CSETech

May 1999

GuardHouse: locking the stable door ahead of the Trojan Horse

Steven M. Beattie

Andrew P. Black

Crispin Cowan

Calton Pu

Lateef P. Yang

GuardHouse: Locking the Stable door ahead of the Trojan Horse

Steven M. Beattie, Andrew P. Black,
Crispin Cowan, Calton Pu, Lateef P. Yang

Department of Computer Science & Engineering,
Oregon Graduate Institute of Science & Technology

Abstract

Once attackers have penetrated a system, they will usually take advantage of their position by extending their reach to compromise other systems (e.g., by sniffing passwords from the network), and by installing "back doors" that will enable them to regain access even if the original insecurity is repaired. A common approach is to install a modified version of a standard system daemon such as *telnetd*. It is also common for attackers to attempt to cover their tracks by installing doctored versions of standard programs like *ls*, *ps* and *sum*. Programs like this, which conceal something harmful inside a harmless looking exterior, are called Trojan Horses.

GuardHouse detects Trojan Horses as they are about to be activated. It is similar in its goals to Kim and Spafford's Tripwire, but detects intrusions with lower latency. That is, rather than laying dormant until the system administrator next runs a Tripwire integrity check, GuardHouse would detect a Trojan Horse as soon as it is run. Another difference is that GuardHouse can be configured to prevent the Trojan Horse from running at all.

Deploying a Trojan Horse involves installing code crafted by the attacker or imported from another site. In principle, this can be detected by requiring that all code that runs on a system be *certified*. Using Public Key Cryptography, it is possible to *sign* executable binaries to indicate that they are genuine, and to check the signature on a binary before it is executed. The Trojan Horse would then be exposed before the system is put at risk by loading it.

But what is possible in principle does not always work well in practice. Our goals in developing GuardHouse was to deploy code certification as an intrusion detection tool in a real system (LINUX version 2.0.36) so that its practical impact can be

This work was partially supported by DARPA Grant F30602-96-1-0331 and by NSF Grant Number CDA-9703218

evaluated. The paper makes three contributions. First, we show that the implementation of binary code certification in a modular system like LINUX is simple. Second, we discuss configurations that trade different degrees of security for different degrees of inconvenience, but in all cases we are able to find reasonable points of compromise. Finally, we show that the impact of the additional checks on system performance is small enough to encourage the wide adoption of this technology.

Introduction and Motivation

The idea that Public Key Cryptography can be used to digitally sign a document, thus establishing its authenticity, is as old as public key cryptography itself [1]. Toolkits that implement the basic cryptographic algorithms are now widely available, for example, PGP [2] and GPG [3]. GuardHouse, described in this paper, represents the practical application of these ideas in a way that prevents unauthorized programs from running on the protected system. Alternatively, GuardHouse can be configured to allow unauthorized programs to run, but to generate a log record detailing the attempt.

When intruders gain access to a system, they frequently try to down-load and run programs that will either seek to discover further weaknesses in the system, or install a back door, or will hide these very activities from legitimate users and system managers. Using GuardHouse, the running of these programs can be prevented or detected.

GuardHouse is similar in many respects to Tripwire [4, 5]. Tripwire and GuardHouse both collect vital statistics about files, which are later used to detect tampering with those files. Tripwire operates “off-line”, in the sense that comparison between the current version of the file and the previously recorded version is done when explicitly requested by the system administrator, for example, at 8am every morning. GuardHouse is “on-line”, in the sense that the comparison is done every time that the binary is executed. However, Tripwire is more general than GuardHouse, since it can be applied to any file (for example, configuration files), while GuardHouse protects only executable files.

THE BASIC IDEA

The basic idea behind digital signatures is simple. Suppose that Alice wants a digital signature. Alice first obtains a pair of keys k_{pub} and k_{priv}, known as her public and private keys. These keys have the property that if a file (or other sequence of bytes) is first encrypted with k_{priv}, and the result is then encrypted a second time, with k_{pub}, the original file is obtained.

Knowledge of k_{priv} is like possession of a rubber stamp bearing Alice’s signature. Knowledge of k_{pub} is like the ability to compare the imprint of a stamped signature on a document with the real thing. k_{pub}, as its name implied, is intended to be public, for example, Alice might publish it in an Internet directory, so that anyone who is presented with a document d that claims to be signed by Alice can test that fact by obtaining Alice’s public key k_{pub}, and encrypting d with k_{pub}. If the result makes sense, d was genuine. If the result is gibberish, then d was a forgery. (Note that this is just the opposite of what one would do to maintain secrecy; in that case the sender would encrypt with the public key of the recipient, and the appropriate private key would be needed to read the contents.)

In practice, a transmitted document normally contains both the unencrypted and the encrypted version of the original file, so that it is easy to test whether the second encryption actually generates the original text. Because this would make the transmitted document much larger, it is usual to perform the encryption not on the entire file, but instead on a “digest”, a hash of the file contents produced by a digest function.

A digest is a function that takes a (usually large) file and produces a (usually much smaller) result called a hash value. A simple example is a CRC or checksum function, which adds the bytes in the original file together, modulo 2^{32}or 2^{64}. In general, digests are designed to resists inversion—finding another input that produces a known output— and collision—finding two inputs that generate the same hash value. A good digest will depend on every bit of the input, because if it does not, it would be possible to tamper with some of the bits without being detected. MD5 [6] is a digest function that generates a 128-bit hash from a file of arbitrary length. MD5 is widely regarded as being secure, and software implementations are now widely available.

APPLYING THE IDEA

Given this background, it is easy to see how to use digital signatures to check the authenticity of a binary program.

1. On a secure (possibly non-networked) computer *S*, Alice generates the binary in the usual way, using her favourite compiler and linker. On LINUX, the result is a loadable file in Executable Linkable Format (ELF).
2. Alice computes the MD5 digest of the executable portions of the ELF, and then encrypts the digest with k_{priv}. The result is called a *certificate* for that file
3. She then appends the encrypted digest to the original file, creating a signed version of the binary file.
4. The signed file can now be copied from *S* to the system *P* that requires guarding. If S is not on the network, the file can be transmitted by “sneaker net” using a removable medium disk.
5. On *P*, before the binary is executed, the operating system kernel first checks that the MD5 digest computed from the executable portions of the file and the MD5 digest obtained by decrypting the certificate with k_{pub} are identical.
6. If they are identical, then we can assume with high confidence that the file was indeed signed by Alice, or at least by someone who has Alice’s private key. If they are not, then the file is a forgery, or has been corrupted in some way. In either case, an alert should be sounded.

GuardHouse may thus be thought as having two components: a set of utility programs that run on *S*, and a kernel modification that is inserted into *P*. Of course, it is possible for *S* and *P* to be the same machine; this is convenient, but opens the possibility that an attacker may manage to capture the secret key, and thus gain the ability to forge Alice’s signature. We do not recommend running GuardHouse in this configuration.

The Implementation

Our first finding is that implementing GuardHouse on LINUX was relatively simple. The bulk of the work was done in three months by an undergraduate student intern, Lateef Yang. The encryption and verification functions were taken from Gnu Privacy Guard (GPG) [3], a GPL implementation of the OpenPGP standard [7]. We use the El Gamal encryption algorithm; the key length is chosen at key generation time. The MD5 code was extracted from the Redhat Package Manager [8].

The LINUX ELF format already provided meta-data to distinguish executable code from other information (such as symbol table data); loadable data is contained in PT_LOAD segments. Thus it was easy to compute the digest function on these segments only. The ELF data schema has a note segment type (SHT_NOTE); this is used for storing the signed digest, so inserting it was not difficult.

An alternative implementation would be to store the signatures in an auxiliary file, either a system-wide auxiliary table or one additional file per executable. This has the advantage of generality; for example, it could be used for shell scripts and other executables that are not ELF files. It has two disadvantages: it is harder to maintain, because an additional file or table entry must be installed along with every executable, and it is less efficient, because two files must be accessed at *exec* time rather than one. A possible compromise (which has not yet been implemented) would be to use embedded certificates for ELF files, and an auxiliary table for other executables.

The software developer is provided with three commands:

- `gh_insert` <*filename*>
- `gh_check` <*filename*>
- `gh_keygen --gen-key`

`gh_keygen` is derived from the GPG command, except that by default it writes the public and private keys to files in the directory `/etc/.guardhouse`. `gh_insert` computes a digest for the file that is provided as its argument, signs the digest with the key from `/etc/.guardhouse`, and inserts the resulting certificate back into the file.

`gh_check` is a utility function that enables the developer to ascertain whether a file has never been certified, has been certified with a valid certificate, or contains a certificate that does not match its contents.

KERNELGUARD

On the protected system, the function of checking the validity of the certificate is performed by a loadable LINUX kernel module called KernelGuard. KernelGuard can be configured to check every ELF binary that is passed to it before it is run, or to check only some binaries. In addition to the KernelGuard module, some changes to the LINUX kernel itself were necessary to provide an interface into which KernelGuard can be plugged. In addition calls to KernelGuard were added to *exec*. In total, the additions to the LINUX kernel amounted to about 100 lines of highly commented code.

The KernelGuard module itself is 123 kB of binary code. The huge size is due to the inclusion of substantial cryptographic support, including arbitrary-precision arithmetic libraries to support exponentiation for the El Gamal algorithms. KernelGuard needs two inputs to do its work: the public key (corresponding to the private key used by the developer to sign the file originally), and configuration data.

The location of the public key can be specified at the time that the KernelGuard module is loaded. The key is read into kernel memory the first time that KernelGuard runs, and then remains in memory. It is not read again until the operating system is rebooted. This provides some protection against the public key being spoofed. Even if an attacker did succeed in inserting a Trojan Horse binary in to the file system, signing it with a new private key and planing the corresponding new public key into the file system, the operating system would have to be rebooted before the new public key would have any effect. Because there is only a single system-wide public key, attempts to execute any of the legitimate system programs (such as `init`, `login`, etc.) would then fail. For the reboot to succeed, every GuardHouse-protected binary in the file system would have to be replaced by a new version that had been signed by the new, fraudulent, key.

Configuration

KernelGuard can be configured to require valid certificates in all or some binary files. The decision of whether or not to require a certificate is currently based on the user id under which the binary will be run.

The simplest and most secure configuration is to require a certificate in every binary. This might be appropriate for a server machine, such as a web server or a mail server, on which users are not expected to run their own code; any attempt to do so would indicate an attack. This configuration might also be appropriate for a desktop machine in a corporate environment, where users do not write their own code and where only code approved by the system administration staff is permitted. This setting would prevent naive users from, for example, down-loading and running dangerous binary code form the internet.

A common, more permissive, configuration is to set KernelGuard to require signatures on all binaries that will run as root, whether they are *suid root* or simply inherit *root* context from their parent. This would enable students, or software developers, to continue to compile and test programs under their own user names, but would prevent one of them, for example, from writing and running a network sniffer, even if they were able to obtain root privilege. In general, in a system where different user identifiers have different levels of privilege, the more privileged users are those that should be protected by GuardHouse.

The action that the protected system takes when an un-certified (or incorrectly certified) binary is found might also be varied. One possibility is to abort the attempt to *exec* the program. This protects the system, but alerts the attacker. Another possibility is to run the unsafe program anyway, and to generate a silent alarm that is transmitted to system administrators or other intrusion detection systems. GuardHouse can thus be used as a source of intrusion events that then trigger other systems in the network to take protective action.

Note that if the secret key is stored on the same machine that is being protected, it is very likely that attackers who are able to become root on that machine would also be able to steal the secret key, and thus would be able to certify their own binaries. For this reason we consider such a configuration unsafe, and do not recommend it.

Performance

Our measurements of the performance impact of GuardHouse are so far preliminary. All of the numbers reported here are measured on a 120 MHz Pentium processor with no level 2 cache and 32MB of RAM—a very modest configuration. 1024-bit keys were used for the EL Gamal encryption.

The time taken to sign a binary is low, compared to the overall time taken by the compilation process. We have not worked to speed-up the operation of `gh_insert`; it takes (very approximately) one half-second per Megabyte to sign a typical binary, in addition to set up time for `gh_insert`. Signing `emacs` (version 20.3.1, from the Redhat 5.2 distribution) takes 3.9 s with a cold file system cache. It took 7 minutes 5 seconds to generate signed versions of all 914 ELF binaries from the Redhat 5.2 `/usr/bin` directory; sizes of these binaries ranged from 3.4 kB for `dumpreg` to 12 MB for `netscape`.

GIven these modest costs, it seems entirely reasonable to sign all binaries at the time that they are created. But what of the cost of the certificate check that takes place in KernelGuard whenever a certified binary is run?

Decrypting the digest with the public key is fast, because the digest is only 128 bits long. However, re-computing the digest over all of the loadable code segments in the ELF file is potentially time-consuming, because, at the very least, all of the loadable code in the executable file must be read.

Note that normally (without KernelGuard) code can be loaded on demand: it is not necessary to read all of the code before commencing execution. However, KernelGuard must read all of the code before it can compute the MD5 hash. It turns out that because of a feature of the LINUX dynamic code loading process, the normal LINUX kernel also reads the all of the loadable code eagerly. Thus, KernelGuard does not introduce as much of a performance penalty as one might expect. After all, the kernel is already “handling” every byte of the executable.

Measurements show that the cost of KernelGuard is about 300 ms–400ms for a 3MB binary like *emacs*. This is of course independent of source of the data. When the file to be *exec*’d is already in the file cache, the time for emacs to start up and exit immediately is only 80 ms, so this overhead represents a factor of 4 or 5. However, when the file is not in the buffer cache, starting *emacs* takes about 3 s, so the overhead is closer to 10 or 12 per cent. That is, *fork* +*exec* is 10 or 12 per cent slower. Overall, if a system spends 1 per cent of its time executing *fork* and *exec*, the slowdown due to KernelGuard would be of the order of 0.1 per cent.

It is conceivable that systems executing some applications do repeatedly *exec* the same ELF file from the buffer cache, and will thus experience the more significant (factor of 4 or 5) slowdown. If this turns out to be a problem, it may be possible to remedy it by caching the previously computed MD5 digest in the kernel along with the file. Changing the file on disk would still cause the buffer cache to be refreshed and the digest to be re-computed. However, caching the MD5 digest would open up another possible attack: changing the file image in the kernel’s buffer cache. It would also complicate the inter-module dependencies in the kernel. If the slowdown suffered by repeated executions of the same binary turns out to be a problem, this issue will need to be examined more carefully.

With this exception, our preliminary tests showed that GuardHouse has no observable performance impact at execution time. We plan to carry out more careful measurements of the performance impact of GuardHouse and would like to present the results at the RAID workshop and in the final version of this paper.

Conclusion

We believe that digital signatures can be used for intrusion detection by protecting production operating systems from Trojan Horse attacks. The underlying technology is now widely available, and on modern hardware the performance impact is negligible. The reports generated by GuardHouse are valuable as indications of intrusion in their own right, but can also be used as a source of intrusion events for other components in the IDS arsenal

There is always a trade-off between security and convenience. However, we believe that the security offered by GuardHouse is worth the cost in terms of the extra steps that become necessary to install new software. This is particularly true in a server environment, where all software should in any case be installed by trained system management personnel.

The present implementation of GuardHouse is experimental. We plan to combine GuardHouse with *CoDomain*, another research project that we are conducting at the Oregon Graduate Institute to enhance the survivability of servers. It is our intention to release the combined code to the community for evaluation and testing.

References

[1] W. Diffie and M. E. Hellman, “New Directions in Cryptography,” *IEEE Transactions on Information Theory*, vol. IT-22, pp. 644–654, 1976.

[2] P. R. Zimmerman, *The Official PGP User's Guide*. Boston: MIT Press, 1995.

[3] W. Koch, “The GNU Privacy Guard,” . Düsseldorf, Germany: http://www.d.shuttle.de/isil/gnupg/, 1999.

[4] G. H. Kim and E. H. Spafford, “Experiences with Tripwire: Using integrity checkers for intrusion detection,” presented at Systems Administration, Networking and Security Conference III, 1994.

[5] G. H. Kim and E. H. Spafford, “The Design and Implementation of Tripwire: A File System Integrity Checker,” presented at 2nd ACM Conference on Computer and Communications Security, Fairfax, Virginia, 1994.

[6] R. Rivest, “RFC 1321: The MD5 Message-Digest Algorithm.,” RSA Data Security, Inc. April 1992.

[7] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer, “RFC 2440: OpenPGP Message Format,” , Proposed Standard November 1998.

[8] RPM, “The RPM Home Page,” http://www.rpm.org/ April 1998.