To appear in S. Shenoi & M. Pollitt (Eds.), *Advances in Digital Forensics*. International Association of Information Professionals.

Recovering Digital Evidence from Linux Systems

Philip Craiger, Ph.D.
Assistant Director for Digital Evidence
National Center for Forensic Science &
Department of Engineering Technology
University of Central Florida

ABSTRACT

As Linux kernel-based operating systems gain market share there will be an inevitable increase in Linux systems that law enforcement agents must process at cybercrime scenes. The skills and expertise required to recover evidence from Microsoft Windows-based systems do not necessarily translate to Linux systems. Although the procedures required to identify, recover, and examine evidence on Windows and Linux systems may appear similar at an abstract level, the "devil is in the details" as they say. This paper provides an introduction to digital forensic procedures to recover evidence from Linux systems. In particular we demonstrate: methods of identifying and recovering deleted files from disk and volatile memory; identifying notable and Trojan files; finding hidden files; and files with renamed extensions. All procedures are accomplished with Linux command line utilities and require no special or commercial tools. We close by describing a taxonomy of digital artifacts that is appropriate to serve as the basis for an automated method of identifying and recovering digital evidence in large-scale data systems.

Introduction

As Linux kernel-based operating systems gain market share there will be an inevitable increase in Linux systems that law enforcement agents must process at cybercrime scenes. The skills and expertise required to recover evidence from a Microsoft Windows-based system do not necessarily translate to the same tasks on a Linux system. Although the procedures required to identify, recover, and examine evidence on these systems may appear similar at an abstract level, the "devil is in the details" as they say. For instance, the Microsoft NTFS, FAT, and Linux

EXT2/3 file systems work differently enough that understanding one tells you little about how the other functions.

In this paper we demonstrate digital forensics procedures for Linux systems using Linux command line utilities. Although not as user-friendly -- or expensive -- as commercial tools, these utilities allow the forensic examiner to accomplish the same tasks as their more expensive counterparts, and in addition, support the recovery of evidence from live as well as powered-down systems. The ability to gather evidence from a running system is particularly important as evidence in RAM can be lost if a forensics first responder does not prioritize the collection of live evidence.

The forensic procedures we demonstrate include:

1. Several methods of identifying and recovering deleted files from:
   (a) RAM
   (b) magnetic media
2. Identifying notables files and Trojans
3. Finding:
   (a) hidden files
   (b) renamed files (files with renamed extensions)
   (c) warez

We begin by describing recovering deleted files from RAM on a live (running) Linux system. Because Linux systems are usually employed as servers, most of the demonstrations are directed toward activities and techniques that intruders are known to use after breaking into a system.

Recovering Files from RAM

A deleted file whose contents have been overwritten on disk may still be recovered. For instance, an intruder may execute a program and then delete it from disk to hide its existence.

This happens, for example, when an intruder opens a backdoor on the victim's system by running the *netcat* utility, then deleting it from the disk after it is executed. As long as the program remains running in memory it can be recovered. The file is recoverable because the Linux kernel uses a pseudo file system to track the general state of the system, including running processes, mounted file systems, kernel information, and several hundred other pieces of critical system information [WAR03]. This information is kept in virtual memory and is accessible through the */proc* directory. The listing below shows the contents of the */proc* directory on a running Linux system. Each of the numbered directories below corresponds to the process identification number (PID) of a process running in memory.

```
# ls /proc

1      4      4513  4703  4777        execdomains  mdstat      swaps
1693   40     4592  4705  acpi        fb           meminfo     sys
2      4045   4593  4706  asound      filesystems  misc
2375   41     4594  4708  buddyinfo   fs           mm          sysvipc
2429   4163   4595  4709  bus         ide          modules     tty
2497   4166   4596  4712  cmdline     interrupts   mounts      uptime
2764   4186   4597  4713  config.gz   iomem        mtrr        version
29     42     4620  4715  cpufreq     ioports      net         vmstat
2915   4225   4659  4716  cpuinfo     irq          partitions
3      4237   4660  4719  crypto      kallsyms     scsi
3221   4406   4686  4721  devices     kcore        self
3237   4418   4689  4723  diskstats   kmsg         slabinfo
3884   4436   4691  4725  dma         loadavg      splash
39     4449   4694  4727  driver      locks        stat
```

To illustrate the recovery of a deleted file, say that an intruder has downloaded a password cracker and is attempting to crack system passwords -- a very common goal for an intruder. The intruder runs the john password cracker with a list of passwords in a file called 'pass.' The intruder subsequently deletes both the executable and the text file containing the passwords, the executable remains running in memory until the process is killed.

The `ps` command displays running processes. The listing below shows the executable 'john' has been called with the 'pass' file at 10:10AM, has been running for 22 seconds, and is owned by root.

```
# ps aux | grep john

root   5288 97.9  0.0  1716  616 pts/2  R+  10:10   0:22 ./john pass
```

According to the listing above the executable's process ID (PID) is 5288. The directory */proc/5288* will contain information regarding the running process, as displayed in the listing below.

```
# ls -al /proc/5288

total 0
dr-xr-xr-x    3 root root 0 Jan 17 10:11 .
dr-xr-xr-x  108 root root 0 Jan 17 04:00 ..
dr-xr-xr-x    2 root root 0 Jan 17 10:12 attr
-r--------    1 root root 0 Jan 17 10:12 auxv
-r--r--r--    1 root root 0 Jan 17 10:11 cmdline
lrwxrwxrwx    1 root root 0 Jan 17 10:12 cwd -> /j
-r--------    1 root root 0 Jan 17 10:12 environ
lrwxrwxrwx    1 root root 0 Jan 17 10:12 exe -> /j/john (deleted)
dr-x------    2 root root 0 Jan 17 10:11 fd
-rw-------    1 root root 0 Jan 17 10:12 mapped_base
-r--r--r--    1 root root 0 Jan 17 10:12 maps
-rw-------    1 root root 0 Jan 17 10:12 mem
-r--r--r--    1 root root 0 Jan 17 10:12 mounts
-rw-r--r--    1 root root 0 Jan 17 10:12 oom_adj
-r--r--r--    1 root root 0 Jan 17 10:12 oom_score
lrwxrwxrwx    1 root root 0 Jan 17 10:12 root -> /
-r--r--r--    1 root root 0 Jan 17 10:11 stat
-r--r--r--    1 root root 0 Jan 17 10:12 statm
-r--r--r--    1 root root 0 Jan 17 10:11 status
dr-xr-xr-x    3 root root 0 Jan 17 10:12 task
-r--r--r--    1 root root 0 Jan 17 10:12 wchan
```

Directory */proc/5288* contains several files, the most important of which is 'exe,' which is a symbolic link (note the 'l' in the very first column of the permissions) to the running password cracker. The operating system (helpfully) displays a note indicating that the file was deleted from disk. Nevertheless, we can recover the file by copying the 'exe' from the directory to a separate directory.

```
# cp /proc/5288/exe ./john.recovered

# md5sum ./john.recovered ./john.original

83219704ded6cd9a534baf7320aebb7b  ./john.recovered
83219704ded6cd9a534baf7320aebb7b  ./john.original
```

In the example above we copied the 'exe' from */proc/5288* to another directory, and then compared the MD5 hash of the executable with a hash of a known copy of John. We see the hashes are the same, indicating that we successfully recovered the file. This method of file recovery works for any type of file as long the process remains in memory.

Accessing Unallocated Space

The contents of a deleted file will remain on disk until overwritten. For recovery purposes, "time is of the essence" as the disk space composing a deleted file is marked as reusable (free) by the operating system, and the space can be reused by the operating system at any time.

Once a file is deleted – by a user or the operating system – the blocks composing the deleted file are a part of unallocated space, and cannot be easily accessed at the file system level. There are two methods of accessing unallocated space. The preferred method depends upon: a) whether it is possible to shutdown the machine without disrupting users and/or business, and b) the partition on which the files resided.

If it is possible to power down the system, the forensic examiner can reboot the machine with a bootable Linux CD, and use the UNIX/Linux `dd` utility to create a bitstream duplicate of the partition on which the deleted file(s) resides. (See Appendix for a description of this procedure). The advantage to this method is the shutdown of the machine prevents files from being overwritten. A frequently occurring problem is that business managers are reluctant to powerdown a critical business system because it can take several hours for large (100GB+) hard drives, potentially disrupting customers.

A second method of accessing unallocated space is more efficient and relies on the assumption that partition on which the deleted file resides is on a separate partition from the operating system. For instance, on Linux servers it is common practice -- although not required -- to create separate partitions for directories that are volatile, such as the */var* directory that stores log files, and the */home* directory that stores user directories and files. If an examiner must recover a file from, say, the */home* directory, she can unmount the */home* partition and access it as a physical device through its entry in the */dev* directory.

Recovering Files by Type

We can manually recover a file by searching unallocated space for the file's header, which is located at the beginning of a file. For instance, say we know that an intruder deleted a directory containing several hundred bitmap graphics. We can search through unallocated space for a sector beginning with "BM," the signature for a bitmap graphic. When found we can manually recover the file using the Linux *dd* command. The success of this procedure is assumes that: a) we can identify header information; b) the file has not been overwritten; and c) the file is not fragmented. If the file is fragmented, we will only be able to recover part of the file, as we will be unable to identify the blocks that previously comprised the file.

In this demonstration we have an image (or unmounted partition) that contains several deleted *.jpg files. First we must identify the first sector of each *.jpg file, which we do by searching for the text 'JFIF,' commonly found in a *.jpg file. Figure 1 shows a starting sector for a deleted *.jpg file. (Note: The only part of the header required for a *.jpg file are the first three bytes: `ffd8ff`. In experiments we found that deleting the 'JFIF' in the header does not prevent applications from accurately identifying the type of file, although removing any of the first three bytes does.)

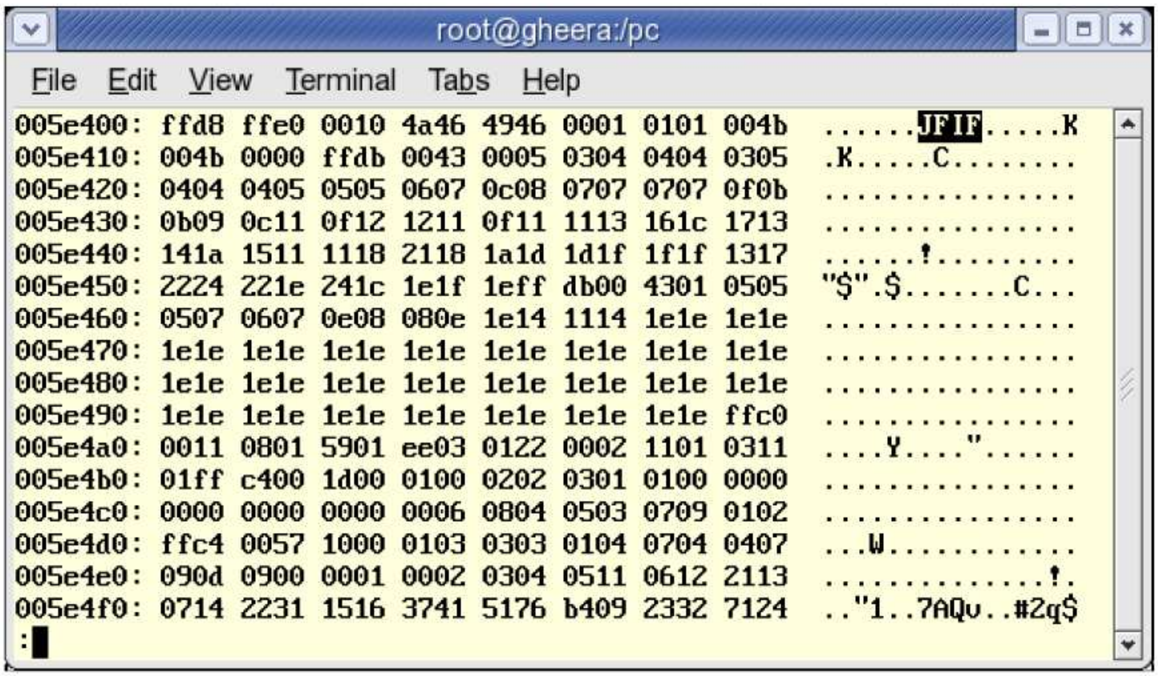

Figure 1. Start of JPG File on Disk as Viewed with a Hex Viewer

Figure 1 shows that the file starts at 0x5e400 (hex). We convert this to decimal, 386,048, and divide by 512 (the number of bytes in sector) resulting in 754, which is the *starting sector number* of the file, i.e., from the beginning of the image.

Under many circumstances we won't know the **exact** size of a deleted file, requiring us to make an educated guess as to its size. If we guess too low and **under recover** the file, viewing the file in its application will show that too few sectors were recovered, and the file will not appear complete. If we recover too many sectors, we have an **over recovery**. In our experience recovering too many sectors does not harm the file. Once we recover the file we can view it in the appropriate application to determine the accuracy of our guess.

We use the UNIX/Linux *dd* command to carve the file from the image. We specify the input file to be our image (`if=image.dd`), and we choose a name for the recovered file (`of=recovered1.jpg`). We must specify the starting sector to begin carving. According to our earlier calculation the image starts at physical sector 754. The default block size in *dd* is 512 bytes, which we will leave as is. Finally we must specify the number of consecutive blocks to recover. In this instance we will guess 30 blocks of size 512 bytes each, so we are recovering files of size 15K.

```
# dd if=image.dd of=recovered1.jpg skip=754 count=30
30+0 records in
30+0 records out

# file recovered1.jpg
recovered1.jpg: JPEG image data, JFIF standard 1.01
```

We successfully recovered 30 consecutive sectors per file. The file command shows we recovered the header successfully. Figure 2 shows the recovered file.

<u>Figure 2. File Under Recovery.</u>

As Figure 2 demonstrates, we have under estimated the file size, resulting in recovery of only half the file. We increased the count to 100 (50K file size) resulting in Figure 3.

```
# dd if=image.dd of=recovered2.jpg skip=754 count=100
100+0 records in
100+0 records out
```

<u>Figure 3. File Over Recovery</u>

The actual file size of the file was 29K. As Figure 3 demonstrates an over recovery does nothing to harm the file contents, and the file is readable as long as its contents are sequential.

This recovery method can be used with any type of file, as long as the file's header information remains intact. The success of this method depends, again, on the lack of file fragmentation and some luck as to whether any blocks of the file have been reused.

A General File Recovery Procedure

If a file's header has been overwritten and the file is primarily text, we can use a more general recovery procedure that only requires that we know some keywords for which to search, and of course, that the file has not been **completely** overwritten.

For this demonstration we will recover the Linux general log file */var/log/messages*. This file is often targeted by an intruder as it will contain evidence of the intruder's tracks. Novice intruders will delete the entire file, which is clearly evidence to an administrator that an intrusion occurred. In contrast, skilled intruders will surgically remove lines that point to their break-in, keeping the remaining contents.

To recover the log file we must identify keywords contained in the file. Ideally we identify keywords that are unique to the file, thus reducing the number of false-positive hits. For this example we are likely to encounter some false-positives because log files are rotated on a frequent basis, so our search is likely to pick up keywords from previous versions of the log file.

We unmount the partition that contains the directory `/var` where *messages* resides. This is simple if `/var` is on its own partition:

```
# umount /dev/hda3
```

If `/var` is on the same partition as the root directory we will need to reboot the system using a Linux bootable CD such as Knoppix (www.knoppix.com), Local Area Security (www.localareasecurity.com) or Helix (www.e-fense.com), and perform the procedures from the boot disk [CRA05].

Next use we use *grep* to search for the keywords on the physical device (unmounted partition). We are using the physical device because we must access unallocated space through the physical device:

```
# grep -ia -f keywords -C 2 /dev/hda3
```

The flag 'i' specifies a a case insensitive search. The flag 'a' specifies to treat the input (contents of the physical device */dev/hda3*) as ASCII text; if we don't then *grep* will only indicate whether the file contains the keyword or not. The flag 'f' specifies that what follows is a text file that contains a list of keywords for which to search. We are essentially conducting a simultaneous search for multiple keywords, which we might use, for example, if we are unsure as to exactly what keywords our deleted file contains. The flag 'C 2' specifies that we want two lines of context – two lines before and after a keyword hit. Finally we specify the physical device to search which contained the */var* directory.

For this demonstration we assume that the attacker made several unsuccessful attempts to log into the root account -- a common occurrence in an intrusion. These unsuccessful login attempts will be noted in the *messages* log file.

The results of our search are displayed below:

```
Dec 18 19:12:59 gheera messagebus: messagebus startup succeeded
Dec 18 19:13:00 gheera cups-config-daemon: cups-config-daemon startup
succeeded
Dec 18 19:13:09 gheera gdm(pam_unix)[2727]: authentication failure; logname=
uid=0 euid=0 tty=:0 ruser= rhost=  user=schmoopie
```

```
Dec 18 19:13:13 gheera gdm-binary[2727]: Couldn't authenticate user
Dec 18 19:13:16 gheera gdm(pam_unix)[2727]: session opened for user schmoopie
by (uid=0)
--
...
Dec 20 18:33:29 gheera gdm(pam_unix)[2752]: authentication failure; logname=
uid=0 euid=0 tty=:0 ruser= rhost=  user=schmoopie
...
--
...
Dec 21 18:16:55 gheera gdm(pam_unix)[2750]: authentication failure; logname=
uid=0 euid=0 tty=:0 ruser= rhost=  user=schmoopie
...
--
Dec 22 17:38:21 gheera kernel: lp0: using parport0 (polling).
Dec 22 17:38:21 gheera kernel: lp0: console ready
Dec 22 17:49:33 gheera gdm(pam_unix)[2756]: authentication failure; logname=
uid=0 euid=0 tty=:0 ruser= rhost=  user=schmoopie
Dec 22 17:49:36 gheera gdm-binary[2756]: Couldn't authenticate user
Dec 22 17:49:48 gheera gdm(pam_unix)[2756]: session opened for user schmoopie
by (uid=0)
```

The keywords are in bold. It appears that user 'schmoopie' unsuccessfully attempted to log in as root on December 18, 19, 21, and 22. Note the two lines of context both before and after each search hit. In practice we would not want such a limited result: We would rather recover the entire contents of the log file, which we could to do by requesting a much larger value for context, e.g., -C 100. Because we do not know *a priori* how large the file is this will be trial and error effort.

To appear in S. Shenoi & M. Pollitt (Eds.), *Advances in Digital Forensics*. International Association of Information Professionals.

Recovering Files from EXT2 Formatted Disks

A final recovery method assumes that the file system is EXT2, a somewhat common Linux file system (although it is being replaced by more efficient journaling file systems). In this method we can use the system debugger to find and recover the file. For this example, say a recently terminated employee deleted an important file from his directory under */home*. (Not an uncommon event for terminated employees.) Say we are informed that the file was a zip archive. We must determine the hard drives geometry, including the number of partitions, how the partitions are formatted, before we begin the file recovery process. The Linux command `fdisk -l` provides this information:

```
# fdisk -l

Disk /dev/hda: 30.0 GB, 30005821440 bytes
16 heads, 63 sectors/track, 58140 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes

Device Boot     Start   End      Blocks     Id  System
/dev/hda1           1   41613    20972826   83  Linux
/dev/hda2       57147   58140      500976    f  W95 Ext'd (LBA)
/dev/hda3       41613   52020     5245222+  83  Linux
/dev/hda5       57147   58140      500944+  82  Linux swap
```

We see that we have a single 30GB IDE hard drive with four partitions. The first partition (*/dev/hda1*) is a primary partition formatted in EXT2. The second partition (*/dev/hda2*) is an extended partition (hence the *Ext'd* notation) containing two logical partitions, one an EXT2 file system (*/dev/hda3*) and the second a Linux swap file (*/dev/hda5*).

Next we need to know which directories are mounted on which partitions. We run the `mount` command which displays this information.

```
# mount | column -t

/dev/hda1  on  /              type  ext2    (rw,acl,user_xattr)
proc       on  /proc          type  proc    (rw)
tmpfs      on  /dev/shm       type  tmpfs   (rw)
devpts     on  /dev/pts       type  devpts  (rw,mode=0620,gid=5)
/dev/hda3  on  /home          type  ext2    (rw,acl,user_xattr)
/dev/hdc   on  /media/cdrom   type  subfs
(ro,nosuid,nodev,fs=cdfss,procuid,iocharset=utf8)
```

The mount command shows us that the */home* directory is mounted on */dev/hda3* device. We unmount the */home* directory, or remount it read-only so that there is no possibility of overwriting the deleted file. The more quickly this can be done the better; as the file has a good chance of being overwritten the longer the partition remains mounted. To unmount the directory, we issue the command:

```
# umount /home
```

We use the debugger `debugfs` to open the partition and recover the deleted file. In the debugger we execute the `lsdel` command to display inode information on all the deleted files on the partition. (An inode is a data structure that holds file metadata. See [CRA99] and [BUC03] for more information on inodes.)

```
# debugfs /dev/hda3

debugfs 1.35 (28-Dec-2004)
debugfs:  lsdel

Inode  Owner  Mode    Size    Blocks   Time deleted
272319     0 100755 3383328  828/ 828 Thu Dec 23 23:45:22 2004
1 deleted inodes found.
lines 1-3/3 (END)
```

The `lsdel` command indicates that a file represented by inode number 272319 was deleted on December 23 and was of size 3MB (comprising 828 blocks). Once we have the inode number we can get more detailed information with the `stat` command:

```
debugfs: stat <272319>

Inode: 272319 Type: regular Mode:  0755   Flags: 0x0  Generation: 92194859
User:     0   Group:     0   Size: 3383328
File ACL: 0   Directory ACL: 0
Links: 0   Blockcount: 6624
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x41cb9ee2 -- Thu Dec 23 23:45:22 2004
atime: 0x41cb9d68 -- Thu Dec 23 23:39:04 2004
mtime: 0x41cb9d68 -- Thu Dec 23 23:39:04 2004
dtime: 0x41cb9ee2 -- Thu Dec 23 23:45:22 2004
BLOCKS:
(0-11):582122-582133, (IND):582134, (12-826):582135-582949
TOTAL: 828
```

The `stat` command provides us with a variety of information, including the modified, accessed, changed, and deleted date and times of the deleted file. (Unlike NTFS and FAT file systems, the Linux EXT2 file system tracks a file's deleted date and time.) The `stat` command also shows us the number of direct, indirect, and doubly indirect blocks under the BLOCKS section. (For a more thorough explanation of the EXT2 file system see [CRA99], [BUC03], and [PAR04]). It appears that all blocks are intact, i.e., no blocks have been overwritten, meaning we can recover the entire file. The `dump` command takes as argument an inode number and a name to call the recovered file:

```
debugfs:  dump <272319> hda3.recovered
```

Once we exit the debugger we determine the recovered file's type with the `file` command. The `file` command uses the header information to determine the type of file.

```
# file hda3.recovered
hda3.recovered: Zip archive data, at least v1.0 to extract
```

Our recovered file is a ZIP archive, as expected. We determine the success of our procedure by comparing the hash of our recovered file with the hash of the original file (which we happen to have for our demonstration here). The hashes match indicating that we successfully recovered the file. (Or when an MD5 doesn't exist of the original, simply 'unzipping' the file, in is case.)

```
# md5sum original.file.zip hda3.recovered

ed9a6bb2353ca7126c3658cb976a2dad  original.file.zip
ed9a6bb2353ca7126c3658cb976a2dad  hda3.recovered
```

The success of this procedure depends on a number of critical factors. First is the time interval between when the file is deleted and attempted recovery. The longer the time between deletion and recovery, the more likely part or the entire file will be overwritten. A second factor is file size. Smaller files (that fit in the direct blocks) have a higher probability of being recovered than larger files that may also require the use of indirect and doubly indirect blocks.

Identifying Notable Files and Trojans

The two primary goals of intruders are to effectuate a break in, and to remain on the victim's system as long as possible. Remaining hidden on the system is usually accomplished by installing a rootkit. A rootkit replaces several important system files with 'Trojaned' versions. The Trojaned versions work like the original system files with the exception that they fail to display any traces of the intruder, such as running processes, open files or open sockets. Utilities that are commonly Trojaned include *ps* (to display system processes), *netstat* (to display sockets and network connections), and *top* (display process information sorted by activity), among others.

A simple way to identify Trojaned files is through a hash analysis. A hash analysis compares the one-way cryptographic hashes of "notable" files with hashes of files on the system. If two hashes match it indicates that a file has been replaced with a Trojaned version. Below we show the contents of a file containing MD5 hashes of several Trojaned versions of Linux utilities.

```
# cat rootkit.md5

e25684245306fba8e37117a52e1c4ce5  netstat
7728c15d89f27e376950f96a7510bf0f  ps
9e3ca9002efbda4e008d451d6a457553  ben
4d957974e91f8f3bc757e3b4b657b1cd  core
```

Next we compare these hashes to those of the utilities on the hard drive. The md5deep utility (md5deep.sourceforge.net) is freeware that takes a file that contains MD5 hashes and compares them recursively to the hashes of files on a system. This is demonstrated in the listing below.

```
# md5deep -r -m rootkit.md5 /

/bin/netstat
/bin/ps
```

According to the hash analysis *ps* and *netstat* have been replaced with Trojaned versions. We are certain that the two files were replaced with Trojaned versions because their MD5 hashes matched exactly the MD5 hashes of known Trojaned files.

A second method of identifying Trojans is by comparing inode numbers of files within a directory. An inode number that is substantially out-of-sequence with the inode numbers of other files in a directory could be an indication that the file has been replaced.

When a file is saved to the hard drive it is assigned an inode number. Files that are saved in short succession will have inode numbers that are consecutive or nearly so. This is demonstrated below, which displays a directory listing of the contents of the /bin directory, sorted by inode number (located in the first column).

```
# ls -ali /bin | sort
130085 -rwxr-xr-x   1 root root   2680 Nov  4 00:48 doexec
130086 -rwxr-xr-x   1 root root  47252 Jun 15  2004 ed
130087 -rwxr-xr-x   1 root root 253148 Jun 29  2004 pgawk
130088 -rwxr-xr-x   1 root root  25060 Oct  5 11:50 mkdir
130089 -rwxr-xr-x   1 root root  16792 Oct  5 11:50 echo
130090 -rwxr-xr-x   1 rpm  rpm   82944 Nov  1 21:54 rpm
130091 -rwxr-xr-x   1 root root  59100 Oct  5 11:50 cp
130092 -rwxr-xr-x   1 root root  15516 Oct  5 11:50 unlink
130093 -rwxr-xr-x   1 root root 161380 Oct 11 09:25 tar
130094 -rwxr-xr-x   1 root root  16556 Oct  5 11:50 rmdir
130095 -rwxr-xr-x   1 root root  26912 Oct  5 11:50 ln
130096 -rwxr-xr-x   1 root root  10804 Sep 30 08:49 hostname
130097 -rwxr-xr-x   1 root root 307488 Sep 21 17:26 tcsh
130098 -rwxr-xr-x   1 root root  15064 Aug  9 09:14 mt
130099 -rwxr-xr-x   1 root root  23072 Oct  5 11:50 mknod
130100 -r-xr-xr-x   1 root root   8972 Oct  6 10:19 mktemp
569988 -rwxr-xr-x   1 root root  76633 Jun 29  2004 ps
569990 -rwxr-xr-x   1 root root  92110 Jan 18  2004 netstat
```

The order in which the files were saved to the hard disk is clear as shown by the increasing sequence of inode numbers. It is clear we have an abnormality with the inode numbers for the *ps* and *netstat* commands.

A file's inode number will change when it is replaced with a different file. Because the Trojan was installed well after the original file the Trojan's inode number will be higher than that of the original file. Thus, a simple method of identifying Trojans is looking for inode numbers that are 'outliers,' particularly for those files that are likely to be part of a rootkit. As demonstrated above, the *ps* and *netstat* have inode numbers that are significantly out-of-sequence with the inode numbers of the other files, indicating the possibility that the original utility was replaced with a Trojan version. This is not a guarantee, unlike the hash analysis above, that the files are known Trojan horses. Regardless, further scrutiny is warranted.

Identifying Files with Renamed Extensions

A simple means of hiding a file is by renaming the file's extension. For instance, changing the file *chix.jpg* to *homework.doc* takes a file of questionable content and turns it into a file that appears innocuous. This technique can be particularly effective within Windows because Windows will display an icon that is based on the extension of a file, regardless as to whether a file's extension is a true reflection of the file's type.

As described previously, a file's type is reflected in its header (sometimes called *signature*). A file's header is a sign to applications as to how to handle the file. For instance, all modern Microsoft Office files begin with the following 8-byte signatures (in bold):

```
0000000: d0cf 11e0 a1b1 1ae1 0000 0000 0000 0000  ................
0000010: 0000 0000 0000 0000 3e00 0300 feff 0900  ........>.......
```

One way find graphic files whose extension has been changed is to combine three GNU utilities: `find`, `file`, and `grep`. The best way to explain the procedure is through a demonstration.

Step 1: Use the `find` command to find all "regular files" on the hard drive. Pipe the results of this command to the:

Step 2: `file` command, which displays the type of file based on header information. Pipe the results of this command to the:

Step 3: `grep` command to search for graphical-related keywords.

Below we combine the three utilities to identify all graphical images that have a renamed extension:

```
# find /  -type f ! \( -name '*.jpg' -o -name '*.bmp' -o -name '*.png' \) -
print0 | xargs -0 file | grep -if graphics.files
```

This is simpler to understand if partitioned into steps:

1. The / argument specifies the directory in which to start, here the root directory.

2. The flag `-type f` specifies that we are interested in regular files as opposed to special files such as devices or directories. The `find` command is recursive by default so it is essentially recursively finding all regular files beginning at the / (root) directory.

3. The exclamation mark (!) modifies the contents within the parenthesis, and indicates that we want to process files whose extension is not *.jpg, or *.png, or *.bmp, or *.tiff.

4. The `-print0` is a special formatting command that is required to format the output of `find` for piping to the next command.

5. We pipe the results – a list of files whose extension is not *.jpg, *.bmp, etc. -- to `xargs -0`, which sends each file name to the *file* command. `file` evaluates each file's signature, returning a description of the type of file.

6. These results are piped to *grep* to search for the specific keywords that are contained within the *graphics.files* file. The arguments for *grep* include –i for case insensitive search, and the –f *graphics.files*, the file containing the list of keywords: 'PNG,' 'GIF,' 'bitmap,' 'JPEG,' and 'image.'

Our search identified three files with misleading names and extensions:

```
# find / -type f ! \( -name '*.jpg' -o -name '*.bmp' -o -name '*.png' \) -
print0 | xargs -0 file | grep -if graphics.files

/var/cache/exec:           JPEG image data, JFIF standard 1.01
/var/log/ppp/0x12da2:      PC bitmap data, Windows 3.x format
/var/log/ppp/README.txt:   PNG image data,8-bit/color RGB
```

The search correctly identified three files, a *.jpg, a *.bmp, and a *.png, whose name and/or extension were changed in an effort to obfuscate their true type. This technique will work correctly as long as the files signature remains intact.

gThumb is a application that can use a file's header, as opposed to its extension, to determine the type of file. Figure 4 below demonstrates that *gThumb* is able to determine that README.txt is actually a graphical image and treats it appropriately.

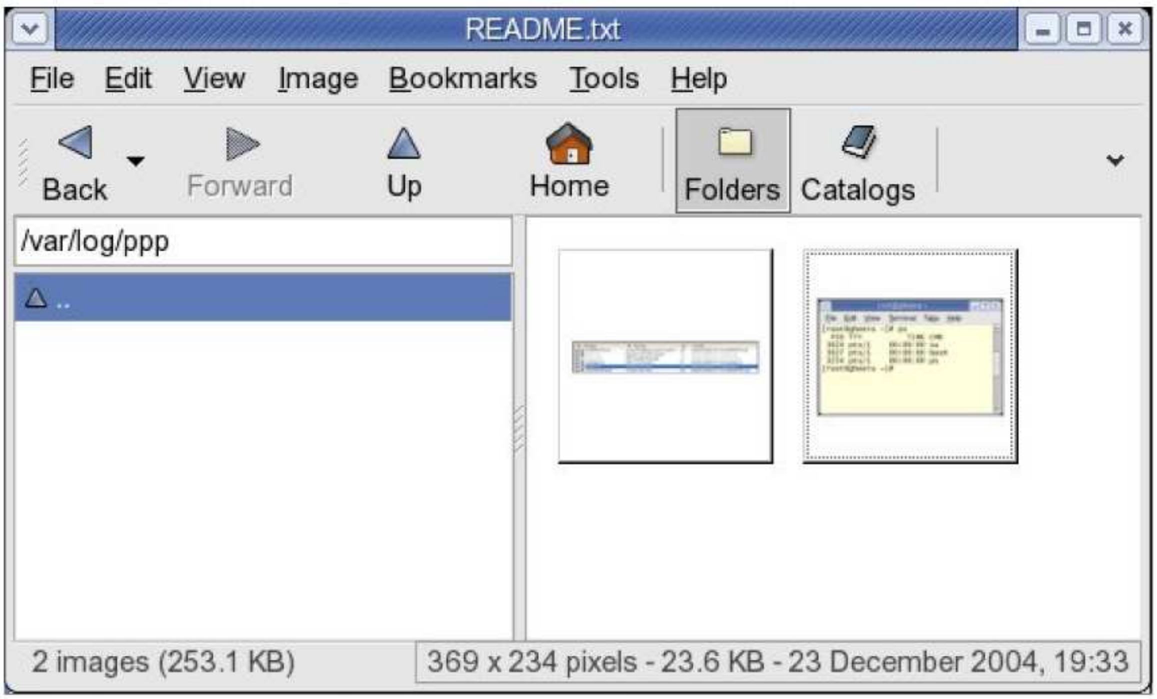

Figure 4. Files with Renamed Extensions

Identifying Hidden Files

Intruders often use the victimized computer to store files that the intruder doesn't want to store on his own computer. For instance, intruders have been known to upload pirated versions of commercial software ('warez'), illegal copies of recorded music and videos, pornography, and the like. The intruder typically creates a backdoor on the system, allowing access to the (illegal) files that cannot be traced back easily to its source.

How does an intruder upload hundreds of large files to the victim's computer without the victim noticing? The intruder uses several common methods of hiding files. For instance, a file that begins with a period (e.g., .file) is not displayed by the UNIX `ls` command. Here's an initial directory listing:

```
# ls
.   ..
```

The directory appears to be empty, only the current ('.') and parent directory ('..') are displayed. Now we request a 'long' listing that includes all files in the directory.

```
# ls -alR
.:
total 853
drwxr-xr-x   3 root root    1024 Jan 17 10:19 .
drwxr-xr-x  31 pc   users   2280 Jan 17 10:16 ..
drwxr-xr-x   2 root root    1024 Jan 17 10:20 .lotr
-rw-r--r--   1 root root  864256 Jan 17 10:19 .hackers.mpg

./.lotr:
total 2286
drwxr-xr-x  2 root root    1024 Jan 17 10:20 .
drwxr-xr-x  3 root root    1024 Jan 17 10:19 ..
-rw-r--r--  1 root root 1368064 Jan 17 10:18 .towers.mpg
-rw-r--r--  1 root root  958464 Jan 17 10:17 .hobbit.mpg
```

Adding the -a (all) -l (long) and -R (recursive) flags show all files recursively, including those that begin with a period. The owner may never be aware that an intruder is using his computer as 'free storage' as long as the intruder is careful not to upload too many files.

A user could use this manual technique to search for hidden files; however, its labor intensive, not efficient, and prone to error. There are several applications that can be used to check on changes or additions to the file system, Tripwire (www.tripwire.com) being the most famous of these applications.

Alternatively, one can use the `find` command to locate hidden files. Too illustrate: A common way to hide files is in "plain sight." For example, make a file appear similar to a known good file or directory, such as naming a directory ". ." (that's a period, a space, followed by a period). A user is likely to overlook this directory as it looks similar to the current ('.') and parent directories ('..'), and because the user is typically not 'primed' to look for hidden files.

There are dozens of possible combinations of periods and spaces that can be used to create hidden directories. The following command line finds several different combinations of periods and spaces:

```
# find / -type d -name '.[. ]*'

/var/cache/.            (AUTHORS NOTE: ONE PERIOD AND TWO SPACES)
/etc/sysconfig/...
/usr/lib/. ..
```

In this example we searched for directories (indicated by *-type d*) whose name began with a period and the remaining characters were either periods or spaces. The search located three hidden directories. The name of the hidden directory found under */var/cache* is a period followed by two spaces, which makes it look very similar to the parent directory ('.').

Finding Warez

Warez tend to be large in size (*.mpg movies, *.mp3 music files, applications, etc.). One can use the `find -size` flag to search for files larger or smaller than a specific size. In the demonstration below we search for regular files whose name begins with a period and that are larger than 1MB.

```
# find / -type f -name '.?*' -size +1000k

/opt/.. ./.gone-with-the-wind.mpg
/opt/.. ./.star-trek2.mpg
/opt/.. ./.star-trek1.mpg
/opt/.. ./.lotr/.two-towers.mpg
/opt/.. ./.lotr/.return-of-the-king.mpg
/windows/d/backups/pc/.y2log-2          (NOTE: These last three files are valid)
/windows/d/backups/pc/.y2log-1
/windows/d/backups/pc/.y2log-3
```

We can use the flags `ctime` (change time) or `cdate` (change date) flag if we know when the intruder was on the system. For instance, say we know that an intruder was on our system within the last 24 hours, and we suspect that he downloaded files. We search for files beginning with a period, and whose change time was less than an hour ago (`-ctime -1`).

```
# find / -type f -name '.?*' -ctime -1

/opt/.. ./.gone-with-the-wind.mpg
/opt/.. ./.star-trek2.mpg
/opt/.. ./.star-trek1.mpg
/opt/.. ./.hackers.mpg
/opt/.. ./.lotr/.two-towers.mpg
/opt/.. ./.lotr/.return-of-the-king.mpg
./dev/.udev.tdb                (Note: These last two files are valid system files)
./etc/lvm/.cache
```

In this example we believe that an intruder was on our computer less than an hour ago. We can search for files whose change time is less than an hour (`-ctime -1`) from the current time. (To search for files greater than an hour ago the notation would be -ctime +1. For files with changed times more than 24 hours ago we can use `-ctime +24` or `-cdate +1`.)

Future Research

The techniques described in this paper work well in identifying and recovering digital evidence for a large portion of the cases law enforcement agents will encounter. Changes in technology, particularly the growth of storage capacity, are beginning to create problems for law enforcement agencies, however. For instance, the FBI computer analysis and response team (CART) saw a **three-fold increase in cases** from 1999 to 2003; the **amount of data however increased 46-fold** [CRA05]. It is not uncommon for agents to encounter servers storing terabytes of data, equating to millions of documents, each of which is a potential piece of evidence. The critical question for law enforcement is: which of the millions of digital artifacts is probative 'evidence,' and which is not?

The techniques described in this chapter do not scale well to such tremendous data systems. Although some forensic procedures are automated – such as the hash analysis and searches – many require manual input or human interpretation. In fact, almost no conventional digital forensic techniques scale well to terabyte-sized systems. As the amount of data grows, automated procedures for identifying, recovering, and examining digital evidence will be required to process evidence in a reasonable time period.

Below we describe a taxonomy of digital artifacts that could serve as the basis for an automated system to identify probative evidence in large-scale systems. The taxonomy conceptualizes digital artifacts based on three attributes: a) the artifact's contents, b) its associated metadata, and c) ambient information. A digital artifact's values for these attributes are both digital and identifiable, that is, knowing the identifier for a digital artifact (e.g., file name or inode number) one can identify, and therefore recover, the artifact's contents, metadata, and ambient information. Consequently, it is conceivable that an automated procedure can be developed that is capable of recovering these values, obviating the need for any manual input or interpretations.

Contents

A digital artifact's contents can be separated into two sources: a) overt contents and b) latent contents. Overt content conveys information directly through perception. Examples of overt content are the text you are reading on this page, graphical images, perceptible audio, etc. In contrast, latent content is not meant to be directly perceived. Rather, latent content often serves the purpose of formatting overt content. For instance, the electronic version of this document contains several hundred kilobytes of formatting, metadata, and other information that is not meant to be directly viewed, yet serves a distinct purpose beyond the overt content. Often a digital artifact's latent information exceeds its overt contents.

Metadata

Metadata is data about data. In the case of digital artifacts, it includes the time and date of creation, modification, and last access of the file; file size; header information (i.e., the type of file); file attributes (read-only, archive, hidden, etc.); who wrote the document (e.g., contained in the properties sheet of documents created by certain applications), etc. The listing below illustrates a portion of the metadata associated with the digital version of this paper.

```
# stat IFIP.doc

  File: `IFIP. doc'
  Size: 311296          Blocks: 616         IO Block: 4096   regular file
```

```
Device: fd00h/64768d    Inode: 377747      Links: 1
Access: (0664/-rw-rw-r--)  Uid: (  500/      pc)   Gid: (  500/      pc)
Access: 2005-01-17 12:28:18.780753080 -0500
Modify: 2005-01-17 12:28:34.151416384 -0500
Change: 2005-01-17 12:28:34.153416080 -0500
```

The listing above represents file system-bound metadata including the file's size, modified, accessed, and changed date/times, inode number, and the user ID and group ID of the owner of the document. As described previously, the document itself contains metadata, including the author, the title of the document, comments, revision history, etc.

Ambient Data

Finally a digital artifact can be characterized by its ambient data; ambient meaning that which surrounds the file, either logically or physically. Ambient data may provide insight about a digital artifact just as the environmental surroundings of a crime scene may provide clues about the perpetrator of a crime. An interesting non digital example of ambient data used to solve a crime was an Arizona murder case. In this case police conducted a DNA analysis on a seed from a tree that was present at the scene of the murder and a similar seed that was found in the back of the murder suspects pickup truck. The DNA analysis indicated that the seed from the back of the suspect's truck was from the same tree from scene of the murder, placing the suspect at the scene of the crime.

Ambient information can be partitioned into the logical and the physical. Logical ambient information refers to two characteristics: a) the place in which the file resides in the file hierarchy, and b) the surrounding files in the directory. Physical ambient information involves the low level information surrounding the digital artifact, that is, at the sector level. To see how physical and logical ambient information are distinct: Files saved in short succession to a hard drive are typically saved physically contiguous to each other (if enough contiguous disk space is available). However, logically the files could be located in entirely different directories. Moreover, files in the same directory are not necessarily physically contiguous.

Identifying Digital Evidence

Once a data artifact has been catalogued according to the attributes above, an algorithmic solution may be applied to discover relationships among the values in an attempt to identify digital artifacts of potential probative value. Data mining techniques, statistical clustering, neural networks, or rule-based expert systems, could serve as the inference engine of such a system, with digital artifact's attribute values serving as the content of the system.

Statistical clustering and neural network use machine learning techniques to equate input-output pairs or to cluster similar cases. In contrast, rule-based expert systems are built upon manually-derived rules from human experts. An advantage to the machine-learning-based techniques is they automatically learn the input-output pairs (inferences) based on mathematical derivations of the differences and similarities among input-output pairs. A disadvantage of these machine-learning techniques is that the inference mechanism between input and output is mathematical in nature and doesn't easily lend itself to human language explanation (particularly problematic with neural networks which can be highly non linear).

The listing below shows four simple rule-based examples that demonstrate how digital artifacts attribute values can be used to infer a probability of the probative value of a digital artifact.

Rule Associations

RULE 1:
IF
Large Number of Files with Misnamed Extensions AND
Files are graphics AND
The files reside in a single directory AND

OR

The file are physically contiguous

THEN

Potential evidence with probability of .65

RULE 2:

IF

Large Number of Files with Misnamed Extensions AND
The files are graphics AND
The files reside in a single directory AND
Date/time stamps close to date/time stamp of cookie for illegal web site

THEN

Potential evidence with probability of .90

RULE 3:

IF

Large number of graphics files in unallocated space

THEN

Potential evidence with probability of .05

RULE 4:

IF

Large number of graphics files in unallocated space
Cookie for illegal web site

THEN

Potential evidence with probability of .35

Appendix (Adapted from [CRA05])

`dd` takes as input a stream of bits, and outputs a stream of bits, making an exact physical duplicate of a file, drive, etc. In the example below `dd` is reading from the device */dev/hda1*, a logical device associated with the 1st ATA (IDE) hard drive on the primary controller, and writing it to a file we have named *evidence.dd*. In Linux physical devices are logically associated with a file residing in the */dev* directory. These associations are shown in Table 1. Note that the names of the logical devices may differ between Linux distributions, or between versions of UNIX.

```
# dd if=/dev/hda1 of=evidence.dd conv=noerror,notrunc,sync
```

Logical Device	**Physical Device**
/dev/hda	1st IDE hard drive on primary controller
/dev/hda1	1st partition 1st hard drive on primary controller
/dev/hdb	2nd IDE hard drive on primary controller
/dev/hdc	1st IDE hard drive on secondary controller
/dev/hdd5	5th partition on 2nd hard drive on secondary controller
/dev/sda	1st SCSI device
/dev/sda1	1st partition on 1st SCSI device
/dev/cdrom	1st CDROM drive
/dev/fd0	1st floppy disk

Table 1. Mapping from logical to physical device

The argument `if=` specifies the source image, here the logical device associated with the floppy drive. The argument `of=` specifies the output file's name. The `bs=` argument specifies the block size to read and write, and is optional. The default block size is 512 bytes. The `conv` argument specifies other command line arguments to include. For imaging we include: a) `noerror`, continue after a read error; b) `notrunc`, do not truncate the output in case of an error, and c) `sync`, in case of a read error, pad input blocks with zeros.

To appear in S. Shenoi & M. Pollitt (Eds.), *Advances in Digital Forensics*. International Association of Information Professionals.

References

[BUC03] Brian Buckeye and Kevin Liston. (2003). Recovering Deleted Files in Linux. *SysAdmin*. http://www.samag.com/documents/s=7033/sam0204g/sam0204g.htm.

[CRA05a] Philip Craiger. (2005a). Computer Forensics Procedures And Methods. To appear in H. Bigdoli (Ed.), *Handbook of Information Security*. John Wiley & Sons.

[CRA05b] Philip Craiger, Mark Pollitt, & Jeff Swauger. (2005b). Digital Evidence and Digital Forensics. To appear in H. Bigdoli (Ed.), *Handbook of Information Security*. John Wiley & Sons.

[CRA99] Aaron Crane. (1999). Linux Undelete How-to http://www.praeclarus.demon.co.uk/tech/e2-undel/html/howto.html

[HER04] Hermelito Go. (2003). *UnixGuide.net*. http://www.unixguide.net/linux/faq/04.16.shtml

[PAT03] Steve Pate (2003). *UNIX Filesystems: Evolution, Design, and Implementation.* New York: Wiley

[WAR04] Trevor Warren. (2003) Exploring /proc. http://www.freeos.com/articles/2879/