Signed Executables for Linux

Leendert van Doorn
leendert@watson.ibm.com
IBM T.J. Watson Research Center
Yorktown, NY

Gerco Ballintijn [*]
gerco@cs.vu.nl
Vrije Universiteit
Amsterdam, The Netherlands

William A. Arbaugh [†]
waa@cs.umd.edu
University of Maryland
College Park, MD

June 4, 2001

CS-TR-4259

Abstract

We describe the design and implementation of signed executables for Linux, which provide the following strong integrity guarantees: the inability to tamper with executables and the inability to add new unauthorized executables. Unlike other implementations, ours covers statically and dynamically linked executables as well as executable scripts. In addition, we reduced the overhead of signature verification to almost zero by caching the successful verification results. The negligible overhead enables signature verification to be used as a basic building block for other applications of which some are described in this paper.

1 Introduction

The ability to authenticate the originator of a network connection and verify the integrity of the transmitted data are considered the basic building blocks of secure distributed systems. When it comes to executable files on such a system, we appear to be satisfied with much weaker integrity guarantees, even though executables should be considered part of a secure foundation as well. Hence, executables must be authenticated and protected against integrity attacks. This is conveniently achieved by digitally signing them.

Signed executables have a number of interesting properties. They prevent intruders from replacing executables with unsigned versions that have a backdoor installed in them, they allow system administrators to determine what executables their users run and, if so desired, restrict the execution of them- essentially providing mandatory access control.

We have implemented digitally signed executables for the Linux operating system. While the concept of a signed executable is a straightforward one, the implementation of it in a real system raises many interesting and unanswered questions. Among these are how to effectively deal with:

- Dynamically linked executables.
- Executable scripts, and
- Performance.

Our system performs the digital signature check by augmenting the activation process. Before a binary file is executed, an embedded signature is verified and when it is valid the binary is executed. This mechanism is easy to implement for statically linked executables. Dynamic executables, unfortunately, are problematic in this model because they load and run additional code after verification. That is, what is verified is a subset of what is running. We solve this problem by the introduction of delegation certificates and make sure

[*]This work was done as part of an internship at IBM's T.J. Watson Research Center.

[†]Portions of this work were funded by an IBM Faculty Fellowship.

the signed portions of the executables verify the dynamically loaded code before executing it. A similar technique is used for executable scripts.

In order to ensure widespread adoption, the signature verification must have an almost negligible performance overhead. In our system we achieve this by introducing a signature cache which contains the results of all previous valid signature verification checks. Once an executable has been verified successfully, this result is cached and future verifications are skipped-provided the file has not been modified. This cache dramatically reduces the performance overhead of our system.

In the next section, we discuss the issues involved in using digital signatures and the guarantees our system provides. Section 3 describes the implementation of our system: static and dynamically linked signed executables, signed executable scripts, and the signature cache. Section 4 discusses some of our experiences with signed executables and includes performance measurements. Section 5 describes a number of useful applications our work and is followed by a future work discussion in Section 6. Related work is described in Section 7.

2 Design Issues

In the design of our system we were primarily focused on providing the following two integrity guarantees:

- Prevent the modification of authorized executables, and
- Prevent the addition of unauthorized executables.

Central to these guarantees is the notion of an external, possibly off-line, authorization process that determines whether an executable is allowed to be executed on a given set of systems. The exact nature of this process is outside the scope of this paper. In this paper, we are primarily concerned with the implementation of the enforcement mechanism.

In our system we opted for digitally signing individual executables rather than using extended filesystem attributes [8] or a signature database. The use of extended attributes has the advantage that it allows *all* files to be signed (*i.e.*, configuration files, databases, C programs), but has the disadvantage that it does not work on filesystems that do not support extended attributes or remote file systems. Remote filesystems are especially problematic since we need an additional mechanism to ensure that the remote server is presenting the true extended attributes. In addition, the implementation overhead for extended attributes is considerable, and we wished to keep our implementation limited to a small set of changes.

The use of a signature database has the advantage that it does not require the modification of the executable itself, but has the disadvantage that it needs to be updated every time a new executable is added or modified. The entire database has to be signed which requires invoking the authorization process on every update. The other disadvantage is that the system administrator has to manage two separate files, the executable content and its signature, instead of one.

In our system, we attached the signature to the executable content. This has the advantages that we only deal with a single container, and the implementation requires only a small number of kernel modifications. To permit flexibility, we added attributes to the signature, and rather than inventing our own formats used standards as much possible. Hence we use ELF [9] for our executable binaries, the PKCS#7 [10] format for storing our signatures, and the X.509 [21] format for storing public key certificates.

Our system ensures its integrity guarantees for executables at load time. It does not provide protection against run-time attacks such as *code injection* attacks (*e.g.*, buffer overflows). These should be handled at a different level.

The current implementation of the system is vulnerable to two attacks. The first attack replaces the public key used to verify signatures with a new public key known by the attacker. The attacker then re-signs some of the binaries he/she is interested in. This attack is possible since we currently use a file, `/etc/certificate`, to store the public key. This attack can be countered by using a secure boot mechanism, as described in Section 5.

The second attack is an overwrite or downgrade attack. In this attack, the hacker has gained access to a machine and copied, for example, the current signed version of the ftp daemon, say *wuftpd*. After a month or so, when the next *wuftpd* bug is discovered, the system administrator installs a new and improved signed version of the ftp daemon. The attacker now replaces the new signed version with the older signed version, which has the known bug and for which the attacker presumably can exploit. This attack is undetected by our current system since it does not keep any state on individual files.

Preventing this kind of attack either requires creating a new key-pair and resigning all executables or keeping a signed revocation list. Since this list will grow arbitrarily, revocation records cannot be deleted, we need to purge the list by creating a new key and resign all executables. None of this is supported by our

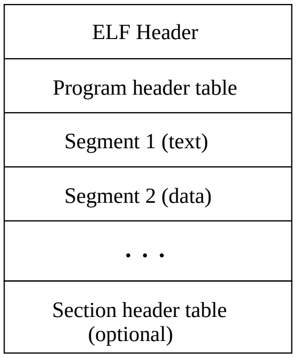

Figure 1: ELF object file format (execution view).

```
OBJECT IDENTIFIER signedData          // content type
INTEGER 1                             // version
OBJECT IDENTIFIER md5 NULL            // digest algorithm
OBJECT IDENTIFIER data                // content type
INTEGER 3                             // version
OCTET STRING                          // key identifier
  68 78 2A 64 3D E2 50 47 B7 E7 90 94 21 F9 F5 FF
  B6 94 D2 BF
OBJECT IDENTIFIER md5 NULL            // digest algorithm
OBJECT IDENTIFIER rsaEncryption NULL // encryption algorithm
OCTET STRING                          // signature data
  0E 64 20 D1 2D 23 0F 26 61 B1 86 39 02 8F 12 27
  0F CA 97 0A B0 A2 C2 5E E5 7D 4F 3D DA 96 39 B9
  62 7F 1D 60 70 64 7F CE B2 D6 62 F4 74 61 CD 68
  F2 A2 FB A3 03 5F 7F 6A 66 88 C6 6B 4B 6F 70 E5
  81 11 C8 35 DE D2 B4 6A EF 9F AE 76 CC DB 74 AD
  D7 85 6A EC 64 A9 2A 5A F9 19 5E E1 EA 67 B1 12
  EC C4 7A 30 B8 4F 99 40 A5 F7 68 62 C5 CB DE BB
  BD 64 3E F6 29 C2 45 09 01 C3 63 51 81 36 B7 DA
```

Figure 2: Simplified dump of `/bin/ls`'s (CMS) signature.

current implementation.

3 Implementation

We implemented the signature checking for executables using the ELF format [9]. The Executable and Linking Format (ELF) standard, describes the structure of object files. It distinguishes three types: executable files, shared object files, and relocatable files. Our work focuses on the first two types since relocatable files are not used in program execution– only during program creation. The ELF format is a convenient implementation vehicle since it allows extensions to the basic format. Adding signatures to other formats, such as COFF or `a.out`, is possible but requires a certain amount of shoehorning. Our current kernel only supports the ELF format so that applications cannot bypass the signature verification mechanism.

All ELF object files follow the same general structure. This structure distinguishes two views of an object file: linking, and execution. Since we deal with signatures on a per executable granularity, we are interested only in the execution view. In the execution view, an ELF object file consist of four parts, as shown in Figure 1: a general ELF header, the program header table, a sequence of segments, and an optional section header table. The general ELF header gives global information on the object file, like its type and intended platform. The program header table lists the segments, and provides their characteristics, such as the type of segment, size, and offset. The segments contain the actual code and data that will be loaded into memory when the executable is started. The file ends with the optional section header table that stores information used during program creation.

For our signature checking implementation, we introduce a new type of ELF segment: the signature segment. This segment contains the digital signature of those parts of the executable that are used during execution. More specifically, the signature covers the ELF parts listed below:

- ELF header.
- All program headers.
- All loadable segments, and
- The interpreter segment.

The rationale behind signing these ELF portions is described in the sections below.

We use the PKCS#7 [10] format for our digital signatures implementation. A human readable example of such a signature is shown in Figure 2 which is taken from the `/bin/ls` program. We chose the PKCS#7 format since it is extensible, allowing us to store extra information in the future. It is also useful that it is a well-known format with publicly available implementations, allowing us to implement our system quickly. The signature is created using the MD5 secure hash function and the RSA encryption scheme [13]. However, given the possible weaknesses of MD5 [4], we expect to use a different secure hash algorithm in the future, e.g. SHA1.

3.1 Statically Linked Executables

The actual signature verification is performed during the `execve()` system call. During this system call the memory image of the current process is discarded, and a new memory image is created using the executable file, given as a parameter. When `execve()` is called by a process, the kernel first determines the actual binary format of the specified executable. Once determined, a loader for the specific binary format is called– in our case that is the ELF binary loader.

The ELF loader continues the loading process by first loading the general ELF header and then the program header table. A scan of the program header table indicates which segments are needed to create the new memory image. These segments are marked as loadable. The Linux kernel does not actually *load* the loadable segments into memory, but instead uses the kernel's memory mapping capabilities. Mapping the loadable segment into the process' memory is more efficient, because it results in loading only those pages that are actually used. After the segments are loaded, the kernel returns control to the process at the start address specified in the general ELF header.

To secure the execution of an executable file, we need to sign those sections of the file that can actually influence its execution. For a statically linked executable that includes the general ELF header, the program header table, and the loadable segments. A potential attacker cannot interfere by modifying, creating, or deleting segments, without also invalidating the executable's signature.

Signature verification is straightforward in the case of statically linked executables. It consists of computing the secure hash function over the general ELF header, program header table, and loadable segments, and using the public key to verify the outcome. The public key used during signature verification was already loaded from the `/etc/certificate` file during kernel initialization.

A slight problem arises when the verification fails. Since the original memory image has been discarded, there is no running program to return an error code. We decided to let the process die on a signal to allow its parent process to notice the error condition.

An important assumption that we made is that an executable cannot be changed during execution. Since the verification occurs prior to execution, there is the possibility that an attacker might change the executable file while it is in use. This is particularly important in the presence of demand loaded executables. Writing to the executable file after it is verified, might allow unverified code to be introduced when pages are reloaded from the executable by the virtual memory system.

Normally, this is not a problem since the kernel does not allow an executable to be changed during execution. This requirement can, however, only be enforced for files on a local filesystem. Files on a remote filesystem, such as NFS, can be changed without the local kernel being aware of it. The only way to avoid this problem is to actually *load* the executable prior to verification. This way a local copy is made of the contents that cannot be changed after verification.

3.2 Dynamically Linked Executables

The execution model for a dynamically linked executable adds two steps to the static one. As in case for a statically linked executable, the kernel begins with loading the general ELF header and program header table. The `execve()` system call identifies a dynamically linked executable when it finds an *interpreter* segment in the program header table. The interpreter segment stores the path to the dynamic linker, usually `/lib/ld.so` on Linux.

After mapping the loadable segments, the kernel also maps the dynamic linker into the process' memory image. The kernel then passes control to the dynamic linker, allowing it to load some or all of the dynamically linked libraries. The dynamic linker is either an executable or shared object file. Note that the dynamic linker is actually part of the process' memory image.

The content of a process' memory image is determined by three sources: the executable, the dynamic linker, and the dynamically linked libraries. To ensure the proper creation of the memory image, we must sign the loadable segments from these sources. But, we also must sign the interpreter segment and the linking information used by the dynamic linker. This information is stored in the dynamic segment.

The signature verification occurs immediately after the segments are mapped. For dynamically linked executables, this means that the kernel must verify both the executable, and the dynamic linker must verify the dynamically linked libraries. This results in the verification of dynamically linked libraries in user space.

The addition of the interpreter segment to the lists of segments requiring signing is a simple extension of the static executable verification model. There is, however, no need to add the dynamic segment to the signature check since it is always located inside another loadable segment, and is thus covered by the fact that we sign all loadable segments. If an executable or shared object file would have a separate dynamic segment, that segment would need to be signed and verified as well.

3.3 Script Executables

For script executables, we use the same indirection step as used for dynamically linked libraries. When a script

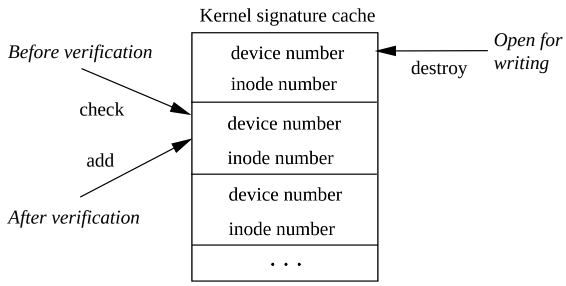

Figure 3: Kernel signature cache operations.

is started, the kernel searches for a binary loader, as usual. However, when the kernel determines that the executable file is actually a script, it loads the script *interpreter* instead, giving the script as parameter. The interpreter will then load and interpret the script. To support scripts we must modify the script interpreter to verify the signature of the script. The script interpreter then performs a role similar to the dynamic linker.

Script executables do pose several problems. The first problem is that we need signature verification functionality in every script interpreter. This can lead to a rapid increase in the number of places where signatures are verified, compared to the base solution where only the kernel and the dynamic linker performed the check. A second problem is that scripting languages frequently allow user input as executable content. This content is not signed, and will thus allow the execution of arbitrary code.

3.4 Signature Cache

Signature verification unfortunately significantly slows the startup time of programs. Since all segments are completely loaded to compute the secure hash function, the performance gain of dynamic loading of executables is lost. To avoid this loss of performance, we need to avoid the signature verification when possible. The signature actually does not need to be computed every time it is executed. If the kernel knows it has verified an executable file before, and can determine that the file has not changed afterwards, it can simply reuse the previous result. This is, in fact, a signature cache– amortizing the verification cost across multiple invocations.

The kernel uses the signature cache to associate a verification result with an executable file. When the kernel is about to verify an executable file, it looks in the cache for a previous result, and after verifying an executable file the results are stored in the cache. The kernel does, however, need to know that an executable file has changed since its verification result needs to be purged from the cache when that happens. This means we have to check every potential file change. Since this implies checking every `write()` system call and a significant performance degradation, we chose to simply look at the `open()` call. When a file is opened for writing, its signature will be purged, as shown in Figure 3.

Remote filesystems also pose a problem for the signature cache. Since the kernel cannot see all the changes made to a file on a remote filesystem, it can potentially cache a verification result while the file has been modified. For this reason, the signature cache can not be used with remote filesystems.

A second problem is that a signature cache can only be effective if its results are actually reused. This is not a problem for the kernel, but since the dynamic linker runs in user space, it can not trust the verification results of the dynamic linkers in other processes. A simple way to avoid this problem is to let the kernel perform the actual signature verification, and present this functionality to a process via a new system call. This new system call, `verify()`, takes an open file descriptor as argument and validates it's signature and returns the result. If the signature is valid, it is also added to the cache. This way the dynamic linker does not have to verify a signature itself– it simply asks the kernel. This is straightforward for both executable and shared object files since they all share the ELF format. Scripting languages fit less well in this model, and their result is currently not cached.

4 Performance

We measured the performance of our system using a set of sample applications that are indicative for a development system. Table 1 summarizes the performance measurements of our system under a typical application load. We measured the execution time in three situations: the signature verification turned off, the signature verification enabled but the signature cache disabled, and the both signature verification and cache enabled. The first measurement gives an indication of the original performance while the later two show the impact of signature verification and the impact of the cache.

The system we used for our measurements consisted of a 900MHz Athlon with 128MB memory running RedHat 6.2 and a 2.2.16 Linux kernel. For our implementation we tried to reuse as many available components as possible. For signature verification in the kernel, we used the RSAREF [19] library and modified the existing ELF loader. These modifications were small and consisted only of 482 lines of code.

Program	*Unverified*	*Verified*	*Verified+Cache*
`ls /`	1261	31799	1243
`sh /dev/null`	2710	58084	2670
`gcc hw.c`	294634	522311	294634
`vi -c :q`	4377	49389	4359

Table 1: Application execution time (in μsec).

Program	*Unverified*	*Verified*	*Verified+Cache*
boot to login prompt	20	35	21

Table 2: Execution time (in sec).

The overhead for signature verification (without the cache) is significant[1], in some cases even 96% of the execution time is due to signature checking. This overhead disappears completely as soon as we introduce the signature cache– since no verification is required with a cache hit. The fact that signed executables with a signature cache are slightly faster than unverified executables is a curious result. While the numbers are within the margin of error they might be caused by some as yet unexplained prefetch effect within the VM subsystem.

Table 2 shows the impact of signature verification during the system bootstrap process. The performance numbers are consistent with the previous results and show that the overhead with a signature cache becomes negligible.

5 Applications

The signed executables described in this paper form the building blocks for a number of interesting applications and extensions. In this section we will look at four of them. They include: secure boot, system administration, capabilities, and application identification,

Secure boot [1] is a procedure whereby the initial program loader (usually the BIOS) verifies the signature of the bootstrap loader before it actually executes it. In turn, the bootstrap loader will verify the signature of the operating system kernel before bootstrapping it. Our work is a logical continuation of this secure bootstrap process. Where the secure boot procedure guarantees that only appropriately signed kernels are started, our work extends this by guaranteeing that only appropriately signed applications are executed.

Integrating signed executables with a secure boot mechanism prevents an attack on the root certificate that is used by the kernel to verify the signatures. In our current system the root certificate is stored in a well-known file, `/etc/certificate`, which could potentially be replaced by an attacker with a different certificate. When using secure boot, the bootstrap loader would pass the certificate to the kernel or verify that the one stored in the filesystem is valid before actually booting the kernel. Another way to prevent this attack is to store the certificate on a token such as a smartcard, but this too requires a secure boot mechanism to ensure that the certificate is actually used.

The second use of signed executables is for system administration. Signed binaries can be used to control what the user is permitted to execute on a system. It is easy to imagine multiple profiles such as, a web server, a firewall, a developer machine, or a secretary machine. Each executable would be classified into the groups it belonged. These groups would then be stored as signature attributes in each executable. At boot up, the administrator would specify the desired domain and the machine would only execute the binaries belonging to that group. For example, a secretary would be able to execute a mail client and an office suite, but not the C compiler, Perl, or any other system utility. A firewall would be even more restrictive. The advantage of using signature attributes is that a single software distribution suffices for many different uses and hence simplifies the maintenance job for the administrator. An alternative application of this mechanism would be to enforce software licenses without having to keep different software distributions.

Closely associated with tagged attributes is the tagging of capabilities. Rather than storing the `setuid` or `setgid` properties of an executable in the filesystem they could be stored in the signature attributes. This would prevent a potential attacker from marking programs setuid since that would require the possession of the off-line secret key used to sign the binaries.

So far we have used digital signatures as a way to authenticate an executable to the kernel. They can

[1]Part of this overhead is due to the use of RSAREF rather than an optimized implementation of RSA.

also be used to identify themselves to other applications. This is especially useful for remote applications where the client wants to establish the identity of the server. A remote server can present its digital signature as proof of this. Of course, this in itself is not sufficient, it has to be signed by the kernel it is running on in order to show that the server is not spoofing the certificate. This works recursively. We also need to know the authenticity of the kernel which requires the bootstrap loader to vouch for it. This requires authenticity guarantees for the bootstrap loader and therefore requires a secure boot mechanism. Eventually this leads to a PKI with a shared root between the client and the server from which trust is acquired.

The application identification mechanism lets a client enforce to which version of the server it wishes to communicate, on which version of the operating system it runs, and which version of the firmware it uses. This is especially useful in the area of secure cryptographic coprocessors [20] that are used to store highly sensitive data. It is crucial for a client to reliably establish trust in a server running on these devices before committing data to it. In fact, knowing what server is currently running on the device is often not sufficient. Depending on the type of application, additional information about what else is running on the device and what ran in the past may all be used by the client to determine its trust in the device.

6 Future Work

In the previous section we described a number of applications that all use the signature mechanism as a basic building block. In this section we look at more immediate future work.

In our current system we do not sign and verify kernel modules. Including these in our system is straightforward and uses the same delegation mechanism we use for shared libraries and scripts. Kernel modules are loaded and relocated by a separate program, `insmod`, before the prepared module image is mapped into the kernel address space. To verify kernel modules we need to enhance `insmod` to verify the module's signature before processing it.

The redirection of standard input into a script raises several problems with only two possible solutions. The first potential solution eliminates redirection from all authorized interpreters. And, the second requires that each IO stream begins with an authorized signature. Unfortunately, both solutions prevent the use of the *command line* or *on the fly* scripts. While this will annoy the system administrators, this functionality violates one of our design principles– preventing the execution of unauthorized code. As a result, we will be implementing the second solution which, fortunately, maps closely with the approach already taken with interpreted scripts.

7 Related Work

Integrity checking for applications has a long history, but it is only recently that processor performance has become sufficient to support the use of public key cryptography within the kernel. In this section, we present background information on the related work to our effort.

7.1 Locus

The first to propose the use of integrity checks based on message authentication codes and digital signatures were Pozzo and Gray in 1986 and 1987 [18, 17, 16]. Their goal was to prevent viruses, and they implemented their system as part of the Locus distributed operating system. The idea was to place a digital signature on each application. The kernel then took the responsibility for validating the signature. If the signature did not match, then the application was not executed. Thus a virus could infect a file, but it could not propagate- essentially cutting off the viruses vector. Unfortunately, the computing power available to Pozzo and Grey at that time was not sufficient to support the use of public key cryptography. As a result, the initial prototype only used the UNIX *crypt* function to create a four byte fingerprint for each "signed" file. The authors recognized that such a mechanism was insufficient, but the inadequacy of processing power at the time prevented a more robust solution. Fortunately, the computing power now exists such that a mechanism as proposed by Pozzo and Grey is now possible. But, Pozzo and Grey only addressed the problem with a monolithic kernel. They never addressed issues such as kernel modules, shared libraries, nor shell redirection. Pozzo and Grey did, however, identify that the use of signatures on executables could implement a strong form of access control.

7.2 Tripwire

Tripwire®provides an excellent means for ensuring the integrity of a filesystem [12, 11]. It has been used for years by many sites to successfully detect the effects of intrusions, i.e. the modification of binaries and/or configuration files. Unfortunately, Tripwire's fundamental flaw is that it relies on the validity of the operating system, the Tripwire binary, and the database of signatures. If any of these items is modified to provide incorrect results to hide malicious changes, then the analysis by Tripwire is suspect and likely to produce false negatives [7, 6]. Another major short coming of Tripwire is that it does not perform its integrity checks in *real time*, i.e. prior to the file or application being used. As a result, the compromise of a site could be missed up to the length of time between Tripwire checks. While Tripwire does perform functions beyond the work described this paper, e.g. integrity protection for non-executable files, the drawbacks to Tripwire remain significant.

7.3 AEGIS Kernel

A precursor to this work was a signed execution prototype produced by one of the authors while employed by the U.S. Department of Defense [2]. Modifications were made to the SunOS 4.X kernel tree so that the integrity of a file was verified prior to execution. The prototype used RSA, MD5, and a simple certificate format appended to the end of *a.out* files. In 1995, the SunOS prototype was ported to FreeBSD where the concept of a verification cache was introduced to amortize the verification cost. Both prototypes identified the numerous issues with interpreted scripts, shared libraries, and loadable kernel modules, but neither prototype implemented a solution.

7.4 IBM 4758

The IBM 4758 secure cryptographic coprocessor [20] uses a signed packaging mechanism to load executables into the device. The package is signed by a developer key which was generated by the developer and signed off-line by the IBM root key. This root key is only used by the card to verify the developer signature, which executable can be loaded is controlled by the developer key. The signing information on the executable is used in a mechanism called *outgoing authentication* which is an initial version of application identification. The later still requires further study, especially in the area of dynamic kernels and multiple device owners.

7.5 Authenticode

Microsoft currently uses several types of integrity checking. First, they have placed digital signatures on device drivers for Windows 2000 and before that Windows 98 [15]. The windows kernel will not load a driver without a valid signature. While this feature could be viewed as a security feature, its implementation was probably more of a configuration management issue, i.e. prevent the loading of unapproved drivers so the machine doesn't crash. Finally, Microsoft implements a code signing mechanism entitled *Authenticode* that places a digital signature on ActiveX controls [14]. The signatures on the controls are used in conjunction with local policy to determine if the control will be executed, or ignored. In essence, providing a limited form of mandatory access control as in the early Java security architecture [3].

7.6 Java Code signing

The signing of Java applications and applets initially only supported an *all or nothing* approach to access control as in Authenticode. Applications on the local filesystem were completely trusted with or without a signature. Remote code was *untrusted* unless it contained a valid signature. The current security architecture for Java, however, provides for a fine grained access control mechanism [5]. In the current security architecture, a local policy file determines what, if anything, resources applications and applets can access. The policy can, for instance, allow only those applications signed by certain public keys to access local files.

8 Conclusions

While integrity has long been a desirable property for network and distributed security efforts, it has, for the most part, been ignored within modern operating systems. The result is that *add-on* mechanisms such as Tripwire were employed by a large set of users requiring strong integrity guarantees. We believe that such integrity mechanisms belong in the kernel and below–for once integrity is lost, it can not be easily regained. One of the may reasons why strong integrity guarantees are not present in modern operating systems is the incorrect perception that such mechanisms incur a large performance penalty. We have shown in this work that such beliefs are misguided, and that through the use of a signature cache strong integrity guarantees can be provided with only a negligible performance penalty.

Acknowledgments

We would like to thank Peter Gutmann for helpful comments on the paper. The authors are working towards a public release of the software described in this paper.

References

[1] W. Arbaugh, D. Farber, and J. Smith. A Secure and Reliable Bootstrap Architecture. In *1997 IEEE Sym-*

posium on Security and Privacy, pages 65–71. IEEE, 1997.

[2] W. A. Arbaugh. Signed execution. Office of INFOSEC Research and Technology Program Status Review, February 1994.

[3] D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *1996 IEEE Symposium on Security and Privacy*, pages 190–200. IEEE, 1996.

[4] H. Dobbertin. Cryptanalysis of MD5 Compress, May 1996. Presented at the rump session of Eurocrypt '96.

[5] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 1999.

[6] Halflife. Bypassing Integrity Checking Systems. In *Phrack*, volume 7. 2600, September 1997.

[7] Hoglund. Windows rootkit. *http://www.rootkit.com*.

[8] IEEE. POSIX 1003.1e Draft Standard, Access Control Lists, October 1997. Withdrawn, *http://www.guug.de/ winni/posix.1e/download.html*.

[9] Intel. Tool interface standard portable formats specification (version 1.1), October 1993. Intel order number 241597.

[10] B. Kaliski. PKCS #7: Cryptographic Message Syntax (Version 1.5). In *Internet Request for Comments (RFC) 2315*. March 1998.

[11] G. Kim and E. Spafford. Experience with Tripwire: Using Integrity Checkers for Intrusion Detection. In *System Administration, Networking, and Security Conference III*. USENIX, 1994.

[12] G. H. Kim and E. H. Spafford. The Design and Implementation of TRIPWIRE: A File System Integrity Checker. Technical Report TR-93-071, Department of Computer Science, Purdue University, November 1993.

[13] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[14] Microsoft. Authenticode Techonology. Microsoft's Developer Network Library, October 1996.

[15] Microsoft. Authenticode Signing of Device Drivers. Presentation at Microsoft Professional Developers Conference, September 1997.

[16] M. M. Pozzo and T. E. Gray. An Approach to Containing Computer Viruses. *Computers and Security*, 6(4):321–331, August 1987.

[17] M. M. Pozzo and T. E. Gray. A Model for the Containment of Computer Viruses. In *1989 IEEE Symposium on Security and Privacy*, pages 312–318. IEEE, 1989.

[18] M. M. Pozzo and T. E. Grey. A Model for the Containment of Computer Viruses. In *1986 Aerospace Computer Security Conference*, pages 11–18, 1986.

[19] RSA Laboratories. RSAREF®: A Cryptographic Toolkit for Privacy-Enhanced Mail, 1994.

[20] S. Smith and S. Weingart. Building a High-Performance, Programmable Secure Coprocessor. In *Special Issue on Computer Network Security*, volume 31, pages 831–860. Elsevier, 1990.

[21] X.509. ITU-T Recommendation X.509 (1997 E): Information Technology - Open Systems Interconnection - The Directory: Authentication Framework, June 1997.