Chapter 19

RECOVERING DIGITAL EVIDENCE FROM LINUX SYSTEMS

Philip Craiger

Abstract As Linux-kernel-based operating systems proliferate there will be an inevitable increase in Linux systems that law enforcement agents must process in criminal investigations. The skills and expertise required to recover evidence from Microsoft-Windows-based systems do not necessarily translate to Linux systems. This paper discusses digital forensic procedures for recovering evidence from Linux systems. In particular, it presents methods for identifying and recovering deleted files from disk and volatile memory, identifying notable and Trojan files, finding hidden files, and finding files with renamed extensions. All the procedures are accomplished using Linux command line utilities and require no special or commercial tools.

Keywords: Digital evidence, Linux system forensics

1. Introduction

Linux systems will be increasingly encountered at crime scenes as Linux increases in popularity, particularly as the OS of choice for servers. The skills and expertise required to recover evidence from a Microsoft-Windows-based system, however, do not necessarily translate to the same tasks on a Linux system. For instance, the Microsoft NTFS, FAT, and Linux EXT2/3 file systems work differently enough that understanding one tells little about how the other functions. In this paper we demonstrate digital forensics procedures for Linux systems using Linux command line utilities. The ability to gather evidence from a running system is particularly important as evidence in RAM may be lost if a forensics first responder does not prioritize the collection of live evidence.

The forensic procedures discussed include methods for identifying and recovering deleted files from RAM and magnetic media, identifying no-

tables files and Trojans, and finding hidden files and renamed files (files with renamed extensions.

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

2. Recovering Files from RAM

A deleted file whose contents have been overwritten on disk may still be recovered. To illustrate the forensic technique, say an intruder may execute a program and then delete it from disk to hide its existence. This happens, for example, when an intruder installs a backdoor on the victim system by executing the `netcat` utility, then deleting the utility from the disk. As long as the program remains as a running process in memory, the original executable 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 [6]. This information is kept in virtual memory and is accessible through the /proc directory. The (partial) listing below shows the contents of the /proc directory on a running Linux system. Each of the numbers below corresponds a processor ID, and is a directory that contains information on the running process.

```
# 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
```

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` (www.openwall.com) 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 process ID (PID) is 5288. The directory /proc/5288 will contain information regarding the running process, as displayed in the (partial) 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 ..
-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)
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
dr-xr-xr-x 3 root root 0 Jan 17 10:12 task
```

Directory /proc/5288 contains several files and directories, 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 `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.

3. Recovering Files by Type

We can manually recover a file by searching unallocated space for the file 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: (i) we can identify header information,

(ii) the file has not been overwritten, and (iii) 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. The list below 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: `ffd8ffe0`. 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.)

```
0004200:  ffd8 ffe0 0010 4a46 4946 0001 0200 0064
```

The listing above shows that the file starts at 0x4200 (hex). We convert this to decimal, 16,896, and divide by 512 (the number of bytes in sector) resulting in 33, which is the starting sector number of the file, i.e., from the beginning of the image. Under many circumstances we will not 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 33. 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=33 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 of the JPG file. The file command shows we recovered the header successfully.

This recovery method can be used with any type of file, as long as the file 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.

3.1 General File Recovery Procedure

If a file 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 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 and perform the procedures from the boot disk [2]. 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 do not 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 19:13:09 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]: Couldnt 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 17:49:33 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]: Couldnt 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.

3.2 Recovering Files from EXT2 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 Extd (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 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 ...
```

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 [1, 4] 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 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 [1, 5]). 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 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 does not 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.

3.3 Identifying Notable Files and Trojans

The two primary goals of intruders are to effectuate a break in, and to remain on the victim 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.

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 (partial) directory listing of the contents of the /bin directory, sorted by inode number (located in the first column).

```
# ls -ali /bin | sort
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
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 inode number will change when it is replaced with a different file.

Because the Trojan was installed well after the original file the Trojan 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.

3.4 Identifying Files with Renamed Extensions

A simple means of hiding a file is by renaming the file 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 extension is a true reflection of the file type. As described previously, a file type is reflected in its header (sometimes called signature). A file 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):

```
d0cf 11e0 a1b1 1ae1 0000 0000 0000 0000
```

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.

1. Use the `find` command to find all regular files on the hard drive.
2. Pipe the results of this command to the `file` command, which displays the type of file based on header information.
3. Pipe the results of this command to the `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 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 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.

4. Conclusions and Future Work

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 increases in 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 [3]. 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: (i) the artifact contents, (ii) its associated metadata, and (iii) ambient information. A digital artifact 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 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.

References

[1] B. Buckeye and K. Liston, Recovering deleted files in Linux (www.samag.com/documents/s=7033/sam0204g/sam0204g.htm), 2003.

[2] P. Craiger, Computer forensics procedures and methods, to appear in *Handbook of Information Security*, H. Bigdoli (Ed.), John Wiley, New York, 2005.

[3] P. Craiger, M. Pollitt and J. Swauger, Digital evidence and digital forensics, to appear in *Handbook of Information Security*, H. Bigdoli (Ed.), John Wiley, New York, 2005.

[4] A. Crane, Linux undelete how-to (www.praeclarus.demon.co.uk/tech/e2-undel/html/howto.html), 1999.

[5] S. Pate, *UNIX Filesystems: Evolution, Design and Implementation*, John Wiley, New York, 2003

[6] T. Warren, Exploring /proc (www.freeos.com/articles/2879/), 2003.