Analysis of LOKI2, Using mtree as a Forensic Tool, and Sharing Data with Law Enforcement

Andrew J. Korty

GCFA Practical Assignment Version 1.2, Parts 1, 2b, and 3

Table of Contents

Abstract

When doing forensics on a compromised system, one often encounters mysterious binaries, either installed as Trojans or hidden away in the filesystem. The first part of this paper will describe the analysis of one such unknown binary.

Next, this paper analyze the **mtree** tool for use in forensic investigation. With a few matches, **mtree** works quite well for gathering evidence or as a lightweight Tripwire, validating filesystems against known good specifications.

Legal issues are encountered in all aspects of forensic investigation. The final part of this paper discusses statutes and policies as they relate to information sharing with law enforcement agencies.

Part 1: Analyze an Unknown Binary

An unknown binary was obtained from a compromised system and transported to an analysis machine archived in a Zip file. Two analysis machines were used: a Pentium II system running Gentoo Linux 1.4. The system was disconnected from the network during testing and analysis to avoid corruption of hosts on the network by the potential malware in the Zip file. A Red Hat 5.2 system was also used in an attempt to compile an identical binary from the LOKI2 source.

Binary Details

It's important for the forensic investigator to try to find out as much as possible while changing as little as possible. So we start with the least invasive tools until we've exhausted what they can do. From there gradually move to more invasive tools. For example, we might start by looking at a file's meta-data before delving into the file data itself. The latter would change the file's access time, so it's important that we looked at the meta-data first to determine what that access time was before we got our hands on the file.

Analyzing the Zip File

Before unzipping the file, I used **zipinfo** to learn about what's inside. Without any command-line options, **zipinfo** displays a one-line summary of meta-information for each file in the archive.

Example 1. Displaying Archive Contents with zipinfo

```
$ zipinfo binary_v1.2.zip
Archive:  binary_v1.2.zip   7309 bytes   2 files
-rw-rw-rw-  2.0 fat       39 t- defN 22-Aug-02 14:58 atd.md5
-rw-rw-rw-  2.0 fat    15348 b- defN 22-Aug-02 14:57 atd
2 files, 15387 bytes uncompressed, 7115 bytes compressed:  53.8%
```

The fields are as follows: See the zipinfo(1) manual page for more information on these

1. File permissions in Unix format (see the ls(1) manual page)
2. Version of **zip** used to create the zip archive
3. Operating system on which the archive was created
4. Size of the file before compression
5. Two characters, the first of which indicates whether the file is presumed to be text (`t`) or binary (`b`) (the second character isn't used in our archive)
6. Compression method
7. Modification date of the file
8. Modification time of the file
9. Filename

The fields are as follows: See the zipinfo(1) manual page for more information on these fields.

So the zip file contains two files. The `atd` file's name, size, and the binary (`b`) flag in the fifth field indicate it's probably a compiled executable. In contrast, `atd.md5`'s name and extension, size, and text (`t`) status mean it probably contains an MD5 signature for `atd`. I can compute my own MD5 sum and compare the two to ensure I received `atd` intact.

Though **zipinfo**'s output lists the modification time, I chose to use its `-v` option, which displays a whole stanza of verbose details about each file. Particularly, it returns a more precise representation of each file's modification time.

Example 2. Displaying Archive Details with zipinfo

```
Central directory entry #2:
---------------------------

atd

offset of local header from start of archive:     75 (0000004Bh) bytes
file system or operating system of origin:        MS-DOS, OS/2 or NT FAT
version of encoding software:                     2.0
minimum file system compatibility required:       MS-DOS, OS/2 or NT FAT
minimum software version required to extract:     2.0
compression method:                               deflated
compression sub-type (deflation):                 normal
file security status:                             not encrypted
```

```
extended local header:                              no
file last modified on (DOS date/time):              2002 Aug 22 14:57:54
32-bit CRC value (hex):                             d0ee3072
compressed size:                                    7077 bytes
uncompressed size:                                  15348 bytes
length of filename:                                 3 characters
length of extra field:                              0 bytes
length of file comment:                             0 characters
disk number on which file begins:                   disk 1
apparent file type:                                 binary
non-MSDOS external file attributes:                 81B600 hex
MS-DOS file attributes (20 hex):                    arc

There is no file comment.
```

Analyzing the Binary's Meta-data

Having learned all I could from the zip archive, I unzipped it to inspect the files inside. Operating as the root user and using **unzip**'s `-X` option is supposed to preserve the user and group ownership of the original file. But **zipinfo** already told us the archive was made on a FAT filesystem, which doesn't support the notion of users and groups. So I wasn't surprised to see the files inherit the user (root) and primary group (root) of the user doing the unzipping, despite `-X`.

Example 3. Unzipping the Archive

```
# unzip -X binary_v1.2.zip
Archive:  ../binary_v1.2.zip
inflating: atd.md5
inflating: atd
# ls -l
total 28
-rw-rw-rw-   1 root     root        15348 Aug 22  2002 atd
-rw-rw-rw-   1 root     root           39 Aug 22  2002 atd.md5
-rw-r--r--   1 ajk      ajk          7309 Mar  1 17:57 binary_v1.2.zip
```

The **stat** command displays MACtimes much more precisely than **ls**.[1]

Example 4. Using stat to Determine Precise MACtimes

```
# stat atd
File: "atd"
Size: 15348        Filetype: Regular File
Mode: (0666/-rw-rw-rw-)         Uid: (    0/    root)  Gid: (    0/    root)
Device:  8,9   Inode: 2846360   Links: 1
```

[1] Some versions of **ls** can display MACtimes as precisely as **stat**. BSD's **ls** has a `-T` for this purpose.

```
Access: Thu Aug 22 14:57:54 2002(00191.03:08:10)
Modify: Thu Aug 22 14:57:54 2002(00191.03:08:10)
Change: Sat Mar  1 18:04:56 2003(00000.00:01:08)
```

Analyzing the Binary

Up to this point, I had retrieved as much information as I could without accessing the files in the archive themselves. To get more clues, I needed tools like **strings**, **grep**, and **md5sum** to examine the files.

Verifying the MD5 Digest of `atd`

When inspecting the contents of the zip archive before, I inferred based on its name and size that `atd.md5` contained an MD5 signature for the original `atd` file. The contents of the file appeared to be the standard format generated by **md5sum**.

Example 5. Contents of the `atd.md5` File

```
$ cat atd.md5
48e8e8ed3052cbf637e638fa82bdc566  atd
```

So I should have been able to generate my own digest from `atd` and compare the two. **md5sum**'s `-c` option automates this process. But when I ran the command, I got some strange error messages:

Example 6. Verifying against `atd.md5`

```
$ md5sum -c atd.md5
: No such file or directory
: FAILED open or read
md5sum: WARNING: 1 of 1 listed file could not be read
```

Not sure what to try next, I ran **md5sum** on `atd`, just to get a look at the digest.

Example 7. Generating an MD5 Digest of `atd`

```
$ md5sum atd

48e8e8ed3052cbf637e638fa82bdc566  atd
```

Those digests *looked* the same, but not trusting my ability to do an “optical grep”, I tried to confirm using **diff**.

Example 8. Using diff to Compare the Digests

```
$ md5sum atd | diff - atd.md5
1c1
< 48e8e8ed3052cbf637e638fa82bdc566  atd
---
> 48e8e8ed3052cbf637e638fa82bdc566  atd
```

diff seemed to think these two lines differed, but they appeared identical. What was going on? Whenever I see two strings that look the same but **diff** thinks are different, I suspect unprintable characters. The **od** tool, whose `-c` option displays unprintable characters as printable escape codes, can help.

Example 9. od Dump of `atd.md5`

```
$ od -c atd.md5
0000000   4   8   e   8   e   8   e   d   3   0   5   2   c   b   f   6
0000020   3   7   e   6   3   8   f   a   8   2   b   d   c   5   6   6
0000040           a   t   d  \r  \n
0000047
$ md5sum atd | od -c
0000000   4   8   e   8   e   8   e   d   3   0   5   2   c   b   f   6
0000020   3   7   e   6   3   8   f   a   8   2   b   d   c   5   6   6
0000040           a   t   d  \n
0000046
```

I should have known! The `atd.md5` file, which was generated on a DOS or Windows system, contains a carriage return not present in our **md5sum** output. DOS and Windows use a carriage return and a newline to delimit lines, whereas Unix just uses a newline. So our Unix-based **diff** program thinks the carriage return is just part of the data on the line.

So why was the output from **md5sum -c** so strange? It was probably trying to display the filename followed by a colon followed by an error message, but the carriage return was causing everything after the filename to be displayed starting at the beginning of the line, overwriting the filename. **od -c** can confirm this hypothesis.

Example 10. od Dump of md5sum's Output

```
$ md5sum -c atd.md5 | od -c
0000000   m   d   5   s   u   m   :       a   t   d  \r   :       N   o
0000020       s   u   c   h       f   i   l   e       o   r       d   i
0000040   r   e   c   t   o   r   y  \n   a   t   d  \r   :       F   A
0000060   I   L   E   D       o   p   e   n       o   r       r   e   a
0000100   d  \n   m   d   5   s   u   m   :       W   A   R   N   I   N
0000120   G   :       1       o   f       1       l   i   s   t   e   d
0000140       f   i   l   e       c   o   u   l   d       n   o   t
0000160   b   e       r   e   a   d  \n
0000170
```

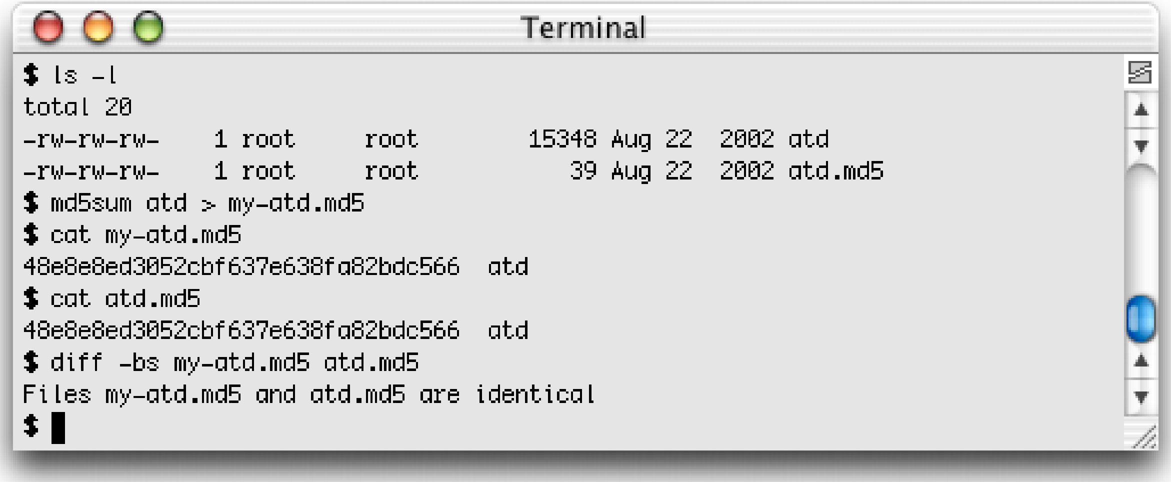
Sure enough, `a t d \r` was right there in the output. Naturally, **md5sum** couldn't find the file using this incorrect name.

So how do we show that the two digests are identical when our `.md5` file contains a bogus character? In this case, we can tell **diff** to ignore differences in the amount of whitespace using the `-b` option. Since the digest value never contains whitespace, using `-b` won't taint our analysis. The `-s` option prints a decisive statement on whether the files are identical.

```
$ md5sum atd | diff -bs - atd.md5
Files - and atd.md5 are identical
```

Courts and law enforcement agencies might prefer to have an screen shot to enter into evidence. In the Figure 1, I've tried to be as straightforward as possible so it's easy to understand what the commands mean.

Figure 1. Screen Shot of `atd` Digest Verification



```
$ ls -l
total 20
-rw-rw-rw-    1 root     root        15348 Aug 22  2002 atd
-rw-rw-rw-    1 root     root           39 Aug 22  2002 atd.md5
$ md5sum atd > my-atd.md5
$ cat my-atd.md5
48e8e8ed3052cbf637e638fa82bdc566  atd
$ cat atd.md5
48e8e8ed3052cbf637e638fa82bdc566  atd
$ diff -bs my-atd.md5 atd.md5
Files my-atd.md5 and atd.md5 are identical
$
```

Extracting Printable Strings from `atd`

My next step in examining the file data was to use the **strings** command on the `atd` binary. **strings** simply displays any printable strings embedded in a binary. Usually, **strings** will show us the text of error messages, prompts, title banners, command strings and other revealing information a program prints out or accepts as input when it's running. So **strings** is a great tool for learning about a binary without actually executing it.[2]

The string `lokid:` begins many lines of output. Since it's common practice for Unix commands to generate error messages prefixed with the program name, this string is my first clue as to the identity of the program. At this point I began to suspect it was a Loki daemon (**lokid**).

Example 11. Strings Beginning with `lokid:` in `atd` binary

```
$ strings atd | grep '^lokid: '

lokid: Client database full
lokid: inactive client <%d> expired from list [%d]
lokid: server is currently at capacity.  Try again later
lokid: Cannot add key
lokid: popen
lokid: client <%d> requested an all kill
lokid: clean exit (killed at client request)
lokid: cannot locate client entry in database
lokid: client <%d> freed from list [%d]
lokid: unsupported or unknown command string
lokid: client <%d> requested a protocol swap
lokid: transport protocol changed to %s
```

Further inspection of the `loki:` strings is consistent with my Loki hypothesis. All these strings appear to be messages or message fragments. The messages speak of clients and servers, and they hint that the server keeps some kind of database of connected clients. One message, `Cannot add key`, implies the use of cryptography.

The next clues I found in the unfiltered **strings** output appeared to be command-line option information, a command usage synopsis, and some title banner text. The first string, `v:p:`, looks like an argument to the POSIX `getopt` library function. This particular `getopt` string specifies that the program takes two options, `-v` and `-p` are permitted, and both take arguments. The next string is a usage synopsis, probably displayed when the user invokes the program with incorrect syntax. It confirms that two options are permitted. If this synopsis follows Unix conventions, the `-p` option is required and takes either the letter `i` or the letter `u` as an argument. The `-v` option is not required, as indicated by the surrounding square brackets, and takes either the number `0` or the number `1` as an argument.

[2] By default, **strings** only displays strings at least four characters long. To display shorter strings, specify the minimum length preceded by a hyphen, for example, **`strings -3`**.

But perhaps the biggest clue of all is the title banner. `LOKI2` indicates that perhaps this program is version 2 of LOKI. The copyright date may indicate the approximate year of the code's release. Finally, `guild corporation worldwide` could be organization that authored the code.

The only other interesting strings I found in the binary were what look like commands. The first one I found was **/quit all**. Many client-server protocols, including Internet Relay Chat (IRC), use the slash (/) character to begin commands. So I used **grep** again to find more occurrences.

Example 12. Looking for Commands in `atd`

```
$ strings atd | grep /

/lib/ld-linux.so.1
/dev/tty
/tmp
/quit all
/quit
/stat
/swapt
```

The first three strings look like system filename specifications. But the last four look more like commands. The **/quit** command appears to take an optional **`all`** argument. Two other possible commands are **/stat** and **/swapt**.

Results

Table 1 summarizes the results found in this part of the analysis.

Table 1. Binary Details

Program Filename	`atd`
Date and Time File was Last Modified (*`mtime`*)	22 August 2002 14:57:54
Date and Time File was Last Accessed (*`atime`*)	22 August 2002 14:57:54
Date Inode was Last Changed (*`ctime`*)	unknown (file came from a FAT filesystem)
File Owner (User and Group)	unknown (file came from a FAT filesystem)
File Size	15,348 bytes
MD5 Hash	48e8e8ed3052cbf637e638fa82bdc566

<table>
<tr><td>Key Words Found in Binary</td><td>loki:
client
database
server
command string
transport protocol
v:p:
lokid -p (i|u) [-v (0|1)]
LOKI2 route [(c) 1997 guild corpora
tion worldwide]
/quit
/stat
/swapt
IP_HDRINCL</td></tr>
</table>

Program Description

The data reported in the last section gives us many clues as to what the program does. It seems pretty obvious from the **strings** output that `atd` is actually a LOKI2 server daemon.

The concept of Loki was introduced in a 1996 issue of *Phrack* magazine. The idea is basic tunneling over ICMP. At that time, firewalls rarely blocked ICMP because programs that used it, such as **ping** and **traceroute**, were widely used for network testing and maintenance. Those protocols that aren't blocked by firewalls are obvious choices for tunneling, and thanks to Loki, ICMP was no exception [daemon9].

Loki uses the client/server model. Usually, a daemon program (**lokid**) would be installed on a compromised host, and a client would be used from a home base to interact with the system. Released a year later, LOKI2 implemented cryptography so attackers could keep their tunneled communications covert [route]. It also offers an option to disguise traffic as DNS transactions. Loki has been used not just for working with individual compromised hosts—it can also be used by denial of service tools such as Trinoo and Tribe Flood Network.

It's hard to say when this particular **lokid** binary was last used. Normally, the access time of a binary would give away when it was last used. But since the digest file has the same access time, we have to allow for the possibility that `atd`'s access time was changed by **md5sum**.

Though I already had a pretty good idea of what **lokid** does, I decided to run it in a controlled environment to learn more. It emitted a “Permission denied” error when run as a mortal user, which makes sense if it tries to open raw sockets for ICMP tunneling, which requires root privileges. After double-checking that the system was not connected to a network, I ran the program as root and got the “guide corporation” copyright message we found earlier. Immediately control was returned to my shell, so I assumed the process either forked or died. I found the former to be true when checking **ps**.

Example 13. Finding atd in the Process Table

```
# ps -x | grep atd
 2723 ?        S      0:00 ./atd
 2740 pts/1    S      0:00 grep atd
```

Once we know the process ID we can use **lsof** to show us what files and sockets it has open.

Example 14. Running lsof on the Process

```
# lsof -p 2723
COMMAND    PID      USER   FD   TYPE     DEVICE    SIZE       NODE NAME
atd        2723     root  cwd    DIR        8,9     8192    163521 /tmp
atd        2723     root  rtd    DIR        8,9     4096         2 /
atd        2723     root  txt    REG        8,9    15348   2846360 /home/ajk/gcf
a/atd
atd        2723     root  mem    REG        8,9    25034    425228 /lib/ld-linux
.so.1.9.5
atd        2723     root  mem    REG        8,9   699832    408885 /usr/i486-lin
ux-libc5/lib/libc.so.5.3.12
atd        2723     root    1u   CHR      136,1                  3 /dev/pts/1
atd        2723     root    2u   CHR      136,1                  3 /dev/pts/1
atd        2723     root    3u   raw                      61545300 00000000:0001
->00000000:0000 st=07
atd        2723     root    4u   raw                      61545301 00000000:00FF
->00000000:0000 st=07
```

In addition to several typical files we would exepct to be open, Example 14 shows two very interesting lines at the end of its output. Two raw sockets have been opened. **lsof**, in its own way, tells us these sockets are both bound to *`INADDR_ANY`* (any and all network interfaces on the system) and they are listening on IP protocols 1 and 255. *`INADDR_ANY`* is designated by the first `00000000` on each line, which represents a wildcard IP address. Following that is the IP protocol number in hex (`0001` and `00FF`). After the arrow is another wildcard IP address and a wildcard protocol number, which just mean the socket is listening.

To put this finding in perspective, let's see what **netstat** has to say. We know these are raw, listening sockets, so we'll use the `--raw` and `--listening` options to keep our output brief (see Example 15). **netstat** gives us one more piece of information, as it turns out: protocol 1 is ICMP. This socket and the raw IP socket (using the pseudo protocol 255) are used for Loki communications.

Example 15. Network Listeners Added by lokid

```
# netstat --listening --raw
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
raw        0      0 *:255                   *:*                     7
raw        0      0 *:icmp                  *:*                     7
```

Now, if we only had a LOKI2 client, we could try to connect to one of those sockets . . . .

Forensic Details

We see from the **lsof** output that **lokid** accesses relatively few files. This stands to reason as it was probably intended to be run without the system administrator's knowledge. The standard C library and dynamic linker are accessed, but that's not very useful or interesting since nearly every other binary on the system does too. Besides, a clever attacker could compile lokid statically and reduce those dependencies. It's marginally interesting that /tmp is open—**lokid** seems to use it as its working directory.

Obviously, the best way to detect a running **lokid** is by looking for the raw sockets. An extra ICMP listener and a protocol 255 listener are a dead giveaway. Plus, Netflow or **tcpdump** records will show the ICMP or DNS packets, which may be larger than legitimate packets. We also tend to see packets in pairs for those protocols, but with Loki, we're likely to see more traffic going from server to client than the other way around. So unpaired ICMP requests or replies are another clue. Most network intrusion detection systems should notice Loki as well.

Program Identification

By now we have more than a suspicion that this program is LOKI2. A Google search tells us this program was published in the September 1997 issue of Phrack. Note that the year 1997 corresponds to the copyright notice we found. Looking through the code, I found several exact matches to strings we found earlier. Also, I was able to learn the meanings of the options. For instance, -p chooses the transport: ICMP (*i*) or DNS/UDP (*u*). The default is ICMP, which is why our running **lokid** was listening for that protocol.

To extract the LOKI2 source code from Phrack, we must use one of the supplied programs. Both a C version and a Perl version are available. I used the Perl version, and found that I needed to change the **mkdir** call so that the permissions were passed as a number (*0777*) rather than a string (*"0777"*). Otherwise, the directories it creates end up with mode r----x--t. Consequently, the **extract** process can't write any files to those directories, and they end up empty.

To build LOKI2, I had to add a #include directive for linux/types.h at the top and re-move the one for `linux/signal.h`. I was able to compile both the client and the server this way, but the server binary did not match `atd`. Looking at the output of **strings --all** (which displays strings from all sections of the file, even those that aren't usually interesting), I found that this binary was compiled with GCC 2.7.2.1, which was released in 1996. To get an identical binary, I was going to have to use a compiler that was at least of the same vintage. Unfortunately, I was unable to get that version of GCC built, so I could not compare the binaries.

On the bright side, I now had a Loki client that I could use to connect to my running server. The client insisted I specify the destination and the protocol.

Example 16. Running the Loki Client

```
$ ./loki -d localhost -p i

LOKI2   route [(c) 1997 guild corporation worldwide]
loki> whoami
root
loki>
```

Scary!

Legal Implications

Because we were only given a Zip archive containing the unknown binary, and because it's probable the binary's access time was clobbered by the **md5sum** process before the archive was made, we can't prove whether the binary was executed. Whether or not the attacker executed the program, the Computer Fraud and Abuse Act [18 USC § 1030] could apply just because the attacker *accessed* the system. But this law does require that some sort of "damage" occurred to a "protected computer". *Damage* is defined in the statute as "any impairment to the integrity or availability of data, a program, a system, or information" [18 USC § 1030(e)(8)]. A *protected computer* is a government computer [18 USC § 1030(e)(2)(A)] or a computer used by a financial institution or for commerce, either domestically or abroad [18 USC § 1030(e)(2)(B)].

Fortunately for the victim, the term "damage" is defined quite broadly in the statute and can include "loss" of up to $5,000, corruption of medical records, bodily injury, threats to public safety, or harm to certain government systems [18 USC § 1030(a)(5)(B)(i)-(v)]. "Loss" can mean all the accumulated costs of incident response, including "the cost of responding to an offense, conducting a damage assessment, and restoring the data, program, system, or other information to its condition prior to the offense" [18 USC § 1030(e)(11)]. Anyone who has worked an incident knows such costs can accumulate quickly. "Loss" also includes lost revenue and cost of service interruption. So it doesn't take long to get up to $5,000.

Table 2. States of Mind for Network Crimes

State of Mind	Intent to Damage	Authorization	Victim Computer Accessed
Negligent	not required	outsider	required
Reckless	required	outsider	required
Intentional	required	insider/outsider	not required

If a protected computer is involved and the $5,000 minimum has been met, the resulting penalty depends on the attacker's state of mind as shown in Table 2. *Negligence* applies if there is no intent to damage and the victim computer is actually accessed. It applies only if the attacker did not have authorization to the victim's computer [18 USC § 1030(a)(5)(A)(iii)]. *Recklessness* also requires access to the victim's computer and only applies if the attacker did not have authorization, but any damage had to be caused intentionally. Finally, *intentional* conduct can apply with or without proper authorization and whether or not the victim's machine was accessed (as is the case with worms, viruses, or denial of service attacks), as long as the act was intentional. Table 3 lists the maximum penalties for each of these cases.

Table 3. Maximum Penalties for Network Crimes

State of Mind	First Offense	Repeat Offense
Negligent	fine + 1 year	fine + 10 years
Reckless	fine + 5 years	fine + 20 years
Intentional	fine + 10 years	fine + 20 years

Interestingly, the Computer Fraud and Abuse Act allows for criminal punishment [18 USC § 1030(c)] and civil relief [18 USC § 1030(g)]. It does not, however, allow for punitive damages, as damages are limited to economic damages only [18 USC § 1030(g)].

Interview Questions

Given the chance to interview the person who installed the mystery binary, I would start by trying to trick the suspect into admitting having used **lokid**. Then I would try to find out more about what actually happened on the system by using misdirection to challenge the suspect's intellect. In an attempt to show me up, hopefully the suspect would reveal the information we seek. The root kit question in particular might prompt the suspect to brag that he or she is too “l33t” to use scripts and root kits. Or the suspect might not be able to resist telling us that LOKI2 also supports DNS tunneling since we keep mentioning ICMP. The last question might lead to bragging about other compromised hosts on the Internet.

> **Where did you learn how to tunnel data over ICMP?**
>
> **Why did you choose such an outdated tool, considering ICMP is blocked most places and most intrusion detection systems can detect LOKI2 traffic?**
>
> **Since LOKI2 uses cryptography, you must have been handling sensitive data. *[chuckling]* Government secrets or something? Or do understand cryptography?**
>
> **Was LOKI2 just a part of a root kit that you installed when you gained access to the system?**
>
> **Since we've caught you on this host, does that pretty much shut down your entire operation?**

Additional Information

For additional information on LOKI2, see the original *Phrack* articles ([daemon] and [route]). The *Fromadia* article [Ka0ticSH] presents a detailed usage example of **loki** and **lokid**. See also the manual pages for strings(1).

Part 2: Perform Forensic Tool Validation

Scope

The **mtree** program lists files contained in a given hierarchy along with their associated properties (size, link count, checksum, *etc.*). The list is called a *specification* because it represents how a hierarchy is supposed to look for a user-defined set of properties. **mtree** can also compare a hierarchy against a specification it created earlier. The forensic investigator might find **mtree** useful in conjunction with data recovery tools because it can build customizable specifications of all the files on a seized computer. Its ability to compare to a specification from a known-good system could also make it a useful intrusion detection tool.

But **mtree** is not just a forensics tool. It can also create a new directory hierarchy with all the appropriate permissions and ownerships based on a given specification. This feature is useful for installing software distributions. In fact, some operating systems (for example, the BSDs) invoke **mtree** to build a skeleton before installing third party software and even the operating system itself. This approach ensures directories are created with the correct permissions and ownerships every time. This paper only covers the **mtree** features directly useful to forensic investigation, but the functions related to hierarchy creation are documented in the manual page.

Tool Description

mtree started life as a part of 4.3 BSD Unix. Today it comes bundled with FreeBSD, NetBSD, OpenBSD, and Mac OS X. Ports are available for Linux.

Specification Format

For its specifications, **mtree** uses a strict format that is both human- and machine-readable. Each file in the hierarchy is represented by a line in the specification file consisting of the file's name and its corresponding properties separated by whitespace. Long lines may be broken with a backslashes (see Example 18). Modern versions of **mtree** escape any whitespace in the filename to distinguish it from a field delimiter. When reading a specification, **mtree** processes it line by line, comparing each file in the hierarchy to its corresponding entry. When a directory appears in the specification, **mtree** descends into it and searches for subsequent entries there. A subsequent entry of `..` by itself on a line tells **mtree** to ascend back up one level.

Each file's properties are represented by certain *keywords* that **mtree** understands. A few of these keywords (those directly useful to the forensic investigator) are listed in Table 4. For the complete list, see the mtree(8) manual page.

Table 4. Some mtree Keywords

Keyword	Description
`cksum`	checksum of the file as returned by cksum(1)
`gid`	numeric id of the file's group
`gname`	symbolic name of the file's group
`link`	for symbolic links, the file referenced by the link
`md5digest`	the MD5 digest of the file's contents
`mode`	the file's permissions
`nlink`	the number of hard links to this file
`rmd160digest`	the RMD-160 digest of the file's contents
`sha1digest`	the SHA-1 digest of the file's contents
`size`	length of the file's contents in bytes
`time`	date and time the file was last modified

Keyword	Description
`type`	file type (`file`, `dir`, `link`, *etc.*)
`uid`	numeric user id of the file's owner
`uname`	symbolic user name of the file's owner

Note that several of these keywords can be used for file integrity checking. The `cksum` keyword uses the default cyclic redundancy check (CRC) algorithm used by the **cksum** command. There are also keywords for generating MD5, SHA-1, and RIPEMD-160 message digests of the files. These keywords give the forensic investigator several options for verifying the integrity of file data.

By default, **mtree** records `gid`, `link`, `mode`, `nlink`, `size`, `time`, `type`, and `uid` for every file.[3] The user can specify other keywords to use at runtime with the `-K` or `-k` option. Keywords can also be set and unset inside the specification itself in the form ***`keyword`*****`=`*****`value`***. One or more of these keyword/value pairs may appear, separated by whitespace, on lines beginning with ***`/set`*** and ***`/unset`***.

Invocation

When no command-line options are supplied, **mtree** reads a specification from the standard input and compares it with the hierarchy rooted in the current working directory. A number of options are supported to change the default behavior (all documented in mtree(8)). Here are a few that are useful when doing forensics:

`-c` Create a specification for the hierarchy rooted in the current working directory. The specification is printed on the standard output.

`-K` *`keywords`* Specify keywords to add to the default list. The *`keywords`* parameter should be a comma- or whitespace-delimited list of the keywords appearing in the next section.

`-k` *`keywords`* Replace the default list of keywords (except `type`, which is always recorded) with those specified.

`-p` *`path`* Specify an alternate root to the hierarchy instead of the current working directory. This option prevents you from having to **chdir** into the root directory of the hierarchy of interest.

`-s` *`seed`* Generate a single checksum for *all* the files in the hierarchy for which `cksum` was specified. It's best to choose a random, hard-to-guess value for *`seed`*, which is used to seed the checksum algorithm. The checksum is printed on the standard error and

[3] On BSD systems, **mtree** supports a `flags` keyword that corresponds to file system flags (for marking files immutable, append-only, *etc.*). On such systems, `flags` is also one of the default keywords.

should be saved for quick comparison after later **mtree** runs.

`-x` Stay local to the current filesystem--don't traverse mount points. This option is useful when creating specifications for individual filesystems to ensure no other filesystems are included.

A forensic investigator could use `-K` to add one or more of the cryptographic hash keywords and `-s` to generate a single checksum for the entire hierarchy. This technique is useful for comparing files from a possibly compromised system against files from a known-good system. It could also be used to create a file system specification of a seized computer for use as evidence. The checksum generated by `-s` can quickly show whether the filesystems differ, and the cryptographic digests indicate exactly which files have changed. The next section describes this technique in more detail.

Creating a Specification

In Example 17, we create a very simple hierarchy and then run **mtree** on it. The **mtree** command will create a specification of the hierarchy rooted in `/tmp/test`, enabling all the checksum and cryptographic digest keywords and printing on the standard error a final checksum of all the files (using the randomly chosen seed `3948572984`). The specification itself will end up in `/tmp/mtree.out`.

Example 17. Creating an mtree Specification

```
$ mkdir /tmp/test
$ cd /tmp/test
$ touch file1
$ touch file2
$ mtree -c -K cksum,md5digest,sha1digest,ripemd160digest -s 3948572984 > /tmp/mt
ree.out
mtree: /tmp/test checksum: 2147483648
```

Example 18. mtree Specification

```
$ cat /tmp/mtree.out
#          user: ajk ❶
#       machine: example.com
#          tree: /tmp/test
#          date: Sun Mar 23 21:49:51 2003

# . ❷
/set type=file uid=707 gid=0 mode=0644 nlink=1 flags=none ❸
```

```
.               type=dir mode=0755 nlink=2 size=512 time=1048474097.0 ❹
    file1       size=0 time=1048474088.0 cksum=4294967295 \ ❺
                md5digest=d41d8cd98f00b204e9800998ecf8427e \
                sha1digest=da39a3ee5e6b4b0d3255bfef95601890afd80709 \
                ripemd160digest=9c1185a5c5e9fc54612808977ee8f548b2258d31
    file2       size=0 time=1048474097.0 cksum=4294967295 \
                md5digest=d41d8cd98f00b204e9800998ecf8427e \
                sha1digest=da39a3ee5e6b4b0d3255bfef95601890afd80709 \
                ripemd160digest=9c1185a5c5e9fc54612808977ee8f548b2258d31
..
```

Example 18 shows the resulting specification. Note:

❶ the comment header, which contains information about this invocation
❷ the directory comment, which appears for every directory in the hierarchy
❸ the `/set` line, which specifies default keyword values to keep the specification from growing too large and uncluttered
❹ per-file keyword specifications, which can override the values previously defined by `/set`
❺ the backslash character (`\`), which is used to break long lines for readability

The resulting specification can be saved for later verification of the hierarchy, either as a normal auditing procedure or in the event of a possible compromise. It's best to digitally sign or encrypt the specification and final checksum and store them offline to thwart tampering.

Verifying a Hierarchy against an Existing Specification

With our saved specification and checksum in hand, we can verify our hierarchy at any time with a simple command. In Example 19, we use `-s` with the same seed as before.

Example 19. Verifying a Hierarchy

```
$ cd /tmp/test
$ mtree -s 3948572984 < ../mtree.out
mtree: /tmp/test checksum: 2147483648
```

Note how the checksum displayed on the standard output matches the original checksum. Example 20 shows what happens if we change the hierarchy by putting some text in one of the files.

Example 20. Verifying a Changed Hierarchy

```
$ cd /tmp/test
$ echo difference > file1
$ mtree -s 3948572984 < ../mtree.out
file1 changed
        size expected 0 found 11
        modification time expected Sun Mar 23 21:48:08 2003 found Wed Mar 26 20:
44:28 2003
        cksum expected 4294967295 found 1265948061
        MD5 expected d41d8cd98f00b204e9800998ecf8427e found ad517beba2f5d57a2a3a
7bd754b10492
        SHA-1 expected da39a3ee5e6b4b0d3255bfef95601890afd80709 found bb4f41c291
c169877307e316e2d77beeaf0d2f0c
        RIPEMD160 expected 9c1185a5c5e9fc54612808977ee8f548b2258d31 found 0a75df
f24727957785ac27431e01c903a6c87345
mtree: /tmp/test checksum: 3034195045
```

mtree tells us right away that `file1` has changed and lists the old and new values for each keyword in the specification. Finally, the overall checksum does not match the original.

System Files and Resources Used by mtree

Compiling mtree Statically

These runtime library dependencies can be eliminated by compiling statically. When compiling and linking with **gcc**, we use the `-static` option.

To compile a static **mtree** from the BSD source tree, we can pass `-static` to **gcc** using the `LDFLAGS` variable.

Example 21. Compiling mtree Statically from the BSD Sources

```
# cd /usr/src/usr.sbin/mtree
# make LDFLAGS=-static
cc -O -pipe  -DMD5 -DSHA1 -DRMD160    -c /usr/src/usr.sbin/mtree/compare.c
cc -O -pipe  -DMD5 -DSHA1 -DRMD160    -c /usr/src/usr.sbin/mtree/../../usr.bin/c
ksum/crc.c
cc -O -pipe  -DMD5 -DSHA1 -DRMD160    -c /usr/src/usr.sbin/mtree/create.c
cc -O -pipe  -DMD5 -DSHA1 -DRMD160    -c /usr/src/usr.sbin/mtree/excludes.c
cc -O -pipe  -DMD5 -DSHA1 -DRMD160    -c /usr/src/usr.sbin/mtree/misc.c
cc -O -pipe  -DMD5 -DSHA1 -DRMD160    -c /usr/src/usr.sbin/mtree/mtree.c
cc -O -pipe  -DMD5 -DSHA1 -DRMD160    -c /usr/src/usr.sbin/mtree/spec.c
cc -O -pipe  -DMD5 -DSHA1 -DRMD160    -c /usr/src/usr.sbin/mtree/verify.c
cc -O -pipe  -DMD5 -DSHA1 -DRMD160    -static -o mtree compare.o crc.o create.o
excludes.o misc.o mtree.o spec.o verify.o  -lmd
gzip -cn /usr/src/usr.sbin/mtree/mtree.8 > mtree.8.gz
# ldd /usr/obj/usr/src/usr.sbin/mtree
ldd: /usr/obj/usr/src/usr.sbin/mtree: not a dynamic executable
```

Test Apparatus

Before using **mtree** as a forensic tool, we should test its effectiveness, both at detecting changes to a hierarchy and at reporting those changes, or the lack of changes. This technique will most likely be used on a Unix or Unix-like filesystem containing system binaries, libraries, and data files, so the filesystems in our test environment should mimic `/`, `/usr`, `/var`, *etc.*, of a real system.

When verifying a hierarchy that has not changed, **mtree** should indicate that nothing has changed. If the hierarchy *has* changed, such as after a root kit has been installed, **mtree** should make it clear to the user exactly what has changed and nothing more. Therefore, we must first create an **mtree** specification of an untouched live filesystem. Then we run **mtree** again to compare the original filesystem to the specification just made to ensure no changes are reported. Next we make a copy of the live filesystem image, mount it, chroot(8) into it, and install a publicly available root kit. Finally, we run **mtree** on the compromised filesystem to compare it to the original specification, and confirm that it reported only those changes made by installing the root kit.

I chose FreeBSD as the operating system in my test environment for two reasons. First, the BSD versions of **mtree** are the most full featured, supporting all the keywords in Table 4. Halfway through testing, however, I found two major bugs in FreeBSD's **mtree** implementation. One of the bugs causes `-s` to report an incorrect checksum when verifying a hierarchy. I developed a patch, which appears in Appendix A, *mtree Patches*, to fix this problem, which seems to exist in every version of **mtree** I've used (Linux, NetBSD, Mac OS X, *etc.*). I will submit this patch to all the appropriate distributions. The other bug has been fixed in NetBSD at least, and there is a fix pending in FreeBSD's problem report database. The bug is excited by filenames containing a hash mark (#), which looks to **mtree** like a comment character. The fix, submitted by Andrew L. Neporada, is to escape hash marks when creating the specification. That patch also appears in Appendix A, *mtree Patches*.

Also, FreeBSD's live filesystem images (distributed with every release) make good test hierarchies because they mimic real world systems. The vn(4) disk driver, configurable using the **vnconfig** command, can map a backing file to a device. Using **vnconfig -s labels**, the device can be configured to use disk labels just like a real disk. Then a label can be applied with **disklabel** and a filesystem created with **newfs**. See vnconfig(8) for more information.

The root kit, which doesn't seem to have a name other than “FreeBSD Rootkit”, I found on Packet Storm. According to the documentation, the root kit worked on FreeBSD 2.2.5, and porting the root kit forward to a recent version of FreeBSD (like 4.6), seemed daunting. The kernel has changed significantly since 2.2.5, and I didn't see the need to further the usefulness of this piece of malware. So I found a 2.2.6 live filesystem CD and used that--the root kit compiled and installed flawlessly.

Environmental Conditions

The tests themselves will be conducted on a machine that is isolated from any network. All the actual testing will be done using filesystem images which can be mounted with restrictive options. Whenever possible, filesystems will be mounted read-only. When they must be writable, some combination of the *nodev*, *noexec*, and *nosuid* options will be used.

All code retrieved from the Internet will be downloaded first on a different machine, inspected, and then transferred to the test machine. Of course, it's difficult to be sure no Trojan code has sneaked into our test apparatus, but the fact that we'll only be executing shell commands and not binaries mitigates that threat somewhat. Most of the known Trojaned distributions released of late would have been fairly easy to detect just by inspecting makefiles and configure scripts.

Description of the Procedures

I used the following procedure to validate **mtree** as a forensic tool:

Procedure 1. Validation of mtree as a Forensic Tool

1. Obtain a copy of the live filesystem CD or ISO image circa 2.2.5. The CD I used was from the 2.2.6 release. For images, use the vn(4) interface to mount the image. Throughout this procedure, we use the *nodev*, *nosuid*, *noexec* mount options wherever possible to prevent devices, setuid executables, or any executables whatsoever from being used. Using these options is a must when working with root kits, but we're also mounting images of a very outdated operating system that may have vulnerabilities.

Example 22. Mounting the ISO Image

```
# chmod 0700 /mnt
# umask 077
# vnconfig -c -s labels vn0 /gcfa/image/2.2.6-RELEASE-live
# disklabel -r -w vn0 auto
# mkdir /mnt/iso
# mount -r -t cd9660 -o nodev,noexec /dev/vn0c /mnt/iso
```

2. Create a UFS filesystem image to represent a clean, uncompromised system. We start by using **dd** to create a zero-filled image from `/dev/zero`--700 megabytes should be enough to hold the contents of the CD. Then we use **vnconfig** to attach the image to `/dev/vn1` as if it were a disk with a label. We auto-configure the label and create a filesystem on the image with **newfs**. Finally, we mount the image using the *async* and *noatime* options for speed.

Example 23. Creating a UFS Filesystem Image

```
# dd if=/dev/zero of=/gcfa/image/original bs=1m count=700
700+0 records in
700+0 records out
734003200 bytes transferred in 33.300977 secs (22041491 bytes/sec)
# vnconfig -c -s labels vn1 /gcfa/image/original
# disklabel -r -w vn1 auto
# newfs vn1c
Warning: Block size and bytes per inode restrict cylinders per group to 89.
/dev/vn1c:      1433600 sectors in 350 cylinders of 1 tracks, 4096 sectors
        700.0MB in 4 cyl groups (89 c/g, 178.00MB/g, 22144 i/g)
super-block backups (for fsck -b #) at:
 32, 364576, 729120, 1093664
# mkdir /mnt/original
# mount -o async,noatime,nodev,noexec /dev/vn1c /mnt/original
```

3. Copy the contents of the live filesystem CD to the blank **UFS** filesystem we just created. I chose **pax**, a modern, POSIX-compliant archiving tool, for this purpose. **pax**'s `-r` and `-w` options, when used together, copy a hierarchy from one location to another, similar to using **tar -c** and **tar -x** in a pipeline. We can also tell **pax** to preserve all file attributes (permissions, ownerships, *etc.*) with `-p` *`e`*. Immediately after the copy, we unmount the UFS filesystem to avoid further changes.

Example 24. Copying the Live Filesystem to the UFS Filesystem

```
# cd /mnt/iso
# pax -r -w -p e . /mnt/original
# umount /mnt/original
```

4. Make a copy of the UFS filesystem to compromise with the root kit. All we have to do is copy the image and associate another vn device with it. Then we mount the filesystem on `/mnt/compromised/`.

Example 25. Creating a Filesystem to Compromise

```
# cp /gcfa/image/original /gcfa/image/compromised
# vnconfig vn2 /gcfa/image/compromised
# mkdir /mnt/compromised
# mount -o nosuid /dev/vn2c /mnt/compromised
```

5. Retrieve the FreeBSD root kit. It's available from Packet Storm, but two header files are missing. I found those files in other distributions of the same root kit on the Internet. Before installing a root kit or any malware, be sure to use a dedicated computer not connected to any network, and inspected the build scripts to ensure no malicious code will compromise the test environment.

   When developing this step of the procedure, I had several false starts building the root kit. I would prepare the root kit to the point I thought it would work, transfer it to the chrooted environment on the test machine, and try to build it. When it failed, either because of missing files or bad permissions, I would revert to the copy of the retrieval machine, make the necessary changes, and repeat the process. In this procedure, I list the steps as if it were known what preparations had to be made.

   a. Retrieve the Packet Storm version of the root kit. The files on Packet Storm are presented not as one archive file but as a hierarchy for browsing. The **wget --recursive** command can be used to retrieve such a hierarchy. For this particular hierarchy, the `--no-parent` option is necessary to keep **wget** constrained to the depth of the hierarchy specified. Without it, links back to the Packet Storm home page would cause the recursion to bleed back to the very top of the hierarchy and eventually harvest the entire site. The `--reject` *`html`* option strips out all the `index.html` files that are used on the web site but aren't necessary for installation of the root kit. The **wget** command creates a deep hierarchy--one level for every subdirectory in the URL. The **mv** command can be used to flatten the hierarchy for convenience.

      It's best to retrieve the root kit files on a separate machine rather than directly into the test environment. The machine used for testing should be disconnected from the network, and the root kit can be transferred to it by floppy disk or CD-ROM. In the following examples, we use the same convention for naming files and directories on the retrieval machine as on the test machine just to indicate that the root kits should end up in the `/gcfa/rk/` directory on the test machine.

      ## Example 26. Harvesting the Root Kit from Packet Storm

      ```
      $ mkdir /gcfa/rk
      $ cd /gcfa/rk
      $ wget --recursive --reject html --no-parent http://packetstormsecurity.nl/mag/crh/fr
      sd/rootkit/
      $ mv packetstormsecurity.nl/mag/crh/freebsd/rootkit/ packetstorm
      $ rm -r packetstormsecurity.nl
      ```

   b. Inspect the distribution for malicious code triggered by building the root kit. Read configure scripts and makefiles line by line, looking for code that would attempt to contact remote hosts, create cron jobs or system startup scripts, *etc.*

c. Fix permissions. Some of the scripts in the root kit, such as `configure` and `install.sh`, are meant to be executable, but **wget** sets permissions the same on all the files it downloads. After several false starts of building the root kit, I knew which files needed to be made executable. This **find** command does it all at once:

Example 27. Fixing Permissions in the Root Kit Distribution

```
$ find packetstorm \( -name configure -o -name config.sub -o -name config.guess -o -
install.sh \) -print | xargs chmod u+x
```

d. Supply missing files. When trying to build the root kit, error messages indicated two header files that were missing, `rootkitnetstat.h` and `rootkitsyslogd.h`. There are several other distributions of this root kit on the Internet, though the ones I found appeared to be earlier versions than on Packet Storm. Still, they contained the missing header files.

Example 28. Copying Missing Files from an Older Version of the Root Kit

```
$ wget http://www.atomicfrog.com/archives/exploits/rootkits/fbsd.rootkit.1.2.tar.gz
$ pax -rzf fbsd.rootkit.1.2.tar.gz
$ cp fbsdrootkit-1.2/rootkitnetstat.h packetstorm/netstat
$ cp fbsdrootkit-1.2/rootkitsyslogd.h packetstorm/syslogd
```

6. Install the root kit. We first copy it on to the test machine in `/gcfa/rk/` just as on the retrieval system. Then we copy it into place on `/mnt/compromised/`, **chroot**, build, and install. The **chroot** step ensures that all the malware is confined to our test image and is not installed anywhere else on the machine. Finally, we delete the root kit source as an attacker would, and we even use the `-P` option to overwrite the file data with patterns. After exiting the **chroot** environment, we remount the filesystem read-only so no other changes can take place.

Example 29. Installing the Root Kit

```
# cp -rp /gcfa/rk /mnt/compromised/tmp
# chroot /mnt/compromised /bin/csh
# cd /tmp/rk/packetstorm
# make
# make install
# cd ../..
# rm -Pfr rk
# exit
# mount -u -o ro,nodev,noexec /mnt/compromised
```

7. Create a specification for the original, clean filesystem. First, we remount the original image in read-only mode. Then we create a session directory under `/gcfa/mtree/` to store our command line (`cmd`), the resulting checksum from the **mtree** run (`cksum`), and the specification itself (`spec`). We use a shell variable, `$run1`, to store the current date and time, which we will use to name the subdirectory for this run. We also use `$seed` to store the argument to `-s` that we will use for this run and the verify run. We choose a random seed using our **rand_uint** script.

Example 30. Creating the Specification

```
# mount -r -o nodev,noexec /dev/vn1c /mnt/original
# run1=/gcfa/mtree/`date +%Y-%m-%d-%T`
# mkdir -p $run1
# cd $run1
# seed=`rand_uint`
# echo mtree -c -K cksum,md5digest,sha1digest,ripemd160digest -p /mnt/original -s $seed >
d
# sh cmd 1>spec 2>cksum
```

8. Verify the original filesystem. We create another session directory for the results. We use the same seed as last time, but we eliminate the `-c` and `-K` options from the **mtree** command line since we're verifying based on the specification we created (`$run1/spec`).

Example 31. Verifying the Original Filesystem

```
# run2=`date +%Y-%m-%d-%T`
# mkdir $run2
# cd $run2
# echo mtree -p /mnt/original -s $seed > cmd
# sh cmd <$run1/spec 1>report 2>cksum
```

9. Verify the compromised filesystem. Again we create a new session directory and use `$seed` to refer to the original value.

Example 32. Verifying the Compromised Filesystem

```
# run3=`date +%Y-%m-%d-%T`
# mkdir $run3
# cd $run3
# echo mtree -p /mnt/compromised -s $seed > cmd
# sh cmd <$run1/spec 1>report 2>cksum
```

10. When all testing is completed, unmount the filesystems and delete the images.

```
# umount /mnt/compromised
# umount /mnt/original
# umount /mnt/iso
# rm -P /gcfa/image/compromised /gcfa/image/original
```

Criteria for Approval

Since nothing should have changed under `/mnt/original/`, the report from Step 8 should be blank, and the checksum should match that of the previous run, in Step 7.

In Step 9, we verified the filesystem we compromised against the original specification. In that step **mtree** should report a detailed listing of how the root kit changed the compromised filesystem, and the checksums should not match. For all the installed root kit binaries, the report should show differing checksum and digest values. Their sizes and modification times will not differ because of precautions taken by the root kit. It does not, however, reset modification times on the directories where the Trojan binaries are installed, so the report should show those discrepancies. It should also show differing modification times for the directory where built the root kit, `/tmp/`.

Data and Results

As expected, the checksum generated in Step 8 matches that of Step 7, and the report is blank:

Example 33. Results from Step 8

```
# cat $run2/report
# cat $run1/cksum
mtree: /mnt/original checksum: 2093295857
# cat $run2/cksum
```

```
mtree: /mnt/original checksum: 2093295857
```

As for Step 9, the checksums do not match, as we expected.

Example 34. Checksum Results from Step 9

```
# cat $run1/cksum
mtree: /mnt/original checksum: 2093295857
# cat $run3/cksum
mtree: /mnt/original checksum: 2542779770
```

The report also matches our expectations:

Example 35. Report from Step 9

```
# cat $run3/report
bin changed ❶
        modification time expected Tue Mar 24 12:51:05 1998 found Thu Apr  3 16:
45:51 2003
bin/ls changed ❷
        cksum expected 2268745294 found 1141276009
        MD5 expected cafc3c8b6dde53014bf92242495fe8ec found e36db0e8557eb315c207
9c03bcfb3c71
        SHA-1 expected 6f6dba1cf44eeb90e897339e2a0d14c33b37859d found b19b9948ee
0abe1b35fe8bd8ec9821a31e184292
        RIPEMD160 expected 3cc150b867ff92bc8c4609c683fecb728b6a791f found c4a8a4
c90b76760cdf0e086d93c9c5bc6d763bf9
bin/ps changed
        cksum expected 310943921 found 3675457541
        MD5 expected 3e7fe9075da05e7d9f12673b079813d9 found 66e3dea7659f7c8319bf
be767ecd6e21
        SHA-1 expected 84d9b94f593233311d6a272e4b5adf80282b222b found 462f87490b
be41f5f16dac3196b11af0377940b8
        RIPEMD160 expected 74aa4846e7442fd8b8ac2c194bbcb3c149860bfe found 3e6920
c29165e6b55fd3b50e46abdfe3d18208a3
sbin changed
        modification time expected Tue Mar 24 12:53:17 1998 found Thu Apr  3 16:
45:51 2003
sbin/ifconfig changed
        cksum expected 2884771185 found 731393516
        MD5 expected 6acffdf9d1047b11cf3a609f98bbf496 found cf6bfcbe5635415634ff
a235a9bb4c48
        SHA-1 expected 23cd2e66e43e1ad22488a6d3bc2e81c3789ddf21 found e1c6248321
5a2a4153e7e36565996ee644d025b3
        RIPEMD160 expected bc781413b66853e52b2ce45a71efa6e400310d4b found eb59c6
6c7b992ca80eee7b709f3a58e894196d4f
tmp changed ❸
        modification time expected Tue Mar 24 12:49:00 1998 found Thu Apr  3 16:
```

```
46:44 2003
usr/bin changed
        modification time expected Tue Mar 24 12:56:41 1998 found Thu Apr  3 16:
45:51 2003
usr/bin/login changed
        cksum expected 207547773 found 408507498
        MD5 expected e2503f49e49f42a4f461d1a6cc25f28c found 308df8f1596fbc48c2d9
9384a4d166a0
        SHA-1 expected a4e532ce97c94110e6fa165eea4813cf973e27ff found faa48092ac
f57c4835f238adb950232ff595ea2d
        RIPEMD160 expected ab489bc8ff8d71c36faec09e0661f9c8a2fed7c1 found 99a2a1
6fff01bc7bbeccb92d44ff0ac0993ab24f
usr/bin/netstat changed
        cksum expected 1552604713 found 3593833430
        MD5 expected b2a050ba96154f61ad525d666d17365c found 75363381eed15ee80555
f9a953ff3bea
        SHA-1 expected 77f93f0376600eb413472b8d762ba349cc8398e4 found 3ccd3fea8a
d3f0863a7d0c62fd68f68b8e6aa65f
        RIPEMD160 expected 850c6de27135911c33715fdf32cfe19c85a90acb found 7015b9
d4586bea3a69f2dae348e93c0e0b3dc2eb
usr/libexec changed
        modification time expected Tue Mar 24 13:46:21 1998 found Thu Apr  3 16:
45:51 2003
usr/libexec/rshd changed
        cksum expected 2716520514 found 2582926996
        MD5 expected 3e8cda66b906588906e714248e2ccddd found f12c3da1f35bb7749f27
12402657b8e0
        SHA-1 expected e59626cf4e3e2fa93af1ea5a6336e255f0dd7bcd found a39a1a1dc2
99666722787a435ee6b03bc2034f0a
        RIPEMD160 expected 9168b8f787de6f807e5b70255acc2aa0ad3d8f53 found 796842
68dd9f9855826b9154d9dee92657d822c9
usr/sbin changed
        modification time expected Tue Mar 24 13:02:38 1998 found Thu Apr  3 16:
45:51 2003
usr/sbin/inetd changed
        cksum expected 257727007 found 2380850430
        MD5 expected 043d0cced32d6cb081b2de0aaa0ee1a4 found a65e90393b48028b5194
cc7882f50679
        SHA-1 expected 8a8172691823c2314509e40eba7eed469cefb143 found 9eb8ec0268
5728a5a9a8acae7cc3c50dfabb506e
        RIPEMD160 expected 913364e5393ec92b422de11c008882559f710fc5 found 0a6c18
643ca436f9b93dc29b56e7dfa8ef53dfaa
usr/sbin/syslogd changed
        cksum expected 2882548084 found 1156011432
        MD5 expected d7b95f5d68d81e5f84c2116fa01b8faa found 474a10f313be2a722cba
26ba6ac347eb
        SHA-1 expected fd23759c2c96b0ee59496001ac546d51e5afa0e8 found 291d905460
da5e51ad7c40a15496895b8306c5a8
        RIPEMD160 expected fa114f2a1b5a4dbd82e2cc48dbb83f249989a5a6 found e43ddc
efe9b0d4b9c91dea72b19acafb7ec8e3c4
var/tmp changed ❹
        modification time expected Tue Mar 24 12:49:04 1998 found Thu Apr  3 16:
45:44 2003
```

Just a few notes on the report:

❶ The modification time on the `bin` subdirectory has changed. This was caused by files being installed into `bin`. Though the root kit changed the modification times of the Trojans themselves to match those of the good binaries, it did not take such pains with the destination directories.

❷ **ls** was one of the binaries Trojaned. **mtree** reports differing checksums and cryptographic digests. It does not report differing sizes or modification times because the root kit adjusted these to match the good binary.

❸ The modification time of `/tmp/` changed because that was our working directory for building the root kit.

❹ It is unclear why the modification time on `/var/tmp/` changed, but it was probably caused by temporary files created by the compiler when building the root kit. The compiler automatically removes its temporary files when they're no longer needed, which is why the files themselves didn't show up in the report.

Analysis

A forensic analyst poring over the data found in the last would no doubt reach the correct conclusions. Looking at the checksum and report from Step 8, an analyst would conclude that the filesystem had not changed. Of course, it's unlikely there would be *zero* changes on a production system unless very little time had elapsed between creation of the specification and verification against it. An appropriate conclusion on such a system might be that the specification, report, or checksums had been tampered with or that the **mtree** binary used in either part of the process had been Trojaned. But when verifying a filesystem that is supposed to have remained static, such as one being kept in a read-only state as evidence, the investigator would conclude that the data is still safe.

The checksum generated in Step 9 would probably be ignored by the investigator on a production filesystem that is almost sure to change. The investigator is likely to go straight to the report in that case. But for a filesystem in evidence, this differing checksum is a huge red flag to the investigator. In either case, the report speaks volumes. The particular binaries changed are very common root kit Trojans. An astute investigator might also be able to detect *which* root kit was used by comparing the checksums and cryptographic digests.

Presentation

mtree produces output readable by both humans and machines. It has several output modes that need to be considered:

- the specification created with `-c`
- the report generated when verifying against a specification
- the checksum generated by `-s`

The specification itself is probably the most confusing to the uninitiated. Luckily, it probably never has to be shown to anyone. If it did, most of the explanation would involve describing the various file properties and digest algorithms that are represented. The `*keyword*=*value*` syntax itself is fairly straightforward. Readability can be improved with the `-i` option, which causes each level of the hierarchy to be indented by four spaces.

Reports generated when verifying a hierarchy use plain language so there is no doubt what has changed and how. When a file has changed, **mtree** simply reports *`filename`* `changed`. Users who are interested can look below that line at the indented lines of the form *`keyword`* `expected` *`old_value`* `found` *`new_value`*. This output, which is likely to be most useful in court, needs very little explanation.

As shown in Section , “Data and Results”, the `-s` option emits on line on the standard error of the format `mtree:` *`hierarchy`* `checksum:` *`checksum`*, which anyone with a basic knowledge of checksums could interpret and compare to other values. One would need to explain to the court that the checksum is just meant for quick verification of changes. It's fairly plausible to craft changes to produce a matching checksum, but differing checksums definitely indicate changes to the system. But it should be easy for the court to visually compare the 32-bit checksum values from two-separate runs as we did in Example 33 and Example 34.

Conclusion

FreeBSD's current implementation of **mtree**, without the patches we applied, would not be appropriate for use as a forensic tool. Once patched to escape special characters in filenames and produce correct checksums with `-s`, however, **mtree** becomes an excellent tool for gathering evidence. It succeeds in quickly telling us exactly what has changed on a filesystem and does not produce superfluous, confusing noise. It does alter access times on the filesystem, but nothing else. The access times can be protected by mounting the filesystem being analyzed as read-only or with the *`noatime`* option.

Since **mtree** can easily be built as a static binary and it relies on no other data files, it makes a great candidate for a CD of forensic tools for an incident response jump kit. One problem is that the specifications and reports can be fairly large, and the more keywords you use, the larger they become. When using a forensic CD, it might be necessary to set up a secure network channel (*e.g.*, using encrypted **netcat**) to another computer where this data can be stored.

Apart from the patches already discussed, I have one suggestion for improving **mtree**. Currently, the `-s` option is not as useful as it could be because the checksum algorithm it uses is not cryptographically sound. This option should be modified to use any of the digest algorithms already supported (MD5, SHA-1, or RIPEMD-160). Though it might also be nice to support signing and encryption of specifications, such a feature would be overkill because **gpg** can be used easily on **mtree**'s output.

Part 3: Legal Issues of Incident Handling

In general, the Fourth Amendment protects people's rights to privacy and from “unreasonable” searches and seizures. The U.S. Supreme Court has said “the Fourth Amendment protects people, not places” [Katz v. U.S., 389 U.S. 347 (1967)]. Therefore, if users have an expectation of privacy, their information cannot be freely shared. This right does not just cover a person's home computer. Business and commercial premises are also covered by the Fourth Amendment [*See v. City of Seattle*, 387 U.S. 541 (1967)].

Information Sharing Rules for the Initial Contact with Law Enforcement

Where federal law is concerned, sharing communications records with law enforcement is governed by three statutes [Salgado, 29]. The Wiretap Act [18 USC §§ 2510-22] limits interception of the contents of real-time communications. The Pen/Trap statute [18 USC §§ 3121-7] applies to meta-information associated with real-time communications, such as protocol and e-mail headers and logs. Stored data, be it the contents of that data or meta-information, is protected by the Electronic Communications Privacy Act (ECPA) [18 USC § 2510 *et. seq.*].

In this situation, the employee contacted by law enforcement is considered to be acting under “color of law”, meaning the statutes apply to the employee just as they do to law enforcement. Out of the several exceptions to the Wiretap Act under which an ISP could share real-time communications data with law enforcement under the Wiretap Act, only one applies—the consent exception [18 USC § 2511(2)(c)]. If at least one of the parties has consented to interception of communications and sharing with law enforcement, the employee acting under color of law may do so. It follows that the exception also applies if the employee is one of the individuals taking part in the communications.

The Pen/Trap statute would also be restrictive in this situation unless one of the parties had given prior consent [18 USC § 3121(b)(3)]. Without consent, this statute prevents sharing protocol headers, e-mail headers, logs, and other non-content data in this situation.

The ECPA specifies whether information relating to stored communications, be it the communications data itself or meta-information, can be shared with law enforcement. The ECPA makes the distinction between a *public* and a *private* provider. In this situation, the ISP would be considered a public provider because it provides its services to the public (regardless of whether a fee is paid for those services). Under the ECPA, a private provider is able to share any stored data, but a public provider may only do so if one of the exceptions, which vary based on the type of data, applies.

The ECPA allows a public provider to disclose communications content if any of the exceptions under 18 USC § 2702(b) apply. Five of the exceptions might apply in this situation:

1. the law enforcement agent was the intended recipient [18 USC § 2702(b)(1)]

2. one of the parties to the communication has consented to sharing the data [18 USC § 2702(b)(3)]

3. sharing the data will protect the rights and property of the ISP [18 USC § 2702(b)(5)]

4. disclosure is required by the Crime Control Act of 1990, § 227 [18 USC § 2702(b)(6)(B)]

5. immediate disclosure would prevent serious injury or death [18 USC § 2702(b)(6)(C)]

The list of exceptions differs slightly for sharing data that pertains to users themselves and not actual content. Exceptions 2, 3, and 5 above apply [18 USC § 2702(c)(2)-(4)], but the ISP may also disclose user data to anyone other than the government [18 USC § 2702(c)(5)].

The common theme among the exceptions for all of these statutes is consent. If a provider wants to be helpful to law enforcement, it should adopt a policy of requiring users to consent to monitoring and disclosure of communications and records [Salgado, 33]. If the ISP in this hypothetical situation had such a policy, it could share all the user's records and real-time and stored communications with the law enforcement agent over the phone (*i.e.*, without yet having issued a warrant or subpoena). (Actually, some states require that *all* parties to communication consent to wiretaps, but Indiana, where I work, is not one of them.) Otherwise, it's unlikely the ISP, as a public entity, would be able to share anything unless one of the above exceptions applied.

Preservation of Evidence

According to 18 USC § 2703(f)(1), all the officer must do is request the evidence be preserved. The evidence must be preserved for a 90-day period which law enforcement can extend by an addition 90 days by renewing the request [18 USC 2703(f)(2)].

Legal Authority Required for Sharing Logs

There are several ways a law enforcement officer can get the authority to request logs, defined in 18 USC § 2703(c)-(d). The officer must:

1. provide a warrant issued “by a court with jurisdiction over the offense under investigation or equivalent State warrant” [18 USC § 2703(c)(1)(A)]

2. get a court order [18 USC § 2703(c)(1)(B)], which the court may choose to quash or amend based on a motion by the ISP [18 USC § 2703(d)]

3. get the consent of the user [18 USC § 2703(c)(1)(C)]

Though other provisions are outlined in subsection (c), they apply to user information rather than logs.

Other Permitted Investigative Activity

The provider exception of the Wiretap Act permits the ISP to monitor real-time communications in two circumstances [18 USC § 2511(2)(a)(i)]. One of them applies when the monitoring is an unavoidable side effect of rendering services. That exception doesn't really apply because the ISP is actively investigating. The other exception allows a provider to monitor to protect its “rights and property” [18 USC § 2511(2)(a)(i)]. Precedential case law says an employee of the ISP can continue to monitor an attack to ensure no further damage is done [*U.S. v. Mullins*, 992 F.2d 1472 (9th Cir. 1993)], though *U.S. v. McLaren* [957 F.Supp. 215, 219 (M.D. Fla. 1997)] limits the scope of the monitoring to only that which protects the ISP's rights and property.

Again, prior consent would allow the ISP to investigate further communications, whether they be real-time or stored. If written consent is on file and banners are posted on network hosts warning users of their implied consent to monitoring and disclosure, the ISP's investigative powers are much less limited.

Unauthorized Access

Up to this point, we've assumed for the sake of argument that the user being investigated was legitimate. But if the evidence suggests otherwise, a new exception to the Wiretap Act applies. This exception was created in 2001 by the USA Patriot Act, which introduces the concept of an unauthorized attacker or “computer trespasser”, who have “no reasonable expectation of privacy” [18 USC § 2510(21)(A)]. Because in this situation we assume the ISP consents to the investigation and is otherwise acting appropriately under color of law, the employee of the ISP may intercept any communications pertaining to the investigation [18 USC 2511(2)(I)-(III)]. While investigating, the employee may not intercept any communications other than those in which the trespasser are involved [18 USC 2511(2)(IV)].

Therefore, if a trespass were evident, I would monitor the attacker's real-time communications, being careful to intercept only the data that pertains to the attacker and the investigation. Providing my institution consented, I would share this data with law enforcement.

A. mtree Patches

By itself, **mtree** has bugs that prevent it from being used reliably as a forensic tool. But those bugs are easy to fix with the following patches.

The first patch ensures that files are visited in the same order when creating a specification and verifying a hierarchy against a specification. Without the patch, a sort function is used when creating a specification to control how files appear in the specification, but no sort function was used when verifying. Since the CRC algorithm used by the `-s` option is affected by the order in which individual file checksums are computed, the final checksum is often incorrect. This patch, which causes **mtree** to use the same sort function when verifying, corrects the final checksum but does not appear to change the resulting report or lengthen run time.

```
diff -u mtree.orig/create.c mtree/create.c
--- mtree.orig/create.c Fri Apr  4 17:08:33 2003
+++ mtree/create.c Thu Apr  3 18:38:37 2003
@@ -79,7 +79,6 @@
 static mode_t mode;
 static u_long flags = 0xffffffff;

-static int dsort __P((const FTSENT **, const FTSENT **));
 static void output __P((int, int *, const char *, ...));
 static int statd __P((FTS *, FTSENT *, uid_t *, gid_t *, mode_t *,
                            u_long *));
@@ -399,7 +398,7 @@
         return (0);
 }

-static int
+int
 dsort(a, b)
         const FTSENT **a, **b;
 {
diff -u mtree.orig/extern.h mtree/extern.h
--- mtree.orig/extern.h Tue Jun 27 21:33:17 2000
+++ mtree/extern.h Thu Apr  3 18:39:37 2003
@@ -39,6 +39,7 @@
 void  cwalk __P((void));
 char *flags_to_string __P((u_long));

+int  dsort __P((const FTSENT **, const FTSENT **));
 char *inotype __P((u_int));
 u_int  parsekey __P((char *, int *));
 char *rlink __P((char *));
diff -u mtree.orig/verify.c mtree/verify.c
--- mtree.orig/verify.c Fri Jan 12 14:17:18 2001
+++ mtree/verify.c Thu Apr  3 18:39:52 2003
@@ -85,7 +85,7 @@

         argv[0] = ".";
         argv[1] = NULL;
- if ((t = fts_open(argv, ftsoptions, NULL)) == NULL)
+ if ((t = fts_open(argv, ftsoptions, dsort)) == NULL)
                 err(1, "line %d: fts_open", lineno);
         level = root;
         specdepth = rval = 0;
```

This patch, submitted to FreeBSD by Andrew L. Neporada, escapes hash marks (#) when creating specifications

```
Index: create.c
===================================================================
RCS file: /home/ncvs/src/usr.sbin/mtree/create.c,v
retrieving revision 1.22
diff -u -r1.22 create.c
--- create.c 5 Jul 2001 07:52:56 -0000 1.22
+++ create.c 16 Mar 2002 21:59:58 -0000
@@ -159,7 +159,12 @@
         escaped_name = calloc(1, p->fts_namelen * 4  +  1);
         if (escaped_name == NULL)
                 errx(1, "statf(): calloc() failed");
- strvis(escaped_name, p->fts_name, VIS_WHITE | VIS_OCTAL);
+ len = strvis(escaped_name, p->fts_name, VIS_WHITE | VIS_OCTAL);
+ if (escaped_name[0] == '#') {
+         for (val = len; val; val--)
+                 escaped_name[val] = escaped_name[val-1];
+         escaped_name[0] = '\\';
+ }

         if (iflag || S_ISDIR(p->fts_statp->st_mode))
                 offset = printf("%*s%s", indent, "", escaped_name);
Index: spec.c
===================================================================
RCS file: /home/ncvs/src/usr.sbin/mtree/spec.c,v
retrieving revision 1.14
diff -u -r1.14 spec.c
--- spec.c 17 Jun 2000 14:19:33 -0000 1.14
+++ spec.c 16 Mar 2002 22:24:52 -0000
@@ -65,6 +65,7 @@
         register char *p;
         NODE ginfo, *root;
         int c_cur, c_next;
+ u_long i, len;
         char buf[2048];

         centry = last = root = NULL;
@@ -147,6 +148,11 @@
 #define MAGIC "?*["
                 if (strpbrk(p, MAGIC))
                         centry->flags |= F_MAGIC;
+         if (p[0] == '\\' && p[1] == '#') {
+                 len = strlen(p);
+                 for(i = 0; i < len; i++)
+                         p[i] = p[i + 1];
+         }
                 if (strunvis(centry->name, p) == -1) {
                         warnx("filename %s is ill-encoded and literally
 used",
                             p);
```

B. MD5 Digests of Files Used in this Paper

```
MD5 (image/2.2.6-RELEASE-live.img) = d4c4ee92138cbd33964a08e712b024ba
MD5 (image/original) = 17435a3282e6c0315eb56bc64cd4a2c4
MD5 (image/compromised) = 3c4569ba7fde99bcb0a4500c7a79aec6
MD5 (rk/fbsd.rootkit.1.2.tar.gz) = 0bb36973fbd3697677c0757e834ecee5
MD5 (rk/fbsdrootkit-1.2/rootkitnetstat.h) = 2b442d4c6a0f2724d93b42c6d8b4fb76
```

```
MD5 (rk/fbsdrootkit-1.2/rootkitsyslogd.h) = 4096efe29fa2ed651fab81c997a4fd8d
MD5 (rk/packetstorm/addlen.c) = bd837cbf7a8518dca2ae3ee6854f3ee0
MD5 (rk/packetstorm/bindshell.c) = fda3e105e87c60f64868e30f9ef2b168
MD5 (rk/packetstorm/config.h) = e88973e91ac45410dd05fde3f21dffd8
MD5 (rk/packetstorm/du/du.1) = 607a0efa3d4eba64d4ebe07d7c633eba
MD5 (rk/packetstorm/du/du.c) = 203c0b63a3e3ca0b8fae0e6fe524ad2e
MD5 (rk/packetstorm/du/Makefile) = 3cdc70c3d61f85ba0edaaec42abfae08
MD5 (rk/packetstorm/fix.c) = 5f5b5dfafa4236ff5b49bfd3e9e3950d
MD5 (rk/packetstorm/ifconfig/ifconfig.8) = 02ade5d945c2872f88b00ffe33b32b5e
MD5 (rk/packetstorm/ifconfig/ifconfig.c) = feb8a38e5e1a3c9295709529a135a84c
MD5 (rk/packetstorm/ifconfig/ifconfig.h) = 79d84989e108d0ff2f8d290f36b6982b
MD5 (rk/packetstorm/ifconfig/ifmedia.c) = c0d9621b7e6ebe93675ee55e70cfee2f
MD5 (rk/packetstorm/ifconfig/Makefile) = 7d790d0cb855e1b2d5727b12dd2265aa
MD5 (rk/packetstorm/inetd/inetd.8) = 32be797cbb2b9235ae77f895623bc93f
MD5 (rk/packetstorm/inetd/inetd.c) = a85e863c199e96a65b446c873085771d
MD5 (rk/packetstorm/inetd/Makefile) = 4bf83a35cb1d590ab03db0e7a628adfb
MD5 (rk/packetstorm/inetd/pathnames.h) = 7310121b88aceea4fa141866c400a667
MD5 (rk/packetstorm/install.sh) = a0b1b486540ede6a9069fb26ee81810e
MD5 (rk/packetstorm/login/klogin.c) = 0839182847fc95e7785725af4dbffe91
MD5 (rk/packetstorm/login/login.1) = 99391bcca40b6e76a1cb630780d01887
MD5 (rk/packetstorm/login/login.access.5) = 7a06cc20967db99a73881b537ca782a5
MD5 (rk/packetstorm/login/login.c) = 7fb41feb7444c8f4e36971e6b42ae8c7
MD5 (rk/packetstorm/login/login_access.c) = 286cea2bb0032f06fb0ca02667d6968f
MD5 (rk/packetstorm/login/login_fbtab.c) = c4119ba044a24bfc1271ae517bc66ab2
MD5 (rk/packetstorm/login/Makefile) = db8585ec12fb3501a7566a6f716d7f33
MD5 (rk/packetstorm/login/pathnames.h) = 13c77c82ac16eca2a657dec83c164b20
MD5 (rk/packetstorm/login/README) = db8c29f85c4580406c305b0cc23e40f6
MD5 (rk/packetstorm/ls/cmp.c) = 6b2e54745d9507be4bc7434a5cfdbd41
MD5 (rk/packetstorm/ls/extern.h) = 1c84a48703f5c27cdbd6d7e588f3085f
MD5 (rk/packetstorm/ls/ls.1) = d3eaf32e38cfaa3111fcbe524caccc99
MD5 (rk/packetstorm/ls/ls.c) = 735172c2406a7e2aa4b7e621ba81477c
MD5 (rk/packetstorm/ls/ls.h) = 33c20b0c8fa0d8e9c6fef3f878022f79
MD5 (rk/packetstorm/ls/Makefile) = 4849d1bc68f798f97497286e83851415
MD5 (rk/packetstorm/ls/print.c) = 93eb7d4671986fc6412ed8a1742666e3
MD5 (rk/packetstorm/ls/stat_flags.c) = c6f42e64da46aaec6c462d6bf53cedfe
MD5 (rk/packetstorm/ls/util.c) = da5186bfac5354ed33e9c0699df6a1eb
MD5 (rk/packetstorm/Makefile) = 22f3e5a0f02122230f78d3392fa2f70c
MD5 (rk/packetstorm/marryv11.c) = f82d1bf7f57716d45184e4d0b033f819
MD5 (rk/packetstorm/netstat/atalk.c) = bd86c084a134bcffbcdda71d7d4cb950
MD5 (rk/packetstorm/netstat/if.c) = a4da2f06e56d8b3ff9663394e659c0c9
MD5 (rk/packetstorm/netstat/inet.c) = 75aec6c26160319c9cc354783e4a28f2
MD5 (rk/packetstorm/netstat/ipx.c) = 45b5dca03005606bcb7939bd02b8082f
MD5 (rk/packetstorm/netstat/iso.c) = 7bfc1b6deb21a9a92245d668f7fc1d20
MD5 (rk/packetstorm/netstat/main.c) = 85ae9a8c107fce99cf3ddff04dc5e5fd
MD5 (rk/packetstorm/netstat/Makefile) = f189c98017fd6c362f63a1314a986b87
MD5 (rk/packetstorm/netstat/mbuf.c) = 3180e495490dd082137431b3e3fdd57d
MD5 (rk/packetstorm/netstat/mroute.c) = 916c6859161ff6f813178488023a15b6
MD5 (rk/packetstorm/netstat/netstat.1) = 4d2a4426379a42d4f66901ab3c137391
MD5 (rk/packetstorm/netstat/netstat.h) = cc41e511ca198fee92482599cddd485f
MD5 (rk/packetstorm/netstat/ns.c) = ae7606b3d14cc86de9b70ed65e4efbf3
MD5 (rk/packetstorm/netstat/route.c) = 64c3be6f3a21d655688b906dd0f7ddbc
MD5 (rk/packetstorm/netstat/unix.c) = 62622a5f5c94387963ad7aa5fb5e6829
MD5 (rk/packetstorm/netstat/rootkitnetstat.h) = 2b442d4c6a0f2724d93b42c6d8b4fb76
MD5 (rk/packetstorm/ps/devname.c) = e282616729870c3351d30ec001221c2f
MD5 (rk/packetstorm/ps/extern.h) = b86580056c9807630c97c62c67386bec
MD5 (rk/packetstorm/ps/fmt.c) = ea5c1c71516fc5b795aab9f9b0dd31c1
MD5 (rk/packetstorm/ps/keyword.c) = 3aa37e2efe2812e235a1a3289948a0a9
MD5 (rk/packetstorm/ps/Makefile) = 9640c674566dea23d6e54e4babe5ec82
MD5 (rk/packetstorm/ps/nlist.c) = c642d1d0d942c7c6a8d886f197bb6631
```

```
MD5 (rk/packetstorm/ps/print.c) = 19606ff1ce9d3f797cd5d8b15bc58c52
MD5 (rk/packetstorm/ps/ps.1) = 61b0817aa4ccf13e97e68697f86a5fa9
MD5 (rk/packetstorm/ps/ps.c) = 885a00aa26873b85b4f139b56b57cbc4
MD5 (rk/packetstorm/ps/ps.h) = 01e8c81d1808d45294fd82ed43804b9f
MD5 (rk/packetstorm/ps/rootkitps.h) = ac23e5fe1db4b5f1ac5faa8bd247db96
MD5 (rk/packetstorm/README) = 2a473f739fd0f69eeedf4e623f73faad
MD5 (rk/packetstorm/rootkitutil.h) = 94835edc62343b56f94cf792ed2565b9
MD5 (rk/packetstorm/rshd/Makefile) = 639caa8d986da383ab1e59c7ebf205f4
MD5 (rk/packetstorm/rshd/rshd.8) = 52c1919b63330486dae5a296854bf144
MD5 (rk/packetstorm/rshd/rshd.c) = 1c3fb47d4a4cb902760c55ca5524689e
MD5 (rk/packetstorm/sniffit.0.3.5/config.guess) = 86829804843dc9788c59930d9154fb
98
MD5 (rk/packetstorm/sniffit.0.3.5/config.sub) = 9b6423290a5b9e54465d120adc3ccd9b
MD5 (rk/packetstorm/sniffit.0.3.5/configure) = 2cc69739288f84632912357f04878838
MD5 (rk/packetstorm/sniffit.0.3.5/configure.in) = c67ab52f818fa154650f7d863e32a3
02
MD5 (rk/packetstorm/sniffit.0.3.5/dns_plugin.plug) = 0c99bb23d532533e0eed84f534b
7aea8
MD5 (rk/packetstorm/sniffit.0.3.5/dummy_plugin.plug) = 7d852a9bdffb67b4be2e8add4
3cd1cff
MD5 (rk/packetstorm/sniffit.0.3.5/install-sh) = ca4c8e08be31723608758c97c4001862
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/acsite.m4) = dd0352f56e19f3f3f6efa
660bd8dc6d1
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/bpf/net/bpf.h) = 442ba24daa8446a5c
ff95c57f3f6cd88
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/bpf/net/bpf_filter.c) = 2ea132c17a
e5b394f8a45e5a2e645c4d
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/bpf_filter.c) = 2ea132c17ae5b394f8
a45e5a2e645c4d
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/bpf_image.c) = eb8bf7466d1477b63b8
3afc2bc229e50
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/CHANGES) = 2787141636b64c156e5193c
39d255364
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/config.guess) = 86829804843dc9788c
59930d9154fb98
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/config.sub) = 9b6423290a5b9e54465d
120adc3ccd9b
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/configure) = adb666b5e69cce7427df3
befcf18ec0c
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/configure.in) = 8a33df3aa1af1cc017
6c095fa72cf4ab
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/etherent.c) = b99a1d2d30581ca65092
c1af29e98e2c
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/ethertype.h) = 1c2b909b9833bed6b91
0c8c719026d77
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/FILES) = 3cdf2363b1e8932b5a44bc277
43d184e
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/gencode.c) = d5ee039b7a43a46e12804
f9d49df6b69
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/gencode.h) = f04ac2a48b766001f7824
1c9bcf9530b
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/gnuc.h) = ae904251370cecddfb2a1cbd
b368e08b
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/grammar.y) = eb159b1b546cf7c76424c
75c1ee73cb2
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/inet.c) = 7159cc2bb7dd9859f83e132c
bd5869a9
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/INSTALL) = 73c6446ac4282bf74f60209
84901d2ae
```

```
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/install-sh) = ca4c8e08be3172360875
8c97c4001862
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/lbl/gnuc.h) = ae904251370cecddfb2a
1cbdb368e08b
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/lbl/os-solaris2.h) = b2b7e2da17cb6
65baae0d2756f34a18b
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/lbl/os-sunos4.h) = a73188352a3c6d6
e5ea1c008b0814420
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/lbl/os-ultrix4.h) = d372d089532c5c
1a05a51baf0fa1bbcd
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/linux-include/netinet/if_ether.h)
= 5a1b8c615e7a3fe27676c3b4b526a22d
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/linux-include/netinet/ip_var.h) =
7881ab2e16731ce492013c9e3e6c854d
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/Makefile) = 4391ec7b0f7301399ea4de
2d6feb27f0
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/Makefile.in) = 37153215f512d445a40
c201492382a71
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/mkdep) = 906406b4df9d9f4170453a7ec
abfa7f2
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/nametoaddr.c) = bdc077ad2009bd320b
3fbf02ae6dccf3
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/net/bpf.h) = 442ba24daa8446a5cff95
c57f3f6cd88
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/net/bpf_filter.c) = 2ea132c17ae5b3
94f8a45e5a2e645c4d
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/optimize.c) = 4ffe10ea42aba9941182
e413ef1ea125
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap-bpf.c) = ff044cc418436c5aa5a2
032289d14ed0
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap-dlpi.c) = d8ed6dede10213b0581
8d9e06f6f05e5
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap-enet.c) = 58a1294c776931c6659
2e1551ff1a408
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap-int.h) = d22d4301869d81ef1d7d
4293f1650a38
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap-linux.c) = 38331b218ae15092ac
e809b67d108e28
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap-namedb.h) = 92ff3b83bc6b09ccd
f956003718ca1c7
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap-nit.c) = ee9fab5a51001682e6c0
4bf8693bde4e
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap-nit.h) = 7c0274ec710b49f814a6
4fc68b689805
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap-null.c) = fc988a12b6562f8cc4a
3982ff9d89f00
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap-pf.c) = f0ef77d03c5a4b64029c5
9ccdac5e60f
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap-pf.h) = 7d593a279bbc29b6b32f8
742476192c4
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap-snit.c) = dbf1f8fa3bce954759c
ea2395da50a7f
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap-snoop.c) = 73f8eb46552a9ecde7
d33021d04c1272
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap.3) = f570541c34169bde558daeef
9b5f4949
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap.c) = e3151cdebfe247315851b6f5
1a2193c6
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/pcap.h) = 72a0b707143405d951f7ab68
```

```
39fe4dc9
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/README) = 0141c47a0473eb42781393d0
3f567e64
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/savefile.c) = f95db48a7d5e7d1e5719
87a47d3033d5
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/scanner.l) = fea3dc1f1c827b1750c47
ed08e8f9f33
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/SUNOS4/nit_if.o.sparc) = c4a0a0d67
e33268c206b9a2b3eae3ee1
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/SUNOS4/nit_if.o.sun3) = 681ce39bb8
8e217507909a619ea6e157
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/SUNOS4/nit_if.o.sun4c.4.0.3c) = fd
97ddaa7664ce9813a96568bfb4be9b
MD5 (rk/packetstorm/sniffit.0.3.5/libpcap-0.3/VERSION) = 588f8b45134362a0dfc0931
64b839024
MD5 (rk/packetstorm/sniffit.0.3.5/Makefile) = 426a884e950e7e38fe005e1ec4581691
MD5 (rk/packetstorm/sniffit.0.3.5/Makefile.in) = 711987b38f826f1fbb26162e42bdccc
d
MD5 (rk/packetstorm/sniffit.0.3.5/pcap.h) = 72a0b707143405d951f7ab6839fe4dc9
MD5 (rk/packetstorm/sniffit.0.3.5/PLUGIN-HOWTO) = 9c223b56c7ceee0c56c47d74071e53
22
MD5 (rk/packetstorm/sniffit.0.3.5/README.FIRST) = f25b994f5e64ed751829ccf26d5f7c
cb
MD5 (rk/packetstorm/sniffit.0.3.5/sample_config_file) = 17910c210037d2320a79cf89
af657c27
MD5 (rk/packetstorm/sniffit.0.3.5/sniffit.0.3.5.c) = c1dc459664b33b9ea4633718704
f2925
MD5 (rk/packetstorm/sniffit.0.3.5/sniffit.0.3.5.c.orig) = ed70299840968f93f55600
f052da5b11
MD5 (rk/packetstorm/sniffit.0.3.5/sniffit.0.3.5.patch.1) = 58ffac5bf7fb5921c0fc7
b2fc6402054
MD5 (rk/packetstorm/sniffit.0.3.5/sniffit.5) = d1c73772104b80012e965be26c4bc56a
MD5 (rk/packetstorm/sniffit.0.3.5/sniffit.8) = 048c20da32af67f9d9cc0c6cc81c05e9
MD5 (rk/packetstorm/sniffit.0.3.5/sniffit.h) = a994173f8c824218db7374f341298b29
MD5 (rk/packetstorm/sniffit.0.3.5/sn_cfgfile.c) = c9cb1aebb54898da9091ad329b4d20
55
MD5 (rk/packetstorm/sniffit.0.3.5/sn_cfgfile.h) = aa4bc760c599e3b6cff77379dc2bba
04
MD5 (rk/packetstorm/sniffit.0.3.5/sn_config.h) = 3a9b8099b80c5912f8d8f975d568b8d
e
MD5 (rk/packetstorm/sniffit.0.3.5/sn_curses.h) = 601ebcf92fbdc2ab6d049bfe9c5d556
b
MD5 (rk/packetstorm/sniffit.0.3.5/sn_data.h) = 5834421faaa23eaec47e51224a25c04d
MD5 (rk/packetstorm/sniffit.0.3.5/sn_defines.h) = ca310e4cde626e4b1f0bab6f836bd7
0e
MD5 (rk/packetstorm/sniffit.0.3.5/sn_defines.h.orig) = 318225bdc0b08f369e29cb67a
fad4cae
MD5 (rk/packetstorm/sniffit.0.3.5/sn_defines.h.rej) = 797d3673dab14cd0d6397462da
1433f2
MD5 (rk/packetstorm/sniffit.0.3.5/sn_generation.c) = 010fe6994019d1eec1e05ef5532
3c46d
MD5 (rk/packetstorm/sniffit.0.3.5/sn_generation.h) = 17ad4d2a714c15ff9e59ea7c8d9
2879d
MD5 (rk/packetstorm/sniffit.0.3.5/sn_global.h) = 8f5a4f32655809146af8d14198bb42e
e
MD5 (rk/packetstorm/sniffit.0.3.5/sn_interface.c) = 9fd7aa7706799a700803829c93b9
a7f0
MD5 (rk/packetstorm/sniffit.0.3.5/sn_interface.h) = 95351ec6be614713fd80239087bd
85b5
```

```
MD5 (rk/packetstorm/sniffit.0.3.5/sn_logfile.c) = 6d6bc97624154eeb421c285ee361c9
cd
MD5 (rk/packetstorm/sniffit.0.3.5/sn_logfile.h) = 0f2c6787f485871cd954bfcce1f638
93
MD5 (rk/packetstorm/sniffit.0.3.5/sn_packets.c) = 68aec22fc15e7d634db8419b042071
9b
MD5 (rk/packetstorm/sniffit.0.3.5/sn_packets.c.orig) = 074f79834e34ff711aba3335b
4411707
MD5 (rk/packetstorm/sniffit.0.3.5/sn_packets.h) = 7dad48deb0793904eb1a192a72dc1a
8b
MD5 (rk/packetstorm/sniffit.0.3.5/sn_packetstructs.h) = d7fa5d49b1a0eced5dc409fa
90bc13df
MD5 (rk/packetstorm/sniffit.0.3.5/sn_packetstructs.h.orig) = 142d7943de553da5378
31397afb1a3b3
MD5 (rk/packetstorm/sniffit.0.3.5/sn_plugins.h) = fc83c1261532860f819017b339a485
48
MD5 (rk/packetstorm/sniffit.0.3.5/sn_structs.h) = d4f47f72854ceeb717558a97b0099f
0d
MD5 (rk/packetstorm/syslogd/Makefile) = d0a565ef649ac85431981652f7640890
MD5 (rk/packetstorm/syslogd/pathnames.h) = 418340c7a2fcccdb94e6a25ed9f30f25
MD5 (rk/packetstorm/syslogd/syslog.conf.5) = 6e1478cd83dd9c89ea579a32a6c9046c
MD5 (rk/packetstorm/syslogd/syslogd.8) = e52ced88fe321b67596ae947e93f2b99
MD5 (rk/packetstorm/syslogd/syslogd.c) = 7ae4dd6739249918cac31a4e744c24d5
MD5 (rk/packetstorm/syslogd/wall/Makefile) = 4276a7209b31161e2ea2ea55ba2527a0
MD5 (rk/packetstorm/syslogd/wall/ttymsg.c) = d548c799d5feb4ca93dcbed63cb9051b
MD5 (rk/packetstorm/syslogd/wall/wall.1) = 479b3a5ea298a9a3740ae8392ff2c9c2
MD5 (rk/packetstorm/syslogd/wall/wall.c) = ade74f8b205eaf988a07fd5ca34228d2
MD5 (rk/packetstorm/syslogd/rootkitsyslogd.h) = 4096efe29fa2ed651fab81c997a4fd8d
```

List of References

Bonnie, Richard J., Anne M. Coughlin, John C. Jeffries, Jr., and Peter W. Low. *Criminal Law*. Westbury, New York: The Foundation Press, Inc., 1997.

chroot(8). *FreeBSD System Manager's Manual*. FreeBSD 4.6.1. 2002.

cksum(1). *FreeBSD System Manager's Manual*. FreeBSD 4.6.1. 2002.

daemon9. “Project Loki: ICMP Tunneling”. *Phrack*. URL: <http://www.phrack.org/show.php?p=49&a=6>. 8 November 1996 (28 March 2003).

Dalton, Matthew. *A Forensic Tool Validation of the Coroner's Toolkit's* ***mactime***. SANS Institute, 2002.

Ka0ticSH. “Diggin Em Walls (part3)—Advanced/Other Techniques for ByPassing Fire-walls”. *Fromadia*. URL: <http://www.fromadia.com/newsread.php?newsid=469>. 10 May 2002 (4 April 2003).

Kamisar, Yale, Wayne R. LaFave, Jerold H. Israel, and Nancy J. King. *Basic Criminal*

Procedure. St. Paul, Minnesota: West Group, 1999.

Katz v. U.S. 389 U.S. 347. 1967.

ls(1). *GNU File Utilities Manual*. 2003.

mtree(8). *FreeBSD System Manager's Manual*. FreeBSD 4.6.1. 2002.

Owen, Gary. *Analysis of Unknown Binary*. The SANS Institute, 2002.

route. “LOKI2: (the implementation)”. *Phrack*. URL: <http://www.phrack.org/show.php?p=51&a=6>. 1 September 1997 (28 March 2003).

Salgado, Richard P. *Forensics and Incident Response*. SANS Track 8 course material. The SANS Institute, 2002.

See v. City of Seattle. 387 US 541. 1967 .

stat(1). *GNU File Utilities Manual*. 2003.

strings(1). *GNU Development Tools Manual*. 2003.

18 USC § 1030. 2001.

18 USC §§ 2510-22. 1998.

18 USC §§ 3121-7. 1994.

U.S. Const. Amendment IV.

U.S. v. McLaren. 957 F. Supp. 215, 219. M.D. Fla. 1997. .

U.S. v. Mullins. 922 F.2d 1472, 1478. 9th Cir. 1993.

vn(4). *FreeBSD System Manager's Manual*. FreeBSD 4.6.1. 2002.

vnconfig(8). *FreeBSD System Manager's Manual*. FreeBSD 4.6.1. 2002.

zipinfo(1L). *Info-ZIP Documentation*. Info-ZIP 2.4. 2002.