The Design and Implementation of Tripwire: A File System Integrity Checker*

Gene H. Kim and Eugene H. Spafford

COAST Laboratory
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398

February 23, 1995

Abstract

At the heart of most computer systems is a file system. The file system contains user data, executable programs, configuration and authorization information, and (usually) the base executable version of the operating system itself. The ability to monitor file systems for unauthorized or unexpected changes gives system administrators valuable data for protecting and maintaining their systems. However, in environments of many networked heterogeneous platforms with different policies and software, the task of monitoring changes becomes quite daunting.

Tripwire is tool that aids UNIX system administrators and users in monitoring a designated set of files and directories for any changes. Used with system files on a regular (e.g., daily) basis, Tripwire can notify system administrators of corrupted or altered files, so corrective actions may be taken in a timely manner. Tripwire may also be used on user or group files or databases to signal changes.

This paper describes the design and implementation of the Tripwire tool. It uses interchangeable "signature" (usually, message digest) routines to identify changes in files, and is highly configurable. Tripwire is no-cost software, available on the Internet, and is currently in use on thousands of machines around the world.

1 Introduction

Most modern computer systems incorporate some form of long-term storage, usually in the form of files stored in a file system. These files typically contain all of the long-lived data in the system, including both user data and applications, and system executables and databases. As such, the file system is one of the usual targets of an attack. Motives for altering system files are many. Intruders could modify system databases and programs to allow future entry. System logs could be removed to cover their tracks or discourage future detection. Compromised security could lead to degradation or denial of services. Modification

*This paper is to appear in the Proceedings of the 2nd ACM Conference on Computer and Communications Security, 1994. An earlier version of this paper was released as Purdue Technical Report CSD-TR-93-071.

or destruction of user files might also compromise aspects of the security policy. As such, the security administrator needs to closely monitor the integrity of the file system contents.

The near-ubiquitous UNIX system is an example of a file system where such monitoring is useful. Flaws and weaknesses in typical UNIX systems are well-documented (e.g., [8, 25, 19, 4, 9]). UNIX file systems are susceptible to threats in the guise of unauthorized users, intruders, viruses, worms, and logic bombs as well as failures and bugs. As such, UNIX system administrators are faced with prospects of subtle, difficult-to-detect damage to files, malicious and accidental.

Tripwire is an integrity checking tool designed for the UNIX environment to aid system administrators to monitor their file systems for unauthorized modifications. First made available on November 2, 1992, it has proven to be a popular tool, being portable, configurable, scalable, flexible, manageable, automatable, and secure. It was written in response to repeated break-in activity on the Internet, and the difficulty experienced by affected administrators in finding all of the "backdoors" left by the intruders.

The foundations of integrity checking programs are surveyed in [2]. In simplest terms, a database is created with some unique identifier for each file to be monitored. By recreating that identifier (which could be a copy of the entire file contents) and comparing it against the saved version, it is possible to determine if a file has been altered. Furthermore, by comparing entries in the database, it is possible to determine if files have been added or deleted from the system.

As described in [9], a *checklist* is one form of this database for a UNIX system. The file contents themselves are not usually saved as this would require too much disk space. Instead, a checklist would contain a set of values generated from the original file — usually including the length, time of last modification, and owner. The checklist is periodically regenerated and compared against the saved copies, with discrepancies noted. However, as noted in [9], changes may be made to the contents of UNIX files without any of these values changing from the stored values; in particular, a user gaining access to the *root* account may modify the raw disk to alter the saved data without it showing in the checklist.

Efficiently detecting changes to files under these circumstances can be done by storing a value calculated from the contents of the files being monitored. If this value is dependent on the entire contents of the file and is difficult to match for an arbitrary change to the file, then storing this value is sufficient. This *fingerprint* or *signature* of the file can then be saved instead of the file contents.[1] The signature function(s) used should be computationally simple to perform, but infeasible to reverse. It should signal if the file changes but be sufficiently large as to make a chance collision unlikely. Signature functions and methods are discussed in [24, 18, 9, 17, 4, 7, 16, 23].

Although various candidate signature functions have been studied over the past few years, we were unaware of any tool in general use that used these methods under UNIX. This led to the design of Tripwire.

2 Problem Definition

Ultimately, the goal of integrity checking tools is to detect and notify system administrators of changed, added, or deleted files in some meaningful and useful manner. The success of such a tool depends on how well it works within the realities of the administration environment. This includes appropriate flexibility

[1] Some contend that the term *signature* should be used only when referring to functions that have roots in cryptographic methods. In this paper, we use the term in a more general connotation: the fixed-size "fingerprint" generated by a function using the contents of a file as its input data.

to fit a range of security policies, portability to different platforms in the same administrative realm, and ease of use. We also believe that it is important that any such tool present minimal threat to the system on which it was used; if the tool were to be read or executed by an attacker, it should not allow the system to be compromised.

From this basic view, we identified several classes of issues for further study.

2.1 Administration issues

It is not uncommon for system administrators to have sites consisting of hundreds of networked machines. These machines may consist of different hardware platforms, operating systems, releases of software, and configurations of disks and peripherals. Some machines are critical because of their specialized functions, such as mail and file services. These variables increase the complexity of administration.

Furthermore, system administrators manage these machines within the confines of local policies, dictating backups, user accounts, access, and security. Even small sites may have different policies for machines based on their roles.

To administer these machines, configurations may be classified into logical classes based on their purpose (e.g., desktop machines, file servers). This maximizes potential configuration reuse and reduces opportunities for error.

A well-designed tool must work within these conditions. It must be scalable to networks consisting of hundreds of machines. The tool must be flexible to handle different and unique configurations, at some cost to complexity. However, appropriate support for reuse helps to reduce complexity and exploit existing commonality of logical classes of machines. Thus, an integrity tool should be both able to handle many special-case configurations and to support reuse of configuration information based on common characteristics.

2.2 Reporting issues

To aid in the detection of the appropriate threats, system administrators would use an integrity checker to monitor file systems for added, deleted, and changed files. Meaningfully reporting changed files is difficult, because most files are expected to change: system log files are written to, program sources are updated, and documents are revised. Typically, these changes would not concern system administrators. However, changes to certain files, such as system binaries, might elicit a different reaction.

Similarly, changes to certain file attributes (stored in the file's inode structure [1]) occur frequently and are usually benign. A tool reporting every changed file potentially forces security administrators to interpret large amounts of data. Interpreting needlessly large reports cluttered with unimportant information increases the risk of genuinely interesting and noteworthy reports being lost or missed.[2]

For example, consider the tedium imposed by a scheme that requires system administrators to search for reports of potentially dangerous file ownership changes, obscured by reports of thousands of files whose access timestamp changed. However, in some of those cases, changes to a file's access timestamp may be of great interest. For instance, "trap files" could be placed as tripwires against snooping intruders.[3] If the

[2]This is quite similar to the problem of audit trail reduction.

[3]Hence the original motivation for the name "Tripwire."

system is properly configured, security administrators could learn when an intruder or local "snooping" user has accessed the trapped file, thus unavoidably updating the file's timestamp.

Supplying some form of global filter to the output of the monitor program might help reduce the reports to a more manageable volume. There are difficulties with this approach, however. It may not be possible to write general rules that remove noise while adequately preserving interesting events. Global filter rules may prevent system administrators from carrying out local, and possibly very unusual, policies. We believe it is better to generate only those events of interest rather than filter meaningful events from a collection of all possible events.

2.3 Database issues

The database used by the integrity checker should be protected from unauthorized modifications; an intruder who can change the database can subvert the entire integrity checking scheme. Although the system administrator can secure the database by storing it on some media inaccessible to remote intruders (e.g., paper printout), usability is sacrificed. A database stored in some machine readable format may risk unauthorized modification, but allows the integrity checking process to be automated. Storing the database on read-only media provides the best of both approaches, allowing machine access but preventing changes. This also will allow users to use the tool to monitor their own files, if they wish.

After a reported file addition, deletion, or change is determined to be benign, the database should be updated to reflect the change. This prevents the change from appearing in future reports. Furthermore, comparisons for changed files should be made with up-to-date information. Updating a database stored on read-only media poses obvious procedural difficulties. The integrity checking protocol must allow some mechanism or procedure for the secure installation of updated databases.

Because files systems are dynamic in nature, their associated databases may require updating often. Therefore, updating specific entries should not require regenerating the entire database. As many files may change, enumerating each file to be updated could be tedious. Tedium should be avoided to encourage and support use of the tool.

The database should contain no information that allows an intruder to compromise the integrity checking scheme. This allows databases to be shipped with software distribution packages, whose circulation can not be easily restricted.

2.4 File signature issues

Selection of appropriate signatures to use in an integrity checking tool should help engender trust in the tool. Thus, it is important to address issues related to the selection of one or more functions to generate the file signatures.

2.4.1 Change detection

A simple method for detecting a changed file is comparing it against a previously made copy. This has the advantage of giving system administrators the ability to tell exactly what change was made to the file. However, this method is resource and time intensive, potentially doubling the space used by the file system and necessitating further support from system administration staff. In many cases, knowing that a change has been made is all that is necessary.

A more efficient method would record the file's fixed-size signature in the database. One consequence of fixed-sized signatures is multiple mappings: for any given signature generated by a file, there are many (possibly infinite) other files of varying sizes that also generate that same signature. What is important here is that the functions be chosen such that it is highly unlikely that an attacker could alter a file in such a way that it coincidentally retains its original signatures.

2.4.2 Signature spoofing

Intruders could modify a file and remain undetected in an integrity checking scheme using file signatures if the file can be further modified to generate the same signature as the original. Two methods for finding such a modification are brute force search, and inverting then spoofing the signature function.

Given a modified file, someone using a brute force search would iteratively scan for an offsetting change in the file that yields the desired signature. For a signature of size n bits, on average, one might expect to perform 2^{n-1} attempts to find such a signature collision.

For small files, this search is a trivial operation using high-speed, general-purpose workstations. Consider the case of finding a duplicate signature for the `/bin/login` program under SunOS 4.1. This is a 47 kilobyte binary file. Using a SparcStation 1+ (a common 12.5 MIPS machine), a duplicate 16-bit CRC (*Cyclic Redundancy Checkcode*) signature preserving the file's length can be found in 0.42 seconds. A duplicate 32-bit CRC signature can be found in four hours.

However, exhaustive search becomes unnecessary if one exploits knowledge of the workings of the signature function itself. By understanding how a function generates a signature, one could reverse-engineer the function. For any desired signature, an intruder could reverse the signature function and generate an arbitrary file that also yields that signature (cf. [16]).

For these reasons, message-digest algorithms (also known as one-way hash functions, fingerprinting routines, or manipulation detection codes) as described in [7, 17, 16, 23] become valuable as integrity checking tools. Message-digests are usually large, often at least 128 bits, and computationally infeasible to reverse.

2.4.3 Empirical results

Table 1 shows signature collision frequencies for 254,686 files. These signatures were gathered from file systems residing on five computers at Purdue University and two computers at Sun Microsystems, Inc. These files were in active user directories and source trees, and are a representative sampling of files residing on large, timeshared, general purpose servers and large file servers used as source repositories.

Each file examined had its signatures generated using (in order) the 16-bit SunOS `sum` command, two standard CRC algorithms, the final 64 bits from a DES-CBC[6] encoded version of the file, and the 128-bit values taken from standard message digest functions. The large number of collisions for the 16-bit signatures, and the absence of any collisions for the 128-bit signatures, confirms the expected observation that larger signatures are less likely to collide by accident.

Frequency of Signature Collisions (254,686 signatures)										
Signature	Number of collisions									Total
	1	2	3	4	5	6	7	8	>9	
16-bit checksum (sum)	14177	6647	2437	800	235	62	12	2	1	24375
16-bit CRC	15022	6769	2387	677	164	33	5	0	0	25059
32-bit CRC	3	1	1	0	0	0	0	0	0	5
64-bit DES-CBC	1	1	0	0	0	0	0	0	0	2
128-bit MD4	0	0	0	0	0	0	0	0	0	0
128-bit MD5	0	0	0	0	0	0	0	0	0	0
128-bit Snefru	0	0	0	0	0	0	0	0	0	0

Table 1: Collision frequencies of signatures gathered from file systems at Purdue University and Sun Microsystems, Inc.

We also generated empirical support of the difficulty of spoofing 128-bit signatures. An attempt was made to find a duplicate Snefru[16] signature for the `/bin/login` program using 130 Sun workstations.[4] Over a time of several weeks, 17 million signatures were generated and compared with ten thousand stored signatures, the maximum number of signatures that fit in memory without forcing virtual memory page faults on each search iteration. Approximately 2^{24} signatures were searched without finding any collisions, leaving approximately 10^{15} remaining unsearched.

2.5 Performance and resource issues

Detecting file tampering by comparing each file against a duplicate copy is easy to do, but requires considerable storage and time. Generating and comparing file signatures may require more computation, but it requires less storage. Some signature functions are quite expensive to execute in software, while others are simpler. Local policy should dictate the signatures and resources used to satisfy the level of trust desired.

2.6 Other issues

Security tools should be completely self-contained, needing no auxiliary programs to run. For example, an integrity checker that depends on utilities such as `diff` or `sum` could be subverted if either of those programs were compromised. Thus, by making this tool self-contained, it would be possible to run the program without relying on outside, potentially vulnerable, helper programs.

The database for the tool should be human-readable. This not only provides an alternate means of checking the database for potential tampering (e.g., comparison against a printed copy), but it also provides a means for users to verify individual files. By including a standalone program to apply the signature functions to an arbitrary file, a user could compare this against the signature database.

[4] We measured a Sun SparcStation 1 as capable of generating 37 Snefru signatures per second

The program should be able to run without privilege, possibly on a user's private set of files. Additionally, it should only report, and not effect, changes. Although a user could use the tool's output to drive changes, the tool itself would not provide any explicit means of making alterations to the system. This was also one of the principles at the heart of the COPS tool,[8] and one which we believe contributed greatly to its wide-spread acceptance and use.

3 Existing Tools

Most available UNIX security tools fall into two categories: static audit tools and integrity checkers. Among the most prominent are COPS[8], TAMU[22], crc_check[8], Hobgoblin[15], and ATP[28].[5] A few commercial security tools also exist, but they are comparable to the user-community tools mentioned here. While many of these tools may be outstanding in their own right, most are mismatches for integrity checking in UNIX environments.

3.1 COPS

COPS serves as a good benchmark for static audit tools. Freely distributed since 1989, it is widely used and supports a large number of UNIX platforms. It is comprehensive, configurable, and thorough. However, as a static audit tool, it does not aid in intrusion detection other than identifying known avenues of penetration.

The lack of integrity monitoring in COPS was addressed after its release by the addition of the *crc_check* program. It is a "checklisting" program, similar to the shell scripts described in [9, 4]. It is based on a simple CRC checksum of the files being monitored. Numerous problems prevent it from being a comprehensive integrity checking solution as we have outlined in the previous sections.

Most obvious is the lack of extensibility and flexibility in crc_check. It is impossible to update a database entry without regenerating the entire database. Experience has shown that a more sophisticated program is necessary to be useful. For larger sites, maintaining crc_check is especially tedious.

crc_check does not allow all the fields in the UNIX file inode structure to be monitored. This prevents certain changes from being monitored. Furthermore, the reporting cannot be tailored within crc_check. Although filter programs can be written to transform the output, relying on outside programs that can be subverted introduces another point of compromise.

The use of CRC signatures are poorly suited for integrity checking. Originally intended for hardware-based error-detection, CRC functions were designed to detect multiple bit errors in a data stream (e.g., [3]). Reversing the CRC function to yield a desired signature is a well-understood process, and tools to assist a potential intruder are widely available[10].

3.2 TAMU

TAMU is a set of security utilities being distributed by Texas A&M University.[22] Included in the package is a static audit tool, a signature database to check system binaries against known signatures of patch files, and a sophisticated network traffic analyzer that aids system administrators in assessing outside threats.

[5] SPI, a widely-used tool developed by the U. S. Department of Energy and the U. S. Air Force, is not discussed in this paper; future releases of SPI will supposedly be based on the COPS tool.

TAMU is shipped with a database of signatures for system binaries of popular operating systems. TAMU compares signatures of critical system files against those stored in its database to determine whether they match any of the known versions. TAMU can thus notify the security administrators of binaries with security patches that have not been installed by the operating system vendor as determined by records in its signature database.

TAMU is more specialized than most integrity checkers, but requires that its database be updated as new operating system versions and patches are released. Although this tool provides valuable information to system administrators, it is not a general-purpose integrity checker: it provides no facilities to scan the entire file system for changes.

3.3 Hobgoblin

Hobgoblin was written as tool to aid system administrators in enforcing local file system policies.[15] For instance, when more than one person is allowed to install and delete files, it becomes difficult to track changes. Hobgoblin can assist in tracking these changes.

Hobgoblin uses a template description that specified files and directories are expected to match. It then scans those files to check whether the files match the descriptions. In this manner, any changes can be reported to the system administrator.

Hobgoblin does not have all the capabilities associated with integrity checkers: detecting added and deleted files is not straightforward in Hobgoblin. There is no existing interface for storing a file's signature in the database. Furthermore, Hobgoblin assumes that files in its database do not change often. Because of this, no provisions for updating the database exist. This makes its use in dynamic file systems difficult.

3.4 ATP

A recent paper describes a forthcoming program for UNIX, named ATP.[28] It employs a dual signature to verify files, using a 32-bit CRC and the MD5 message digest algorithm. The ATP database is encrypted using DES in Cipher Block Chaining mode, and is checksummed to detect tampering and prevent unauthorized updates. However, this prevents its use in an automated manner: the secure entry of the encryption key requires human intervention or else storage in the file system — thus compromising the entire program. The lack of any mechanisms for updating the database potentially makes maintenance as tedious as crc_check.

An interesting design decision was the introduction of *action lists*. Having detected a changed file, ATP can automatically change the ownership to `root` and make it inaccessible to all users. This feature makes ATP unique among the security tools listed in this section, because it does more than report potential dangers. Provided that the actions are suitable under local policies, this automated form of damage control could be very useful to system administrators. However, as we noted earlier, this is of questionable utility. Accidental triggering of the rules and malformed actions are two dangers in such a mechanism. Furthermore, a determined attacker might well be able to exploit this mechanism to perform denial-of-service attacks.

4 Implementation of Tripwire

Tripwire was written over a period of two months in 1992. It was released in the fall of 1992 to a group of over one hundred beta testers around the world who provided valuable feedback on its portability and features. Several bugs have been identified, and four updates were released in 1993. In December 1993, the formal release of Tripwire was made. Some of the more involved software engineering aspects of Tripwire construction and maintenance are discussed in [14].

This section describes the structure of Tripwire. A high level model of Tripwire operation is shown in Figure 1. This shows how Tripwire uses two inputs: a *configuration* describing file system objects to monitor, and a *database* of previously-generated signatures putatively matching the configuration. *Selection-masks* (described below) specify file system attributes and signatures to monitor for the specified items.

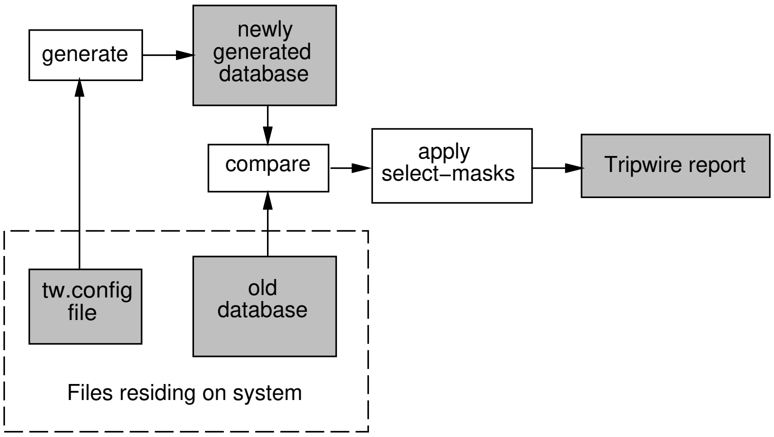

Figure 1: Diagram of high level operation model of Tripwire

4.1 Administrative model

4.1.1 Portability

Because of the heterogeneous nature of computer equipment at most sites, the design of Tripwire emphasized program and database portability. The code is written in the standard K&R C programming language,[12] adhering to POSIX standards wherever possible. The result is a program that compiles and runs on at least 28 BSD and System-V variants of UNIX, including Xenix and Unicos.

Tripwire database files are encoded in standard ASCII and are mostly human readable. They are completely interoperable (i.e., files generated on one platform can be read and used on other platforms). This allows the database files to be printed using standard software, compared using standard text tools, and examined using other standard tools.

Generating correct signatures is complicated by architectural differences in byte-ordering (i.e., big-endian vs. little endian). An automated installation procedure generates macros and header files so that the signatures generated are uniform; the standard "network-order" byte order used in the IP protocol suite is our underlying model. This allows database files to be used on machines different from those on which they are generated, if this should be desired (and as might be the case with some networked file systems and software distributions).

A comprehensive test suite is included in the Tripwire distribution to confirm correct signature generation. The test suite also checks each file in the distribution against those stored in a database, ascertaining each file's integrity. This serves both to check the consistency of the distribution, and to ensure that all features of the Tripwire program are working as expected.

4.1.2 Scalability

Tripwire includes an M4-like preprocessing language [11] to help system administrators maximize reuse of configuration files. By including directives such as "`@@include`", "`@@ifdef`", "`@@ifhost`", and "`@@define`", system administrators can write a core configuration file describing portions of the file system shared by many machines. These core files can then be conditionally included in the configuration file for each machine.

To allow the possible use of Tripwire at sites consisting of thousands of machines, configuration and database files do not need to reside on the actual machine. Input can be read from file descriptors, open at the time of Tripwire invocation. These file descriptors can be connected to UNIX pipes or network connections. Thus, a remote server or a local program can supply the necessary file contents. Supporting UNIX style pipes also allows for outside programs to supply encryption and compression services — services that we do not anticipate including as a standard part of the core Tripwire package.

Tripwire does not encrypt the database file so as to ensure that runs can be completely automated (i.e., no one has to type in the encryption key every night at 3 a.m.). Because the database contains nothing that would aid an intruder in subverting Tripwire, this does not undermine the security of the system. However, if Tripwire is used in an environment where the database is encrypted as a matter of policy, the interface supports this, as described above.

4.1.3 Configurability and flexibility

Tripwire makes a distinction between the configuration file and the database file. Each machine may share a configuration file, but each generates its own database file. Thus, identically configured machines can share their configuration database, but each has its integrity checked against a per-machine database.

Because of the preprocessor support, system administrators can write Tripwire configuration files that support numerous configurations of machines. Uniform and unique machines are similarly handled. This helps support reuse and minimize user overhead in installation.

The configuration file for Tripwire, `tw.config`, contains a list of entries, enumerating the set of directory (or files) to be monitored for changes, additions, or deletions. Associated with each entry is a selection-mask (described in the next section) that describes which file (inode) attributes can change without being reported as an exception. An excerpt from a set of `tw.config` entries is shown in Figure 2.

Prefixes to the `tw.config` entries allow for pruning (i.e., preventing Tripwire from recursing into the specified directory or recording a database entry for a file). Both inclusive and non-inclusive pruning are

```
# file/dir      selection-mask
/etc            R       # all files under /etc
@@ifhost solaria.cs.purdue.edu
 !/etc/lp               #  except for SVR4 printer logs
@@endif
/etc/passwd     R+12    # you can't be too careful
/etc/mtab       L       # dynamic files
/etc/motd       L
/etc/utmp       L
=/var/tmp       R       # only the directory, not its contents
```

Figure 2: An excerpt from a `tw.config` file

supported; that is, a directory's contents only may be excluded from monitoring, or the directory and its contents may both be excluded.

By default, all entries within a named directory are included when the database is generated. Each entry is recorded in the database with the same flags and signatures as the enclosing, specified directory. This allows the user to write more compact and inclusive configuration files. Some users have reported using configuration files of a simple `/`, naming all entries in the file system!

4.2 Reporting model

The `tw.config` file contains the names of files and directories with their associated selection-mask. A selection-mask may look like: `+pinugsm12-a`. Flags are added ("+") or deleted ("-") from the set of items to be examined.

Tripwire reads this as, "Report changes in permission and modes, inode number, number of links, user id, group id, size of the file, modification timestamp, and signatures 1 and 2. Disregard changes to access timestamp."

A flag exists for every distinct field stored in an inode. Provided is a set of templates to allow system administrators to quickly classify files into categories that use common sets of flags:

read-only files Only the access timestamp is ignored.

log files Changes to the file size, access and modification timestamp, and signatures are ignored.

growing log files Changes to the access and modification timestamp, and signatures are ignored. Increasing file sizes are ignored.

ignore nothing self-explanatory

ignore everything self-explanatory

```
changed: -rw-r--r-- root           20 Sep 17 13:46:43 1993 /.rhosts
### Attr        Observed (what it is)         Expected (what it should be)
### =========== ============================= =============================
/.rhosts
st_mtime:       Fri Sep 17 13:46:43 1993      Tue Sep 14 20:05:10 1993
st_ctime:       Fri Sep 17 13:46:43 1993      Tue Sep 14 20:05:10 1993
```

Figure 3: Sample Tripwire output for a changed file

Any files differing from their database entries are then interpreted according to their selection-masks. If any attributes are to be monitored, the filename is printed, as are the expected and actual values of the inode attributes. An example of Tripwire output for changed files is shown in Figure 3.

A "quiet option" is also available through a command-line option to force Tripwire to give terse output. The output when running in this mode is suitable for use by filter programs. This allows for automated actions, similar to those allowed in ATP if it is really desired. One example would be to use the terse output of Tripwire after a breakin to quickly make a backup tape of only changed files, to be examined later.

By allowing reporting to be dictated by local policy, Tripwire can be used at sites with a very broad range of security policies.

4.3 Database model

Tripwire uses two databases: the configuration file and the output database. The design and intended use of both of these files is described in this section.

4.3.1 Inviolability

Tripwire uses an unencrypted database that can be world-readable. To prevent the database from being altered, it should be stored on some tamper-proof media. One method of accomplishing this involves storing the databases on a write-protected disk or on a "secure server" where logins can be strictly controlled. The database could also be made available via a read-only remote file system (e.g., read-only NFS [26]).

Installing an updated database is problematic because intruders might replace the database (or selected entries) with one of their own choosing during the update. Therefore, to best ensure the security of the database, the Tripwire documentation suggests that the machine be operated in single-user mode to install the database. System administrators can thus choose greater security over ease-of-use, allowing for the possible enforcement of even the most severe policies.

4.3.2 Semantics

Changes to the database can be categorized into six cases, as shown in Table 2. For each of these cases, an appropriate action is taken, based on whether the file is a `tw.config` entry, and whether the file exists in the old and newly generated databases.

Updating or deleting a file from the database is straightforward — the database entry for the file is replaced by a new entry reflecting the current state of the file. Adding files is more complex as there is no associated selection-mask for the file (i.e., there is no `tw.config` entry for it). To resolve this, Tripwire scans the list of `tw.config` entries and chooses the "closest" ancestor entry, whose selection-mask it inherits. If no such entry can be found, the file is added with a default selection-mask.

Adding, deleting, and updating entries is also simple. All the files in the database that were generated from the given entry are also added, deleted, or updated, appropriately. The updates are done to a copy of the file in case of some system failure. The user must then replace the old database with the modified version.

4.3.3 Interface

Specifying files to be updated can be done via the command-line. Tripwire also has an interactive update mode where the user is asked whether the database entry should be changed for each changed, added, or deleted file. This allows the system administrator to easily update the database, and ensures that no files are inadvertently updated without review. Updating the database is a process that should not be overly automated because its careful review is as important as reports of changed files.

4.4 Signatures model

Tripwire has a generic interface to signature routines and supports up to ten signatures to be used for each file. The following routines are included in the latest Tripwire distribution: MD5[21] (the RSA Data Security, Inc. MD5 Message-Digest Algorithm), MD4[20] (the RSA Data Security, Inc. MD4 Message-Digest Algorithm), MD2 (the RSA Data Security, Inc. MD2 Message-Digest Algorithm),[6] 4-pass Snefru[16] (the Xerox Secure Hash Function),[7] 128-bit HAVAL[29], and SHA (the NIST Secure Hash Algorithm). Tripwire also includes POSIX 1003.2 compliant CRC-32 and CCITT compliant CRC-16 signatures.

Each signature may be included in the selection-mask by including its index. Because each signature routine presents a different balance in the equation between performance and security, the system administrator can tailor the use of signatures according to local policy. By default, MD5 and Snefru signatures are recorded and checked for each file. However, different signatures can be specified for each and every file. This allows the system administrator great flexibility in what to scan, and when.

Also included in the Tripwire distribution is `siggen`, a program that generates signatures for the files specified on the command line. This tool provides a convenient means of generating any of the included signatures for any file.

The code for the signature generation functions is written with a very simple interface. Thus, Tripwire can be customized to use additional signature routines, including cryptographic checksum methods and per-site hash-code methods. Tripwire has room for 10 functions, and only seven are preassigned, as above.

---

[6] The copyright on the available code for MD-2 strictly limits its use to privacy-enhanced mail functions. RSA Data Security, Inc. has kindly given us permission to include MD-2 in the Tripwire package without further restriction or royalty.

[7] Although we realize that 4-pass Snefru has been shown to be somewhat weaker than desired, it would still take on the order of 2^{88} operations to duplicate a signature, and this is strong enough for our use.[23]

Filename exists in:			Interpreted action
tw.config entry	old database	newly generated database	
		x	Added file
	x		Deleted file
	x	x	Updated file
x			Added entry
x	x		Deleted entry
x	x	x	Updated entry

Table 2: Enumeration of possible Tripwire update states.

4.5 Performance

Tripwire allows local policy to dictate which signatures are compared against the database. Which signatures to be used can be specified at run-time, as well as in the `tw.config`, allowing flexible policies to be used without modifying configuration files. For example, Tripwire could compare CRC32 signatures hourly, and compare MD5 and Snefru signatures daily, needing only two `cron` entries with the appropriate command line arguments to Tripwire.

5 Tripwire usage

This section summarizes the procedure of building, installing, and using Tripwire on a single machine. This procedure assumes a system administrator who is interested in the maximum level of assurance possible using Tripwire.

5.1 Building Tripwire

First, the administrator would load a clean distribution of the operating system and utilities onto an isolated machine (disconnected from any network, and running in single-user mode). After unpacking the Tripwire distribution, the administrator edits the top level Makefile[27] to specify system-specific tools (e.g., compiler, compiler flags, etc.). Next, the user would choose a `conf-machine.h` header file that describes special options for the machine to be monitored. Currently, 23 machine-specific header files are included; writing a customized header file for a machine not included in this group is a simple procedure for someone with moderate programming skill, and we have been encouraging the authors of such files to share them with us for use in later releases.

After configuring Tripwire in this fashion, system administrators type "`make`" to build the Tripwire binaries. After these files are compiled, typing "`make test`" starts the Tripwire test suite. This test suite exercises all the signature routines to ensure correct signature generation, and then compares all the Tripwire source files against a test database to ensure distribution integrity.

5.2 Installing the database

After building Tripwire, the system administrator should build the system database. The file `tw.config` contains a listing of all the directories and files to be scanned, along with their associated selection-masks. Generalized `tw.config` files are provided for eight common UNIX versions (including generic BSD and SVR4). These files cover the most critical system files and binaries.

After choosing and reviewing this file, the administrator can make his own customizations and additions. After all additions have been made, it is time to create the database. In single-user mode still, so that no user can tamper with the files or system, the user types “`tripwire -initialize`” and waits for Tripwire to finish scanning and recording information on the files listed in the `tw.config` file.

When this is completed, Tripwire reports where the database has been stored, and reminds the user to move the database to read-only media. After having done so, and copied the configuration file and Tripwire binary itself to read-only, the system administrator has successfully installed the database, and can bring the machine back up in multi-user mode.

5.3 Checking the file systems

When running in integrity checking mode, Tripwire rereads the `tw.config` and the database files, and then scans the file system to determine whether any files have added, deleted, or changed. System administrators type “`tripwire`” to generate a report of these files. This must be done in such a way as to ensure that the protected, original version of Tripwire is the one that is run.

Alternatively, typing “`tripwire -interactive`” will run Tripwire in interactive update mode. In this mode, Tripwire scans for added, deleted, or changed files, and for each such file, the user is asked whether or not the entry should be updated. A new database is created, and again, a warning notifies the user to install it on read-only media to ensure the security of the database. Note that Tripwire does not overwrite the existing database. Further note that our system administrator should perform this function in stand-alone mode to maximize protection of the database.

Tripwire is designed so that any user can safely execute it — the database file can be public information, and the binaries require no special privileges to run. If local policy deems this inappropriate, both the database and Tripwire binaries can be made readable and executable by only system administrators. However, by disabling use of Tripwire by general users, they are likewise unable to run the program to monitor their own databases and applications which might not otherwise be covered by the system-wide monitoring.

6 Experiences

Since the initial release, four versions have been released to incorporate bug fixes, support additional platforms, and add new features. The authors estimate Tripwire is being actively used at several thousand sites around the world. Retrievals of the Tripwire distribution from our FTP server initially exceeded 300 per week. Currently, five months after the last official patch release, we see an average of 25 fetches per week. This does not include the copies being obtained from the many FTP mirror sites around the net.

More data on active Tripwire usage can be gleaned from bug reports. One recent patch to Tripwire included code to check for certain rare and erroneous boundary conditions, displaying a banner that asked the user to mail the output to the authors when found. Although the associated bug is now fixed and a corrective patch distributed, until recently the authors received about two of the requested bug reports per day. From this information, we can only surmise that Tripwire use is growing. (The error condition was only triggered when very large databases exceeding 7000 entries were used.)

Tripwire has proven to be highly portable, successfully running on over 28 UNIX platforms. Among them are Sun, SGI, HP, Sequents, Pyramids, Crays, NeXTs, Apple Macintosh, and even Xenix. Configurations for new operating systems has proven to be sufficiently general to necessitate the inclusion of only eight example `tw.config` files.[14, 13]

In the past year, the authors have collected feedback from numerous active sites reporting the effectiveness of Tripwire in detecting changed files on their systems. Several cases have been reported to us of Tripwire finding unauthorized intruders. Other cases have been reported to us of system administrators making unannounced file system updates or configuration changes. At least one case of a bad disk being discovered by Tripwire has also been reported to us. All these classes of stories seem to validate our concept of this integrity checking tool. The last two classes of use have proven to be surprising applications of Tripwire that we did not envision at the time we wrote it!.[13]

According to system administrators, the ability to update Tripwire databases is among its most important features. Files seem to change for many unforeseen reasons. Consequently, the database is updated regularly. The addition of the interactive update facility in Tripwire was among the most enthusiastically received features.

System administrators who are concerned about their security seem to appreciate the information provided by Tripwire. They further appreciate the lack of privilege necessary to run Tripwire, and its passive, report-only mode of operation. To ensure its security and inviolability, “secure NFS servers” are the most commonly used configurations for running Tripwire. However, some sites’ distrust of NFS has motivated the addition of a “Tripwire server” which provides network services for fetching databases and configuration files.

Many users have found the Tripwire sources to be legible and malleable. Eleven user-contributed scripts are included with the Tripwire distribution, and we know of several sites where the users have extensively modified Tripwire to fit local needs. Maintenance of Tripwire has proven similarly easy; adding the SHA signature routine to the distribution was accomplished in less than one hour. The early bug reports often had file and line numbers of the faults. This surprising fact lends support that the approximately 13,000 lines of C code is relatively easy to understand.

7 Conclusions

Tripwire has proven to be a highly portable tool that system administrators can build using available tools. It is completely self-contained, and once built, requires no other tools for execution. Tripwire is publically available, is widely distributed, and widely used.

Tripwire has been used by system administrators in large and small sites: we have documented Tripwire’s active use at single machine sites, as well as sites having several hundreds of machines. We have yet to hear a report of a site where Tripwire was installed and then removed because it did not function according to expectation, or because it was too difficult to build or maintain. Coupled with the many positive comments we have received, and the fact that Tripwire has already caught several intruders, leads us to conclude that our analysis and design are successful. We hope this effort serves as a model for others who consider building security tools with similar goals.

References

[1] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[2] Vesselin Bontchev. Possible virus attacks against integrity programs and how to prevent them. Technical report, Virus Test Center, University of Hamburg, 1993.

[3] J. Compbell. *C Programmer's Guide to Serial Communications*. Howard W. Sams & Co., 1987.

[4] David A. Curry. *UNIX System Security: A Guide for Users and System Administrators*. Addison-Wesley, Reading, MA, 1992.

[5] Edward DeHart, editor. *Proceedings of the Security IV Conference*, Berkeley, CA, 1993. USENIX Association.

[6] Data encryption standard. National Bureau of Standards FIPS, 1977.

[7] Paul Fahn. Answers to frequently asked questions about today's cryptography. Technical Report Version 1.0 draft 1e, RSA Laboratories, 1992.

[8] Daniel Farmer and Eugene H. Spafford. The COPS security checker system. In *Proceedings of the Summer Conference*, pages 165–190, Berkely, CA, 1990. Usenix Association.

[9] Simson Garfinkel and Gene Spafford. *Practical Unix Security*. O'Reilly & Associates, Inc., Sebastopol, CA, 1991.

[10] Chuck Gilmore. README file for PROVECRC.EXE. README file with program, 1991.

[11] Brian W. Kernighan and Dennis M. Ritchie. *The M4 Macro Processor*. AT&T Bell Laboratories, 1977.

[12] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[13] Gene H. Kim and Eugene H. Spafford. Experiences with tripwire: Using integrity checkers for intrusion detection. In *Systems Administration, Networking and Security Conference III*. Usenix, April 1994.

[14] Gene H. Kim and Eugene H. Spafford. Writing, supporting, and evaluating tripwire: A publically available security tool. In *Proceedings of the Usenix Applications Development Symposium*, Berkeley, CA, 1994. Usenix.

[15] Scott Leadly, Kenneth Rich, and Mark Sirota. *Hobgoblin: A File and Directory Auditor*. University Computing Center, University of Rochester, 1991.

[16] Ralph C. Merkle. A fast software one-way hash function. *Journal of Cryptology*, 3(1):43–58, 1990.

[17] W. T. Polk and L. E. Bassham. A guide to the selection of anti-virus tools and techniques. National Institute of Standards and Technology report, December 1992.

[18] Yisrael Radai. Checksumming techniques for anti-viral proposed. In Edward Wilding, editor, *Virus Bulletin Conference Proceedings*. Virus Bulletin, Ltd., September 1991.

[19] Robert B. Reinhardt. An architectural overview of UNIX network security. Technical report, ARINC Research Corportation, February 1993.

[20] R. L. Rivest. The md4 message digest algorithm. *Advances in Cryptology — Crypto '90*, pages 303–311, 1991.

[21] R. L. Rivest. RFC 1321: The md5 message-digest algorithm. Technical report, Internet Activities Board, April 1992.

[22] David R. Safford, Douglas Lee Schales, and David K. Hess. The TAMU security package: An ongoing response to internet intruders in an academic environment. In DeHart [5], pages 91–118.

[23] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, Inc, 1993.

[24] Gustavus J. Simmons. *Contemporary Cryptology: The Science of Information Integrity*. IEEE Press, 1992.

[25] Cliff Stoll. *The Cuckoo's Egg*. Simon & Schuster, Inc., New York, 1990.

[26] Sun Microsystems, Inc. *System and Network Administration*, 1990. Part number 800-3805-10.

[27] Steve Talbott. *Managing Projects with make*. O'Reilly & Associates, Inc., 1991.

[28] David Vincenzetti and Massimo Cotrozzi. ATP anti tampering program. In DeHart [5], pages 79–90.

[29] Y. Zhang, J. Pieprzyk, and J. Seberry. HAVAL — one-way hashing algorithm with variable length of output. In *Advances in Cryptology — AUSCRYPT 92 Proceedings*. Springer-Verlag, 1992. As cited in Schneier.