# REMOTE INTEGRITY CHECKING

*How to Trust Files Stored on Untrusted Servers*

Yves Deswarte*, Jean-Jacques Quisquater**, Ayda Saïdane*
* *LAAS-CNRS, France*
** *Université Catholique de Louvain, Belgium*

Abstract: This paper analyzes the problem of checking the integrity of files stored on remote servers. Since servers are prone to successful attacks by malicious hackers, the result of simple integrity checks run on the servers cannot be trusted. Conversely, downloading the files from the server to the verifying host is impractical. Two solutions are proposed, based on challenge-response protocols.

Key words: file integrity checking, intrusion detection, challenge-response protocols.

## 1. INTRODUCTION

The integrity of files stored on servers is crucial for many applications running on the Internet. A recent example has shown that Trojan Horses can be largely distributed in security critical software, due to a successful attack on one server [CERT 2002]. This kind of damage could have been hindered if this particular software distribution was secured by digital signatures (as is currently done for some other software distribution on the Internet). In other cases, damage may vary from web page defacing to circulation of false information, maliciously modified execution of remote services or diverse frauds in electronic commerce. The detection of such wrong behavior can be more difficult than for software distribution, since in most cases it is not possible to just check a signature on a content. The current state of the Internet security is such that most servers are vulnerable to dedicated attacks and in most cases the detection of such attacks happens several hours, days

or weeks after the real attacks have occurred (one or two days in the case cited above). Indeed, new vulnerabilities are discovered frequently on most commercial operating systems and applications, and current intrusion detection systems do not detect or misidentify a too large proportion of attacks, in particular when the attacks are new and slow [Green et al. 1999].

It is thus of tremendous importance for system administrators to check frequently that no critical file has been modified on the servers they manage. The classical way to do so is to reboot the server from a secure storage (a CDROM for instance) and then to launch locally (from the secure storage) a program to compute cryptographic checksums (using one-way hash functions) on the critical files and compare the results with reference checksums stored on a secure storage. Tripwire[1] is a well-known example of such programs. It is important to reboot the system from a secure storage since a malicious hacker could have installed and launched a program on the server which, for instance, could modify the disk drive handler to return the original file content to the integrity checking program instead of a modified version actually stored on the disk. Some viruses and Trojan horses are using such stealth techniques. This is also why such a program has to be run locally, after a secure reboot, before any insecure application (which could have been modified by a hacker) is run.

Such a technique is impractical in most Internet server systems:

- This task requires a lot of time: halting operational applications; halting the system in a safe way; rebooting form a CDROM; running the integrity checking program on all critical files; restarting the operational applications. This is hardly compatible with 24/7 operation required from most Internet servers and, anyway, would be too costly if run frequently.
- Competent system administrators are a scarce resource, and thus most servers are managed remotely: it would be impractical for the administrators to go to each server to execute this routine task in a secure way.

Running remotely an integrity check program is inefficient, since it is impossible to be sure if the program that is run is the original one and not a fake, if the reference checksums are the correct ones[2], and if the operating system has not been modified for some stealth operation.

Conversely, it is impractical for the administrator to download all critical files to his local host, to compute locally the checksums and then to compare the results with reference checksums: this would cause too much overhead on the network and on the administrator host, as soon as the numbers of files

---

[1] Tripwire® is a registered trademark of Tripwire, Inc.

[2] The reference checksums can be signed, but then the integrity of the signature verification key must be also checked.

and of servers are high, the mean file length is large, and this task has to be run frequently.

This paper proposes two solutions to this problem. The first one, described in Section 2, is a challenge-response protocol using conventional methods, which lead to some engineering trade-offs. Section 3 presents another protocol, based on a modular exponentiation cryptographic technique, to solve this problem in a more elegant way, but at the price of more complex computations. These solutions are then compared to alternative solutions and related work.

# 2. A CONVENTIONAL CHALLENGE-RESPONSE PROTOCOL

## 2.1 General approach

In this solution, the administrator's host (called hereafter the *verifier*) sends periodically a request to the server for it to compute a checksum on a file (specified as a request parameter) and return the result to the verifier. The verifier then compares the returned result with a locally-stored reference checksum for the same file.

A naïve implementation of this protocol would be inefficient: a malicious attacker could precompute the checksums on all files he intends to modify, store these checksums, and then modify the checksum computation program to retrieve the original checksum rather than compute it. The attacker can then modify any file, while remaining able to return the expected checksums when requested by the verifier. This protocol has to be modified to guarantee the freshness of the checksum computation.

This can be achieved by adding a *challenge* $C$ in the request parameters. With this new protocol, the server has to compute a *response* $R$ depending on the challenge. More precisely, instead of a checksum computed as the result of a one-way hash function on the content of the file, the response must be computed as the hash of the challenge concatenated with the file content:

$$R = \mathcal{H}(C|File) \qquad (1)$$

Of course, the challenge must be difficult to guess for the attacker. In particular it must be changed at each request. But then the verifier cannot simply compare the response with a reference checksum. A solution would be to maintain a copy of all the original files on the verifier and run the same response computation on the verifier as on the server. But this is impractical

if the numbers of files and of servers are high and the mean file length is large.

A better solution would be for the verifier to use two functions $f$ and $\mathcal{H}'$, of which at least one of them is kept secret[3], such that $\mathcal{H}'$ is a one-way hash function, and $f$ is such that:

$$f(C, \mathcal{H}'(File)) = \mathcal{H}(C|File) = R \qquad (2)$$

Unfortunately, we have not found (yet) functions $f$, $\mathcal{H}$ and $\mathcal{H}'$ satisfying this property.

To workaround this problem, a finite number $N$ of random challenges can be generated off-line for each file to be checked, and the corresponding responses computed off-line too. The results are then stored on the verifier. At each integrity check period, one of the $N$ challenges is sent to the server and the response is compared with the precomputed response stored on the verifier. In order to guarantee that a different challenge is issued at each request, and thus that an attacker cannot predict which will be the next challenge(s), the server has to be rebooted periodically[4] so that:

$$N > \frac{\text{(frequency of challenge-response protocol for the same file)}}{\text{(reboot frequency)}} \qquad (3)$$

A possible way for the attacker to circumvent this freshness checking could be to keep copies of both the original and the modified files. But this should be easy to detect, either by checking the integrity of the concerned directories and system tables, or by intrusion detection sensors tuned to detect this specific abnormal behavior.

The table of precomputed responses, stored on the verifier, is thus composed of $N$ entries with, for each entry, a challenge and the expected corresponding response. It is possible to reduce the size of the table, by exploiting a technique presented in [Lamport 1981]: rather than generating a specific random number as a challenge for each entry, only one random number $C_N$ is generated for a file, and each challenge $C_i$ is computed as $\mathcal{H}(C_{i+1})$ for each $i$ from ($N$-1) to 1 (by step of -1). The precomputed response table contains only the $N$ expected responses, the last challenge $C_N$ and the

---

[3] At least $\mathcal{H}'$ or $f$ must be kept secret, because if both were public, it would be easy for the attacker to precompute all needed $\mathcal{H}'(File)$ and then dynamically compute the expected response $f(C, \mathcal{H}'(File))$.

[4] Our current implementation exploits the fact that the servers are periodically rebooted (e.g., once a day), as a measure for software rejuvenation [Huang et al. 1995]: at reboot all files and programs are restored from a secure copy. This would erase any Trojan horse implemented previously by a malicious hacker, as well as any table of precomputed responses he could have built with previous challenges.

number *N*. The challenges are sent in the increasing order from $C_1$ to $C_N$, each challenge $C_i$ being dynamically computed by the verifier:

$$C_i = \mathcal{H}^{(N-i)}(C_N), \text{ where } \mathcal{H}^k(X) = \mathcal{H}(\mathcal{H}^{k-1}(X)) \text{ and } \mathcal{H}^1(X) = \mathcal{H}(X) \qquad (4)$$

## 2.2 A practical example: integrity checking for a distributed web server

The above protocol has been implemented in a distributed, intrusion-tolerant web server [Valdes et al. 2002]. The system consists of a set of *verifiers* managing and monitoring a bank of web *servers* (see Figure 1).

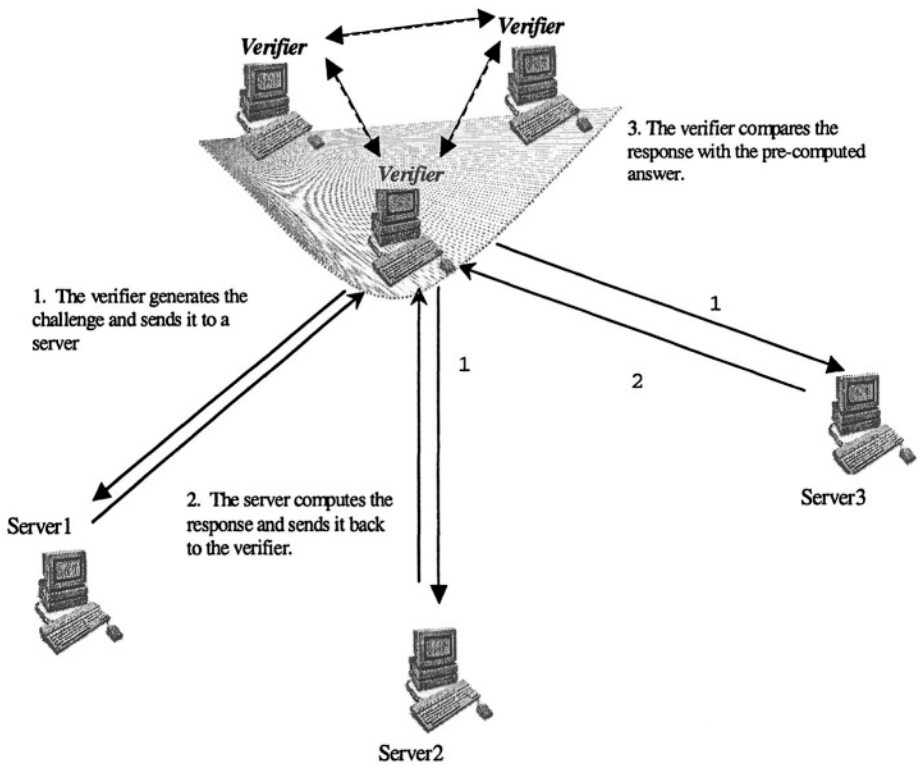


*Figure 1.* Intrusion-tolerant web server architecture

The challenge-response protocol is launched periodically by each verifier to check each server and each other verifier. This protocol is used for three purposes:

- As a *heart beat*: since this protocol is launched periodically on each proxy, if a proxy does not receive a challenge from another proxy for a time greater than the period, it can raise an alarm.
- To check the liveness of the servers and other proxies: if, after emitting a challenge, a proxy does not receive a response within some delay, it can raise an alarm.
- To check the integrity of some files, directories or tables located on remote servers and proxies.

The challenge-response protocol (CRP) was created to check the integrity of some files which are not modified during normal operation, such as sensitive system files (e.g., boot files and OS code files) or security-critical files (e.g., /etc/passwd on a UNIX system). The role of a web server is to produces HTML documents that could be static files (web pages) or dynamically produced (by CGI or ASP scripts) from static data (e.g., a read-only database). So CRP checks important HTML files, scripts and system and security files. It can also check the identity of sensitive active processes on the machine (e.g., httpd, security processes, etc.).

If the server is rebooted periodically for "software rejuvenation", there is a relation between the frequency of the CRP, the duration of one CRP exchange, the number of files to be checked, and the number of servers:

If the server is rebooted periodically for "software rejuvenation", there is a relation between the frequency of the CRP, the duration of one CRP exchange, the number of files to be checked, and the number of servers:

Considering $n$ the number of the checked files, $f$ the frequency of CRP ($f < 1/d$, $d$ being the duration of one CRP exchange), and $N$ the number of challenges per file, the relation is:

$$(n \times N \times \#\text{servers} / f) >= \text{time between reboots} \quad (5)$$

There is a minimal value of $1/f$ that corresponds to the maximal value of the duration of an execution of CRP which is related to a request on the biggest checked file. The next table gives examples of the performance of CRP (when using MD5 as hash function, on Pentium III, 600 Mhz):

*Table 1.* Execution duration

| File size | Duration of one execution of CRP |
|---|---|
| 2,2 Mbytes | ~ 0,66 s |
| 13,5 Kbytes | ~ 0,008 s |

So if we consider that the biggest file to check is about 2 Mbytes, we must choose a value of $1/f > 0{,}66$ sec.

The following table gives the size of the precomputed response table for different values of $n$, $1/f$ and $N$, considering 4 servers and a reboot frequency of one per 24 hours.

*Table 2.* Frequency and size

| n | 1/f | N | Table size |
|---|---|---|---|
| 50 | 5 s | 87 | ~578 Kbytes |
| 500 | 1 s | 44 | ~2. 92 Mbytes |
| 5000 | 0 ,7 s | 7 | ~4.76 Mbytes |

## 3. A SOLUTION BASED ON THE PROTOCOL OF DIFFIE-HELLMAN

We here describe a generic solution based on the well-known cryptographic protocol of Diffie-Hellman for key exchange [Diffie & Hellman 1976].

Let:

- $m$ denotes the value of the file to be remotely verified on a server; it is an integer,
- $N$, a RSA modulus, with two prime factors or more, of length of around 1024 bits; this value is public, that is, considered as known by everybody including any malicious hacker with a lot of computing power,
- $phi(N) = L$ is secret and only known by the verifier; this function is the Euler function (if $N=pq$, then $L = (p-1)(q-1)$),
- $a$, an element between 2 and N-2, randomly chosen and public.

The protocol is the following one:

- the verifier stores the following precomputed value

$$a^m \bmod N = M, \tag{6}$$

this computation is easy thanks to the knowledge of $L$ (the theorem of Euler allows us to replace the exponent $m$ by the short value ($m$ mod $L$) of length around 1024 bits, independent of the length of the protected file) and using, if necessary, the Chinese remainder theorem using the knowledge of the prime factors;

- the verifier chooses a random value $r$ (the domain is the same as $a$) and sends the following value $A$ as a challenge to the server with the file to be verified:

$$a^r \bmod N = A \tag{7}$$

- the server computes

$$A^m \bmod N = B \tag{8}$$

  and sends $B$ to the verifier,
- the verifier computes in parallel

$$M^r \bmod N = C \tag{9}$$

  and verifies if $B = C$ thanks to the equation (10).

It is easy to see that the next equation (10) is correct by using the equations (6) and (9),

$$B = A^m \bmod N = a^{rm} \bmod N = M^r \bmod N = C \tag{10}$$

The security of the protocol follows from the security of the Diffie-Hellman protocol. The freshness of computation (8) on the whole file is guaranteed by the random selection of $r$ by the server. Another paper will describe a lot of optimisations of this generic protocol.

## 4. DISCUSSION

In this section, we discuss how an attacker can defeat the proposed solutions and compare these solutions with conventional signature schemes.

To defeat our solutions, a hacker could save each file before modifying them. In that case, the hacked server would serve modified files to innocent users while still being able to compute fresh responses by using the saved file copies. But counter-measures can easily prevent and/or detect the saving of critical files:

- To prevent the hacker to copy the files, the server file system can be dimensioned in such a way that there would be no room for critical file copies.
- It is easy for a host-based intrusion detection system to discriminate the file copying from the normal server behavior. Moreover, the challenge-response protocol can be applied not only to data files, but also to directories, and even system tables, which stay mostly static on, dedicated servers. This would make the hacker's job much more complex.

An alternative, conventional way to check file integrity consists in signing every file by using a private owner's key, while each user would be

able to check the file integrity by retrieving the corresponding public key through a public key infrastructure. But this solution, while well adapted for software distribution, presents many drawbacks for other applications:

- It is not directly applicable to web services: the replies to http requests are generally not a simple file content, and even when it is the case, the integrity checks would have to be integrated in browsers, with all the complexity associated with PKI management.
- A hacker could still replace the current copies of the files with obsolete copies with their original signatures.
- It would not solve the remote server management problem: the administrator would still have to retrieve the contents of all the files to check their integrity.

## 5. RELATED WORK

Tripwire® [Kim & Spafford 1993] is the most famous file integrity checker. It has been designed to monitor a set of files and directories for any changes according to signatures previously stored. Tripwire proposes a set of signature functions (MD5, MD4, MD2, Snefru and SHA). By default, MD5 and Snefru are stored and checked for each file but the selection-mask can be customized and any of these functions can be selected. The user must first generate, offline, a configuration file containing the list of the files to be monitored and constitute a database of signatures corresponding to this configuration file. When running, Tripwire scans periodically the file system for added or deleted files in the directories specified in the configuration file, and computes the signatures of the monitored files to compare them with the signatures stored in the database. As previously stated, this approach cannot be directly applied to check the integrity of files stored on a remote server: a corrupted server can store locally the signatures of monitored files before modifying them and, on request by the verifier, return these signatures instead of freshly computed ones.

The SOFFIC project (Secure On-the-Fly File Integrity Checker) is carried out at UFRGS (Brasil) [Serafim & Weber 2002]. Their goal is to create a framework for intercepting file system calls and checking the correctness of any request to access a file (read/write/execute). It should be able to deny access to illegally modified files and to protect itself against tampering. The SOFFIC is implemented as a patch to the Linux kernel so the majority of its components resides in the kernel. The idea is to generate off-line hashes for all the files to be checked (Hash List) and generate a list of non-checked files (Trusted File List). Each time a user process attempts to access a file (which is not in the Trusted File List), SOFFIC is activated to

grant or deny the access: the access is denied if the hash stored in the Hash List differs from the hash computed on-the-fly. For writable files, a new hash is computed after modification.

Rather than modifying the kernel, it is possible to insert a middleware layer between the application software and the system kernel. This solution is more portable and easier to maintain than kernel modifications. Jones [Jones 1993] has proposed to implement this approach by Interposing Agents that control all or parts of the system interface. These agents can be used to implement monitors to check the correctness of system calls, in particular for accessing files. Fraser et al. propose a similar approach, based on software wrappers, to augment the security functionality of COTS software [Fraser et al. 1999]. Such wrappers could be used to protect kernel and critical system files from non-authorized changes.

All these approaches suffer the same problems as Tripwire: if a server is corrupted, its kernel can be modified or the middleware can be bypassed to remove all integrity checks.

## 6. CONCLUSION

In this paper, we proposed two methods for remote file integrity checking. The first one is based on a table of multiple challenges and precomputed responses for each file to be checked, the response being computed by the hash of the challenge concatenated with the content of the file. The freshness of the response computation by the server is guaranteed by the fact that a challenge is never reused before reboot of the server. With the second method, a single value is precomputed and stored on the verifier for each file to be checked, and the challenge is generated randomly. This second method requires more computation (modular exponentiation instead of a hash on the content of the file), but does not require a large table to be stored by the verifier. Many optimizations are possible on the second method to reduce the computation cost, and they will be presented in a future article, with performance comparison with the first method.

## ACKNOWLEDGEMENTS

This research was initiated during collaboration between LAAS-CNRS and SRI International, partially supported by DARPA under contract number N66001-00-C-8058. The views herein are those of the authors and do not necessarily reflect the views of SRI International or DARPA. We are deeply

grateful to our SRI International colleagues for the fruitful cooperation on this project, and for their help in improving this paper.

## REFERENCES

[CERT 2002] CERT Advisory CA-2002-24, *Trojan Horse OpenSSH Distribution,* August 1, 2002.

[Diffie & Hellman 1976] W. Diffie and M.E. Hellman, "New Directions in Cryptography", *IEEE Transactions in Information Theory,* 22(1976), pp. 644-654.

[Fraser et al. 1999] T. Fraser, L. Badger and M. Feldman, "Hardening COTS Software With Generic Software Wrappers", Proc. of IEEE Symposium on Security and Privacy, 1999, pp. 2-16.

[Green et al. 1999] John Green, David Marchette, Stephen Northcutt, Bill Ralph, "Analysis Techniques for Detecting Coordinated Attacks and Probes", in *Proc. 1st USENIX Workshop on Intrusion Detection and Network Monitoring,* Santa Clara, California, USA, April 9-12, 1999, available at:
<http://www.usenix.org/publications/library/proceedings/detection99/full_papers/green/green_html/>

[Huang et al. 1995] Y. Huang, C. Kintala, N. Kolettis, N.D. Fulton, "Software Rejuvenation: Analysis, Module and Applications", in *Proc. 25th IEEE Symposium on Fault Tolerant Computing Conference (FTCS-25),* Pasadena, CA, USA, June 1995, pp. 381-390.

[Jones 1993] M. Jones, "Interposition Agents: Transparently Interposing User Code at the System Interface", *Proc. 14th ACM Symp. on Operating Systems Principles,* Operating Systems Review, 27[5], December 1993, pp. 80-93.

[Kim & Spafford 1993] G.H. Kim and E.H. Spafford, *The Design and Implementation of Tripwire: a File System Integrity Checker,* Technical Report CSD-TR-93-071, Computer Science Dept, Purdue University, 1993.

[Lamport 1981] Leslie Lamport, "Password Authentication with Insecure Communication", *Comunications of the ACM,* 24(11), pp. 770-772, November 1981.

[Serafim & Weber 2002] Vinícius da Silveira Serafim and Raul Fernando Weber, *The SOFFIC Project,* < http://www.inf.ufrgs.br/~gseg/projetos/the_soffic_project.pdf>.

[Valdes et al. 2002] A. Valdes, M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saïdi, V. Stavridou and T. Uribe, "An Adaptative Intrusion-Tolerant Server Architecture", in *Proc. 10th International Workshop on Security Protocols,* Cambridge (UK), April 2002, to appear in Springer LNCS Series.