG flags set in the Makefile:

```
EXTRA_CFLAGS += -DDSI_DEBUG -DDSI_DIGSIG_DEBUG -I $(obj)
```

In debug mode, DigSig lets unsigned binaries run. This state is ideal to test DigSig, and also list the binaries you need to sign to get a fully operational system.

Once this precaution has been taken, it is now time to load the DigSig module, with your public key as argument. BSign uses GnuPG keys to sign binaries, so retrieve your public key as follows:

```
$ gpg --export >> my_public_key.pub
```

Then, log as root, and use the digsig.init script to load the module.

```
# ./digsig.init start my_public_key.pub
Testing if sysfs is mounted in /sys.
sysfs found
Loading Digsig module.
Loading public key.
Done.
```

This is it: signature verification are activated. You can check the signed ps executable (ps-test) works:

```
$./ps-test
$ su
Password:
# tail -f /var/log/messages
colby kernel: DSI-LSM MODULE - binary is ./ps-test
colby kernel: DSI-LSM MODULE - dsi_bprm_compute_creds: Found signature section
colby kernel: DSI-LSM MODULE - dsi_bprm_compute_creds: Signature verification
                                                                    successful
```

But, corrupted executables won't run:

```
$ ./ps-corrupt
bash: ./ps-corrupt: Operation not permitted
colby kernel: DSI-LSM MODULE - binary is ./ps-corrupt
colby kernel: DSI-LSM MODULE Error - dsi_bprm_compute_creds: Signatures do not
                                                        match for ./ps-corrupt
```

If the permissive debug mode is set, signature verification is skipped for unsigned binaries. Otherwise, the control is strictly enforced in the normal behavior:

```
$ ./ps
bash: ./ps: cannot execute binary file
# su
Password:
# tail -f /var/log/messages
colby kernel: DSI-LSM MODULE - binary is ./ps
colby kernel: DSI-LSM MODULE - dsi_bprm_compute_creds: Signatures do not match
```

5 DigSig, behind the scene

The core of DigSig lies in the LSM hooks placed in the kernel's routines for executing a binary. The starting point of any binary execution is a system call to sys_exec() which triggers do_execve(). This is the transition between user space and kernel space.

The first LSM hook to be called is bprm_alloc_security, where a security structure is optionally attached to the linux_bprm structure which represents the task. DigSig does not use this hook as it doesn't need any specific security structure. Then, the kernel tries to find a binary handler (search_binary_handler) to load the file. This is when the LSM hook bprm_check_security is called, and precisely when DigSig performs signature verification of the binary. If successful, load_elf_binary() gets called, which eventually calls do_mmap(), then the LSM hook file_mmap(), and finally bprm_free_security().

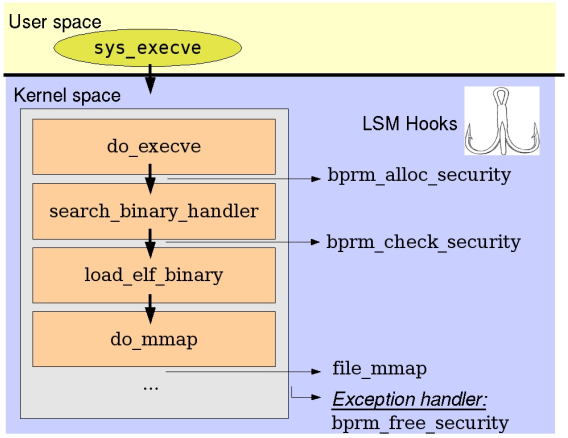

Figure 1: *Control flow in binary execution.*

So, this is how DigSig enforces binary signature verification at kernel level. Now, let's shortly explain the signing mechanism of DigSig's userland counterpart: BSign. When signing an ELF binary, BSign stores the signature in a new section in the binary.

To do so, it modifies the ELF's section header table to account for this new section, with the name 'signature' and a user defined type 0x80736967 (which comes from the ASCII characters 's', 'i' and 'g'). You can check your binary's section header table with the command readelf -S binary. Then, it performs a SHA1 hash on the entire file, after having zeroed the additional signature section. Next, it prefixes this hash with "#1; bsign v%s" where %s is the version number of BSign, and stores the result at the begining of the binary's signature section. Finally, BSign calls GnuPG to sign the signature section (containing the hash), and stores the signature at the current position of the signature section. A short compatibility note: GnuPG adds a 32-byte timestamp and a signature class identifier in the buffer it signs.

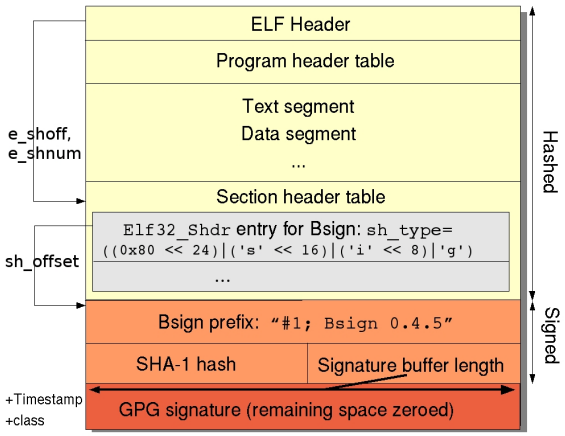

Figure 2: *BSign's signature section as added in an ELF binary.*

On a cryptographic point of view, DigSig needs to verify BSign's signatures, i.e RSA signatures. More precisely, this consists in, on one side, hashing the binary with a one-way function (SHA-1) and padding the result (EMSA PKCS1 v1.5), and, on the other side, "decrypting" the signature with the public key and verifying this corresponds to the padded text.

PKCS#1 padding is pretty simple to implement, so we had no problems coding it. Concerning SHA-1 hashing, we used Linux's kernel CryptoAPI:

- we allocate a crypto_tfm structure (crypto_alloc_tfm), and use it to initialize the hashing process (crypto_digest_init)
- then, we read the binary block by block, and feed it to the hashing routine (crypto_digest_update)
- finally, we retrieve the hash (crypto_digest_final).

The trickiest part is most certainly the RSA verification because the CryptoAPI does not support asymetric algorithms (such as RSA) yet, so we had to implement it...

The theory behind RSA is relatively simple: it consists in a modular exponentation ($m^e mod n$) using very large primes, however, in practice, everybody will agree that implementing an efficient big number library is tough work. So, instead of writing ours, we decided it would be safer to use an existing one and adapt it to kernel restrictions. We decided to port GnuPG's math library (which is actually derived from GMP, GNU's math library) [6]:

- only the RSA signature verification routines have been kept. For instance, functions to generate large primes have been erased.
- allocations on the stack have been limited to the strict minimum.

6 How much does it slow the system down ?

We have performed two different kinds of benchmarks for DigSig: a benchmark of the real impact of DigSig for users (how much they feel the system is slowed down), and a more precise benchmark evaluating the exact overhead induced by our kernel module.

The first set of benchmarks have been performed by comparing how long it takes to run an executable with or without DigSig. To do so, we used the command 'time' over fast to longer executions. The following benchmark has been run 20 times:

```
% time /bin/ls -Al   # times /bin/ls
% time ./digsig.init compile      # times compilation with gcc
% time tar jxvfp linux-2.6.0-test8.tar.bz2 # times tar
```

On a Pentium 4, 2.2 Ghz, with 512 MB of RAM, with DigSig using GnuPG's math library, we have obtained results displayed at Table 2. They clearly show that the impact of DigSig is quite important for short executions (such as ls) but soon becomes completely negligeable for longer executions such as compiling a project with gcc, or untarring sources with tar.

	ls	gcc	tar
Without DigSig	0.006	15.727	36.7345
With DigSig	0.008	16.550	36.7333

Table 2: Average execution time in seconds with or without DigSig.

Second, we have measured the exact overhead introduced by our kernel module. To do so, we have basically compared jiffies at the beginning and at the end of bprm_check_security. In brief, jiffies represent the number of clock ticks since the system has booted, so they are a precise way to measure time in the Linux kernel. In our case, jiffies are in milliseconds. We have run each binary 30 times (see Tab. 3) for DigSig compiled with GnuPG.

The results show that, naturally, the digital signature verification overhead increases with executable's size (which is not a surprise because it takes longer to hash all data).

Binaries	Size	GnuPG
ls	68230	1.61765
ps	70337	1.67742
busybox	248801	4.66666
cvs	672532	11.53571
vim	1894305	31.25000
emacs	4093614	67.41176

Table 3: Average DigSig overhead, in milliseconds, for various binaries.

Finally, to assist us in optimizing our code, we have run Oprofile [7], a system profiler for Linux, over DigSig (see Figure 3). Results clearly indicate that the modular exponentiation routines are the most expensive, so this is where we should concentrate our optimization efforts for future releases. More particularly, we plan to port ASM code of math libraries to the kernel, instead of using pure C code.

```
CPU: CPU with timer interrupt, speed 2398.91 MHz (estimated)
Profiling through timer interrupt
vma      samples  %          image name       app name     symbol name
00001150 401      42.5239    digsig_verif.ko  digsig_verif mpihelp_submul_1
00001090 198      20.9968    digsig_verif.ko  digsig_verif mpihelp_addmul_1
00000e90 109      11.5589    digsig_verif.ko  digsig_verif dsi_sha1_update
00002fa0 77        8.1654    digsig_verif.ko  digsig_verif mpihelp_divrem
00000fd0 32        3.3934    digsig_verif.ko  digsig_verif mpihelp_mul_1
00001300 27        2.8632    digsig_verif.ko  digsig_verif mpihelp_add_n
00001290 15        1.5907    digsig_verif.ko  digsig_verif mpihelp_sub_n
```

Fig 3. OProfile report for DigSig.

7 Conclusion and Future work

We have showed how DigSig can help you in mitigating the risk of running malicious code.

Our future work shall focus on two main areas: performance and features.

Obviously, as signature verification overhead impacts all binaries, it is important to optimize it. There are several paths we might follow such as caching signature verification [5], sporadically verifying signatures or optimizing math libraries.

On a feature point of view, we recently implemented digital signature verification of shared libraries: if malicious code is inserted into a library, all executables (even signed ones) which link to this library are compromised, which is a severe limitation. This implementation is currently in the testing phase and shall be released soon.

References

[1] *Wraight C., Securing your linux environment, Linux World, Vol 1, Issue 2, pages 48-51, November/December 2003.*.

[2] *Tripwire, http://www.tripwire.com*.

[3] *Bsign, http://packages.qa.debian.org/b/bsign.html*.

[4] *Cryptomark, http://www.immunix.org/cryptomark.html*.

[5] *Van Doorn, L., Ballintijn, G., Arbaugh, W.A., Signed Executables for Linux, January 2003.*.

[6] *GnuPG, http://www.gnupg.org*.

[7] *OProfile, http://oprofile.sourceforge.net*.