The DigSig project

The DigSig team

July 15, 2005

Abstract

This working documentation presents the DigSig project, 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 kernels 2.6.8 and above.

1 Introduction

1.1 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.

1.2 Related Work

There has already been several initiatives in this domain (see Table 1), 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.

The advantages we see in the DigSig solution are:

- there are no signature database to maintain. When you want to add a new binary to your system, you only need to sign it. There is no additional command to synchronize a database or a status.

	Real-time signature verification	File type	Level	Availability
Tripwire	No	All	User	Commercial & GPL
Cryptomark	Yes	Binaries	Kernel	Abandonned ?
Signed Executables	Yes	Binaries & scripts	Kernel	Not GPL
Umbrella's [?] DSB (Digitally Signed Binaries)	Yes	Binaries	Kernel	Uses DigSig - GPL
DigSig	Yes	Binaries and libraries	Kernel	GPL

Table 1: Comparison between file signing tools.

- signature verification is automatically enforced. Users do not need to type a special command to verify the binary's signature.
- the kernel does not need to be patched. DigSig is implemented as a kernel module.
- the impact on your system's performance are very light.
- and, of course, it is available for free under the GPL license.

1.3 The DigSig Solution - in brief

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 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.

2 Quick start - How do I use DigSig ?

DigSig is fairly simple to use. We have listed the different steps you should go through. Note all these steps only need to be done once, except loading the DigSig kernel module (which should be done after each system reboot) and signing the binaries (which should be done each time you add/modify a trusted binary).

- Check the requirements (see 2.1)
- Generate a key pair with GnuPG [6] (see 2.2)
- Sign all binaries and libraries you trust (see 2.3)
- Compile the DigSig kernel module (see 2.4)
- Load DigSig (see 2.5)
- Check it works (see 2.6)

2.1 Requirements

- BSign, version 0.4.5 or more [3]
- GnuPG, version 1.2.2 or more [6]
- a 2.6.8 kernel (or more), with CONFIG_SECURITY and CONFIG_SHA1 enabled
- gcc, make etc.

NB. You do **NOT** need DSI. DigSig is an independant project. It uses DSI's CVS for historical reasons.

2.2 Generate a key pair

If you haven't got an RSA key pair yet, generate one with GnuPG:

```
$ gpg --gen-key
```

You may use RSA key pairs up to 2048 bits (included). Keep your private key somewhere same. Then extract your public key:

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

2.3 Sign trusted binaries/libraries

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
```

The following command signs an entire Linux distribution, except some system directories:

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

2.4 Compile the DigSig kernel module

Then, you need to install the DigSig kernel module. To do so, a recent kernel version is required (2.6.8 or more)[1], compiled with security options enabled (CONFIG_SECURITY=y) and SHA-1 (CONFIG_SHA1=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
```

Actually, this is the hard way to do it. The easy way is to use our digsig.init script:

```
$ cd digsig
$ ./digsig.init compile
```

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, make sure to compile DigSig with the DIGSIG_DEBUG flag set in the Makefile (in theory, this is done by default, but still, check it !):

```
EXTRA_CFLAGS += -DDIGSIG_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.

[1] Previous versions of DigSig were known to work with 2.5.66 kernels.

2.5 Load the DigSig kernel module

Once this precaution has been taken, it is now time to load the DigSig module, with your public key as argument. 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.

2.6 Check it works

You can check the signed ps executable (ps-test) works:

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

But, corrupted executables won't run:

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

3 DigSig, behind the scene

3.1 DigSig LSM hooks

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. In former versions of DigSig, this is precisely where DigSig performed its signature verification. However, this has been moved to a later hook (see below) because we have added support for signed libraries and those wouldn't trigger the bprm_check_security hook. If successful, load_elf_binary() gets called. Then, the kernel function do_mmap() is called, which triggers file_mmap(). This is where DigSig actually verifies signature of our binary or library.

Finally, the bprm_free_security() hook is called - which frees any eventual security structure (reminder: we don't have any in DigSig). Note other LSM hooks may be called, such as inode_permission, inode_unlink.

So, this is how DigSig enforces binary signature verification at kernel level. Note signature verification is not triggered only after an execv* but each time the ELF file is mmap'ed (hook do_mmap).

digsig_inode_permission	check it is okay to write in a given inode. If the executable/library is running, forbid write. Remove from signature cache.
digsig_inode_unlink	remove signature from cache.
digsig_file_free_security	called when file is closed. Release write lock.
digsig_file_mmap	Forbid write access to file. Check signature is in the cache. If not, verify signature.

Table 2: LSM Hooks used in DigSig.

3.2 Digital Signature of Shared Libraries

By using the `file_mmap` hook, DigSig can verify shared libraries. Each time a program asks for a library, the kernel maps into memory some part of the library's file. It does this by calling do_mmap. The LSM hook `file_mmap` allow DigSig to intercept the shared library before it is executed and to verify its signature. DigSig can then allow or deny the execution of the shared library. Of course, if DigSig denies execution, the program asking for the library will crash with a segmentation fault error.

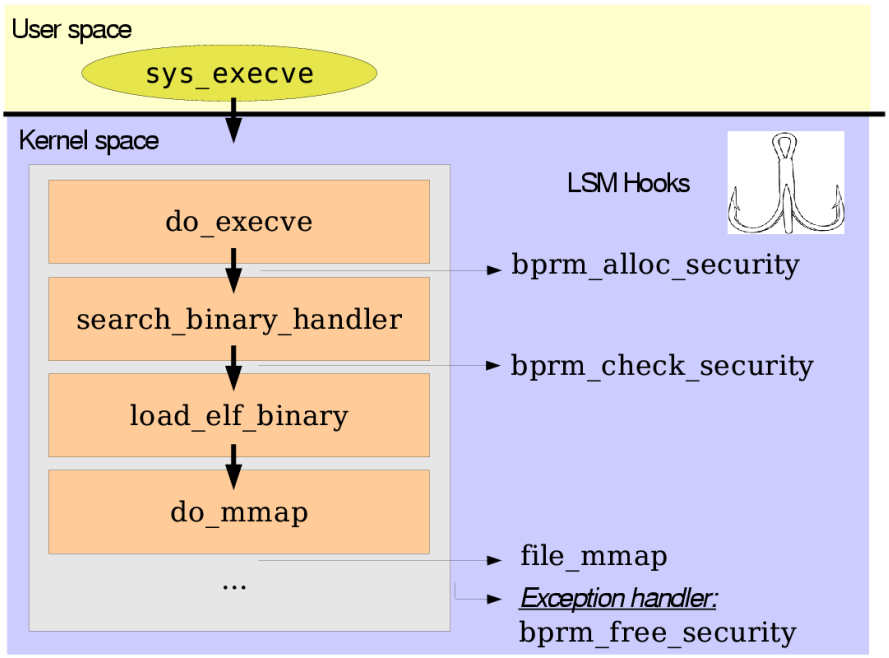

Figure 1: *Control flow in binary execution.*

3.3 Signing an ELF

Now, let's shortly explain the signing mechanism of DigSig's userland counterpart: BSign. When signing an ELF binary (or library), BSign stores the signature in a new section of the binary. To do so, it adds a new entry in the 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.

```
$ readelf -S ./signed-binary
There are 34 section headers, starting at offset 0x1e62:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048114 000114 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048128 000128 000020 00   A  0   0  4
  [ 3] .hash             HASH            08048148 000148 00002c 04   A  4   0  4
  [ 4] .dynsym           DYNSYM          08048174 000174 000060 10   A  5   1  4
...
  [28] .debug_line       PROGBITS        00000000 0013c3 0002a1 00      0   0  1
```

```
  [29] .debug_str        PROGBITS        00000000 001664 0006d4 01  MS  0   0  1
  [30] .shstrtab         STRTAB          00000000 001d38 000132 00      0   0  1
  [31] .symtab           SYMTAB          00000000 0023b2 0006b0 10     32  52  4
  [32] .strtab           STRTAB          00000000 002a62 00045c 00      0   0  1
  [33] signature         LOUSER+736967   00000000 002ebe 000200 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Then, it goes through the following steps:

- zeroize the signature section
- perform a SHA-1 hash of the entire file[2]
- prefix this hash with "#1; bsign v%s" where %s is the version number of BSign,
- store the result at the begining of the binary's signature section.
- call GnuPG to sign the signature section. Currently, GnuPG actually builds an OpenPGP Signature Packet v3 for binary documents. Note the signature is actually performed over #1; bsign v0.4.5, the file's hash, a 4 octet timestamp and a signature class identifier (1 byte)[3] The last two elements are added by GnuPG and comply with the OpenPGP message format.
- store the signature at the current position of the signature section.

3.4 Crypto issues

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)

[2] To be verified: the entire file is hashed except the signature section itself.

[3] Actually, this is bad design. The signature should be performed over the *entire* OpenPGP Signature Packet, including the zeroized part for the signature. Part of this bug has been solved in OpenPGP Signature Packets v4. The other part should be fixed in a newer version of Bsign. There are no harmful exploits known so far, but nonetheless, this is bad.

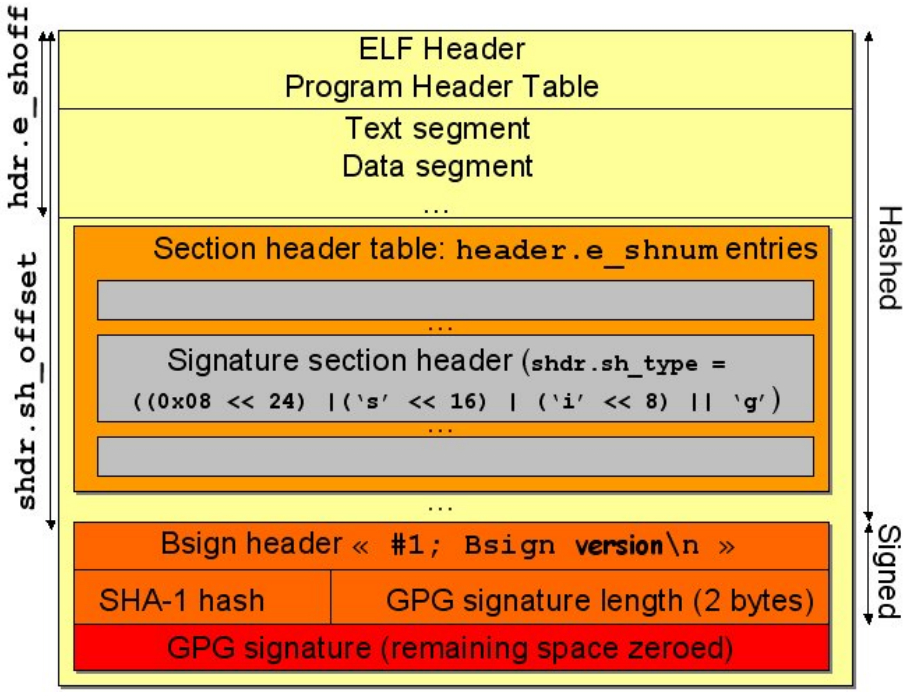

Figure 2: *A Bsign signature section in an ELF binary.*

- 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][4]:

- 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.

[4]Earlier versions of DigSig could alternatively be hooked onto LibTomCrypt [7]. Currently, this is no longer maintained, but we have kept the architecture in case we change our mind and want to re-use LTM.

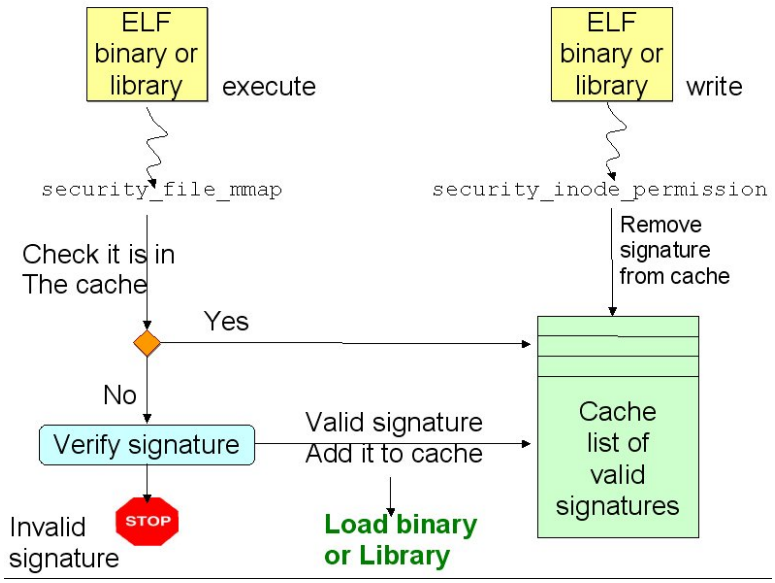

Figure 3: *DigSig's caching mechanism.*

3.5 Caching

Digsig impacts performance only at the beginning of file execution. For long-lived applications which are executed once, such as mozilla, the amortized cost is likely acceptable. However, the cost of repeatedly checking signatures on the same executables (such as `ls`) and libraries (such as `libc`) can become significant depending upon the workload.

To combat this, digsig keeps a cache of validated signature checks When a file's signature has been validated, its inode is added into a hash table. The next time the file is loaded, its presence in this hash table will serve as signature validation without requiring recomputation of the signature.

Caching signature validations can be risky. We must ensure that an attacker cannot use this feature to cause an altered version of a file to be loaded without the (now invalid) signature being checked. In the simplest, case, a new file is copied in place of the validated file. Since Digsig caches decisions based on the inode, and the new file will have a different inode than the old file, the signature will be computed and checked for the new file. If, instead, a process attempts to write to an existing file whose signature validation has been cached, then the signature validation will be cleared. The next time a process executes this file, the signature will be recomputed. Finally, if a process is still executing a file while another process attempts to write to it, the Linux kernel will deny the request for write access.

There is still a risk, however, of the file being overwritten at a lower layer

than the VFS. In particular, this could happen with files mounted over NFS: an NFS mounted file being executed on one client could, for instance, be modified on the server or on any other client. To reduce this threat, DigSig does not cache signature verifications for NFS mounted files.

NB. The signature cache size may be configured at load time using the digsig_max_cached_sigs option:

```
insmod -f digsig_verif.ko digsig_max_cached_sigs=1024
```

3.6 Signature revocation

DigSig also implements a **signature revocation list**, initialized at startup and checked before each signature verification.

At first, signature revocation might seem strange: certificate revocation lists (CRLs) are common, but not signature revocation lists. The idea at stake here is to ease system administrator's task. Suppose the administrator has signed several binaries, but later, a vulnerability is found in one of them. Instead of having the administrator re-sign all his binaries with a new key (what a burden !), we merely ask him to add the signature of the vulnerable executable in the signature revocation list. Of course, the day this list becomes too long, it is time for the administrator to change his key, but that will only happen once in a while, whereas vulnerabilities are (unfortunately) found quite often.

The revocation list is communicated to DigSig using the sysfs filesystem, by writing to the `/sys/digsig/digsig_revoke` file. We only read the revocation list at kernel module startup (so that an attacker cannot modify it once DigSig is in action). TO BE VERIFIED.

To extract the signature from a signed binary, use the extract_sig tool:

```
./tools/extract_sig.sh signed-bin sig
```

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 further precautions, we plan to guard integrity of the signature revocation lists, for instance by signing it with GPG.

3.7 Package description

The DigSig package contains the following directories:

- Makefile: the main Makefile to compile the DigSig kernel module
- README: latest information you should read before installing and running.
- TODO: things we ought to do the day we have some time. Contributions are welcome.

- docs: this directory. Contains the LaTeX documentation.
- gnupg: contains the port of GnuPG's crypto library.
- ltm : contains the port of LibTom's crypto library.
- tools: contains user land tools to extract the public key from your key ring, or extract a signature from a signed binary.

The core implementation of DigSig consists a few files, included directly at the root of the project:

- digsig.c: main for the kernel module. Contains the implementation of all required LSM hooks.
- digsig_cache.c: handles the caching mechanism (see section 3.5).
- digsig_revocation.c: handles the revocation list (see section 3.6).
- dsi_dev.c: handles communication with character device. No longer used.
- dsi_extract_mpi.c: extracts the Multi Precision Integer from the binary's signature. Only used with GnuPG's crypto library.
- dsi_ltm_rsa.c: performs RSA computation using LibTom [7].
- dsi_pkcs1.c: implements EMSA_PKCS_1.5
- dsi_sig_verify.c: implements signature verification using GnuPG's crypto lib.
- dsi_sig_verify_ltm.c: same but with LibTom.
- dsi_sysfs.c: handles communication with sysfs.

3.8 Compilation flags

Compilation flags in DigSig are shown at table 3.

3.9 Features

Currently (v1.4.1), DigSig supports:

- Linux kernels 2.6.8 and above, but requirements should soon move to 2.6.12.
- RSA signatures with keys up to 2048 bits (included)
- SHA-1 hashing (no MD5 or else)
- plugging above GnuPG's and LibTom's crypto library[5]

[5] However, support for LibTom's library hasn't been maintained for a while and is currently broken. But we hope to fix it soon ;-)

Name	Description	Default
DIGSIG_DEBUG	If enabled, unsigned binaries are allowed to run	Yes
DIGSIG_LOG	Activate more intensive logging. Log levels may be configured in digsig.c (`DigsigDebugLevel`). Available levels are listed in dsi_debug.h	Yes
DIGSIG_LTM	Enable use of the LibTom library [7] rather than GnuPG's.	No
DIGSIG_REVOCATION	Enable revocation list	Yes

Table 3: DigSig compilation flags

- support for 32-bit and 64-bit binaries
- signature verification for ELF binaries and libraries. We're currently working on supporting scripts, but that's not completely ready yet.
- signature caching mechanism
- signature revocation list

4 DigSig Performance

All performance measures used a RSA-1024 bit key, and SHA-1.

4.1 Overhead at execution

We benchmarked how long it takes to build three kernels on a non-DigSig system and the same three kernel on a DigSig system. Tests were performed using a Linux 2.6.7 kernel on a Pentium 4 2.4GHz 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 4. The first execution time, both with and without DigSig, appears to reflect extra time needed to load the kernel source data files from disk.

4.2 The efficiency of the caching mechanism

To demonstrate the efficiency of the caching system, we benchmarked the duration of a typical `ls -Al` command. We run the tests a 100 times and display the average execution time, in seconds. The benchmark was run on a Linux 2.6.6 kernel with a Pentium IV 2.2 Ghz, 512 MB of RAM. See Figure 7.

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.

Kernel without DigSig	
real	sys
19m21.890s	1m27.992s
19m9.276s	1m26.584s
19m9.464s	1m26.191s
19m7.717s	1m25.799s
Kernel with DigSig	
real	sys
19m19.957s	1m28.541s
19m7.485s	1m26.832s
19m7.883s	1m26.549s
19m6.494s	1m26.618s

Figure 4: Time required for 2.6.4 kernel "make"

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 5: Time required for "/bin/ls -Al"

Kernel without Digsig			
real	0m59.937s	0m59.175s	0m58.493s
user	0m42.058s	0m42154s	0m42.225s
sys	0m4.005s	0m3.939s	0m3.895s
Digsig without caching			
real	1m0.405s	0m59.361s	0m59.329s
user	0m42.269s	0m42.226s	0m42.190s
sys	0m3.981s	0m3.927s	0m4.005s
Digsig with caching			
real	0m59.660s	0m59.827s	0m59.724s
user	0m42.178s	0m42195s	0m42.120s
sys	0m4.008s	0m3.921s	0m3.940s

Figure 6: Time required for "tar jxvfp linux-2.6.0-test8.tar.bz2"

Actually, caching effects will be most dramatic while doing many quick repeated executions. An example of such a workload is compilation of large packages, which repeat the same sequence of actions on many different files. To measure a best case performance improvement of caching, we timed compilation of Digsig itself in three ways: without DigSig, with DigSig but caching disabled, with DigSig and caching. For each of these three systems, we measured the amount of time required to

- untar the kernel source (see Figure 6),
- perform a directory listing on the top level of the kernel source (see Figure 7),
- compile the actual kernel (with the same configuration each time - see Figure 8).

The benchmark was run on a Pentium IV 2.2 Ghz machine.

The least impact was seen in the `tar` operation. This is because we performed many file creations, which also appear under system time. In contrast, `tar` was a single execution, requiring only one signature validation. Therefore the file operations effectively masked the signature validation check. The impact of the signature check is more dramatic in the other two tests, where Digsig without caching is eight to fourteen times slower than Digsig with caching, or a kernel without Digsig. The latter two performed effectively the same, with Digsig with caching sometimes outperforming a Digsig-free kernel.

Finally, compilation of a full kernel required 592 seconds without Digsig, 588 seconds with caching, and 1029 with digsig but without caching. Caching of signature validations manages very effectively eliminate the performance impact of Digsig under what would ordinarily be its worst workloads.

Kernel without Digsig			
real	0m0.065s	0m0.007s	0m0.006s
user	0m0.001s	0m0.002s	0m0.002s
sys	0m0.005s	0m0.003s	0m0.003s
Digsig without caching			
real	0m0.049s	0m0.053s	0m0.048s
user	0m0.003s	0m0.001s	0m0.003s
sys	0m0.044s	0m0.042s	0m0.043s
Digsig with caching			
real	0m0.025s	0m0.006s	0m0.006s
user	0m0.001s	0m0.000s	0m0.003s
sys	0m0.005s	0m0.003s	0m0.004s

Figure 7: Time required for "/bin/ls -Al"

Kernel without Digsig			
real	0m22.836s	0m15.716s	0m15.700s
user	0m14.291s	0m14.207s	0m14.242s
sys	0m1.449s	0m1.461s	0m1.427s
Digsig without caching			
real	0m42.597s	0m32.629s	0m32.412s
user	0m14.577s	0m14.513s	0m14.501s
sys	0m16.073s	0m16.112s	0m16.158s
Digsig with caching			
real	0m22.996s	0m15.636s	0m15.612s
user	0m14.167s	0m14.179s	0m14.108s
sys	0m1.543s	0m1.408s	0m1.477s

Figure 8: Time required for kernel compilation

4.3 DigSig performance and executable size

The idea in this benchmark is so understand the impact of signed executables' size on DigSig's overhead.

We benchmarked the overhead of DigSig for an executable of 68230 bytes and found a 1.6 ms overhead. Then, we benchmarked the overhead for a big executable of 4093614 bytes, and found a 67ms overhead. On a chart with ms on the x axis and bytes on the y axis, we have two points: SmallExec(1.6, 68230) and BigExec(67,4093614). The line that joins both points is $a.x + b = y$, with $a = 61550$ and $b = -30250$

Then, we approximately verify that a medium sized executable falls on this line: we chose an executable of 672532 bytes and found 11.5ms, which is close to $x = (y - b)/a = (672532 + 30250)/61550 = 11.42$

Of course, we should take more measures, especially on very big executables, but it looks like the overhead induced by DigSig grows linearly with the size of executables, at a very small gradient : 0.0016 microsecond per byte. Again, this is very approximate, and more measures should be done.

Actually, other benchmarks have been done, but with older versions of DigSig (without any caching for instance). Their results corroborate with this idea of DigSig's overhead growing with executable size, but timings cannot be compared with recent ones because machines, kernel versions, DigSig versions have changed too much. Just for your knowledge, we timed 20 executions of ls, gcc compilation and tar:

```
% 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
```

We also counted the number of elapsed jiffies at the begining and at the end of the brpm_check_security hook (which we do not use any longer in recent DigSig versions). We run 30 times several binaries of different sizes (ls, ps, busybox, cvs, vim, emacs...).

4.4 DigSig profiling

Finally, to assist us in optimizing our code, we have run Oprofile [9], a system profiler for Linux, over DigSig (see Table 4). 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 (one day !) to port ASM code of math libraries to the kernel, instead of using pure C code.

5 Tests

DigSig testcases have been added to the Linux Test Project [8]. They are standalone, you do not need to build and compile the whole Linux Test Project.

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

Table 4: OProfile report for DigSig.

Just retrieve the scripts in ltp/testcases/kernel/security/digsig. Then, retrieve DigSig and put it in the ./digsig-latest directory. Run make all and sh test.sh.

So far, we have implemented the following tests:

- make sure it is impossible to write into an executable that is being run.
- make sure it is impossible to execute an executable that is open for write.
- modify byte per byte an executable and check its signature.

6 Contributors

- Axelle Apvrille Axelle_Apvrille@nospam.yahoo.fr
- David Gordon - David.Gordon@ericsson.ca
- Serge Hallyn - serue@us.ibm.com
- Benoit Hamet
- Makan Pourzandi - Makan.Pourzandi@ericsson.ca
- Vincent Roy - Gaspoucho@yahoo.co
- Marco Slaviero
- Chris Wright

Please let us know if somebody's missing...

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] *LibTomCrypt, http://libtomcrypt.org.*

[8] *Linux Test Project, http://ltp.sf.net.*

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

[10] *Umbrella, http://umbrella.sourceforge.net.*

[11] A. Apvrille, M. Pourzandi, D. Gordon, V. Roy, *Stop Malicious Code Execution at Kernel Level*, in Linux World, vol. 2, no. 1, January 2004.

[12] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, V. Roy, *DigSig: Run-time authentication of binaries at kernel level*, in the Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA'04), pp. 59-66, Atlanta, November 14-19, 2004.

[13] A. Apvrille, D. Gordon and the whole DigSig team, *DigSig novelties*, Libre Software Meeting (LSM 2005), Security Topci, Dijon, France, July 4-9, 2005.