Writing, Supporting, and Evaluating Tripwire: A Publically Available Security Tool*

Purdue Technical Report CSD-TR-94-019

Gene H. Kim† *and Eugene H. Spafford*

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

12 March 1994

Abstract

Tripwire is an integrity checking program written for the UNIX environment that gives system administrators the ability to monitor file systems for added, deleted, and modified files. First released in November of 1992, Tripwire has undergone several updates and is in current use at thousands of machines worldwide.

This paper begins with a brief overview of what Tripwire does and how it works. We discuss how certain implementation decisions affected the course of Tripwire development. We also present other applications that have been found for Tripwire. These unanticipated uses guided the demands of some users, and we describe how we addressed some of these demands without compromising the ability of Tripwire to serve as a useful security tool.

We also discuss the process of releasing, and then supporting, a widely available and widely used tool across the Internet, and how meeting users' high expectations affects this process. How these issues affected Tripwire, done as as an independent study by an undergraduate, is also discussed. Software tools that were used in developing and maintaining Tripwire are presented. Finally, we discuss problems that remain unresolved and some possible solutions.

1 Introduction

1.1 Genesis

Tripwire is an integrity checking program written for the UNIX environment that gives system administrators the ability to monitor file systems for added, deleted, and modified files. Tripwire

*This paper appeared as [11]

†Gene Kim is currently at the University of Arizona.

was primarily intended to be used for intrusion detection, and its design and operation are described in detail in [9].

The first version of Tripwire was completed in September 1992. Since then, its design and code have been available for use by the community at large. An intensive beta test period resulted in Tripwire being ported to over two dozen variants of UNIX, including several systems neither author has yet to encounter. Currently entering its seventh (and possibly last) revision, we believe there are interesting lessons to be learned from our experience of design, distribution and support of Tripwire.

Tripwire was conceived somewhat accidentally in 1991. In the fall of 1991, Gene Kim was an undergraduate student at Purdue University looking for a research project. He approached Gene Spafford, who had been his professor in several CS courses, to ask about a doing a research project. Spafford suggested that Kim do a project involving message-digest and cryptographic checksum algorithms for integrity management; hours prior to the visit, Spaf had been considering the problem of how to tell which files had been altered after a system break-in. Gene's visit to Gene[1] was thus fortuitous.

During the spring of 1992, Gene performed initial experiments with several digest algorithms to better understand how they worked. One of these experiments involved collecting the signatures[2] of tens of thousands of files on several machines at the university computing center. A small mistake in coding led to every existing file on the main campus instruction computers being touched, thereby being marked for backup, leading to a shortfall in backup media. This resulted in several panicked phone calls at 2 AM by computing center staff and some administrative repercussions — all much to Gene's chagrin and to Gene's amusement. Experimentation was suspended temporarily.

During the summer of 1992, Gene developed the initial Tripwire program during a summer internship. Upon his return to Purdue in the fall, and consultation with Gene, the program was revised, documented, tested, and distributed to beta testers. Officially released on November 2, 1992, Tripwire is now being actively used at thousands of sites around the world. Published in volume 26 of `comp.sources.unix` and archived at numerous FTP sites around the world, Tripwire is widely available and widely distributed.

1.2 Design

Ultimately, the goal of Tripwire is to detect and notify system administrators of changed, added, and deleted files in some meaningful and useful manner. These reports can then be used for the purposes of intrusion detection and recovery. Changed files are detected by comparing the file's inode information against values stored in a previously generated baseline database. Detecting altered files beyond inode attribute checking is provided by also storing several signatures of the the file — hash or checkum values calculated in such a way that it is computationally infeasible to invert them.

A high level model of Tripwire operation is shown in Figure 1. This illustrates how the Tripwire

[1]For the remainder of this paper, both authors are referred to as "Gene" for purposes of symmetry and to more evenly distribute references of blame, chagrin, and scholarship.

[2]Throughout the Tripwire documentation, we refer to secure hashes, message digests, and cryptographic checksums under the generic term "signature."

program uses two inputs: a *configuration* describing the file system objects to monitor, and a *database* of previously generated signatures putatively matching the configuration.

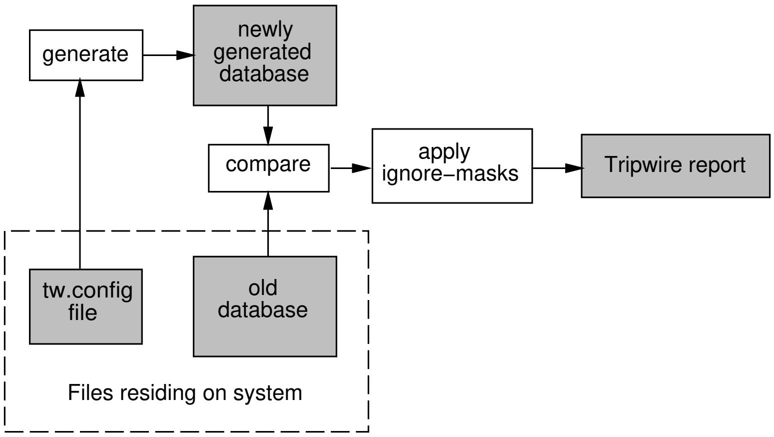

Figure 1: Diagram of high level operation model of Tripwire

In its simplest form, the configuration file contains a list of files and directories to be monitored, along with an associated selection mask (i.e., a list of attributes that can be safely ignored if changed). The database file is initially generated by Tripwire, containing a list of entries with filenames, inode attribute values, signature information, selection masks, and the configuration file entry that generated it.

Tripwire uses these files in all its modes of operation. These modes handle the building of the baseline database, the reporting of all changed, added, and deleted files by comparing a newly generated database against the baseline database, and the updating of the baseline database by replacing out-of-date entries.

More detailed descriptions of operation and features of Tripwire are given in [9] and [10].

2 Theory vs. Practice

As with most research projects, what we thought we wanted and what we finished with were somewhat different.

2.1 What we thought we wanted

Tripwire was envisioned by one of us (Spafford) as a tool to help with intrusion detection and recovery. It was to be a small tool (or set of tools) with a small set of functions. The initial scope of the project was to investigate design issues and to implement a prototype of the tool.

Based on previous experience with various tools, and after some discussion, we agreed that the following goals would be sought after in the design and implementation of Tripwire:

Functionality We wanted to build a tool that could be used to find unauthorized changes in a UNIX file system. The tool should be simple enough to use that most system administrators would (and could) use it.

Portability Among the primary requirements of Tripwire was that it was to be be publically available and freely redistributable. This implied that even the "lowest common denominator" of UNIX systems could build and run Tripwire, underscoring the perceived necessity of a highly portable design. This also meant that the database files for Tripwire would be readable text rather than some fixed-format binary records.

Configurability We wanted a design that would allow site administrators to select what files to monitor and what attributes to monitor. We knew that the security policies and needs of sites would vary considerably, and wanted Tripwire to support those differences.

Flexibility We wanted to make it possible to choose which signatures to generate, or to substitute new signature routines not part of the release. In particular, we want to make it possible for someone to add in one or more cryptographic checksum methods requiring a password for each execution.

Safety We wanted to build a system that required no special privileges to run, and that consisted of source that would be easy to read and understand. Spafford's previous experience with setting this as a design goal for the COPS system[6] has been well-received; system admins are more comfortable running security tools if they can examine the source and customize it if they feel the need.

2.2 What we actually got

Tripwire coding started near the end of the Spring 1992 semester. Semester deadlines for grades undoubtedly motivated some of the early implementation shortcuts that were made at that time. Implementation continued throughout the summer. An intensive testing period began in September 1992, involving two hundred users around the world who has responded to a USENET post.

The first official version of Tripwire was released in November 1992 on the anniversary of the Internet Worm. Since then, seven subsequent versions have been released to incorporate bug fixes, support additional platforms, and add new features. We estimate that 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, seven 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 Internet.

We have received considerable feedback on Tripwire design, implementation, and use. We believe that version 1.1 of Tripwire has succeeded in meeting most of our goals:

Functionality Tripwire appears to meet of the needs of system administrators for an integrity checking tool. We have gathered reports of at least seven cases where Tripwire has alerted system administrators to intruders tampering with their systems. (Experiences with Tripwire for intrusion detection is presented in [10].) The continuing interest in new releases, and the endorsement of Tripwire by various response teams has also confirmed the utility of Tripwire.

Portability Tripwire has proven to be highly portable, successfully running on over 30 UNIX platforms. Among them are Sun, SGI, HP, Sequents, Pyramids, Crays, NeXTs, Apple Macintosh, and even Xenix. Although this has necessitated some awful hacks (some of which are detailed below), the user needs to do very little to configure for a specific machine.

Configurability Tripwire is being used at large homogeneous sites consisting of thousands of workstations, as well as at sites consisting of a single machine. Tripwire allows considerable flexibility in the specification of files and directories to be monitored. Specifying which file attributes can change without being reported allows Tripwire to run silently until a noteworthy filesystem change is detected.

Flexibility Tripwire includes seven signature routines to supplement the file inode information stored in the database to augment the ability to detect changed files in a non-spoofable manner. Because signature routines are often slow cryptographic functions, Tripwire allows system administrators to specify which signatures routines to use for files, and when they should be checked.

Users report that the ability to choose which signatures to use is appreciated and used. We have yet to receive a report of someone (other than ourselves) integrating a cryptographic checksum or a new message digest algorithm, however.

Safety Because Tripwire sources are publically available and freely distributable, they are available for scrutiny by the community at large. Possible weaknesses have been discussed in the literature (e.g., [16]) and by private communication (e.g., [2]). These evaluations were written when our design document was not yet publically available, and reflects positively that our sources are adequately readable.[3]

Furthermore, we have received many reports of system administrators modifying Tripwire, sometimes extensively, to suit their local site needs. In general, such changes have not necessitated changes throughout the sources. Instead, changes have been restricted to one or two files.

As time went on, we discovered that there was one important goal we had completely overlooked in our design: scalability. We failed to recognize the immense size of some installations, and the resulting problem of managing Tripwire data across hundreds of platforms.

Based on our experience with Tripwire, we would encourage designers of security tools to consider carefully all of these goals for their own efforts. We found that reference to these high-level goals helped us resolve questions when they arose, especially when we were evaluating some new feature or extension to be added to the code.

3 What we learned along the way

All my ideas are good, it's only the people who put them into practice that aren't.

Amos Brearly, character in British TV show[5]

[3] All reported weaknesses have been addressed by changes to the code or documentation.

In the previous section, we presented several key design aspects of Tripwire that have supported its widespread use. In this section, we present some of the key (mis)decisions of Tripwire development that significantly affected its development.

3.1 Providing portability

Tripwire was designed and written to run on any reasonably implemented UNIX operating system. The underlying philosophy was to write Tripwire for the lowest common denominator of all existing UNIX systems: building and compiling Tripwire should require only tools usually bundled with the operating system (e.g., K&R C compiler, `lex`, `yacc`). Furthermore, the Tripwire program was designed to be completely self-contained, using no system utility programs such as `grep` or `awk`. This restriction allows system administrators to run Tripwire on machines where the integrity of the system utilities may be in question (e.g., on machines with evidence of intruder tampering).

3.1.1 Theory

Several "meta-configuration" utilities exist, such as the `metaconfig` package originally written by Larry Wall. Generally these packages assist software writers by providing a utility that discovers any quirks or non-standard implementations of the underlying operating system and libraries. Using such a package, the programmer writes for one canonical interface, using the information gathered by the meta-configuration script to customize the code at compile-time.

Against the repeated advice of Gene, the other Gene decided to forego using `metaconfig`. His perception of the simplicity of the Tripwire sources coupled with the perceived complexity of `metaconfig` led him to believe that it would be less work to make small changes to the source code. Thus, provisions were made by grouping UNIX systems into two categories: those derived from AT&T System V and those derived from BSD. While this delineation may seem simple, this partitioning soon proved insufficient to allow straightforward compilation on most machines.

3.1.2 Implementation

In September 1992, we sent the first beta version of Tripwire to over two hundred testers. These testers tried Tripwire on over twenty different platforms, including Suns, HPs, IBMs, NeXTs, Sequents, Crays and PCs running Xenix. They then sent back those changes necessary to allow correct Tripwire compilation and operation on their respective platforms. Although most of the changes were easily merged back into our source tree, virtually all changes to the header sections (i.e., containing the `#include` directives) conflicted with each other.

Creating a framework for correctly including header files in the `/usr/include` hierarchy thus proved surprisingly tedious. Although these header files are intended to hide system-specific data structures from user programs, certain files are needed by virtually every program (e.g., `stdio.h`). However, names of many header files are not consistent across all platforms (e.g., `string.h` and `strings.h`), while others are absent on many machines (e.g., `stdlib.h`). Furthermore, on certain machines, inclusion of two include files may be mutually exclusive. Because many machines used by the initial Tripwire testers (and the Unix-using population at large) predated standards dictating what data structures and definitions reside in which file (e.g., POSIX 1003.2), no assumptions about these include files could be made.

This fact led to the header sections being replaced by hand-generated `#ifdef` blocks, which have since been modified almost beyond recognition. The first test release of Tripwire allowed its compilation on BSD and System V implementations as interpreted by SunOS 4.1.1 and Solaris 1.0. As testers modified the header sections to allow its compilation on their platforms, its complexity grew rapidly. As a consequence of their changes to the header structure, another platform that previously had no problems compiling would often fail.

These *ad hoc* modifications of the header sections and repeated distribution and testing of merged changes eventually produced a reasonable framework. However, only the patience and persistence of our testers allowed this iterative method to succeed. The first several patches striving to add "portability" to the test version of Tripwire invariably would prevent correct compilation on many machines.

A header section extracted from a Tripwire source file is shown in figure 2.

3.1.3 Lessons learned

Tripwire now compiles "out of the box" on most platforms, and adding configurations for new platforms within this framework is relatively easy. However, the tedium required to build this framework is hard to justify when an existing tool could have automated this process.

The configuration framework in Tripwire remains versatile enough to allow the addition of system specific capabilities. Special file operations for Apollo Domain/OS and HP/UX CDF (Context Dependent Filesystem) have been added to Tripwire by users, and are now included in the Tripwire distribution without affecting users of older versions.

One clear choice we could have made was to have declared certain implementations as "unsupported" and thus ignored the portability questions. In one sense, this was done: to support some of the architectures on which Tripwire runs, we depend on our users to provide the necessary configuration information and specialized code. We have no access to the architectures involved to do the work ourselves.

3.2 You administer how many systems?

When Tripwire was distributed to beta testers in the September 1992, it did not have a built-in preprocessor. Because Tripwire was originally intended to monitor small numbers of files for changes, we did not envision any need for such a mechanism.

During the testing period and early deployment, numerous system administrators ran Tripwire on many machines at their site. Given large enough numbers of machines, the time required to configure Tripwire on all machines proved prohibitive. Exacerbating this is that their Tripwire configurations typically cover thousands of files, and that machine disk configurations are rarely uniform enough to support direct reuse.

Thus, a preprocessing language to allow Tripwire to share common configuration files among multiple machines was added. The impetus for this was when a system administrator wrote about

```
#include "../include/config.h"
#include <stdio.h>
#ifdef STDLIBH
#include <stdlib.h>
#include <unistd.h>
#endif
#include <fcntl.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/stat.h>
#ifndef NOGETTIMEOFDAY
# include <sys/time.h>
#else
# include <time.h>
#endif  /* NOGETTIMEOFDAY */
#ifdef DIRENT
# include <dirent.h>
#else
# ifndef XENIX
#  include <sys/dir.h>
# else         /* XENIX */
#  include <sys/ndir.h>
# endif        /* XENIX */
#endif  /* DIRENT */
#if (defined(SYSV) && (SYSV < 3))
# include <limits.h>
#endif  /* SVR2 */
#ifdef STRINGH
#include <string.h>
#else
#include <strings.h>
#endif
#include "../include/list.h"
#include "../include/tripwire.h"

#if defined(SYSV) && (SYSV < 4)
#ifndef HAVE_LSTAT
#  define lstat(x,y) stat(x,y)
#endif
#endif         /* SYSV */
```

Figure 2: Sample of header file declarations.

his ambitions for running Tripwire on the 700 Sun workstations at his site — a moumental task without sharing, requiring generation of 700 different configuration files! We had not envisioned an environment like this when we originally designed Tripwire.

3.2.1 Implementation

To allow scalable use of Tripwire at large sites (e.g., up to thousands of heterogeneous machines), support was added to allow the reuse of configuration files. This was implemented by adding an M4-like preprocessor [8] for the configuration files. Directives such as "`@@define`", "`@@ifdef`", "`@@ifhost`", and "`@@include`" were provided to allow multiple machines to use a configuration file.

We did not use the M4 processor itself, or the `cpp` compiler preprocessor. To do so would have been in contradiction to our earlier efforts to keep Tripwire free of external dependencies and thus, potential vulnerabilities. Furthermore, we could not be certain that these utilities would be present on each platform where Tripwire was to be run. Thus, the decision was made to build a small preprocessor into Tripwire itself.

3.2.2 Lessons learned

Because commands interpreted by the preprocessor reside in the space meant for filenames, a sentinel character must be used to denote the preprocessor directives. This effects how file names containing the sentinel character must be encoded by the user.

One of us (Gene), in his haste to complete the Tripwire prototype before the start of the Fall 1992 semester, added a literal implementation of the preprocessor functionality suggested in the e-mail sent to us. However, this suggestion, while containing the kernel of a good idea, was an especially poor choice of implementation. The most obvious miscue was the use of a two character sentinel sequence. This is semantically unnecessary — a single character would have been sufficient. However, for compatibility reasons, Tripwire continues to use this sentinel sequence.

During one of our meetings, one of us (Gene) was astonished to discover that the Tripwire parser was not implemented using `lex` or `yacc`. Instead, Gene had decided to hand-code a small parser; he believed that it would take too much effort to generate new files using additional tools. Furthermore, he did not believe that the parser was complex enough to require more than a simple hand-build routine.

However, events proved Gene's estimation to be incorrect. Several small bugs and special cases appeared that led to rewrites of the parser. After discussing the contents of future releases, and after a short discussion with Gene about using `lex` and `yacc`, a machine-generated parser was incorporated to allow these future additions.

3.3 Other surprises: orders of magnitude

3.3.1 Do "static file systems" really exist?

The original Tripwire design and implementation stressed the need to maximize the integrity of the baseline database. As the reports generated by Tripwire are only as secure and reliable as its input data, we stressed that the baseline database be immediately moved to some *hardware-enforced read-only* media to prevent tampering. Allowing the database to be updated in any automated manner seemed inherently unsafe: it could allow an intruder to modify the database and thus subvert the entire integrity checking scheme.

The first group of Tripwire testers perceived this inability to update the database as a serious shortcoming. Their filesystems would undergo small, but frequent changes. This would require them to reinitialize the entire Tripwire database after each change so as to get a clean report. This was tedious, at best. However, the perceived need for running an integrity checker on their systems outweighed their complaints.

The first documented complaint of the inability to make database updates was registered in September 1992, shortly after the beta test period began. We ignored it, as we assumed this was an exceptional case that we would not try to support. After we received a steady stream of similar complaints from testers, however, we realized that those static file systems that Tripwire was monitoring were rarely actually static: files seem to change for many unforeseen reasons.

Shortly before the official November 1992 release of Tripwire, we acknowledged that adding database update capabilities was a practical necessity. A command line interface was provided as a means to update database entries that changed. A more convenient mechanism for database updates was added in a test patch released in May 1993, and was not made official until the December 1993 release.

This capability to interactively make database updates was among the most well-received of enhancements we made to Tripwire. However, adding the change made us uncomfortable because it contradicts the precepts of baseline database security that we so stridently emphasized in our documentation. As such, it took almost one year for us to acquiesce and fully implement a convenient mechanism. That Tripwire is used for so many applications outside of intrusion detection is made possible, at least in part, by the ability to easily maintain consistency between the database and the file system.

3.3.2 You do it how often?

Tripwire is essentially a static audit tool because it detects file changes after the fact. It does this by comparing the file's state against information stored in the baseline database. In our design document, we recommended running Tripwire on a regular basis to support a timely notification of file system changes. We suggested "daily" as a reasonable interval.

One can run Tripwire more frequently and thus reduce the time before noteworthy changes are actually reported. Conceptually, if run frequently enough (e.g., every five seconds), Tripwire can function as though it were a real-time intrusion detection tool. However, this was not an application that we were addressing when writing Tripwire. It was certainly not a mode of normal operation that we thought reasonable.

However, during the testing process, several system administrators complained that Tripwire was running too slowly on their machines to allow file systems to be checked hourly. Because they opted to check numerous signatures for each file in the database, the Tripwire runs were not completed by the time the next Tripwire run started! This motivated the addition of a command line option to selectively skip signatures per invocation.

Here, our expectations of the frequency of Tripwire runs differed by an order of magnitude with those of some system administrator. However, by adding a command line option, we were able to accommodate and support their application of Tripwire.

It is possible that careful optimization of the Tripwire code in a future release would allow sessions to be run with full signature checking every hour. Our original development of Tripwire stressed correctness and portability more than speed, and there are many places where the code could reasonably be optimized.

3.3.3 How large is the database?

A request that the authors received early in the testing process was for database compression. Because we were confident that the database files generated by Tripwire would always remain relatively small, these requests were given little attention. At that time, we still assumed that Tripwire would typically be used to monitor several hundreds of files at most.

A year after official release of Tripwire in November 1992, we started to receive more mail requesting database compression. Users would write about how Tripwire would fill up the entire root partition on many of their machines. One of us (Gene) replied that this was probably a system configuration error on their part; it was inconceivable to us that Tripwire database files would ever grow so large as to impact disk usage. Gene then challenged the user to show how large how the databases were, claiming that databases should be "not that large; certainly no larger than a moderately-sized GIF file."

A refutation came in the form of directory listings of 15 database files, collectively taking up 45 MBytes of disk space. Apparently, many sites have huge archives of files on server machines on which they run Tripwire. Finally understanding this, we conceded that mechanisms for external programs providing services such as data compression and encryption would be provided in a future release. We also changed the database format to use a more compact representation of file signatures and inode values by using base-64 instead of hexadecimal.

Here, our expectations of database sizes were three orders of magnitude smaller than those found at some sites. It may be the case that users requesting data compression a year prior to this incident needed compressed databases, but they failed to provide either Gene with a clear understanding of that need.

3.4 Good surprises

> *The mark of a good tool is that it is used in ways that its author never thought of.*
>
> Brian Kernighan[4]

Potentially less exciting than the stories of intruders being caught, but equally inspiring, are the dozens of stories we have received of sites using Tripwire as a system administration tool. System administrators report having found hundreds of program binaries changed on their systems, only to discover that another person with system-level access had made the changes without following local notification policy.

[4] Brian Kernighan has said this, in one form or another, in several of his presentations and written works. This particular version was in private e-mail to one of us in response to a citation request.

There has also been one reported case of a system administrator detecting a failing disk with Tripwire. The normal system log reporting the failure was not read very often by the system administrator, but the Tripwire output was surveyed daily.

We used Tripwire to discover a bug in the `patch` program that causes context diffs to be applied incorrectly in a certain special case. This was discovered when the Tripwire test suite failed on a patched file. (See section 4.1.3.)

All these classes of stories further validate the theory behind integrity checking programs. Although the foundations of integrity checkers in UNIX security have been discussed in [3, 4, 7], when Tripwire design was started in late 1991, no usable, publically available integrity tools existed — providing one of the primary motivations for writing Tripwire.

Another application we note uses Tripwire to help salvage file systems not completely repaired by `fsck`, the program run at system startup that ensures consistency between file data and their inodes. In cases where inodes cannot be bound to a file name in a directory, they are placed in the `lost+found` directory and named some (less than useful) number. If a Tripwire database of file signatures is available, this file can be rebound to its original name by searching the database for a matching signature.

Many system administrators use `siggen`, a supplementary program included with Tripwire, as a convenient program for generating commonly used signatures. For example, recent security alerts and patches from response teams such as the CERT and CIAC often contain message digest signatures of patch files; `siggen` can be used to verify these signatures against the announcement.

Because providing a useful tool to system administrators was one of the primary goals of writing Tripwire, the variety of applications of Tripwire outside the domain of intrusion detection has been especially surprising and satisfying for us. We are still collecting other stories of novel use of the Tripwire package.

4 Patch, package, and release (repeat as necessary)

We have found that the process of building and releasing new versions of Tripwire is one of the most difficult tasks we have faced in the last three years.

4.1 Providing a rigorous test suite

To ensure the correct operation of Tripwire on platforms to which we do not have access, we include a test suite with the distribution. The test suite was originally one shell script that exercised all the signature functions. It would compare the signatures of all the files in the distribution against those included in a test database. As a side-effect, this would ensure the integrity of the files in the distribution.

As time has gone on, we have expanded the self-tests performed. Tripwire currently has a suite of seven shell scripts that also test for correct functioning of various Tripwire operations including database updating, reporting, preprocessing, and using alternate modes of input.

4.1.1 When tests know what files are supposed to look like

Generating signature functions that operated correctly across machines of differing architectures (i.e., big- vs. little-endian) proved exasperating; several of the releases bore patches to make the signature routines work on yet more architectures. In part, we can blame this on the fact that we incorporated existing versions of some of these signature routines as coded by other authors. This was done to avoid transcription errors. It is also traceable to the notion that it would be less error-prone than if we coded the routines ourselves. Unfortunately, we discovered too late that the original code was not written with portability in mind.

This rash of signature function errors motivated us to write a test script to ensure the correct generation of signatures of known files — namely, the Tripwire source files. This test evolved into a general Tripwire run, checking the source file signatures against those stored in an included test database.

This test script has been appreciated by our Tripwire users because it demonstrates Tripwire operation on a smaller (if contrived) scale. Users have commented that it is gratifying to see their modified Makefile and configuration files being reported as changed. However, having a test script that knows about all the files in the distribution has required a far more regimented and structured procedure for building Tripwire releases than we anticipated.

For instance, consider what happened when the first Tripwire patch distributed to testers changed many files. After testers applied the patch, they almost universally reran the test suite (an action that we did not anticipate). Not surprisingly, Tripwire reported all patched files as having changed. This was disconcerting to the testers, and they requested that all future patches install new database entries in the test script.

This new requirement of ensuring database consistency complicated the process of patch and release generation to such an extent that the next three patches released were incorrect in at least one way. Consider the process for putting together a patch release. First, because "checking in" a file under RCS control changes the contents of the file (the RcsId tag is incremented), all files must be first checked in. Next, as the Tripwire test database should be generated using the most recent version of Tripwire, a new Tripwire executable must be compiled. After compiling the sources, the resulting executable is used to generate a test database. Completing this, the patch is then generated (including a patch for the database file). Next, the distribution and patch is moved to a clean directory, compiled, and then tested by running the test script on a variety of platforms. If it completes correctly without any errors, the database is checked in and the entire patch is regenerated.

On a Sequent Symmetry (with 16 MHz 80386 processors), this entire patch build process takes almost thirty minutes. The high iteration time further frustrated a process that was difficult to do correctly.

4.1.2 It's hard to get right

The story of how the first several patches released during the test period is presented here to motivate our decision to fully automate the process of release generation.

Before any patch was released to testers, Gene would send the (hand assembled) completed patch to the other Gene, who would then test it on a number of different machines available to him. Although this first patch applied correctly, the package failed on a certain machine, necessitating a change to the Tripwire source. A new patch was assembled and again sent to Gene, who then reported that the patch failed to apply. Over the course of the next two days, Gene would send six more patches that would also fail for one reason or another, necessitating another build. This process was typical, and so prone to error that Gene insisted on sending out only those patches that he had personally verified.

However, mistakes still occurred. One time, some miscommunication resulted in Gene sending out a patch that was not "checked in" under RCS control. Because of this, subsequent patches until the next official release had to be hand-edited (correcting all the incorrect RcsId tags) so they would apply properly. To fully understand the scope of this problem, consider that the state of Tripwire sources held by users no longer mapped to any files in our source tree. Checking in the distributed files into our source tree would change their RcsId tags, making all future patches to these files fail to apply correctly.

Many of these problems occurred again when working on the second patch release. This motivated the writing of 440 lines of shell scripts in seven files to automate the generation of patches. As we write this, neither Gene knows how to manually generate a patch release without these scripts.

As each "last minute change" that we try to squeeze in right before a patch release necessitates a full rebuild, much time could be saved by using the Tripwire "database update" command to alter only the one changed entry. However, so convoluted is the entire procedure of generating distributions and patches that Gene has not risked breaking the scripts by touching them.

4.1.3 Other sources of problems

One patch release uncovered a bug in the `patch` program itself that would misapply the output of a contextual "diff". The problem was discovered when a user called one of us (Gene) on the telephone, alarmed that the Tripwire test script reported a corrupted header file. Inspecting the file revealed that a newline had been incorrectly inserted by `patch`.

A workaround was provided by hand-editing the patch file so that the output file would match the intended file.

4.2 Growing better test suites

Since the initial release of Tripwire, seven more test scripts have been added to the Tripwire distribution. The addition of these test scripts was motivated by the desire for canonical test cases in the Tripwire distribution for bug reporting purposes. While tracking a few persistent problems we realized that we needed some method to test all Tripwire features on each new architecture. The result was our test suite.

The entire test suite on a Sun workstation now takes as long as twenty minutes to execute. Yet, to the best of our knowledge, running the test suite is the first thing that most users do after building Tripwire. One of us (Gene) expected that users would find it too time-consuming and use Tripwire without running the test suite; the other Gene knew from experience that the actual behavior was the likely case.

The testing environment scripts for the functional testing scripts were originally written in 70 lines of Perl. To ensure that sites without Perl can run the Tripwire test suite, the test suite was rewritten as 160 lines of Bourne shell script with functions. When one of us (Gene) learned that many older machines do not support inline functions in Bourne shell script, it was rewritten as a 240 line "classic Bourne" shell script that ran four times again as slowly. Even with this change, some sites report system oddities that prevent the script from working (including malfunctioning `test` commands).

4.3 Good tools give great mileage

A number of software tools added to the quality of Tripwire code. We made extensive of several publically available tools in writing Tripwire. Those that we feel added significantly to the quality of Tripwire code are discussed below.

4.3.1 Pedantic compilers

A large portion of changes to the Tripwire code during the Tripwire testing period was from users compiling Tripwire on SGI workstations. More than any of the compilers used, those pedantically rejected any marginally unhealthy coding convention. The changes necessary to get Tripwire to compile on SGI machines would be invariably sent to us, where it would get merged into the Tripwire sources. Tripwire now "lints clean," thanks to an ambitious programmer in Australia who sent us a 70 Kbyte patch file in October 1992.

4.3.2 ANSI C prototypes

Using ANSI-style C prototypes saved the authors countless hours of debugging. By enforcing argument consistency in function calls, numerous errors that would have resulted in non-obvious program misoperation generated errors at compile-time.

However, unlike `lint`, ANSI compilers must have function prototypes defined to catch these argument calling errors. Therefore, prototypes to all functions in Tripwire are stored in one header file. Because of this, changing even one prototype in this header file has a side effect of rebuilding all the Tripwire object files because of our Makefile-driven builds. (Having multiple prototype header files was rejected by Gene as being too ugly.)

4.3.3 cproto

The easy generation of ANSI C prototypes is made possible by the `cproto` tool by Chin Huang[5] This program uses a partial C grammar to automatically generate correct ANSI prototypes. The resulting header files will be appropriately wrapped with macros so K&R C compilers will interpret them as conventional function declarations.

[5] Chin Huang can be reached at chin.huang@canrem.com. His `cproto` program can be found in volume 31 of the `comp.sources.misc` archive.

4.3.4 Source control

Another tool used extensively by Gene is the CVS (Concurrent Versions System) tool by Brian Berliner [1].[6] CVS is a front-end to the well-known RCS utilities [15], providing useful "release revision" abstractions beyond the file-by-file revisions afforded by RCS. Among other things, CVS allowed for the easy generation of patch files against any previously released version of Tripwire (modulo the contortions necessary for the self-test database).

4.3.5 Commercial tools

Tripwire development also benefitted from the use of two commercial products: CodeCenter by CenterLine Software and Purify by Pure Software. CodeCenter (formerly known as Saber-C) provides an interpreted C environment, where even the most seemingly benign errors (e.g., function argument misuse, potentially erroneous C statements, etc.) are reported at compile- and run-time. One of the most subtle bugs in Tripwire (the error condition described in section 5.2) was eventually fixed with the aid of CodeCenter.

Purify is a library that is linked at compile-time. When the program is run, it detects memory access errors and instances of memory leaks. It not only helped detect numerous errors before each of the later releases of Tripwire, it also allowed for the elimination of conditions where memory was not being reclaimed. (It also detected an error in the `libc` library being shipped with SunOS 4.1.3: calling `ctime()` results in a warning about writing to an illegal memory location.)

Using commercial tools involved more restrictions than the other tools we used, mostly because of licensing and platform problems. For instance, Gene did not have accounts on machines where these packages were installed. So, the other Gene would run the programs on his machine, and then send the output to Gene via e-mail. (Gene appreciates that he was not ever required to submit punch cards. He was, however, forced to read stack traces, without line numbers, generated by Purify.)

4.4 Finding veteran guinea pigs

The Tripwire development process has been greatly aided by the keen interest generated in the system administrator community. The 200+ beta testers who assisted the authors in the summer and fall of 1992 not only tested Tripwire, but suggested the addition of significant enhancements that undoubtedly have helped lead to its widespread use today.

Since the official release, Tripwire has benefitted from similar groups who tested patches for the five test releases. Generally, we have limited these groups to under fifty people, so as to reduce the number of repeat bug reports to a manageable level and to simplify the distribution process. Invariably, a number of bugs are found and a number of features are added at the last minute.

It is interesting to note that very few members involved in Tripwire testing opt to volunteer for the next callout. It is gratifying that there always seem to be an eager group of Tripwire users

[6] Brian Berliner can be reached at Brian.Berliner@Sun.COM. His CVS program can be obtained from the GNU archives via anonymous FTP at `prep.ai.mit.edu` in `./pub/gnu`.

happy to take part in the release testing given a moment's notice. At the same time, we wonder what has dissuaded the previous testers from volunteering again.

The authors feel very fortunate to have had such an eager and talented group of testers. There is no doubt in our minds that they have made a significant and substantial impact on the direction of Tripwire development. We have tried to credit all of them in the documentation shipped with Tripwire to recognize their contributions.

4.5 Receiving gifts from the code elves

Tripwire has benefited from generous and helpful code reviews provided by people who both Genes respect highly. Code changes received from these people include hash table speedups, base 64 routine speedups, and some inevitable comments on undesirable coding practices.

We have received two patches of almost 80 Kbytes in length that have added significantly to the Tripwire tool. The first was the previously mentioned patch that made Tripwire generate only minimal output when checked by `lint`. The second patch we received from another person in January 1994 added handling for symbolic links — a previously documented weakness of Tripwire. The patch also fixed a few lurking bugs, and offered useful comments on design issues untouched since the summer of 1992.

However, not all of the contributions we have received are so spectacular in their contents. One of us (Gene), having been made more wary of blindly implementing user suggestions, has patches that do not produce compilable files, produce clearly undesired side-effects, destroy data structure assertions, or are just wrong. We both usually write back, thanking the users for their time and consideration.

4.6 Lingua franca, s'il vous plait, bitte?

The test distributions of Tripwire included a tentative design document that one of us (Gene) developed before starting actual coding. This document served to inform testers of the design goals of Tripwire — issues that the incomplete Tripwire program conveyed far less directly. The official November 1992 release of Tripwire did not include this document, being unfit for widespread distribution or scrutiny, but promised that one would be "made available soon."

Hundreds of requests for this document were saved until November 1993 when the design document was finally completed. However, unlike the original document that was written using `troff` and the almost ubiquitous "ms" macros [14, 13], the new document was written using the LaTeX formatting system [12].

The fact that some users would be unable to generate printable text from a LaTeX source file made its inclusion in the Tripwire distribution problematical. We eventually included the design document formatted in PostScript. We resolved the problem for the lowest common denominator (i.e., those without PostScript printers) by making the document available as a technical report via surface mail.

The LaTeX source file without diagrams is 52 Kbytes. The generated PostScript file is 223 Kbytes, and adds to an already already large 7 Mbyte distribution.

5 What we still don't know how to do

5.1 Running Tripwire on the Sasquatch Kumquat Mark VIIa/MP

Tripwire has proven to be highly portable, successfully running on over 28 UNIX platforms. Among them are Sun, SGI, HP, Sequents, Pyramids, Crays, Apollos, NeXTs, BSDI, Lynix, 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.

However, potentially challenging situations result when we receive requests from system administrators asking for help compiling Tripwire on machines that neither of us have ever heard of. In one case, this was a machine only sold in Australia and shipped with incorrect system libraries. Other instances included an especially ignoble machine that has not been sold since 1986 (predating college for one of us), and numerous machines with non-standard compilers, libraries, system calls, and shells.

In all but two cases (of the last variety), we have incorporated changes in Tripwire sources to accommodate these machines. In most cases, there has been a sufficiently large group of system administrators with similarly orphaned machines who put together a suitable patch to allow correct Tripwire compilation and operation.

It is interesting to analyze the time needed to fully support a configuration. Full support for Sun's new Solaris operating system was added two months after the initial Tripwire release. A workaround for the two aforementioned Australian machines was released six months after the problems were first reported. However, some Tripwire users running machines from a large workstation vendor continue to be unable to find a compiler that correctly generates a Tripwire executable that passes the entire test suite; investigation has determined that this is because of non-standard and broken compilers and libraries on those platforms.

5.2 Invalidating all those old versions

We often get mail from users running old versions where the bugs they report have been fixed in a later release. In one case, this was a patch released in 1993 including a section of code that checks for a rare boundary condition. Having detected such a condition, Tripwire would print a warning banner:

```
added:   -rwxr-xr-x root        16384 Jul 23 13:44:55 1992 /usr/etc
### Why is this file also marked as DELETED?
### Please mail this output to (genek@cc.purdue.edu)!
```

Although an important bug was discovered through this invariant testing and a corrective patch distributed, the authors continued to receive daily reports of this bug two months after the fix was released. The Tripwire distribution available through our local FTP server includes the fixes for the problem, but older versions are also available at numerous mirror FTP sites around the world, as well as through any `comp.sources.unix` archive – information servers (e.g., `archie`) can lead users to out-of-date sources.

There is no mechanism for invaliding the older versions of Tripwire that still reside at these FTP servers. One year after the corrective patch, e-mail about this bug has finally abated. However, an e-mail dated February 17, 1994 is evidence that these older versions are still present on some FTP servers.

5.3 Maintaining a useful mailing list

To assist in the dissemination of information on Tripwire, a mailing list was established. Intended for the fielding of Tripwire questions, answers, and usage, the authors expected the mailing list to be very active, reflecting the traffic on certain USENET newsgroups such as `comp.unix.security` and `alt.security`.

However, throughout its lifetime, the mailing list remained mostly quiescent, disturbed only by misaddressed requests to subscribe to the mailing list. When bursts of correspondence did take place discussing the addition of certain features, a flurry of electronic mail from users clamoring to be dropped from the mailing list resulted. The electronic mail traffic asking to be dropped from the mailing list always dwarfed the amount of traffic devoted to Tripwire discussions.

As a result, we have discontinued the mailing list, although personal correspondence with regards to Tripwire to both Genes remain high. However, not all users use personal e-mail correspondence to communicate with the Tripwire authors. There are a surprisingly large number of people who attempt to contact us by posting a USENET article, sometimes to clearly inappropriate groups. This strikes us as the equivalent of walking into a grocery store and yelling, "Could I speak to the person who invented Charmin?" Nonetheless, we note that we have personally counted at least four instances of this in February 1994 alone, and that we have replied to those messages — can we not conclude their methods sometimes work?

5.4 Keeping track of all that documentation

With the exception of an interactive database update mode, the user interface to Tripwire is via the command-line. Ensuring consistency between available command-line options and all associated documentation seems straight-forward, but has been increasingly tedious as more options and features are added.

Tripwire is documented primarily through its manual page and its usage message. However, as new features are added, the manual page is usually the last to be updated. Recently, some new features have instead been documented in a `README` file, and even a `WHATSNEW` for those items that must stand out among those features described in the `README` file. (Considerable discipline was required to restrain from then creating multiple `WHATSNEW` files!)

Maintaining consistency between the program itself and its usage message requires considerable discipline, especially considering the lack of locality between the program code, usage message, and manual page. Several tools might help to automatically generate manual pages and usage messages from the program code, but the authors have not used them. Given the tedium of these issues, further Tripwire development would do well to investigate these tools.

One of us (Gene) recently reviewed the source code to check his commenting consistency. There were many instances of old comments lingering about, despite the pertinent code having been deleted. There are still several sections of code `#ifdef`ed out, but not deleted for fear of deleting something actually important.

5.5 Handling those bug reports

Associated with the problems of handling bug reports are their tracking. There have been a number of tools developed to track bugs. However, in the case of Tripwire, bugs have been tracked exclusively with mailers.

With very few exceptions, bugs are reported via e-mail, having been sent to either Gene or both Genes. Because of the informal nature of our bug tracking, there have been several bug reports lost or misplaced. Consequently, the bug reports that were sent to both Genes have had a better response rate. During our meetings, one of the topics always discussed was whether replies have been made to all people inquiring about certain Tripwire functionalities or bug reports.

5.6 Working with the other Gene

Many of the design misdecisions presented earlier came about because of the lack of contact between both Genes. The combination of Spafford's travel schedule and Kim's lack of diurnal habits made regular contact difficult. When Kim was avoiding Spafford for two months because of diminished progress with Tripwire, even this contact disappeared. Several times, Tripwire progress languished for almost an entire semester, but then experienced frantic progress in the final weeks (or sometimes, days). Not surprisingly, these conditions do not consistently produce good code or good decisions.

In many cases, the perception of the state of Tripwire held by each Gene would diverge, meeting only when enough misunderstandings prompted a long meeting to discuss the true state of Tripwire. This would often occur when both Genes would reply independently to an e-mail request and notice that their answers disagreed.

5.7 Moving the source base

Another consequence of working as an undergraduate student involved in internship programs was the lack of stable resources (e.g., workstation, disks). The Tripwire source tree has resided on six different machines at four institutions over three years. Although this has made the Tripwire distribution tools portable, considerable time and effort has been invested in portions of Tripwire that users will never see.

5.8 Purging American biases

We faced a surprising aspect of portability when we received a request from a user in Australia. He complained that Tripwire would always ignore his command line invocation: "`tripwire -initialise''`. After a day of frustrations, he finally sent one of us e-mail asking us to support non-American spellings of "initialize."

5.9 Other problems of scale

By most measures, the Tripwire project has surpassed our expectations. Among them is the amount of time. To date, almost three years have passed from the start of design study to product delivery and continuing support. Both Genes are relieved that this "one semester project" was not a graduation requirement for Gene Kim.

5.10 Other lowest common denominators

We continue to be amazed how low the "lowest common denominators" for UNIX systems are. Less charitably, but perhaps equally valid, is the same observation for users.

We have received a number of e-mail messages that have titles similar to "Tripewire setup allow read acess?" with messages that apparently contain a question, but no discernible way to glean what the question might be. We have received e-mail asking how to get Tripwire running on a machine, and the only data we are provided with is the name of their machine and that "Tripwire doesn't work." At least one inquiry was received totally in a foreign language not spoken by either Gene. Responses seeking clarification of these requests usually bounce.

Both Genes are very proficient in deleting e-mail.

6 Conclusion

In this paper we have described our initial design goals of Tripwire, what we ended up with, how we got there, and discussed some of the surprising issues that arose on the way.

Some of these challenges we handled well, and others not quite as well — which is probably the more interesting of the two. Among the factors to which we attribute these mishandlings are gross misestimates of scale (how Tripwire was used, how often, how large, etc.), miscommunication, inexperience, and semester deadlines.

However, in the past three years, we have substantially improved our ability to put out releases that are free of errors introduced by procedural mistakes and awkward compatibility issues. We have a rigorous test suite that exercises major functions and performs regression testing, and we have established a testing procedure that has allowed us to find most bugs before the majority of users even get access to Tripwire.

However, other issues remain that are quite separate from applications development that we would eventually like to address. Among them are supporting the most obscure (if not baroque) UNIX machines, controlling and invalidating copies in out-of-date FTP archives, and providing a good forum for discussion of Tripwire issues.

Perhaps the aspect of Tripwire development of which we are most proud is that the majority of Tripwire users remain completely unaware of the problems we have had to overcome. Since November 1992, we have been able to provide a tool for system administrators that is freely available, widely distributed, applicable, and well-supported, and that addresses a significant security need.

7 Availability

Tripwire source is available at no cost.[7] It is available via anonymous FTP from many sites; the master copy is at URL "`ftp://ftp.cs.purdue.edu/pub/spaf/COAST/Tripwire`". The source and several papers about Tripwire can be accessed via WorldWide Web at URL "`http://www.cs.purdue.edu/homes/spaf/coast.html`". Those without Internet access can obtain information on obtaining sources and patches via e-mail by mailing to `tripwire-request@cs.purdue.edu` with the single word "help" in the message body.

We regret that we do not have the resources available to make tapes or diskette versions of Tripwire for anyone other than COAST Project sponsors. Therefore, we ask that you not send us media for copies – it will not be returned.

References

[1] Brian Berliner. CVS II: Parallelizing software development. In *Proceedings of the Winter Conference*, Berkely, CA, 1990. Usenix Association.

[2] Matt Bishop, November 1993. personal communication (11/6/1993).

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

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

[5] Paul Dickson. *The New Official Rules.* Addison-Wesley, 1989.

[6] 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.

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

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

[9] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: A file system integrity checker. Technical Report CSD–TR–93–071, Purdue University, nov 1993.

[10] 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.

[11] 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.

[7] It is not "free" software, however. Tripwire and some of the signature routines bear copyright notices allowing free use for non-commercial purposes.

[12] Leslie Lamport. LaTeX*: User's Guide & Reference Manual.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[13] M. E. Lesk. *Using the -ms Macros with Troff and Nroff.* AT&T Bell Laboratories, 1976.

[14] Joseph F. Ossana. *Troff User's Manual.* AT&T Bell Laboratories, 1976.

[15] Walter F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering.* IEEE Press, September 1982.

[16] David Vincenzetti and Massimo Cotrozzi. ATP anti tampering program. In Edward DeHart, editor, *Proceedings of the Security IV Conference*, pages 79–90, Berkeley, CA, 1993. USENIX Association.