

The Law Enforcement and Forensic Examiner's Introduction to Linux

A Comprehensive Practitioner's Guide to Linux
as a Digital Forensics Platform



Version 4.93
April 2020

Barry J. Grundy

Contents

Legalities	7
Acknowledgements	7
Foreword	9
A word about the GNU in "GNU/Linux"	10
Why Learn Linux?	11
Where are all the GUI Tools?	12
The Hands-on Exercises	12
Conventions Used in this Document	13
1 Installation	14
1.1 Distributions	15
1.2 SLACKWARE and Using this Guide	16
1.3 Installation Methods	17
1.4 Slackware Installation Notes	17
1.5 System Users	19
1.5.1 Adding a Normal User	19
1.5.2 The Super User [root]	20
1.6 Desktop Environment	21
1.7 The Linux Kernel	22
1.8 Kernel and Hardware Interaction	23
1.8.1 Hardware Configuration	23

1.8.2	Kernel Modules	24
1.8.3	Hotplug devices and UDEV	26
2	Linux Disks, Partitions, and the File System	27
2.1	Disks	27
2.2	Device Node Assignment - Looking Closer	30
2.3	The File System	33
2.4	Mounting External File Systems	36
2.4.1	The mount Command	36
2.4.2	The File System Table (/etc/fstab)	39
2.4.3	Userspace Mounting	40
3	Basic Linux Commands	45
3.1	Very Basic Navigation	45
3.1.1	Additional Useful Commands	47
3.2	File Permissions	49
3.3	Pipes and Redirection	51
3.4	File Attributes	53
3.5	Command Line Math	55
3.5.1	bc - the Basic Calculator	55
3.5.2	Bash Shell - Arithmetic Expansion	57
3.6	Bash 'globbing'	58
3.7	Command Review and Hints	58
4	Editing with Vi	59
4.1	The Joy that is vi	59

4.2	The vimtutor Tutorial	59
4.3	vi Command Summary	60
5	The Linux Boot Sequence (Simplified)	62
5.1	Init vs. Systemd	62
5.2	Booting the Kernel	62
5.3	System Initialization	64
5.4	Runlevel	64
5.5	Global Startup Scripts	65
5.6	Service Startup Scripts	66
5.7	Bash	67
6	Configuring a Forensic Workstation	68
6.1	Securing the Workstation	68
6.1.1	Configuring Startup Services	69
6.1.2	Host Based Access Control	71
6.1.3	Host Based Firewall with iptables	74
6.2	Updating the Operating System	78
6.2.1	Slackware's pkgtools	78
6.2.2	slackpkg for automated updates	79
6.3	Installing and Updating "External" Software	81
6.3.1	Compiling From Source	81
6.3.2	Using Distribution Packages	83
6.3.3	Building Packages with SlackBuilds	84
6.3.4	Using the automated package tool sbotools	88

7 Linux and Forensics	94
7.1 Evidence Acquisition	94
7.2 Analysis Organization	95
7.3 Write Blocking	97
7.4 Examining Physical Media Information	98
7.5 Hashing Media	103
7.6 Collecting a Forensic Image with dd	104
7.6.1 dd and Splitting Images	107
7.7 Alternative Imaging Tools	110
7.7.1 dc3dd	111
7.7.2 libewf and ewfacquire	118
7.7.3 Media Errors - ddrescue	128
7.8 Imaging Over the Wire	137
7.8.1 Over the wire - dd	139
7.8.2 Over the Wire - dc3dd	141
7.8.3 Over the Wire - ewfacquirestream	142
7.9 Compression - Local and Over the Wire	145
7.9.1 Compression on the Fly with dd	146
7.10 Preparing a disk for the Suspect Image - Wiping	151
7.11 Final Words on Imaging	154
7.12 Mounting Evidence	154
7.12.1 Structure of the Image	154
7.12.2 Identifying File Systems	156
7.12.3 The Loop Device	158
7.12.4 Loop option to the mount command	158

7.12.5	losetup	159
7.12.6	Mounting Full Disk Images with losetup	161
7.12.7	Mounting Multi Partition Images with kpartx	164
7.12.8	Mounting Split Image Files with affuse	168
7.12.9	Mounting EWF files with ewfmount	173
7.13	Basic Analysis	175
7.13.1	Anti Virus - Scanning the Evidence with clamav	175
7.13.2	Basic Data Review on the Command Line	179
7.13.3	Making a List of File Types	186
7.13.4	Viewing Files	187
7.13.5	Searching All Areas of the Forensic Image for Text	191
8	Advanced (Beginner) Forensics	195
8.1	Manipulating and Parsing Files	195
8.2	Fun with dd	202
8.2.1	Data Carving with dd	203
8.2.2	Carving Partitions with dd	206
8.2.3	Reconstructing a Subject File System (Linux)	210
9	Advanced Analysis Tools	214
9.1	The Layer Approach to Analysis	215
9.2	The Sleuth Kit	217
9.2.1	Sleuth Kit Installation	219
9.3	Sleuth Kit Exercises	220
9.3.1	Sleuth Kit Exercise 1A: Deleted File Identification and Recovery (ext2)	221
9.3.2	Sleuth Kit Exercise 1B: Deleted File Identification and Recovery (ext4)	231

9.3.3	Sleuth Kit Exercise 2A: Physical String Search & Allocation Status (ext2)	236
9.3.4	Sleuth Kit Exercise 2B: Physical String Search & Allocation Status (ext4)	243
9.3.5	Sleuth Kit Exercise 3: Unallocated Extraction & Examination	246
9.3.6	SleuthKit Exercise 4: NTFS Examination - File Analysis	251
9.3.7	Sleuth Kit Exercise 5: NTFS Examination of ADS	257
9.3.8	Sleuth Kit Exercise 6: Physical String Search & Allocation Status (NTFS)	261
9.4	bulk_extractor - comprehensive searching	266
9.5	Physical Carving	274
9.5.1	scalpel	274
9.5.2	photorec	283
9.5.3	Comparing and De-duplicating Carve Output	290
9.6	Application Analysis	292
9.6.1	Registry Parsing Exercise 1: UserAssist	293
9.6.2	Registry Parsing Exercise 2: SAM and Accounts	299
9.6.3	Application Analysis: prefetch	303
10	Integrating Linux with Your Work	306
11	Conclusion	311
12	Linux Support	312
12.1	Places to go for Support	312
	List of Figures	312
	List of Command Examples	315

Legalities

All trademarks are the property of their respective owners.

© 1998-2020 Barry J. Grundy (bgrundy@linuxleo.com): This document may be redistributed, in its entirety, including the whole of this copyright notice, without additional consent if the redistributor receives no remuneration and if the redistributor uses these materials to assist and/or train members of Law Enforcement or Security / Incident Response professionals. Otherwise, these materials may not be redistributed without the express written consent of the author, Barry J. Grundy.

Acknowledgements

As always, there is no possible way I can thank everyone that deserves it. Over the years I have learned so much from so many. A blog post here, a returned email there. Help on IRC, online forums, and colleagues in the office. The contributions I receive from others in the field that take time out of their own busy days to assist me in growing as an investigator and forensic examiner, are simply too numerous to catalog. My heartfelt thanks to all.

The list of colleagues that have contributed over the many years has grown. I remain grateful to all that have given their time in reviewing and providing valuable feedback, and in some cases, simple encouragement to all versions of this guide over the years. My continued thanks to Cory Altheide, Brian Carrier, Christopher Cooper, Nick Furneaux, John Garriss, Robert-Jan Mora, and Jesse Kornblum for helping me lay the foundation for this guide. And for more recent assistance, I'd like to thank Jacques Boucher, Tobin Craig, Simson Garfinkel, Andreas Guldstrand, Bill Norton, Paul Stephens, Danny Werb, and Robby Workman.

Special thanks to Dr. Nhien An Le Khac for providing the motivation to continually update this guide over the past couple of years.

My continued thanks to the Linux Kernel, various distribution, and software development teams for their hard work in providing us with an operating system and utilities that are robust and controllable. What horrors would I be living without their dedication?

The LinuxLEO logo was designed by Laura Etter (WillowWispDesign@yahoo.com).

Finally, I cannot go without thanking my wife Jo and my sons Patrick and Tommy for the seemingly endless patience as the work was underway.

Foreword

The first version of this guide was posted to the Ohio Peace Officer's Training Academy FTP site back in 1999. Since then we've seen significant changes to our profession, and a massive growth in the development of software and techniques used to uncover evidence from an ever expanding universe of devices. The purpose of this document, however, remains unchanged. Here we look to provide an easy to follow and accessible guide for forensic examiners across the full spectrum of this forensic discipline; law enforcement officers, incident responders, and all computer specialists responsible for the investigation of digital evidence. This guide continues to provide an introductory overview of the GNU/Linux (Linux) operating system as a forensic platform.

Above all, this remains a beginner's guide. An introduction. It is not meant to be a full course on conducting forensic examinations. This document is about the tools and the concepts used to employ them. Introducing them, providing simple guidance on using them, and some ideas on how they can be integrated into a modern digital forensics laboratory or investigative process. This is also a hands on guide. It's the best way to learn and we'll cover both basic GNU/Linux utilities and specialized software through short exercises.

The content is meant to be "beginner" level, but as the computer forensic community evolves and the subject matter widens and becomes more mainstream, the definition of "beginner" level material starts to blur. This guide makes an effort to keep the material as basic as possible without omitting those subjects seen as fundamental to the proper understanding of Linux and its potential as a digital forensic platform. If you've been doing forensic examinations for five or ten years, but never delved into Linux, then this is for you. If you're a student at University and you are interested in how forensic tools are employed, but cannot afford thousands of dollars in licenses...then this is for you.

However, this is by no means meant to be the definitive "how-to" on forensic methods using Linux. Rather, it is a (somewhat extended) starting point for those who are interested in pursuing the self-education needed to become proficient in the use of Linux as an investigative tool. Not all of the commands offered here will work in all situations, but by describing the basic commands available to an investigator I hope to "start the ball rolling". I will present the commands, the reader needs to follow-up on the more advanced options and uses. Knowing how these commands work is every bit as important as knowing what to type at the prompt. If you are even an intermediate Linux user, then much of what is contained in these pages will be review. Still, I hope you find some of it useful.

GNU/Linux is a constantly evolving operating system. Distributions come and go, and there are now a number of "stand out" Linux flavors that are commonly used. In addition to balancing the beginner nature of the content of this guide with the advancing standards in forensic education, I also find myself trying to balance the level of detail required to actually teach useful tasks with the distribution specific nature of many of the commands and configurations used.

As we will discuss in further detail later in this guide, many of the details are specific to one flavor of Linux. In most cases, the commands are quite portable and will work on most any system. In other cases (package management and configuration editing, etc.) you may find that you need to do some research to determine what needs to be done on your platform of choice. The determination to provide specific details on actually configuring a specific system came about through overwhelming request for guidance. The decision to use my Linux distribution of choice for forensics as an example is personal.

Over the years I have heard from colleagues that have tried Linux by installing it, and then proceeded to sit back and wonder "what next?". I have also entertained a number of requests and suggestions for a more expansive exploration of tools and utilities available to Linux for forensic analysis at the application level. More recently, there have been numerous requests for configuration guidelines for a baseline Linux workstation. You have a copy of this introduction. Now download the exercises and drive on. This is only the start of your reading. Utilized correctly, this guide should prompt many more questions and kick start your learning. In the years since this document was first released a number of excellent books with far more detail have cropped up covering open source tools and Linux forensics. I still like to think this guide will be useful for some.

As always, I am open to suggestions and critique. My contact information is on the front page. If you have ideas, questions, or comments, please don't hesitate to email me. Any feedback is welcome.

This document is occasionally (infrequently, actually) updated. Check for newer versions (numbered on the front page) at the official site:

<https://www.LinuxLE0.com>

A word about the GNU in "GNU/Linux"

When we talk about the "Linux" operating system, we are actually talking about the GNU/Linux operating system (OS). Linux itself is not an OS. It is just a kernel. The OS is actually a combination of the Linux kernel and the GNU utilities that allow us and our hardware to interact with the kernel. Which is why the proper name for the OS is "GNU/Linux". We (incorrectly) call it "Linux" for convenience.

Why Learn Linux?

One of the questions heard most often is: "why should I use Linux when I already have *[insert Windows GUI forensic tool here]*?" There are many reasons why Linux is quickly gaining ground as a forensic platform. I'm hoping this document will illustrate some of those attributes.

In short, even if you are a seasoned mainstream operating system user, be it Mac or Windows, Linux provide an alternative and entirely unique way of approaching your forensic workflow. Not only is the environment different from what you may be accustomed to, but the *way* you work can also be a complete departure from what you are used to - if you have the patience and courage to allow it to be different. In many cases, with some up-front effort, your workflow can be exponentially more efficient.

- Control - not just over your forensic software, but the whole OS and attached hardware.
- Flexibility - boot from a CD (to a complete OS), file system support, platform support, etc.
- Power - A Linux distribution is (or can be) a forensic tool.

Another point to be made is that simply knowing how Linux works is becoming more and more important. While many of the Windows based forensic packages in use today are fully capable of examining Linux systems, the same cannot be said for the examiners.

As Linux becomes more and more popular, both in the commercial world and with desktop users, the chance that an examiner will encounter a Linux system in a case becomes more likely (especially in network investigations). Even if you elect to utilize a Windows forensic tool to conduct your analysis, you must at least be familiar with the OS you are examining. If you do not know what is normal, then how do you know what does not belong? This is true on so many levels, from the actual contents of various directories to strange entries in configuration files, all the way down to how files are stored. While this document is more about Linux as a forensic tool rather than analysis of Linux, you can still learn a lot about how the OS works by actually using it.

There is also the issue of cross-verification. A working knowledge of Linux and its forensic utility can provide an examiner with alternative tools on an alternative platform. These can be used as a method to verify the findings of other tools on other operating systems. Many examiners have spent countless hours learning and using common industry standard Microsoft Windows forensic tools. It would be unrealistic to think that reading this guide will give an examiner the same level of confidence, sometimes built through years of experience, as they have with their traditional tools of choice. What I can hope is that this guide will provide enough information to give the examiner "another tool for the toolbox", whether it's imaging, recovering, or examining evidence. Linux as an alternative forensic platform provides a perfect way to cross check your work and verify your results, even if it is not your primary choice.

We also need to consider the usefulness of Linux in academic and research applications. The open nature of Linux and the plethora of useful utilities included in a base system make it an almost tailor made platform for basic digital forensics. This is especially true in an academic environment where we find Linux provides a low cost solution to enable access to imaging tools and file examination utilities that can be used to cover the foundations of digital investigations using tools in an environment that supports multiple formats and data types. For example, we can use the **dd** program for simple imaging and carving; **grep** to locate and examine file system structures and text string artifacts, and the **file** command again with **xxd** for signature identification and analysis. This provides us with much the same set of simple tools needed to present the very basics of digital forensics while still teaching Linux command line familiarity. Linux as a forensic platform can easily provide a primary means for digital investigations education. And in fact, prior versions of this guide have been referenced in many advanced degree and law enforcement programs that teach basic digital forensics.

Where are all the GUI Tools?

As much as possible, the tools represented in this guide are callable from and require user interaction through the command line environment. This is not simple sadism. It's a matter of actually learning Linux (and in some ways UNIX as a by-product). This point will be made throughout this document, but the goal here is to introduce tools and how to interact through the command line. Reliance on GUI tools is understandable and is not being wholly disparaged here. If you are making the effort to read and follow along with this guide, then an assumption is being made that you want to learn Linux and the power the command line brings. There are two main points that we can focus on here:

The first is that Linux (and UNIX) find their foundation at the command line. Modern Linux and UNIX implementations are still, at their hearts, driven by system that is most accessible from a command line interface. For this reason, knowing how to interact with the command line provides examiners the widest range of capabilities regardless of the distribution or configuration of Linux encountered. Yes, this is about forensic tools and utilities, but it's

also about becoming comfortable with Linux. It is for this reason that we continue to learn a command line editor like **vi** and simple bit level copying tools like **dd**. There is a very high probability that any Linux/UNIX system you come across will have these tools.

The second point is that knowing and understanding the command line is, in and of itself, a very powerful tool. Once you realize the power of command pipes and flow control (using loops directly on the command line), you will find yourself able to power through problems far faster than you previously thought. Learning the proper use and power of utilities like **awk**, **sed**, and **grep** will open some powerful techniques for parsing structured logs and other data sources. This guide should provide some basic understanding of how those can be used. Once you understand and start to leverage this power, you will find yourself pining for a command line and its utilities when one is not available.

Keep these points in mind as you go through the exercises here. Understand why and how the tools work. Don't just memorize the commands themselves. That would miss the point.

The Hands-on Exercises

As with previous versions of this guide, you will see some old (but still useful) exercises remain. The output and tool usage are always refreshed to reflect the current versions of the tools used. While somewhat aging, these exercises and the files used to present them remain useful and have not been removed.

Exercises have also been added in more recent versions to cover application layer analysis tools and other recent additions to the Linux forensics arsenal. Keep in mind that while this document does cover some forensic strategies and basic fundamentals, it is really about the tools we use and the concepts behind employing them. As such some of the older exercise files may seem a bit dated but they still serve the purpose of providing a problem set on which we can learn commands regardless of the target.

Conventions Used in this Document

When illustrating a command and its output, you will see something like the following:

```
root@forensicbox:~# command
output
```

This is essentially a command line (terminal) session where...

```
root@forensicbox:~#
```

...is the command prompt, followed by the **command** typed by the user and then the command's output.

In Linux, the command prompt can take different forms depending on the environment settings. The defaults can differ across various Linux distributions, and can be further modified by the user. But in general you will normally see something like the format in our example above. This format is

```
user@hostname [present working directory]$
```

This means that we are the user **root** working on the computer (host) named *forensicbox* and our present working directory is **/root** (this is the **root** user's home directory - signified by the shorthand representation of the tilde) note that for a **root** login, the command prompt's trailing character is **#**. We've customised the **root** user's login prompt to show a red username to further highlight when **root** is logged in¹. If we log in as a regular user, the default prompt character changes to a **\$** as in the following example.

```
user@forensicbox:~$  command
output
```

This is an important difference. The **root** user is the system "superuser" or administrator. We will cover the differences between user logins later in this document.

Where you see blue ellipses ('...'), it indicates removed output for the sake of brevity or clarity:

```
root@forensicbox:~# command
...                <--removed output for brevity
output
...                <--removed output for brevity
```

¹Applying colors and other customizations to your command prompt can be done a number of ways. We use a red root prompt as a visual cue in this guide.

1 Installation

One of the old complaints about Linux has been its seeming inability to 'autodetect' modern hardware. Over the years, the development of the Linux kernel (the 'brain' of the operating system) has been quite robust. Hardware detection and configuration used to present some unique challenges for Linux novices. While issues can still occasionally arise, the fact is that setting up a Linux machine as a simple workstation is no longer the potentially frustrating exercise it once was. Automatic kernel detection of hardware has become the norm, and most distributions of Linux can be installed with a minimum of fuss on all but the most cutting edge hardware (and usually even then).

For the vast majority of computers out there, the default kernel drivers and settings will work "out of the box" for both old and new systems. The range of online help available for any given distribution is far wider now than it was even ten years ago, and most problems can be solved with a targeted Internet search. For the most part, solutions that are effective on one distribution will be effective across the board. This may not always be the case, but if you are familiar with your system, you can often interpret solutions and apply them to your particular platform.

If your Linux machine is to be a dual boot system with Windows, you can use the Windows Device Manager to record all your installed hardware and the settings used by Windows. Most of the recent versions of Linux distributions have extraordinary hardware detection, so normally this simply won't be necessary. But it still helps to have a good idea of the hardware you are using so if problems do arise your support queries can be targeted.

At a minimum, you are going to want to know and plan for:

- Hard drive partitioning
 - sizes and partition layout
- Network Configuration
 - adaptor compatibility
 - Network Management (DHCP, static, etc)

Most distributions have tons of documentation, including online help and printable manuals. Do a Web search and you are likely to find a number of answers to any question you might have about hardware compatibility issues. A list of useful Linux educational resources is provided at the end of this guide. Use them. And always remember to research first before jumping into a forum and asking questions.

1.1 Distributions

Linux comes in a number of different "flavors". These are most often referred to as a "Linux distribution" or "distro". Kernel configuration, available tools, and the package format (the software install and upgrade path) most commonly differentiate the various Linux distros.

It is common to hear users complain that device *X* works under one distribution, but not on another, etc. Or that device *Y* did not work under one version of a distribution, but a change to another "fixed it". Most often, the difference is in the version of the Linux kernel being used and therefore the updated drivers, or the patches applied by the distribution vendor, not the version of the distribution (or the distribution itself).

One thing we have seen more and more of lately are somewhat specialized distros, or in some cases, distros that are perceived as specialized. There are still your "general workstation" flavors of Linux - Arch, CentOS, Debian, Ubuntu, Slackware, Gentoo, etc., but we also have specialization now - full distributions designed and distributed specifically for a target audience like pen-testers, enterprise admins, etc.

Some examples of specialized distributions that may be of interest to readers of this document:

- [Parrot OS](#) - A security and forensics distribution.
- [SANS SIFT Workstation](#) - An advanced incident response and digital forensics distribution that is widely supported, frequently updated, and well stocked with all the tools you'll need to conduct digital triage, incident response, and digital forensic examinations.
- [Blackarch Linux](#) - Another security distribution based on Arch Linux with a massive repository of security related tools
- [Kali Linux](#) - An excellent penetration testing and security distribution.

There are many others, including other selections for security focused bootable distros and "lightweight" distros. Don't let the options confuse you, though. Find a mainstream distribution, install it and learn it.

Almost all Linux distros are perfectly suitable for use as a forensic platform. A majority of people new to Linux are gravitating toward Ubuntu as their platform of choice. The support community is huge and a majority of widely available software for Linux forensics is specifically built for and supported on Ubuntu (though not exclusively in most cases). On a personal note, I find Ubuntu less than ideal for learning Linux. This is NOT to say that Ubuntu or its variations don't make excellent forensic platforms. But this guide is focused on learning, and part of that journey includes starting with a clean slate and understanding how the operating system works and is made to suit your environment. For that we focus on a more 'manual' and Unix-like distribution.

If you are unsure where to start, will be using this guide as your primary reference, and are interested mainly in forensic applications of Linux, then I would suggest Slackware. The original commercial distribution, Slackware has been around for decades and provides a good standard Linux that remains true to the Unix philosophy. Not over-encumbered by GUI configuration tools, Slackware aims to produce the most "UNIX-like" Linux distribution available. One of my personal favorites, and in my humble opinion, currently one of the best choices for a forensic platform. (<http://www.slackware.com/>). This guide is tailored for use with a Slackware Linux installation.

One thing to keep in mind: As mentioned earlier, if you are going to use Linux as a forensic workstation, then try not to rely on GUI tools too much. Almost all settings and configurations in many Linux distributions are maintained in text files (usually in either your home directory, or in `/etc`). By learning to edit the files yourself, you avoid problems when either the X window system is not available, or when the specific GUI tool you rely on is not on a system you might need to access. In addition, knowledge of the text configuration files (where they exist) will give you insight into what is "normal", and what might have been changed when you examine a subject Linux system (though that is not the focus of this document). Learning to interpret Linux configuration files is all part of the experience.

1.2 SLACKWARE and Using this Guide

Because of differences in architecture, the Linux distribution of your choice can cause different results in commands' output and, in some cases, different behavior overall. Additionally, some sections of this document describing configuration files, startup scripts or software installation might appear somewhat different depending on the distro you select.

If you are selecting a Linux distribution for the sole purpose of learning and following along with this document, then again, I would suggest Slackware. Slackware is stable and does not attempt to enrich the user's experience with cutting edge file system hacks or automatic configurations that might hamper forensic work. Programs and binaries included with Slackware are generally left unchanged from 'upstream', meaning Slackware specific patches are not applied. This can make getting support from upstream developers a bit easier. Detailed sections of this guide on the inner workings of Linux will be written toward a basic Slackware 64 bit installation.

By default, Slackware's current installation routine leaves initial disk partitioning up to the user. There are no default schemes that result in surprising "volume groups" or other complex disk management techniques. The resulting partition table and file system table (also known as `fstab`) are entirely user driven.

Slackware Linux is stable, consistent, and simple. As always, Linux is Linux. Any distribution can be changed to function like any other (in theory). However, my philosophy has always been to start with an optimal system, rather than attempt to "roll back" a system heavily modified and optimized for the desktop rather than a forensic workstation.

1.3 Installation Methods

Download the needed bootable media files (usually in the form of an ISO), burn to a DVD or write the image to a removable drive and boot the media. This is the most common method of installing Linux. Most distros can be downloaded for free via http, ftp, or torrent. Slackware is available at <http://www.slackware.com>. Have a look at <http://distrowatch.com> for information on downloading and installing other Linux distributions.

With most modern installation routines much of the work is done for you or you are prompted for what is needed and relatively safe defaults are provided. As mentioned earlier, hardware detection has gone through some great improvements in recent years. Many (if not most) Linux distros are far easier and faster to install than other "mainstream" operating systems. Typical Linux installation is well documented online (check your specific distribution's website for more information). There are numerous books available on the subject, and most of these are supplied with a Linux distribution ready for install.

Familiarize yourself with Linux disk and partition naming conventions (covered in Chapter II of this document) and you should be ready to start.

1.4 Slackware Installation Notes

If you do decide to give Slackware a shot, here are some simple guidelines. The documentation provided on Slackware's website is complete and easy to follow. Read there first...please.

Decide on standalone Linux or dual boot. Install Windows first in a dual boot system. Determine how you want the Linux system to be partitioned. A single root partition and a single swap partition are fine. You might find it easier when first starting out to install Linux in a virtual machine (VM), either through VirtualBox or VMware for example. This will allow you to snapshot along the way and recover from any errors. It also provides you with access to community support via the host while installing your Linux system in a VM. Using Linux in a virtual machine is a perfectly acceptable way to follow this guide, and probably the easiest if you are an absolute beginner. In courses I teach, students are encouraged to use virtual machines while learning.

READ through the installation documentation before you start the process. Don't be in a hurry. If you want to learn Linux, you have to be willing to read. For Slackware, have a look through the installation chapters of the updated "Slack Book" located at <http://www.slackbook.org/beta>. There are *detailed* instructions there if you need step by step help, including partitioning, etc. For a basic understanding of how Slackware works and how to use it, the Slack Book should be your first stop.

Following is some installation advice. Read through this and then read the **Installation** section of the Slackbook linked above.

1. Boot the install media

- Read each screen carefully
- Accepting defaults works in most cases
- Your hardware will be detected and configured under most circumstances
- Online support is extensive if you have problems
- If specific hardware causes problems during the install process (not detected, not functioning, etc.) It does NOT mean it is unsupported. Complete the install if you can and troubleshoot.
- The Slackware install media boots a kernel version with support for a huge selection of hardware (the kernel is called **huge.s**). Hit the **F2** key at the **boot:** prompt for more info.
- Once the system is booted, you are presented with the keyboard map prompt followed by the **slackware login:** prompt. READ THE ENTIRE SCREEN as instructed. Login as **root**, and continue with your install routine.

2. Partition and format

- You will partition your Slackware Linux system using **fdisk** or **gdisk** (if you prefer a GPT layout). You might want to research using those prior to starting.
- This step is normally part of the installation process, or is covered in the distribution's documentation. You can partition however you like. Start with just two partitions for simplicity:
 - *root* as type "Linux"
 - *swap* as type "Swap". For the swap partition you can use 2x your system memory as a general rule for the size (up to 16GB where the performance returns are said to be limited).

3. Package Installation

- The main install routine for Slackware is started with the command **setup**. You will need to ensure that you have your disk properly partitioned before you enter the setup routine.
- Take the time to read each screen completely as it comes up.
- When asked to format the partitions, I would suggest selecting the **ext4** file system.
- When asked which packages to select for installation, it is best for to select a full installation. This allows you access to all the software that is meant to come with the system, along with multiple X Window desktop environments. For a learning box it will give you the most exposure to available software for experimentation and additionally ensures that you don't omit libraries that may be needed for software to be installed later.

4. Installation Configuration

- Understand your boot method
 - The boot loader selects the operating system to be booted
 - * Be mindful of UEFI vs. legacy BIOS options. The easiest route on a physical computer (as apposed to a virtual machine) is to set your system to legacy mode
 - * LILO (the LInuxLOader) is installed by default in Slackware, but installing GRUB is possible if desired.
 - * Normally select the option to install LILO to the master boot record (MBR) unless you are using a different boot manager in a dual boot situation.
 - * If you must use EFI, skip the LILO install and finish your installation. GRUB or eLILO can be installed for EFI after installation is complete but *before* you reboot. You should read the UEFI readme that is included on the Slackware install media's root directory before beginning.
- Create a username for yourself. Do not use the root login for normal operations.
- For more information, check the file **CHANGES_AND_HINTS.TXT** on the install media. This file is loaded with useful hints and changes of interest from one release to another.

1.5 System Users

Linux is a multi-user system. It is designed for use on networks (it is based on Unix and its original TCP/IP stack). The **root** user is the system administrator, and is created by default during installation. Exclusive use of the **root** login is DANGEROUS. Linux assumes that **root** knows what he or she is doing and allows **root** to do anything he or she wants, including destroy the system. Don't log in as **root** unless you must. Having said this, some of the work done for forensic analysis will be done as **root** to allow access to raw devices and system commands.

1.5.1 Adding a Normal User

Forensic analysis (most notably acquisitions) and basic system administration will sometimes require **root** permissions. But simply logging in as **root** and conducting your analysis, particularly from an X Window session, is not advisable. We need to add a normal user account. From there you can use **su** to log in as **root** temporarily (covered in the next section).

Slackware comes with a convenient script, **adduser**, to handle the details of setting up our additional account. Some of the items set by this script include:

- Login Name

- UID (User ID)
- Initial group and group membership
- Home directory
- User shell
- Account longevity
- Account general info (name, address, etc.)
- Initial password

For the most part, the defaults are acceptable (even the default groups - be careful not to skip this part). You invoke the script with the command **adduser** (run as **root**, obviously) and the program will prompt you for the required information. When it asks for additional groups, be sure to use the up arrow on your keyboard to display available groups. Accepting the default groups is fine for our purposes.

Once complete, you can log out completely using the **exit** command and log back in as a normal user.

1.5.2 The Super User [root]

So, we've established that we need to run our system as a normal user. If Linux gives you an error message indicating a permissions problem, then in all likelihood you need to be **root** to execute the command or edit the file, etc. You don't have to log out and then log back in as **root** to do this. Just use the **su** command to give yourself **root** permissions (assuming you know **root**'s password). Enter the password when prompted. You now have **root** privileges (the system prompt will reflect this). When you are finished using your privileges, return to your original login by typing **exit**. We can use the **whoami** command to print our current user. Here is a sample **su** session:

```
barry@forensicbox:~$ whoami
barry

barry@forensicbox:~$ /sbin/fdisk -l /dev/sda
fdisk: cannot open /dev/sda: Permission denied

barry@forensicbox:~$ su -
Password:

root@forensicbox:~# whoami
root
```

```

root@forensicbox:~# /sbin/fdisk -l /dev/sda
Disk /dev/sda: 931.5 GiB, 1000204886016 bytes, 1953525168 sectors
Disk model: Samsung SSD 850
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x7a54a740

Device      Boot      Start         End      Sectors   Size Id Type
/dev/sda1                2048        526335       524288    256M 83 Linux
/dev/sda2           526336      17303551      16777216     8G 82 Linux swap
/dev/sda3           17303552     151521279     134217728    64G 83 Linux
/dev/sda4           151521280     1953525167    1802003888   859.3G 83 Linux

root@forensicbox:~# exit
logout

barry@forensicbox:~$

```

Note that the "-" after **su** allows Linux to apply **root**'s environment (including **root**'s path) to your **su** login. So you don't have to enter the full path of a command. Actually, **su** is a "switch user" command, and can allow you to become any user (if you know the password), not just **root**. Notice that after we type **exit** as **root**, our prompt indicates that we are back to our normal user.

A word of caution: Be VERY judicious in your use of the **root** login. It can be destructive. For simple tasks that require **root** permission, use **su** and use it sparingly.

1.6 Desktop Environment

When talking about forensic suitability, your choice of desktop system can make a difference. First of all, the term "desktop environment" and "window manager" are NOT interchangeable. Let's briefly clarify the components of a common Linux GUI.

- X Window - This is the basic GUI environment. Commonly referred to as "X", it is the application that provides the GUI framework, and is NOT part of the OS. X is a client / server program with complete network transparency.
- Window Manager - This is a program that controls the appearance of windows in the X Window system, along with certain GUI behaviors (window focus, etc.). Examples are Kwin (for KDE), Metacity, XFWM, Enlightenment, i3, etc.

- Desktop Environment - A combination of Window Manager and a consistent interface that provides the overall desktop experience. Examples are XFCE, GNOME, KDE, etc.
 - The default window manager for KDE is KWin
 - The default window manager for XFCE is XFWM

These defaults can be changed to allow for preferences in speed, resource management and even workflow preferences over the desire for "eye-candy", etc. You can also elect to run a Window Manager without a desktop environment. For example, the Enlightenment Window Manager is known for it's eye-candy and can be run standalone, with or without KDE or GNOME, etc.

Slackware no longer comes with GNOME as an option, though it can be installed like any other application. During the base Slackware installation, you will be given a choice of KDE, XFCE, and some others. I would like to suggest XFCE. It provides a cleaner interface for a beginner to learn on. It is leaner and therefore less resource intensive. You still have access to many KDE utilities, if you elected to install KDE during package selection. You can install more than one desktop and switch between them, if you like. The easiest way to switch is with the **xwmconfig** command.

1.7 The Linux Kernel

The Linux kernel is the "brain" of the system. It is the base component of the Operating System that allows the hardware to interact with and manage hardware drivers and system resources.

As with all forensic tools, we need to have a clear view of how any kernel version will interact with our forensic platforms and subject hardware.

You can determine your current kernel version with the **uname** command:

```
root@forensicbox:~# uname -a
Linux forensicbox 5.4.30 #1 SMP Thu Apr 2 16:03:32 CDT 2020 x86_64 Intel(R)
Core(TM) i5-3550 CPU @ 3.30GHz GenuineIntel GNU/Linux
```

The key to the safe forensic use (from an evidence standpoint) of ANY operating system is knowledge of your environment and proper testing. Please keep that in mind. You MUST understand how your hardware and software interact with any given operating system before using it in a "production" forensic analysis. If for some reason you feel the need to upgrade your kernel to a newer version (either through automated updates or manually), make sure you read the documentation and the changelog so you have an understanding of any significant architectural, or driver changes that may impact the forensic environment.

One of the greatest strengths Linux provides is the concept of "total control". This requires thorough testing and understanding. Don't lose sight of this in pursuit of an "easy" desktop experience.

1.8 Kernel and Hardware Interaction

In this section, we will focus on the hardware configuration knowledge for baseline understanding of a sound forensic environment under current Linux distributions. We will briefly discuss hardware configuration and inventory, device node management (eudev or udev) and the desktop environment.

1.8.1 Hardware Configuration

It's always useful to know exactly what hardware is on your system. There will be times when you might need to change or select different kernel drivers or modules to make a piece of hardware run correctly. Because there are so many different hardware configurations out there, specifically configuring drivers for your system will remain outside the scope of this guide. Kernel detection and configuration of devices (network interfaces, graphics controllers, sound, etc.) is automatic in most cases. If you have any issues, make note of your hardware (see below) and do some searching. Google is your friend, and there is a list of helpful starting places for assistance at the end of this guide.

There are a number of ways to determine what specific hardware you are running on your system. You can use **lspci** to get more detailed information on specific devices attached to your system. **lspci** (list PCI devices), is for those devices specifically attached to the PCI bus. If you have hardware issues and you search for something like *"network card not detected in linux"*, and you follow a link to a support forum, you will almost always find the request to *"post the output of lspci"*. It's one of the first diagnostic steps for determining many hardware issues in Linux. This command's output can get increasingly detailed (or "verbose") by adding the options **-v**, **-vv**, or **-vvv**. Note that you can run **lspci** from the installation disk prior to running the setup program

Sample summary output for **lspci**:

```
root@forensicbox:~# lspci
00:00.0 Host bridge: Intel Corporation Haswell-ULT DRAM Controller (rev 09)
00:02.0 VGA compatible controller: Intel Corporation Haswell-ULT Integrated
Graphics Controller (rev 09)
00:03.0 Audio device: Intel Corporation Haswell-ULT HD Audio Controller (rev 09)
00:14.0 USB controller: Intel Corporation 8 Series USB xHCI HC (rev 04)
00:16.0 Communication controller: Intel Corporation 8 Series HECI #0 (rev 04)
00:1b.0 Audio device: Intel Corporation 8 Series HD Audio Controller (rev 04)
00:1c.0 PCI bridge: Intel Corporation 8 Series PCI Express Root Port 3 (rev e4)
```

```
00:1d.0 USB controller: Intel Corporation 8 Series USB EHCI #1 (rev 04)
00:1f.0 ISA bridge: Intel Corporation 8 Series LPC Controller (rev 04)
00:1f.2 SATA controller: Intel Corporation 8 Series SATA Controller 1 [AHCI mode] (
    ↪ rev 04)
00:1f.3 SMBus: Intel Corporation 8 Series SMBus Controller (rev 04)
04:00.0 Network controller: Intel Corporation Wireless 7260 (rev 6b)
```

Reading through this output you can see the fact that the network interface in this system is an Intel Corporation Wireless 7260 chipset. This is useful information if you are having issues with getting the interface to work and you want to search for support. You are far more likely to get useful help if you search for "*Linux Intel wireless 7260 not working*" rather than "*Linux network card not working*".

This brings us to the subject of kernel modules.

1.8.2 Kernel Modules

As mentioned previously, the kernel provides the most basic interface between hardware and the system software and resource management. This includes drivers and other components that are actually small separate pieces of code that can either be compiled as modules (loaded or unloaded dynamically) or compiled directly in the kernel image.

There may come a time when you find that the kernel is loading a less than ideal module for a specific piece of hardware, perhaps causing it to either fail to work, or in some cases work at less than optimal performance. Wireless network cards can be a common example.

Looking back out our previous **lspci** output, for example, recall that our output showed an Intel Corporation Wireless 7260 chipset.

If I wanted to see exactly which kernel module is being used to drive this device, I can use the **-k** option to **lspci** (output abbreviated):

```
root@forensicbox:~# lspci -k | less
...
04:00.0 Network controller: Intel Corporation Wireless 7260 (rev 6b)
    Subsystem: Intel Corporation Dual Band Wireless-AC 7260
    Kernel driver in use: iwlwifi
    Kernel modules: iwlwifi
```

This time the output provides some additional information, including which modules are loaded when the device is detected. This can be an important piece of information if I'm trying to troubleshoot a misbehaving device. Online help might suggest using a different driver altogether. If that is the case, then you may need to "blacklist" the currently loaded module in order to prevent it from loading and hindering the correct driver (that you may

need to specify). Blacklisting is normally done in `/etc/modules.d/` by either creating a `blacklist-[modulename].conf` file or making an entry in `blacklist.conf`, depending on your distribution. In Slackware, you can read the `README` file in `/etc/modules.d` and the `man` page for `modules.d` for more information. Since the steps for this vary wildly depending on the driver, its dependencies, and the existence of competing modules, we won't cover this in any more depth. Specific help for individual driver issues can be found online.

Note that if you are using a laptop or desktop with a USB wireless adapter, it likely won't show up in `lspci`. For that you'll have to use `lsusb` (*list USB* - there's a pattern here, see?). In the following output, `lsusb` reveals info about a wireless network adapter. Use the `-v` option for more verbose output (bold for emphasis):

```
root@forensicbox:~# lsusb
...
Bus 001 Device 054: ID 2109:2812 VIA Labs, Inc. VL812 Hub
Bus 001 Device 004: ID 174c:2074 ASMedia Technology Inc. ASM1074 High-Speed hub
Bus 001 Device 079: ID 1b1c:1a06 Corsair
Bus 001 Device 003: ID 046d:c077 Logitech, Inc. M105 Optical Mouse
Bus 001 Device 007: ID 11b0:6598 ATECH FLASH TECHNOLOGY
Bus 001 Device 120: ID 148f:5372 Ralink Technology, Corp. RT5372 Wireless Adapter
Bus 001 Device 005: ID 174c:2074 ASMedia Technology Inc. ASM1074 High-Speed hub
Bus 001 Device 050: ID 046d:c31c Logitech, Inc. Keyboard K120
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
...
```

Or use the script `usb-devices`, which organizes the information from `/sys/bus/usb/devices/usb` into a (mortal) human readable format. Note that it also returns the kernel module in use, much like `lspci -k` does for PCI bus devices (bold for emphasis). We use the pipe (`|`) to the `less` command to page the output for reading (more on this later):

```
root@forensicbox:~# usb-devices | less
...
T:  Bus=01 Lev=01 Prnt=01 Port=05 Cnt=05 Dev#=120 Spd=480 MxCh= 0
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P:  Vendor=148f ProdID=5372 Rev=01.01
S:  Manufacturer=Ralink
S:  Product=802.11 n WLAN
C:  #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=450mA
I:  If#= 0 Alt= 0 #EPs= 7 Cls=ff(vend.) Sub=ff Prot=ff Driver=rt2800usb
...
```

Note that the commands covered here are largely portable across distributions, but the locations of files and methods for managing modules may differ. The process of identifying modules and hardware should mostly be the same. Manual pages (read using the `man`

command) and distribution documentation should always be relied on for primary problem solving.

Keep in mind that these same commands can be run against a subject computer by using Linux based forensic boot media. If you have the time, it's a great way to inventory a subject computer either prior to seizure or if you cannot seize the computer (only image it for whatever reason), but still wish to have a full hardware inventory.²

1.8.3 Hotplug devices and UDEV

In modern distributions, Linux device management has been handed over to eudev or (or udev under Systemd)³. In the past, the device nodes (files representing the devices, located in the `/dev` directory) were static, that is they existed at all times, whether in use or not.

So why do we cover device hot-plugging in a forensic guide? A basic knowledge of device access is critical to understanding how devices (external disks, subject hard drives, etc.) are detected and presented to the examiner.

udev creates device nodes "on the fly". The nodes are created as the kernel detects the device and the `/dev` directory is populated in real time. In addition to being more efficient, udev also runs in user space. One of the benefits of udev is that it provides for "persistent naming". In other words, you can write a set of rules that will allow udev to recognize a device based on individual characteristics (serial number, manufacturer, model, etc.). The rule can be written to create a user-defined link in the `/dev` directory, so that for example, my thumb drive can always be accessed through an arbitrary device node name of my choice, like `/dev/my-thumb`, if I so choose. This means that I don't have to search through USB device nodes to find the correct device name if I have more than one external storage device connected. I can connect 4 USB devices and instead of searching through `/dev/sdc`, `sdd`, `sde`, and `sdf` - I can just go to `/dev/my-thumb`. For a nice, if somewhat outdated, explanation of udev rules, see: .

On Slackware, [e]udev runs as a daemon from the startup script `/etc/rc.d/rc.udev`. We will discuss these startup scripts in more detail later in this document. We will not do any specific configuration for udev on our forensic computers at this time. We discuss it here simply because it plays a major part in device handling and as such is of interest to forensic examiners that want to know what their system is doing. udev does NOT involve itself in auto mounting or otherwise interacting with applications. It simply provides a hardware to kernel interface. Disks and other storage media are handled by `udisks` or `udisks2`. We will cover this in more detail later.

²We will cover more 'verbose' commands for hardware inventory later

³eudev is actually a non-Systemd port of udev - I use udev in this document to avoid differentiating every time. eudev was developed by Gentoo and is used in Slackware as well as a replacement for the Systemd bound udev

2 Linux Disks, Partitions, and the File System

As you go through the following pages, please pay attention to your userid...you'll need to be root for some of this.

2.1 Disks

In Linux, devices are enumerated as files. This is an important concept for forensic examiners. It means, as we will see later on, that many of the commands we can use on regular files, we can also use on disk "files". We can list them, hash them and search them in much the same way we do files in any standard user directory. The special directory where these device "files" are maintained is `/dev`.

As we saw earlier, with the adoption of eudev and udisks, disks are now assigned device node names dynamically, meaning that the names do not exist until the device (a thumb drive, for example) is connected to the system. Of course when you boot a normally configured computer you usually have at least one "boot" drive already connected. Under most circumstances this will be named `sda`. These device nodes are populated under the `/dev` directory. The partitions are simply numbered.

When referring to the entire disk, we use `/dev/sda`. When referring to a *partition* on that disk, we use the disk name and the number of the partition, `/dev/sda1` for example.

Device	File Name
First Disk	<code>/dev/sda</code>
partition 1	<code>/dev/sda1</code>
partition 2	<code>/dev/sda2</code>
Second Disk	<code>/dev/sdb</code>
partition 1	<code>/dev/sdb1</code>
partition 2	<code>/dev/sdb2</code>
Optical Media	<code>/dev/sr0</code>

The pattern described above is fairly easy to follow. If you are using a standard SATA disk, it will be referred to as `sdx` where the `x` is replaced with an `a` for the first detected drive and `b` for the second, etc. In the same way, the CD ROM or DVD drives connected via the SATA bus will be detected as `/dev/sr0` and then `/dev/sr1`, etc.

Note that the `/dev/sdx` device nodes will include USB devices. For example, a primary SATA disk will be assigned `sda`. If you attach a USB disk or a thumb drive it will normally be detected as `sdb`, and so on.

There are some more modern storage devices with more complex naming schemes. PCIe based storage devices like NVMe include newer concepts like "name spaces" where the traditional conventions illustrated in the table above are extended beyond simple partition numbers. If you have an NVMe storage device with two partitions in a single name space, the partitions would be named as follows:

Device	File Name
First NVMe Disk / Name space	/dev/nvmen1
partition 1	/dev/nvmen1p1
partition 2	/dev/nvmen1p2

We will discuss NVMe storage media a little later on.

A simple way to see the disks and partitions that are attached to your system is to use the **lsblk** command:

```
root@forensicbox:~# lsblk
NAME MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda   8:0    0 111.8G  0 disk
├─sda1 8:1    0   64M  0 part
├─sda2 8:2    0  128M  0 part /boot
├─sda3 8:3    0    4G  0 part [SWAP]
└─sda4 8:4    0 107.6G  0 part /
sdb   8:16   0   1.8T  0 disk
└─sdb1 8:17   0   1.8T  0 part /home
sdc   8:32   1   15G  0 disk
└─sdc1 8:33   1   15G  0 part /run/media/barry/Evidence
sr0   11:0   1  1024M  0 rom
```

You can see from the output that disks and partitions are listed, and if any of the partitions are mounted, **lsblk** will also give us the current mount point. In this case we see **/dev/sda2** is mounted on **/boot**, **/dev/sda3** is our swap partition, **/dev/sda4** is our root partition, and we have **/dev/sdc1** mounted as **/run/media/barry/Evidence**. The latter volume is from an external device, plugged in and mounted via the desktop.

Another option for disk information is **lsscsi**. Although **lsscsi** does not show partitions, it does give more information about the actual media:

```
root@forensicbox:~# lsscsi
[0:0:0:0] disk ATA INTEL SSDSC2CT12 300i /dev/sda
[2:0:0:0] disk ATA Hitachi HDS72302 A5C0 /dev/sdb
[3:0:0:0] cd/dvd HL-DT-ST DVDROM GH24NS90 IN01 /dev/sr0
[8:0:0:0] disk hp v125w 1.00 /dev/sdc
```

You can see in the output above that this particular system has a USB device and two internal drives. One drive is an INTEL SSD, the other a Hitachi hard disk. There are also a DVD drive (`/dev/sr0`,) and an HP v125w thumb drive (`/dev/sdc`). **ls SCSI** does not come on most platforms by default (although it does on Slackware). If your system does not have it by default, check your distribution's package manager and install it.

There are other names, using links, that can access these device nodes. If you explore the `/dev/disk` directory you will see links that provide access to the disk devices through volume labels, disk UUID, kernel path, etc. These names are useful to us because they can be used to access a particular disk in a repeatable manner without having to know what device node a disk will be assigned (`/dev/sdc` or `/dev/sdd` for example). For now, just be aware that you can access a disk by a name other than the simple **sdx** assigned node. Also note that some of the assigned nodes might not yet have media attached. In many cases media readers can be detected and assigned nodes before media is inserted.

Now that we have an idea of what our disks are named, we can look at the partitions and volumes. The **fdisk** program can be used to create or list partitions on a supported device. This is an example of the output of **fdisk** on a Linux workstation using the "list" option (`-l` [dash "el"]):

```
root@forensicbox:~# fdisk -l /dev/sda
Disk /dev/sda: 111.8 GiB, 120034123776 bytes, 234441648 sectors
Disk model: INTEL SSDSC2CT12
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 08FD4B34-555D-4CFF-A32A-0218C1287428
```

Device	Start	End	Sectors	Size	Type
/dev/sda1	2048	133119	131072	64M	BIOS boot
/dev/sda2	133120	395263	262144	128M	Linux filesystem
/dev/sda3	395264	8783871	8388608	4G	Linux swap
/dev/sda4	8783872	234441614	225657743	107.6G	Linux filesystem

fdisk -l /dev/sdx gives you a list of all the partitions available on a particular drive. Each partition is identified by its Linux name. The beginning and ending sectors for each partition are given. The number of sectors per partition are displayed. Finally, the partition type is displayed.

Note that the output of **fdisk** will change depending on the disklabel type of the media being queried. The above output shows a disk with a GPT label. If you have a standard DOS style MBR, the output will show slightly different fields. For native handling of GPT partition labels, you can use **gdisk**. Here is the output of **gdisk** on the same drive:

```
root@forensicbox:~# gdisk -l /dev/sda
```

```
GPT fdisk (gdisk) version 1.0.4
```

```
Partition table scan:
```

```
  MBR: protective
  BSD: not present
  APM: not present
  GPT: present
```

```
Found valid GPT with protective MBR; using GPT.
```

```
Disk /dev/sda: 234441648 sectors, 111.8 GiB
```

```
Model: INTEL SSDSC2CT12
```

```
Sector size (logical/physical): 512/512 bytes
```

```
Disk identifier (GUID): 08FD4B34-555D-4CFF-A32A-0218C1287428
```

```
Partition table holds up to 128 entries
```

```
Main partition table begins at sector 2 and ends at sector 33
```

```
First usable sector is 34, last usable sector is 234441614
```

```
Partitions will be aligned on 2048-sector boundaries
```

```
Total free space is 2014 sectors (1007.0 KiB)
```

Number	Start (sector)	End (sector)	Size	Code	Name
1	2048	133119	64.0 MiB	EF02	BIOS boot partition
2	133120	395263	128.0 MiB	8300	Linux filesystem
3	395264	8783871	4.0 GiB	8200	Linux swap
4	8783872	234441614	107.6 GiB	8300	Linux filesystem

BEFORE FILE SYSTEMS ON DEVICES CAN BE USED, THEY MUST BE MOUNTED!

Any file systems on partitions you define during installation will be mounted automatically every time you boot.

Even when not mounted devices can still be written to. Simply not mounting a file system does not protect it from being inadvertently changed through your actions or via mechanisms outside your control. While general forensic processes are outside the scope of this guide, it should be common practice to test your procedures and document every action.

2.2 Device Node Assignment - Looking Closer

Another common question arises when a user plugs a device in a Linux box and receives no feedback on how (or even if) the device was recognized. One easy method for determining how and if an inserted device is registered is to use the **dmesg** command.

For example, if I plug a USB thumb drive into a Linux computer I may well see an icon appear on the desktop for the disk. I might even see a folder open on the desktop allowing me to access the files automatically (undesirable behaviour for a forensic workstation). If

I'm at a terminal and there is no X desktop, I may get no feedback at all. I plug the disk in and see nothing. I can, of course, run the **ls SCSI** command to see if my list of media refreshed. But I may want more info than that.

So where can we look to see what device node was assigned to our disk (**/dev/sdc**, **/dev/sdd**, etc.)? How do we know if it was even detected? Again, this question is particularly pertinent to the forensic examiner, since we may likely configure our system to be a little less "helpful" in automatically opening folders, etc.

Plugging in the thumb drive and immediately running the **dmesg** command provides the following output (abbreviated for readability):

```
root@forensicbox:~# dmesg
...
[2382040.400775] usb 1-4.1: new high-speed USB device number 4 using xhci_hcd
[2382040.491674] usb 1-4.1: New USB device found, idVendor=03f0, idProduct=3307,
    ↪ bcdDevice=10.00
[2382040.491677] usb 1-4.1: New USB device strings: Mfr=1, Product=2, SerialNumber
    ↪ =3
[2382040.491679] usb 1-4.1: Product: v125w
[2382040.491681] usb 1-4.1: Manufacturer: hp
[2382040.491683] usb 1-4.1: SerialNumber: 002354C611A8AC3162CF005C
[2382040.492327] usb-storage 1-4.1:1.0: USB Mass Storage device detected
[2382040.492501] scsi host8: usb-storage 1-4.1:1.0
[2382044.130418] scsi 8:0:0:0: Direct-Access hp v125w 1.00 PQ:
    ↪ 0 ANSI: 4
[2382044.132140] sd 8:0:0:0: [sdc] 31334400 512-byte logical blocks: (16.0 GB/14.9
    ↪ GiB)
[2382044.133608] sd 8:0:0:0: [sdc] Write Protect is off
[2382044.133610] sd 8:0:0:0: [sdc] Mode Sense: 2f 00 00 00
[2382044.135212] sd 8:0:0:0: [sdc] Write cache: disabled, read cache: enabled,
    ↪ doesn't support DPO or FUA
[2382044.141413] sdc: sdc1
[2382044.146351] sd 8:0:0:0: [sdc] Attached SCSI removable disk
```

The important information is in bold. Note that this particular thumb drive (an HP v125w) provides a single volume with a single partition (**/dev/sdc1**). The **dmesg** output can be long, so you can pipe through **less** (**dmesg | less**) or scroll through the output if needed.

You can also follow the output of **dmesg** in real time by watching the output of **/var/log/messages** with **tail -f**, which essentially means "watch the tail of the file and follow it as it grows". Start the following command and *then* plug in a USB device. You'll see the messages as the kernel detects it.

```
root@forensicbox:~# tail -f /var/log/messages
...<plug in a device and watch the kernel messages>
```

We mentioned NVMe storage media before and how they have a different naming scheme as a result of 'name spaces' and other architectural changes. On a system with NVMe media installed you might see the following using **dmesg**:

```
root@forensicbox:~# dmesg
...
[ 0.677521] nvme nvme0: pci function 0000:3d:00.0
[ 0.904343] nvme0n1: p1 p2
...
```

The above dmesg output shows a single NVMe storage device with a single namespace containing two partitions.

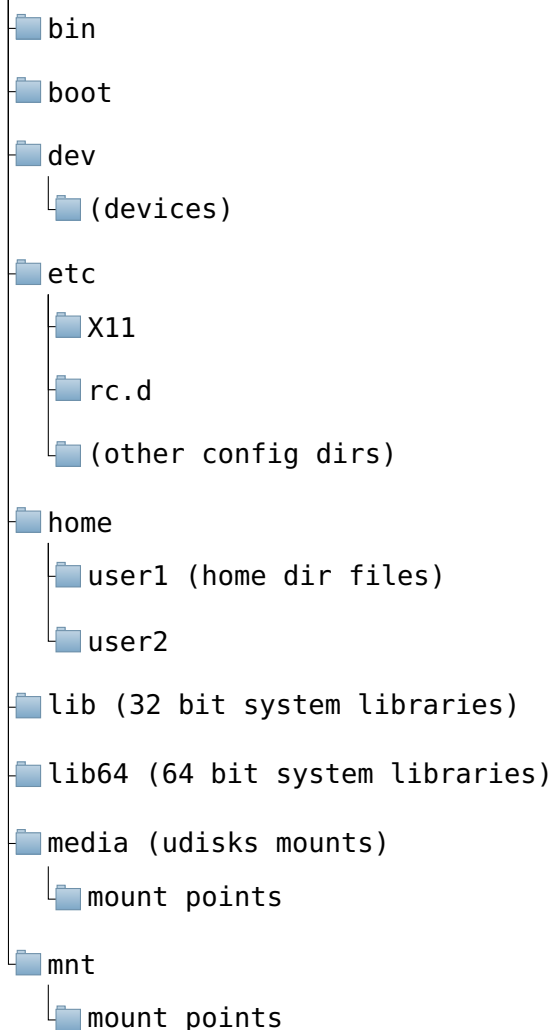
This section covered the identification of devices detected by the Linux kernel. We will discuss collecting information about these devices in later sections.

2.3 The File System

Like the Windows file system, the Linux file system is hierarchical. The "top" directory is referred to as "the root" directory and is represented by `/`. Note that the following is not a complete list, but provides an introduction to some important directories.

On most Linux distributions, the directory structure is organized in the same manner. Certain configuration files and programs are distribution dependent, but the basic layout is similar to this. Note that the directory "slash" (`/`) is opposite what most people are used to in Windows.

`/` (root of the tree - not to be confused with `/root` (root's home))



```
/ (root - continued)
├── opt (software installed here on some systems)
├── root (root's home)
├── run
│   └── media (dynamic udisks2 mount points)
├── sbin (system binaries)
├── usr
│   ├── local
│   └── lib
└── var (logging)
```

Directory contents can include:

- **/bin**
 - Common commands - **ls**, **cd**, etc.
- **/boot**
 - Files needed at boot - including the kernel images pointed to by LILO (the LInux LOader) or GRUB.
- **/dev**
 - Files that represent devices on the system. These are the device nodes.
- **/etc**
 - Administrative configuration files and scripts.
- **/home**
 - User's home directories. Each user directory can be extended by the respective user and will contain their personal files as well as user specific configuration files (for X preferences, etc.).
- **/lib**
 - 32 bit software libraries
- **/lib64**
 - 64 bit software libraries

- `/media`
 - Provides a standard place for system wide removable media.
- `/mnt`
 - Provides temporary mount points for external, remote and removable file systems. This is the legacy directory for manually mounting volumes
- `/opt`
 - Directory for external or "optional" software. Some programs (Google Chrome, for example) will install into this directory.
- `/root`
 - The root user's home directory.
- `/run`
 - Dynamic run time files for system daemons like `[e]udev` and `udisks`. This directory is where you might find external volumes mount from the desktop or via any `udisks2` interface.
- `/sbin`
 - Administrative and system commands. (**`fdisk`**, **`ifconfig`**, etc.)
- `/usr`
 - Contains locals software, libraries, games, etc.
- `/var`
 - Logs and other variable files will be found here.

Another important concept when browsing the file system is that of relative versus explicit paths. While confusing at first, practice will make the idea second nature. Just remember that when you provide a path name to a command or file, including a `"/` in front means an explicit path, and will define the location starting from the top level directory (root). Beginning a path name without a `"/` indicates that your path starts in the current directory and is referred to as a relative path. More on this later.

One very useful resource for this subject is the File System Hierarchy Standard (FHS), the purpose of which is to provide a reference for developers and system administrators on file and directory placement. Read more about it at <http://www.pathname.com/fhs/>.

2.4 Mounting External File Systems

There is a long list of file system types that can be accessed through Linux. This is accomplished using the **mount** command. Linux has a couple of special directories used to mount file systems to the existing Linux directory tree. One directory is called **/mnt**. It is here that you can *manually* attach new file systems from external (or internal) storage devices that were not mounted at boot time. Typically, the **/mnt** directory is used for temporary mounting. Other available directories are **/media**, and **/run/media** which provide a standard location for users and applications to mount removable media. Actually you can mount file systems anywhere (not just on **/mnt**, but it's better for organization. Since we will be dealing with mostly temporary mounting of potential evidence volumes, we will use the **/mnt** directory for most of our work, but there are other equally valid options. This provides a brief overview.

For manual mounting using the **mount** command, anytime you specify a mount point you must first make sure that that directory exists. For example to mount a USB disk under **/mnt/evidence** you must be sure that **/mnt/evidence** exists. After all, suppose we want to have a CD ROM and a USB drive mounted at the same time? They can't both be mounted under **/mnt** (you would be trying to access two file systems through one directory!). So we create directories for each device's file system under the parent directory **/mnt**. You decide what you want to call the directories, but make them easy to remember. Keep in mind that until you learn to manipulate the file **/etc/fstab** (covered later), only **root** can mount and unmount file systems (manually and explicitly).

Newer distributions usually create mount points for you, but you might want to add others for yourself (mount points for subject disks or images, etc. like **/mnt/data** or **/mnt/analysis**). Note that you must be **root** to create mount points in **/mnt**:

```
root@forensicbox:~# mkdir /mnt/analysis
```

2.4.1 The **mount** Command

The **mount** command uses the following syntax:

```
mount -t <filesystem> -o <options> <device> <mountpoint>
```

One of the options we pass to the **mount** command, using **-t**, is the file system type ⁴. But what if you don't know what file system is on a device you've been handed? First, we need to know the partition layout of the device. Is there one partition? Two? Once we've

⁴Actually, modern Linux systems do a pretty decent job of auto detecting file system types, but being explicit is never a bad thing.

selected the partition we want to work with, we need to know what file system might be on there. We can accomplish this with using a series of commands we've already covered in the earlier chapter on disks and disk naming conventions. We use the **lsblk** command to view our devices that have been detected, and show the partitions that we might try to mount. Then we use the **file** command with the **-s** option to determine the file system type we will be mounting. For example, if I insert a thumb drive into my system and I want to manually mount it, I can use the following commands to gather the information I need:

```
root@forensicbox:~# lsblk
```

NAME	MAJ:MIN	RM	SIZE	RO	TYPE	MOUNTPOINT
sdb	8:16	1	119M	0	disk	
└─sdb1	8:17	1	118M	0	part	
sda	8:0	0	1.8T	0	disk	
└─sda2	8:2	0	1.8T	0	part	/home
└─sda1	8:1	0	128M	0	part	/boot
nvme0n1	259:0	0	953.9G	0	disk	
└─nvme0n1p1	259:1	0	64G	0	part	
└─nvme0n1p2	259:2	0	889.9G	0	part	/

```
root@forensicbox:~# file -s /dev/sdb1
```

```
/dev/sdb1: Linux rev 1.0 ext4 filesystem data , UUID=22e4d5cc-7713-4b17-b2df-11b17a73b954, volume name "Win10Image" (extents) (large files) (huge
↪ files)
```

The pertinent output is highlighted in red. **lsblk** shows us that the drive was detected as **/dev/sdb** with a single partition at **/dev/sdb1**. The **file** command reads the signature of the partition and determines it is an EXT4 file system. We will discuss the **file** command extensively later in this guide. For now, just understand that it determines the type of file by its signature (regardless of extension or name), in this case a file system signature. Note the last block device listed in the **lsblk** output is an NVMe device with a single namespace and two partitions.

We can then use that information to mount the drive (this command assumes the directory **/mnt/analysis** exists – if not then create it with **mkdir**):

```
root@forensicbox:~# mount -t ext4 /dev/sdb1 /mnt/analysis
```

Now change directory to the location of the newly mounted file system:

```
root@forensicbox:~# cd /mnt/analysis
```

You should now be able to navigate the thumb drive as usual. Essentially, what we have done here is take the logical contents of the file system on **/dev/sdb1** and made it available to the user through **/mnt/analysis**. You can now browse the contents of the disk.

When you are finished, leave the `/mnt/analysis` directory (if you do the `cd` command by itself, you will return to your home directory), and unmount the file system with:

```
root@forensicbox:~# umount /mnt/analysis
```

Some points to keep in mind:

- Note the proper command is **umount** not **unmount**. This cleanly unmounts the file system. This makes it safe to remove the media from the system.
- If you get an error message that says the file system cannot be unmounted because it is busy, then you most likely have a file open from that directory, or are using that directory from another terminal. Check all your terminals and virtual terminals and make sure you are no longer in the mounted directory.

Another example: Reading an optical disk

- Insert the optical media
- Many optical disks will use the ISO9660 file system. You can check this with the **file** command on the optical drive (in this case `/dev/sr0` with the media inserted:

```
root@forensicbox:~# file -s /dev/sr0
```

- Now we can mount the media and change to the newly mounted file system:

```
root@forensicbox:~# mount -t iso9660 /dev/sr0 /mnt/cdrom
mount: /dev/sr0 is write-protected, mounting read-only
```

```
root@forensicbox:~# cd /mnt/cdrom
```


```
root@forensicbox:~# ls
autorun.inf*  document/  installmanager/  menu/  tools/
```

- You can now navigate the optical media volume as needed. When finished, leave the `/mnt/cdrom` directory (change to your home directory again with `cd` or `cd ~`) and unmount the file system with:

```
root@forensicbox:~# umount /mnt/cdrom
```

If you want to see a list of file systems that are currently mounted, just use the **mount** command without any arguments or parameters. It will list the mount point and file system type of each device on system, along with the mount options used (if any). Note in the output below you can see the optical disk I just mounted (and did not unmount):


```
root@forensicbox:~# mount
/dev/sda3 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
tmpfs on /dev/shm type tmpfs (rw)
/dev/sda1 on /boot type ext4 (rw)
/dev/sr0 on /mnt/cdrom type iso9660 (ro)
```

Alternatively, you can query `/proc/mounts`. Where the **mount** command is actually displaying the contents of `/etc/mtab`, `/proc/mounts` is actually more up to date. The `/proc` file system on Linux is a virtual hierarchical display of system processes and information. Use **cat /**  **proc/mounts** to view the output.

The ability to mount and unmount file systems is an important skill in Linux. We use it to view the contents of a file system, and we use it to mount external storage for collecting evidence files, etc. There are many options that can be used with **mount** (some we will cover later), and a number of ways the mounting can be done easily and automatically. Refer to the **info** or **man** pages for the **mount** command for additional information.

In most modern distributions, optical disks will be auto-detected, and an icon placed on the desktop for it. Of course this will depend on your particular setup, but this is generally the case whether you are using XFCE, KDE, Unity, or some other combination of GUI and window manager. We'll cover more on GUI disk handling in an upcoming section.

2.4.2 The File System Table (`/etc/fstab`)

It might seem like **mount -t iso9660 /dev/sr0 /mnt/cdrom** is a lot to type every time you want to mount an optical disk. One way around this is to edit the file `/etc/fstab` ("file system table"). This file allows you to provide defaults for your mountable file systems, thereby shortening the commands required to mount them. The system also uses `/etc/`  **fstab** to mount default file systems when the computer starts. My `/etc/fstab` looks like this:

```
root@forensicbox:~# cat /etc/fstab'
/dev/sda2  swap          swap          defaults      0      0
```

/dev/sda3	/	ext4	defaults	1	1
/dev/sda1	/boot	ext4	defaults	1	2
/dev/cdrom	/mnt/cdrom	auto	noauto,owner,ro	0	0
devpts	/dev/pts	devpts	gid=5,mode=620	0	0
proc	/proc	proc	defaults	0	0
tmpfs	/dev/shm	tmpfs	defaults	0	0

The columns are:

<device>	<mount point>	<file system type>	<default options>
----------	---------------	--------------------	-------------------

With this in the `/etc/fstab`, I can mount optical media by simply issuing the command:

```
root@forensicbox:~# mount /mnt/cdrom
```

The above **mount** commands look incomplete. When not enough information is given, the **mount** command will look to `/etc/fstab` to fill in the blanks. If it finds the required info, it will go ahead with the mount. To find out more about available options for `/etc/fstab`, enter **info fstab** at the command prompt. After installing a new Linux system, have a look at `/etc/fstab` to see what is available for you. If what you need isn't there, add it. In this particular case, the entry for `/dev/cdrom` (a symbolic link or 'shortcut' to `/dev/sr0`) was uncommented by removing the `#` symbol from the front of the line.

2.4.3 Userspace Mounting

Mounting can also take place via automated or partially automated processes through your desktop environment. Linux has a huge list of available choices in desktop systems and management (XFCE, KDE, Gnome, Mate, etc.). They all have the capability to handle and mount removable devices for the user. This is normally done through the dynamic addition of context capable desktop icons that may appear when removable media is plugged in. Volumes can then be mounted via a right-click menu.

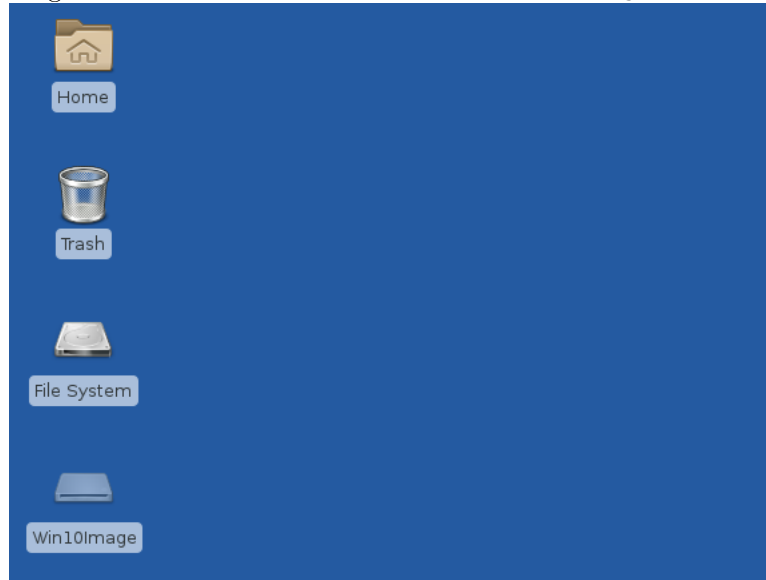
There are a number of useful changes for the general Linux user that makes this sort of desktop capable mounting make sense. First, for general daily use as a desktop workstation, who wants to have to log in as root to mount external devices? What if you are working on a system that you don't have elevated privileges on? In addition to the personal logistics, there's also the fact that the more modern mounting systems will place removable device mount points to a user's personal space rather than a system wide mount point. This offers better security and accessibility for the user.

The following example will show what can happen on an XFCE desktop when a USB drive is inserted. This is just an illustration. Be sure to check your own system for default configurations that might differ from this. You certainly don't want to accidentally mount evidence just because you were unaware the system is doing it for you.

In this case the USB disk has a partition with a volume label **Win10Image** (the volume label can be set by any number of tools when the file system is formatted).

With the USB drive inserted, an icon appears on the desktop.

Figure 1: XFCE with USB volume **Win10Image** inserted



Back in the earlier section on disks and device nodes, we talked about device detection and naming. In addition to the `/dev/sdx` naming, there are other names assigned to the disk by UUID, label, and kernel path. When I see the **Win10Image** label appear on the desk top, a terminal can quickly be opened to see exactly what partition on which disk that label belongs to by accessing the `/dev/disk/` sub-folders, specifically `/dev/disk/by-label`:

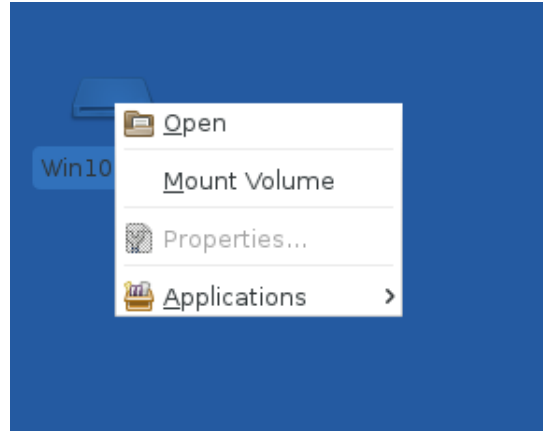
Using `ls -l` we see that the file `/dev/disk/by-label/Win10Image` is a symlink, or shortcut to `/dev/sdb1`.

```
root@forensicbox:~# ls -l /dev/disk
total 0
drwxr-xr-x 2 root root 140 Apr 16 18:28 by-id/
drwxr-xr-x 2 root root  60 Apr 16 18:28 by-label/
drwxr-xr-x 2 root root  80 Apr 16 18:28 by-partlabel/
drwxr-xr-x 2 root root  80 Apr 16 18:28 by-partuuid/
drwxr-xr-x 2 root root  80 Apr 16 18:28 by-path/
drwxr-xr-x 2 root root  80 Apr 16 18:28 by-uuid/

root@forensicbox:~# ls -l /dev/disk/by-label/
total 0
lrwxrwxrwx 1 root root 10 Apr 16 18:28 Win10Image -> ../../sdb1
```

If we right click on the icon and select **Mount Volume** from the menu, the volume is mounted on `/run/media/$USER/$LABEL`. In this case the user is **barry** and the label is **Win10Image**.

Figure 2: Right-click context menu for disk mounting [XFCE]



Once mounted, we can see the results from the terminal using the **mount** command:

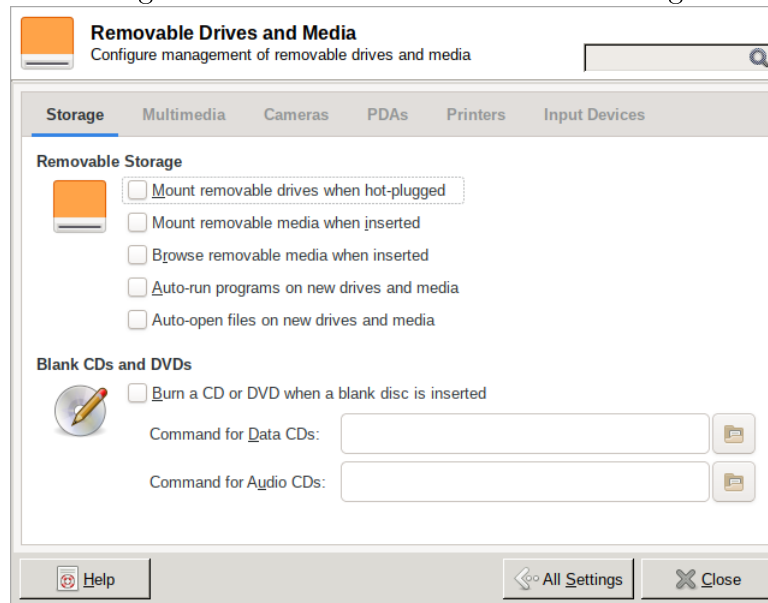
```
root@forensicbox:~# mount
/dev/sda1 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
tmpfs on /dev/shm type tmpfs (rw)
/dev/sdb1 on
/run/media/barry/Win10Image type ext4(rw,nodev,nosuid, uhelper=udisks2)
```

Make sure you know how to control the mounting of disks and volumes within your desktop environment. The XFCE shipped with Slackware does no auto-mounting of any volumes by default. The icons appear on the desktop, but you are free to mount them as you see fit. There are configuration options available to change this behavior, so be careful. Make sure you test what happens when you "hot plug" USB media or other removable storage. For example, some distributions might elect to auto-mount devices on the GUI desktop immediately upon insertion. This is undesirable for digital forensics.

XFCE is a lighter weight (meaning lighter on resources) desktop. And although XFCE is also capable of automatically handling hot plugged devices, it allows for easy control of removable media on the desktop (as do other desktop systems to be sure. I am Just less familiar with them). Consider the following snapshot of an XFCE settings dialog for removable media. By default, on Slackware, devices are NOT auto mounted in the XFCE environment. Not all distributions might be configured this way, however. Be sure to check and test for yourself. As a forensic examiner, you do NOT want your system automatically mounting devices simply because you plugged them into the system. We will cover this in a bit more detail in the second chapter.

You might notice that we need to be logged in as **root** to use the **mount** command to manually

Figure 3: XFCE Removable Media Dialog



mount a volume to `/mnt` (or any other directory we create). The same applies when we try to mount a volume manually that is not defined in `/etc/fstab`. On the other hand, when we insert media and use the desktop icon to mount the device, we can be logged in as a normal user and simply right click on the device icon to have it mount to `/run/media/XXX`. So how is that happening?

Userspace mounting on most Linux systems is handled by the `udisks` package. There are two versions of this: `udisks` and `udisks2` (the newer version). For the sake of simplification, we'll say that if your system is using `udisks` (version 1), it will normally mount to `/media`. If your system is using `udisks2`, it will normally mount to `/run/media/USER`. As we've seen, this userspace mounting can be done via the desktop using the mouse.

What happens if you are at a terminal without a GUI, or if you are using a desktop/window manager that does not allow for dynamic volume icons? You can still use the userspace capabilities of `udisks` on the command line. The following command session shows the use of the `udisks2` command line utility `udisksctl` by a normal user to mount a volume. Note that `udisksctl` mounts to `/run/media/USER` (it's a `udisks2` utility).

```
barry@forensicbox:~$ lsblk
NAME                MAJ:MIN RM  SIZE RO TYPE  MOUNTPOINT
sdb                  8:16   1  14.3G  0 disk
└─sdb1               8:17   1  14.3G  0 part
...

barry@forensicbox:~$ udisksctl mount -b /dev/sdb1
Mounted /dev/sdb1 at /run/media/barry/SANDISK.
```

```
barry@forensicbox:~$ lsblk
NAME                MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINT
sdb                  8:16    1  14.3G  0 disk
└─sdb1               8:17    1  14.3G  0 part  /run/media/barry/SANDISK
...
```

The above session shows the output of **lsblk** to identify the volume we want to mount. The **udisksctl** command with the **mount** and **-b /dev/sdb1** options is then run and the resulting successful mount is displayed. (**-b** specifies the 'block' device to mount). We then use **lsblk** again, this time showing the volume **/dev/sdb1** and its mount point in **/run/media**. Notice that we are only providing a volume to mount, we do not provide a mount point. **udisks** will use the volume label to create a dynamic mount point under **/run/media/USER**. If there is no volume name, then it uses the volume UUID.

We unmount the volume using **udisksctl** as well (and then re-check with **lsblk** just for good measure):

```
user@forensicbox:~$ udisksctl unmount -b /dev/sdb1
Unmounted /dev/sdb1.
```

```
barry@forensicbox:~$ lsblk
NAME                MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINT
sdb                  8:16    1  14.3G  0 disk
└─sdb1               8:17    1  14.3G  0 part
...
```

In this section we've covered enough about dealing with disks to get you started on accessing storage media and volumes. Next, we'll delve into actually being able to manipulate and view some data with an introduction to basic commands.

3 Basic Linux Commands

3.1 Very Basic Navigation

Here we are going to go over the basic commands to allow you to navigate the command line. There are plenty of resources out there to learn the command line. This should just provide a start to get you through the guide.

We'll start with some basic navigation.

Directory Listings:

ls	list the contents of a directory
ls -a	list the contents of a directory, including hidden (dot) files
ls -l	detailed listing
ls -lh	detailed listing with human readable sizes
ls -R	list the contents of a directory recursively

Command options can be combined, so that if we use **ls -lhaR** we'll get a detailed listing of files, with sizes in 'human readable' format, including hidden (or 'dot' files), recursively.

A simple **ls -lah** will give us the following:

```
barry@forensicbox:~$ ls -lah
total 52K
drwxr-xr-x 2 barry users 4.0K Jun  3 14:20 Code/
drwxr-xr-x 2 barry users 4.0K Jun  3 14:20 Documents/
drwxr-xr-x 2 barry users 4.0K Jun  3 14:20 Samples/
drwxr-xr-x 2 barry users 4.0K Jun  3 14:20 Utilities/
drwxr-xr-x 2 barry users 4.0K Jun  3 14:20 Videos/
-rw-r--r-- 1 barry users 6.4K Jun  3 14:21 config.txt
-rw-r--r-- 1 barry users 21K Jun  3 14:21 sample.doc
```

We will discuss the columns in the **ls** output in more detail later on, though some like - file size and file name - are obvious.

Changing Directories:

cd <dir>	Change to directory <dir>
cd	(without arguments) Shortcut to your home directory
cd ..	Change to the parent directory of the current directory (on level up)

cd -	Change to the last directory you were in
cd /dirname	change to the specified directory. Note that the addition of the "/" in front of the directory implies an explicit (absolute)path, not a relative one. With practice, this will make more sense.
cd dirname	change to the specified directory. The lack of a "/" in front of the directory name implies a relative path meaning <i>dirname</i> is a subfolder of our current directory.

Copy files:

cp <source> <dest>	Copy file <source> to <destination>
cp -r <source> <dest>	Copy a directory recursively

Move a file or directory:

mv <source> <dest>	Move or rename a file or directory <source> to <destination>
---------------------------------------	--

Delete a file or directory:

rm <target>	Delete a file
rm -r <targetdir>	Delete a directory recursively (including subdirectories)
rmdir <target>	Delete a directory (if empty)
rm -f <target>	Force removal without prompt

Create a directory:

mkdir <directory>	Create a directory called <directory>
--------------------------------	---------------------------------------

Display command help:

man <command>	Display a "manual" page for the specified <command>. Use 'q' to quit. Remember this one. Very Useful.
----------------------------	---

If you want to find information about a command and its various options, you use the man(ual) page. For example, the **find** command can be daunting. But I can gather lots of helpful information with **man find**:

```
barry@forensicbox:~$ man find
FIND(1)                                General Commands Manual                                FIND(1)

NAME
    find - search for files in a directory hierarchy

SYNOPSIS
    find [-H] [-L] [-P] [-D debugopts] [-Olevel] [starting-point...] [
    ↪ expression]

DESCRIPTION
    This manual page documents the GNU version of find. GNU find searches the
    directory tree rooted at each given starting-point by evaluating the given
    ...
```

Display the contents of a (plain text) file:

cat <filename>	The simplest way to display the contents of a file, this command streams the contents of a file to standard output (normally the terminal). Stands for <i>concatenate</i> and can also be used to combine files.
less <filename>	Page through a file line by line or page by page.

3.1.1 Additional Useful Commands

grep : Searches for patterns.

grep pattern filename

The **grep** command will look for occurrences of <pattern> within the file <filename>. **grep** is an extremely powerful tool. It has hundreds of uses given the number of options it supports. Check the **man** page for more details. We will use **grep** in our forensic exercises later on.

find : Searches for files based on any number of criteria. These can include dates, sizes, name patterns, object type, etc.

find <start dir> <criteria>

A simple search for a file named **fstab** could look something like this:

```
root@forensicbox:~# find /etc -iname fstab
/etc/fstab
```

This means "find, starting at the `/etc` directory" by name, `fstab`. **find** will allow you to search by file type (regular files vs. directories, etc.), or even various file times. We'll use this tool extensively later on. Check **man find** and see if you can find the difference between the **-name**, and **-iname** options.

pwd : Prints the present working directory to the screen. The following example shows that we are currently in the directory `/home/barry`:

```
barry@forensicbox:~$ pwd
/home/barry
```

file : Categorizes files based on what they contain using signature comparison. The categorization occurs regardless of name or extension. The file header is compared to a *magic* file to determine the file type. Here's an example using a standard JPEG image:

```
barry@forensicbox:~$ file mypic.jpg
mypic.jpg: JPEG image data, JFIF standard 1.02, aspect ratio, density 1x1,
segment length 16, progressive, precision 8, 720x960, components 3
```

ps : List current processes. This is a *little* like the task manager in Windows. It gives the process ID number (PID) and the terminal on which the process is running. **ps ax** will show all processes (**a**), including processes without an associated terminal (**x**). Note the lack of a dash in front of the options. Read the **man** page for more details. The below output shows the parent of all processes *init* and 3 virtual terminals running the **bash** shell (some output removed).

```
barry@forensicbox:~$ ps
PID TTY          STAT       TIME COMMAND
   1 ?            Ss          0:00 init [4]
   2 ?            S           0:00 [kthreadd]
...
2446 pts/1      Ss          0:00 -bash
2495 pts/2      Ss+         0:00 -bash
2563 ?           I           0:00 [kworker/0:1-kdmflush]
2572 ?           I           0:00 [kworker/3:2-rcu_gp]
2598 pts/3      Ss          0:00 -bash
```

strings : Prints out the readable characters from a file. This command will print out strings that are at least four characters long (by default) from a file. Useful for looking at data files without the originating program, and searching executables for useful strings, etc. More on this forensically useful command later.

chmod : Changes the permissions on a file ("change mode"). We'll cover this command and its use later on.

chown : Changes the owner (and group) of a file much the same way **chmod** changes permissions.

clear : Clears the terminal.

exit : Exits the terminal and closes it.

shutdown : This command will be used to cleanly exit the system and power down (or restart) the computer. You can run several different options here (and be sure to check the **man** page for additional details).

shutdown -r now - will reboot the system 'now'.

shutdown -h now - will shut down the system 'now'.

3.2 File Permissions

Files in Linux have certain specified permissions. These permissions can be viewed by running the **ls -l** command on a directory or on a particular file. For example:

```
barry@forensicbox:~$ ls -l myfile.sh
-rwxr-xr-x 1 barry users 3685 Apr 15 11:14 myfile.sh
```

If you look close at the first 10 characters, you have a dash (-) followed by 9 more characters. The first character describes the type of file. A dash (-) indicates a regular file. A "d" would indicate a directory, and "b" a special block device, etc.

First character of **ls -l** output:

-	=	regular file
d	=	directory
b	=	block device (disk, volume, etc.)
c	=	character device (serial)
l	=	link (points to another file or directory)

The next 9 characters indicate the file permissions. These are given in groups of three:

<u>Owner</u>	<u>Group</u>	<u>Others</u>
rwX	rwX	rwX

The characters (referred to as 'bits') indicate:

r	=	read
----------	---	------

w = write
x = execute

So for the above `myfile.sh` we have: `rwX` `r-x` `r-x`

This gives the file owner read, write and execute permissions (`rwX`), but restricts other members of the owner's group and users outside that group to only read and execute the file (`r-x`). Write access is denied as symbolized by the "-".

Now back to the **chmod** command. There are a number of ways to use this command, including explicitly assigning `r`, `w`, or `x` to the file. We will cover the octal method here because the syntax is easiest to remember (and I find it most flexible). In this method, the syntax is as follows:

chmod octal filename

For our purpose, the *octal* is a numerical value in which the first digit represents the owner, the second digit represents the group, and the third digit represents others outside the owner's group. Each digit is calculated by assigning a value to each permission:

read (r) = 4
write (w) = 2
execute (x) = 1

For example, the file in our original example has an octal permission value of **755** (`rwX` = 7, `r-x` = 5, `r-x` = 5). This means the owner of the file has all three bits set (read, write and execute), while all others have only read and execute. If you wanted to change the file so that only the owner had read, write and execute permissions, and members of the owner's group and all others would only be allowed to read the file, you would issue the command:

chmod 744 filename

(r=4) + (w=2) + (x=1) = 7 [owner]
(r=4) + (w=0) + (x=0) = 4 [group]
(r=4) + (w=0) + (x=0) = 4 [others]

Changing permissions and then displaying a new long list of the file would show:

```
barry@forensicbox:~$ ls -lh myfile.sh
-rwxr-xr-x 1 barry users 3685 Apr 15 11:14 myfile.sh (permission value is 755)
```

```
barry@forensicbox:~$ chmod 744 myfile.sh
```

```
barry@forensicbox:~$ ls -lh myfile.sh
```

```
-rwxr--r-- 1 barry users 3685 Apr 15 11:14 myfile.sh (permission value is now 744)
```

3.3 Pipes and Redirection

Linux (or more specifically bash) allows you to redirect the output of a command from the standard output (usually the display or "console") to another device or file. This is done with streams. There are three streams we will talk about: **stdin** is the standard input (usually the keyboard); **stdout** is the standard output (usually the display); and **stderr** is standard error (usually the display).

We use specific symbols to redirect these streams:

- **stdin** : `<`
 - **cmd < infile**
 - cmd is taking its input from infile rather than the keyboard.
- **stdout** : `>`
 - **cmd > outfile**
 - cmd is sending its output to outfile rather than the display.
- **stderr** : `2>`
 - **cmd 2> errlog**
 - cmd is sending any error messages to the file **errlog**.

Manipulating streams can be useful for tasks like creating an output file that contains a list of files on a mounted volume, or in a directory. For example:

```
barry@forensicbox:~$ ls -al > filelist.txt
```

The above command would output a long list of all the files in the current directory. You won't see the output, though. Instead of outputting the list to the console, a new file called **filelist.txt** will be created that will contain the list. If the file **filelist.txt** already exists, then it will be overwritten. Use the following command to *append* the output of the command to the existing file, instead of over-writing it:

```
barry@forensicbox:~$ ls -al >> filelist.txt
```

Another useful tool is the command pipe, which uses the `|` symbol. The command pipe takes the *output* of one command and "pipes" it straight to the *input* of another command.

In this case, we are redirecting the output to another command rather than a file. You can see the difference below. I can **echo** a character string to a file with `>`, or I can **echo** to a command with `|`. The **wc** command shown below gives a count of lines, words, and bytes. In the first redirect below, I'm creating a file called `h.txt` with the output of **echo**. In the second, I'm using a pipe, so the output of **echo** goes to the command **wc**. In the third, I'm piping the output of **echo** to **wc** and redirecting the **wc** output to a file:

Follow along below, and experiment. DON'T do this logged in as root. Experimentation can get out of hand quickly.

```
barry@forensicbox:~$ echo hello
hello

barry@forensicbox:~$ echo hello > h.txt

barry@forensicbox:~$ cat h.txt
hello

barry@forensicbox:~$ echo hello | wc
      1      1      6

barry@forensicbox:~$ echo hello | wc > outfile.txt

barry@forensicbox:~$ cat outfile.txt
      1      1      6
```

This is an extremely powerful tool for the command line. Look at the following process list (partial output shown):

```
barry@forensicbox:~$ ps ax
  PID TTY          STAT TIME  COMMAND
    1 ?            Ss   0:06  init [4]
    2 ?            S    0:00  [kthreadd]
    4 ?            I<   0:00  [kworker/0:0H]
    6 ?            I<   0:00  [mm_percpu_wq]
<continues>
```

What if all you wanted to see were those processes ID's that indicated a bash shell? You could "pipe" the output of **ps** to the input of **grep**, specifying **bash** as the pattern for **grep** to search. The result would give you only those lines of the output from **ps** that contained the pattern **bash**.

```
barry@forensicbox:~$ ps ax | grep bash
 9434 pts/4    Ss   0:00 -bash
14746 pts/3    Ss   0:00 -bash
25354 pts/5    S+   0:00 grep bash
```

The above output uses **grep** to display only those lines of **ps** output that contain the string **bash**. So we can see that there are two **bash** sessions and a line that shows the process of our actual **grep** command.

You may have noticed that when we redirect the output of a command to a file that we don't actually see the output on the screen unless we view the file we created. There may be times where you want to see the output of a command displayed on the screen and have it redirect to a file as well. You can do that using the **tee** command.

```
barry@forensicbox:~$ ls | tee filelist.txt
Desktop/
Documents/
Evidence/
winlog.txt

barry@forensicbox:~$ cat filelist.txt
Desktop/
Documents/
Evidence/
winlog.txt
```

In the above session, we've used the **tee** command to both display the output of the **ls** command to the screen, and send it to a file called **filelist.txt**. In the context of a forensic examination, this is useful to capture the output of tools in a log file (remember to use **>>** to append).

Stringing multiple powerful commands together is one of the most useful and powerful techniques provided by Linux (again, actually bash) for forensic analysis. This is one of the single most important concepts you will want to learn if you decide to take on Linux as a forensic platform. With a single command line built from multiple commands and pipes, you can use several utilities and programs to boil down an analysis very quickly.

3.4 File Attributes

Linux file systems (like ext2, ext3, ext4) support what are called *file attributes*. There are quite a few of them, and we will not cover all of them here. There are two that can be very useful for protecting forensic data from haphazard deletion or tampering. These are *append only* (a) and *immutable* (i).

Attributes are flags that can control what file operations are allowed to occur on a file or a directory. Some of them can be changed by a normal user, and some cannot. We can list the attributes of files and directories in our current directory with **lsattr**. In this case, we are going to run the command on a particular file. Note that we are using root for this (you can use the **su** command to become root, as we discussed earlier in section 1.5.2):

```
root@forensicbox:~# lsattr)
-----e----- ./myfile.txt
```

Here I am currently in the `/root` directory (signified by `~` in the prompt because `/root` is the root user's home). This output shows that of all the available attributes, this file has only the *extents* attribute set.⁵

We add and change (or remove) attributes with the **chattr** command, simply using a '+' to add the attribute we want with the file name, or a '-' to remove it. For example, if I want to make the file `myfile.txt` immutable, I can add the **i** attribute like this:

```
root@forensicbox:~# chattr +i myfile.txt

root@forensicbox:~# lsattr
----i-----e----- ./myfile
```

Now if I try to remove the file, even as root, I cannot because the file is *immutable*. Even with the **-f** (force) option, the file cannot be removed:

```
root@forensicbox:~# rm -f myfile.txt
rm: cannot remove 'myfile.txt': Operation not permitted
```

If I want to remove the file, I can use **chattr -i myfile.txt** to remove the immutable attribute.

This can be useful if you have forensic images or other evidence you want to protect from accidental change/removal. Attributes can also be set on directories. With the immutable attribute set on a directory, it cannot be removed and files contained in it cannot be changed or removed.

Now let's have a look at the *append only* attribute. Consider a file we'll call `notes.txt` that we are redirecting output to. We don't want to accidentally overwrite the file while adding command output, so we set the *append only* attribute. Have a look at the following session:

⁵Extents are a method of mapping physical blocks of data in a contiguous fashion. We will not cover extents in this guide. Additional information can be found online.

```
root@forensicbox:~# lsattr
-----e----- ./notes.txt

root@forensicbox:~# chattr +a notes.txt

root@forensicbox:~# lsattr
-----a-----e----- ./notes.txt

root@forensicbox:~# echo 'hello' > notes.txt
-su: notes.txt: Operation not permitted
a single redirect (>) tries to overwrite

root@forensicbox:~# echo 'hello' >> notes.txt
a double redirect (>>) appends and is successful
```

3.5 Command Line Math

When conducting an examination, you'll often find yourself needing a quick way to make a simple calculation (sector offset, etc.). We're going to cover some basic ways to accomplish this via the command line. We do this for two reasons: First is that it's often easier to include command line calculations without having to grab a mouse, open a GUI calculator and type in the numbers – why not just type in the terminal and get your answer? Second, you may find yourself needing to use a terminal session or system that has no GUI. You might as well learn how to use the command line for as much as you can and not rely on external resources. There are a number of ways to do this:

3.5.1 **bc** - the Basic Calculator

If we need to do some calculations on the command line, we can use **bc** and either open an interactive session, or pipe the expression to be evaluated via the **echo** command through **bc**. You'll need these techniques to calculate byte offsets in later exercises. You don't want to have to open a calculator app, do you?

For an interactive session, simply type **bc** at the prompt and you will be dropped into the session. Type the expression and hit **<enter>**. Input below is in bold for clarity.

```
barry@forensicbox:~$ bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017
Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
```

```
2+2
4
512*1000
512000
5/3
1
quit
```

Type **quit** to finish, and you'll exit the session. Pay close attention to the last expression, 5/3. Note that the response is 1, a whole number, rather than the fraction we would assume. This is because **bc** is a fixed precision calculator, and the default scale is 1. You can set the scale with the **scale=x** function, where x is the precision you'd like. If you want your answer rounded to two decimal places, you can use **scale=2**.

```
barry@forensicbox:~$ bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017
Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
scale=2
5/3
1.66
quit
```

Here we see our expected result. You can also invoke **bc -l**, which sets additional functions, but the scale is set to 20 by default, and you'd normally want to set a smaller scale anyway.

If you'd prefer not to use an interactive session, you can pipe your expression to **bc** using **echo**:

```
barry@forensicbox:~$ echo 5/3 | bc
1

barry@forensicbox:~$ echo "scale=2; 5/3" | bc
1.66

barry@forensicbox:~$
echo 2048*512 | bc
1048576
```

The above example shows both the default output and scale setting via **echo**. The last command shows a common calculation for byte offset when given a sector number (or sector offset) in forensic work. We will use this frequently in later chapters.

Finally, we can use **bc** to convert hexadecimal values to decimal values by using the option **ibase=16**, either interactively or via **echo**. Note that alpha characters in the hex expression *MUST* be upper case for **bc** to work. Here are a couple of examples (again our interactive input is in bold):

```
barry@forensicbox:~$ bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017
Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
ibase=16
4c
(standard_in) 2: syntax error <- note chars must be upper case
4C
76
quit

barry@forensicbox:~$ echo "ibase=16;4C | bc"
76
```

3.5.2 Bash Shell - Arithmetic Expansion

If you are dealing with simple integers (or hex conversion), and floating point or decimal responses are not required, you can use more simple bash (shell) *arithmetic expansion*. This is probably the quickest and easiest way to do calculations for simple addition or subtraction where integer volume offsets are needed and you are not likely to encounter fractional evaluations. Note that you need to use the **echo** command to evaluate the expression, or the evaluation itself will be interpreted by the shell as a command. Hexadecimal values should be preceded by **0x** (zero x) Here's an example set of evaluations:

```
barry@forensicbox:~$ echo $((2048*512))
1048576

barry@forensicbox:~$ echo $((5/3))
1
<- note the integer response

barry@forensicbox:~$ echo $((0x4c))
76

barry@forensicbox:~$ echo $((0x4c-70))
6
```

For additional information, see **man bash**.

3.6 Bash 'globbing'

The **bash** shell also supports what many would call 'wildcards'. We refer to this as *globbing* or *file name expansion*. Note that this is NOT the same as 'regular expressions', although they look very similar. Here are a couple to remember:

- * for multiple characters
- ? for single characters
- [] for sets or a range of characters

This can be a complicated but very powerful subject, and will require further reading... Refer to "regular expressions" in your favorite Linux text, along with "globbing" or "file expansion". There are important differences that can confuse a beginner, so don't get discouraged by confusion over what '*' means in different situations.

3.7 Command Review and Hints

1. The shell has a history list of previously used commands (stored in the file named **.bash_history** in your home directory). Use the keyboard arrows to scroll through commands you've already typed.
2. Command line editing is also supported. As above, use the arrow keys to see and edit previous commands so you don't need to retype long commands with errors...just edit them.
3. Commands and filenames are CASE SENSITIVE
4. Learn output redirection for stdout and stderr (> and 2>)
5. Linux uses / for directories, Windows uses \.
6. Use **q** to quit from **less** or **man** sessions.
7. To execute commands in the current directory (if the current directory is not in your PATH), use the syntax **./command**. This tells the shell to look in the present directory for the command. Unless it is explicitly specified, the current directory is NOT part of the normal user path.

4 Editing with Vi

There are a number of terminal mode (non-GUI) text editors available in Linux, including **emacs** and **vi**. You could always use one of the available GUI text editors in an X session, but what if you are unable to start X, or a windowing system is not available? The benefit of learning a text editor like **vi** is your ability to use it from a terminal or a telnet or ssh connection, etc. We are discussing **vi** here. (I don't do emacs :-)). **vi** in particular is useful, because you will find it on all versions of Unix. Learn the basics of **vi** and you should be able to edit a file on any Unix system.

4.1 The Joy that is vi

You can start **vi** either by simply typing **vi** at the command prompt, or you can specify the file you want to edit with **vi filename**. If the file does not already exist, it will be created for you.

vi consists of two operating modes, *normal mode* and *insert mode*. When you first enter **vi** you will be in normal mode. *Normal mode* allows you to search for text, move around the file, and issue commands for saving, save-as, and exiting the editor (as well as a whole host of other functions). *Insert mode* is where you actually input and change text.

In order to switch to insert mode, type either **a** (for append), **i** (for insert), or one of the other insert options listed on the next page. When you do this you will see **--INSERT--** appear at the bottom of your screen (in most versions). You can now input text. When you want to exit the insert mode and return to normal mode, press the escape key.

You can use the arrow keys to move around the file in command mode. The **vi** editor was designed, however, to be exceedingly efficient, if not intuitive. The traditional way of moving around the file is to use the *qwerty* keys right under your finger tips. More on this below. In addition, there are a number of other navigation keys that make moving around in **vi** easier, like using **\$** to move to the end of the current line or **w** to move to the next word, etc.

If you lose track of which mode you are in, hit the escape key twice. You will know that you are in normal mode. There is normally an indicator of your current mode at the bottom of the window.

4.2 The **vimtutor** Tutorial

In most Linux distributions, **vi** is usually a link to some newer implementation of **vi**, such as **vim** (vi improved), **elvis** or **neovim**. If your distribution includes **vim**, it should come with a nice tutorial. It is worth your time. Try typing **vimtutor** at a command prompt. Work

through the entire file. This is the single best way to start learning **vi**. The navigation keys mentioned above will become clear if you use **vimtutor**.

4.3 vi Command Summary

Entering edit mode *from* normal mode:

i	: insert text under the cursor
a	: append text after the cursor
o (the letter 'oh')	: open a new line under the current line (in insert mode)
O (uppercase 'oh')	: open a new line above the current line (in insert mode)

Normal Mode:

0 (zero)	: move the cursor to the beginning of the line
\$: move the cursor to the end of the line
x	: delete (cut) the character under the cursor
X	: delete (cut) the character before the cursor
dd	: delete (cut) the entire line the cursor is on
dw	: delete (cut) to the end of the current word
v	: enter visual mode (select with cursor)
y	: yank (copy)
yw	: yank (copy) to the end of the current word
y\$: yank (copy) to the end of the current line
p	: paste after the cursor
P	: paste before the cursor
:w	: save and continue editing
:wq	: save and quit
:wq!	: save and quit without prompt
:q!	: quit and discard changes
:w fname	: save as fname
gg	: go to the top of the file (first line)
G	: go to the end of the file (last line)
G<NUM>	: go to line number <NUM>

The best way to save yourself from a messed up edit is to hit the escape key followed by **:q!**. That command will quit without saving changes.

Another useful feature in normal mode is the string search. To search for a particular string in a file, make sure you are in normal mode and type

/string

Where **string** is your search target. After issuing the command, you can move on to the next hit by typing **n**.

vi is an extremely powerful editor. There are a huge number of commands and capabilities that are outside the scope of this guide. See **man vi** for more details. Keep in mind there are chapters in books devoted to this editor. There are also complete books devoted to **vi** alone. The forensic importance of **vi** is that you never know when you will find yourself responding to a Unix machine, at a terminal, and needing to change a file. **vi** will almost certainly be there for you.

5 The Linux Boot Sequence (Simplified)

This guide is intended to be an introduction to Linux as a *platform* for digital forensics, not just a tutorial on available tools. One of the important aspects of using and maintaining forensic platforms (and the tools that run on them) is a basic knowledge of the operation of that platform. We pursue this knowledge so that we have a better understanding of our system baseline. This allows us to troubleshoot errors or to track changes that are made to the system over time. Knowing how your platform operates and how to add, remove or change running processes is a fundamental skill in administering your own forensic platform.

As long as we are learning Linux, we might as well educate ourselves on core system functions that may help us should we come across a Linux machine that must be investigated. Knowing how to traverse the directory structure or issue commands is not good enough. Our ability to use Linux as either a platform for forensics *or* as the subject of an analysis is very much reliant on our overall knowledge of the operating system. Knowing that there are plenty of online resources to dive into specific systems and distributions, we will only provide a simple introduction here.

5.1 Init vs. Systemd

As with many of the Linux core components, system initialization (or 'init') and service management have seen some drastic changes over the years. Some distributions, like Slackware, have stayed with tried and true BSD or SystemV *init* and service management systems. Other distributions, like Fedora and Ubuntu, have moved on to newer system and service management systems. One of these newer systems is called *Systemd*. There are other system initialization programs, like *openrc* and *runit*, but since this guide is written around Slackware, we will stick to describing SysV/BSD style init in detail, and briefly describe Systemd as a popular alternative.

5.2 Booting the Kernel

First, however, we need to discuss some other complexities that have been introduced over the years. With the introduction of the Unified Extensible Firmware Interface (UEFI), we find ourselves moving away from traditional BIOS booting, although most systems offer it as a 'legacy' option. This has resulted in a requirement for us to pay attention to how our computer boots. Traditional bootloaders may not work under UEFI, and many distributions have moved toward more flexible options.

Once again, one way to experiment with Linux without the pain of hardware compatibility issues is to use virtualisation. There are free options out there and you won't need to worry about hardware compatibility or UEFI vs. legacy BIOS.

The first step in the (simplified) boot up sequence for Linux is loading the kernel. The kernel image is usually contained in the `/boot` directory. It can go by several different names, including `bzImage` or `vmlinuz`. Sometimes the kernel image will specify the kernel version contained in the image, i.e. `vmlinuz-huge-4.4.19`. Very often there is a soft link (like a shortcut) to the most current kernel image in the `/boot` directory. It is normally this soft link that is referenced by the boot loader, LILO (or GRUB, eLILO, etc.). In a stock Slackware system, the kernel image is `/boot/vmlinuz`.

Note that Slackware uses LILO or eLILO (for UEFI systems) by default. LILO and eLILO are older and far simpler systems for booting, but are somewhat less flexible. The boot loader specifies the 'root device', along with the kernel version to be booted. For example, with LILO, this is all controlled by the file `/etc/lilo.conf`. Each `image=` section represents a choice in the boot screen.

This is an example of a `lilo.conf`:

```
root@forensicbox:~# cat /etc/lilo.conf
append=" vt.default_utf8=0"
boot = /dev/sda                <- our boot device
bitmap = /boot/slack.bmp
bmp-colors = 255,0,255,0,255,0
bmp-table = 60,6,1,16
bmp-timer = 65,27,0,255
prompt
timeout = 1200
change-rules
    reset
vga = normal
image = /boot/vmlinuz          <- the kernel image we are booting
    root = /dev/sda1           <- the partition we boot from
    label = Linux
    read-only
```

The actual boot configuration file on your system (`/etc/lilo.conf`, `/boot/grub/grub.cfg`, etc.) will be far more complex in most cases. In this example, comments in the file (lines starting with a `#`) have been removed for readability. And unless you are running in something like a very simple virtual environment you may have other bootable entries or rescue fallback options as well.

Once the system has finished booting, you can replay the kernel messages that scroll quickly past the screen during the booting process with the command `dmesg`. We discussed this command a little when we talked about device recognition earlier. The output can be piped through a paging viewer to make it easier to see (in this case, `dmesg` is piped through `less` on my Slackware system):


```
root@forensicbox:~# dmesg | less
[    0.000000] Command line: BOOT_IMAGE=/vmlinuz-huge-4.14.20 root=/dev/nvme0n1p2
    ↪ ro
[    0.000000] x86/fpu: Supporting XSAVE feature 0x001: 'x87 floating point
registers'
[    0.000000] x86/fpu: Supporting XSAVE feature 0x002: 'SSE registers'
[    0.000000] x86/fpu: Supporting XSAVE feature 0x004: 'AVX registers'
[    0.000000] x86/fpu: Supporting XSAVE feature 0x008: 'MPX bounds registers'
[    0.000000] x86/fpu: Supporting XSAVE feature 0x010: 'MPX CSR'
[    0.000000] x86/fpu: xstate_offset[2]:  576, xstate_sizes[2]:  256
[    0.000000] x86/fpu: xstate_offset[3]:  832, xstate_sizes[3]:   64
[    0.000000] x86/fpu: xstate_offset[4]:  896, xstate_sizes[4]:   64
[    0.000000] x86/fpu: Enabled xstate features 0x1f, context size
is 960 bytes, using 'compacted' format.
[    0.000000] e820: BIOS-provided physical RAM map:
```

5.3 System Initialization

After the boot loader initiates the kernel, the next step in the boot sequence starts with the program `/sbin/init`. This program really has two functions:

- initialize the runlevel and startup scripts
- terminal process control(respawn terminals)

In short, the **init** program is controlled by the file `/etc/inittab`. It is this file that controls your runlevel and the global startup scripts for the system. This is, again, for a Slackware system. Some systems, like Ubuntu, for example, use **systemd** for system control and configuration. If you are interested in the system startup routine for your particular distro (and you should be interested), then research it online.

5.4 Runlevel

The runlevel is simply a description of the system state. For our purposes, it is easiest to say that (for Slackware, at least. Other systems, such as those using **systemd**, will differ):

- runlevel 0 = shutdown
- runlevel 1 = single user mode
- runlevel 3 = full multiuser mode: text login (DEFAULT)

- runlevel 4 = full multiuser mode: X11 graphical login
- runlevel 6 = reboot

In the file `/etc/inittab` you will see a line similar to:

```
id:3:initdefault:
```

```
root@forensicbox:~# cat /etc/inittab
...
# These are the default runlevels in Slackware:
# 0 = halt
# 1 = single user mode
# 2 = unused (but configured the same as runlevel 3)
# 3 = multiuser mode (default Slackware runlevel)
# 4 = X11 with KDM/GDM/XDM (session managers)
# 5 = unused (but configured the same as runlevel 3)
# 6 = reboot

# Default runlevel. (Do not set to 0 or 6)
id:3:initdefault:

# System initialization (runs when system boots).
si:S:sysinit:/etc/rc.d/rc.S
...
```

It is here in `/etc/inittab` that the default runlevel for the system is set. If you want a text login, set the above value in `initdefault` to 3. This is the default for Slackware. With this default runlevel, you use **startx** to get to the X Window GUI system. If you want a graphical login, you would edit the above line to contain a 4.

Note that for Ubuntu, you can create an `/etc/inittab` file and place the value in there. If it exists, the file will be read and the runlevel changed accordingly. The **systemd** style of management used by Ubuntu does not really utilize "runlevels". It utilizes *targets*. Changes to these targets are made using the **systemctl** command. The configuration and use of Ubuntu is outside the scope of this guide, but this particular issue highlights the fact that Linux systems can vary in how they work.

5.5 Global Startup Scripts

After the default run level has been set, **init** (via `/etc/inittab`) then runs the following scripts:

- `/etc/rc.d/rc.S` - handles system initialization, file system mount and check, encrypted volumes, swap initialization, devices, etc.
- `/etc/rc.d/rc.X` - where X is the run level passed as an argument by **init**. In the case of multi-user (non GUI) logins (run level 2 or 3), this is `rc.M`. This script then calls other startup scripts (various services, etc.) by checking to see if they are "executable".
- `/etc/rc.d/rc.local` - called from within the specific run level scripts, `rc.local` is a general purpose script that can be edited to include commands that you want started at boot up.
- `/etc/rc.d/rc.local_shutdown` - This file should be used to stop any services that were started in `rc.local`. Create the file and make it executable to have it run.

5.6 Service Startup Scripts

Once the global scripts run, there are "service scripts" in the `/etc/rc.d/` directory that are called by the various runlevel scripts, as described above, depending on whether the scripts themselves have "executable" permissions. This means that we can control the boot time initialization of a service by changing its executable status. More on how to do this later. Some examples of service scripts are:

- `/etc/rc.d/rc.inet1` - handles network interface initialization
- `/etc/rc.d/rc.inet2` - handles network services start. This script organizes the various network services scripts, and ensures that they are started in the proper order.
- `/etc/rc.d/rc.wireless` - handles wireless network card setup.
- `/etc/rc.d/rc.sendmail` - starts the mail server.
- `/etc/rc.d/rc.sshd` - starts the OpenSSH server.
- `/etc/rc.d/rc.messagebus` - starts d-bus messaging services.
- `/etc/rc.d/rc.udev` - populates the `/dev` directory with device nodes, scans for devices, loads the appropriate kernel modules, and configures the devices.

Have a look at the `/etc/rc.d` directory for more examples. Note that in a standard Slackware install, your directory listing will show executable scripts as green in color (in a terminal with color support) and followed by an asterisk (*).

Again, this is Slackware specific. Other distributions differ (some differ greatly!), but the concept remains consistent. Once you become familiar with the process, it will make sense. The ability to manipulate startup scripts is an important step in your Linux learning process. At the very least, understanding how your system works and where services are started and stopped is important.

5.7 Bash

bash (*Bourne Again Shell*) is the default command shell for most Linux distros. It is the program that sets the environment for your command line experience in Linux. There are a number of shells available, but we will cover **bash**, the most commonly used in Linux, here. Shells like **zsh** and **fish** are gaining some popularity as well. Once you begin to realize the power of the shell, you can begin to explore some of the other options and their benefits.

There are actually quite a few files that can be used to customize a user's Linux experience. Here are some that will get you started.

- **/etc/profile** - This is the global **bash** initialization file for interactive login shells. Edits made to this file will be applied to all **bash** shell users. This file sets the standard system path, the format of the command prompt and other environment variables.
- **/home/\$USER/.bash_profile**⁶ (**\$USER**) and can be edited by the user, allowing him or her to customize their own environment. It is in this file that you can add aliases to change the way commands respond. Note that the dot in front of the filename makes it a "hidden" file.
- **/home/\$USER/.bash_history** – This is an exceedingly useful file for a number of reasons. It stores a set number of commands that have already been typed at the command line (default is 500). These are accessible simply by using the "up" arrow on the keyboard to scroll through the history of already-used commands. Instead of re-typing a command over and over again, you can access it from the history. From the perspective of a forensic examiner, if you are examining a Linux system, you can access each user's (don't forget **root**) **.bash_history** file to see what commands were run from the command line. Remember that the leading "." in the file name signifies that it is a hidden file.

Keep in mind that the default values for **./bash_history** (number of entries, history file name, etc.) can be controlled by the user(s). Read **man bash** for more detailed info.

The **bash** startup sequence is actually more complicated than this, but this should give you a starting point. In addition to the above files, check out **/home/\$USER/.bashrc**. The **man** page for **bash** is an interesting (and long) read, and will describe some of the customization options. In addition, reading the **man** page will give a good introduction to the programming power provided by **bash** scripting. When you read the **man** page, you will want to concentrate on the **INVOCATION** section for how the shell is used and basic programming syntax.

⁶In **bash** we define the contents of a variable with a dollar sign. **\$USER** is a variable that represents the name of the current user. To see the contents of shell individual variables, use **echo \$VARIABLE**. This script is located in each user's home directory

6 Configuring a Forensic Workstation

There are many excellent guides, books and websites out on the Internet that provide some wonderfully detailed information on setting up a Linux installation for day to day use. We are going to concentrate here on subjects of particular interest to setting up a secure and usable forensic workstation.

As with the rest of this guide, the specific commands presented here are for a Slackware installation. While the commands and capabilities provided by other distributions will differ somewhat (or greatly) the basic concepts should be the same. As always, check your distribution's documentation before running these commands on a non-Slackware system. And let me reiterate. These are just the basics. The guidelines set forth in here offer only a starting point for workstation configuration and security. If you are not using Slackware, do not just skip this section...the information is useful regardless. This is not an exhaustive manual on security and configuration. It is simply the basics to get you started.

6.1 Securing the Workstation

These next few sections on start up scripts, `tcpwrapper` and `iptables` are covered in detail in Slackware documentation (the Slack Book, for example) and elsewhere for other distributions. I'm going to mention them here so that the reader gets a baseline understanding of these subjects. The details can be found through further reading. Again, take note that even if you are not using Slackware, and your distribution of choice is not configured as I'm about to describe, it's still worth following along, as the subject of determining open network ports and tracing what service they belong to is an important one.

Anyone who has been working in the field of digital and computer forensics for any length of time can tell you that forensic workstation security is always a top priority. Some practitioners work on completely "air gapped" forensic networks with no connection to outside resources. Others find this approach too limiting and elect to heavily firewall and monitor forensic workstations while allowing some level of access to external networks. In either case, understanding your workstation's security posture is extremely important. This document does not endorse or suggest any particular approach, and as with all things in this business, the requirements for your particular setup may change day to day depending on the nature of the cases you are working on, the evidence you are handling, the physical or network environment you are working in and the policies set forth by your agency or company.

The goal here is to ensure that, at a minimum, a forensic examiner understands the current security posture of the workstation, or at the very least, is conversant in addressing them. This section is not meant to imply, in any way, that simple host based security is enough to protect your forensic environment. The ideal lab will have edge routers and hardware based appliances to properly secure data and network access. In some cases, contraband analysis and malware investigation for example, air gapping may be the only realistic solution. In

any event, understanding the mechanics of host based security is an often overlooked, but important part of the forensic environment.

6.1.1 Configuring Startup Services

We'll start our security configuration with the most basic steps...disabling services (and/or daemons) that start when the computer boots. It's fairly common knowledge that running programs and network services that you are not using and do not need serves only to introduce potential vulnerabilities. There are all sorts of services running on any given workstation, regardless of distribution or operating system. Some of these services are required, some are optional, and some are downright undesirable for a forensic environment. As previously discussed, this is where you will find quite a difference among the various distributions. Consult your distribution's documentation for more info, and don't neglect this part of your Linux education!

Previously, we discussed the system initialization process. Part of that process is the execution of *rc* scripts that handle system services. Recall that the file `/etc/inittab` invokes the appropriate run level scripts in the `/etc/rc.d/` directory. In turn, these scripts test various service scripts, also in the `/etc/rc.d/` directory, for executable permissions. If the script is executable, it is invoked and the service is started. This can be chained, where `rc.M` checks to see if an *rc* script is executable, and if so the execution of that script checks for more scripts that are executable. For example, the test inside the `rc.M` (multiuser init script) that will run the network daemons initialization script (`rc.inet2`) looks like this (abbreviated):

```
root@forensicbox:~# cat /etc/rc.d/rc.M
...
# Start networking daemons:
if [ -x /etc/rc.d/rc.inet2 ]; then
    /etc/rc.d/rc.inet2
fi
...

```

The code shown above is an `if / then` statement where the brackets signify the test and the `-x` checks for executable permissions. So it would read:

if the file `/etc/rc.d/rc.inet2` is executable, then execute the script `/etc/rc.d/rc.inet2`

Once `rc.inet2` is running, it checks the executable permissions on the network service scripts (among other things). This allows us to control the execution of scripts simply by changing the permissions. If an *rc* script is executable, it will run. If it is not executable then it is passed over. As an example, let's have a look at the OpenSSH (secure shell) portion of `rc.inet2`:

```
root@forensicbox:~# cat /etc/rc.d/rc.inet2
```

```
...  
# Start the OpenSSH SSH daemon:  
if [ -x /etc/rc.d/rc.sshd ]; then  
    echo "Starting OpenSSH SSH daemon. /usr/sbin/sshd"  
    /etc/rc.d/rc.sshd start  
fi  
...
```

Again, this portion of `rc.inet2` checks to see if `rc.sshd` is executable. If it is, then it runs the command `/etc/rc.d/rc.sshd start`. The rc service scripts can have either **start**, **stop** or **restart** passed as arguments in most cases. So, in summary for this particular example:

- `/etc/inittab` calls `/etc/rc.d/rc.M`
- `/etc/rc.d/rc.M` calls `/etc/rc.d/rc.inet2` (if `rc.inet2` is executable)
- `/etc/rc.d/rc.inet2` passes the command `/etc/rc.d/rc.sshd start` (if `rc.sshd` is executable)..

Earlier on we discussed file permissions. Now let us look at a practical example of changing permissions for the purpose of stopping select services from starting at boot time. A look at the permissions of `/etc/rc.d/rc.sshd` shows that it is executable, and so will start when `rc.inet2` runs:

```
root@forensicbox:~# ls -l /etc/rc.d/rc.sshd  
-rwxr-xr-x 1 root root 1814 Oct 1. 2018 /etc/rc.d/rc.sshd*
```

To change the executable permissions to prevent the SSH service to start at boot time, I execute the following:

```
root@forensicbox:~# chmod 644 /etc/rc.d/rc.sshd  
  
root@forensicbox:~# ls -l /etc/rc.d/rc.sshd  
-rw-r--r-- 1 root root 1814 Oct 1. 2018 /etc/rc.d/rc.sshd
```

The directory listing shows that I have changed the executable status of the script, and therefore prevented the service from starting when the system boots. Depending on your color terminal settings, you may also see the color of the file name change in a listing.

You can use this technique to go through your `/etc/rc.d` directory to turn off those services that you do not need. Since I'm not running an old laptop, and don't need PCMCIA services nor do I have wireless network support on my workstation, I'll make sure these do not have executable permissions:

```
root@forensicbox:~# chmod 644 /etc/rc.d/rc.pcmcia
```

```
root@forensicbox:~# chmod 644 /etc/rc.d/rc.wireless
```

If you want to know what each script does, or if you are unsure of the purpose of a service started by a particular rc script, just open the script with your paging program (**less**, for instance) and read the comments (lines starting with a #). Turning off a service you need is just as bad as leaving an unneeded service running. Learn what service each script starts, and why, and enable or disable accordingly.

For example, here's the comments at the beginning of `/etc/rc.d/rc.yip`:

```
root@forensicbox:~# less /etc/rc.d/rc.yip
#!/bin/sh
# /etc/rc.d/rc.yip
#
# Start NIS (Network Information Service). NIS provides network-wide
# distribution of hostname, username, and other information databases.
# After configuring NIS, you will either need to uncomment the parts
# of this script that you want to run, or tweak /etc/default/yip
#
# NOTE: for detailed information about setting up NIS, see the
# documentation in /usr/doc/yip-tools, /usr/doc/yipbind,
# /usr/doc/yipserv, and /usr/doc/Linux-HOWTOs/NIS-HOWTO.
...
```

I would suggest leaving `sshd` running (via the `/etc/rc.d/rc.sshd` script). Even if you do not think you will use SSH, as you become more proficient with Linux, you will find that SSH, the “secure shell”, becomes an important part of your toolbox.

6.1.2 Host Based Access Control

We continue our baseline security configuration discussion with a word on simple host based access control. Note that this is NOT a firewall. This is access control at the host level. In very simple terms, we can control access to our system with two files, `/etc/hosts.deny` and `/etc/hosts.allow`.

For illustration purposes, let's make sure we can connect from an external host to our forensic workstation via SSH. Remember, we left the `rc.sshd` script executable, so the service started when our computer booted, and at this point we have not set any access controls.

The below command results in a successful connection as user **barry** from the external host **hyperion** to our forensic workstation **forensicbox**. Note the change in the command prompt

on the last line:

```
barry@hyperion:~$ ssh -l barry forensicbox
barry@forensicbox's password:
Last login: Mon Jun 18 16:35:05 2017 from hyperion
Linux 4.4.38.

barry@forensicbox:~$
```

We are now logged into our forensic workstation (`forensicbox`) from a different computer (`hyperion`).

As previously mentioned, there are two access control files we will use. These are `/etc/hosts.deny`, which sets the system wide default policy for access denial, and `/etc/hosts.allow`, which can then be used to poke holes in the denied connections. Both of these files take on the same basic syntax:

services:systems

We start with `/etc/hosts.deny` and use it to deny all incoming connections to all services. We do this by editing the file and adding the string `ALL:ALL` on one single line. When you first open the file for editing, you'll notice that there are no lines that do not start with a `#` sign. That means the entire file is just comments with no real content. Once we add our single line, it will look like this:

```
root@forensicbox:~# cat /etc/hosts.deny

#
# hosts.deny    This file describes the names of the hosts which are
#               *not* allowed to use the local INET services, as decided
#               by the '/usr/sbin/tcpd' server.
#
# Version:  @(#) /etc/hosts.deny 1.00    05/28/93
#
# Author:   Fred N. van Kempen, <waltje@uwaltnl.mugnet.org>
#
#
ALL:ALL.                <- The line we added

# End of hosts.deny.
```

Now any incoming connections will be denied. Note that this is NOT a firewall. It is simply access control to services running on the current system. Since this is `hosts.deny`, we are simply saying "DENY all connections from all hosts".

When we try and **ssh** into our workstation from an external host, we get no connection:

```
barry@hyperion:~$ ssh -l barry forensicbox
kex_exchange_identification: read: Connection reset by peer
```

Once again, in the example above, I'm again trying to log into my forensic workstation, **forensicbox** from the computer **hyperion**). The connection is denied.

Now that we have set a “default deny” policy, let's poke a hole in the scheme by adding an allowed service in. We'll continue to use **sshd** as an example, since I like having access via SSH and will leave it open anyway.

To allow access to a service, we edit the **/etc/hosts.allow** file and add a line for each service in the same **services:systems** format.

When we add an SSH exception for our local network to **hosts.allow**, our **sshd** exception will look like this:

```
root@forensicbox:~# cat /etc/hosts.allow
#
# hosts.allow   This file describes the names of the hosts which are
#               allowed to use the local INET services, as decided by
#               the '/usr/sbin/tcpd' server.
#
# Version:   @(#) /etc/hosts.allow   1.00   05/28/93
#
# Author:    Fred N. van Kempen, <waltje@uwalnt.nl.mugnet.org>
#
#
sshd:192.168.86.          <- The line we added

# End of hosts.allow.
```

This basically reads as “ALLOW connections to **sshd** from systems only on the 192.168.86.0 network”. This limits connections to originating from machines on my local forensic network only. Read the **man** page and adjust to your needs.

Understanding **hosts.deny** and **hosts.allow** gives us a good start on our security configuration. For a typical forensic workstation, this is pretty much as simple as it needs to be at the host level. For many forensic practitioners, simply adding **ALL:ALL** to **hosts.deny** and leaving **hosts.allow** totally empty might be sufficient if you have no need of access.

In order to actually filter traffic at the network interface, we'll need to set up a host based firewall.

6.1.3 Host Based Firewall with **iptables**

It is common practice for many forensic practitioners using other operating systems to utilize some sort of host based firewall program to monitor their workstation's network connections and provide some form of baseline protection from unsolicited access. You may want to do the same thing on your Linux workstation, or you may, in some cases, be required to run a host based firewall by agency or corporate policy. In any event, the most commonly used Linux equivalent for this sort of thing is the **iptables** network packet filter.

There are newer (relatively) network packet filters. Over the past few years **nftables**, managed using **nft**, has become more popular and is included in the mainstream Linux kernel. It is included by default in Slackware and most other distributions and more information can be found on the Internet. For the purpose of this guide, we will stick to **iptables** for now.

Of all the subjects covered in this document, this is one of the more complex, with little direct relationship to actual forensic practice. It is, however, too important not to cover if we are going to discuss Linux as a forensic platform and the required workstation security. A host based firewall may not be a requirement for a good forensic workstation, especially given that many agencies and companies are already working in a well protected (or air gapped) network environment. However, in my humble opinion, it's still a very good idea. It's all too common to see novice Linux users rely completely on the notion that Linux is "just more secure" than other operating systems. And I know from personal experience that there are digital forensic practitioners out there that have workstations connected directly to the Internet and don't take these precautions.

Unlike most of the other subjects covered in this configuration section, **iptables** requires a bit more explanation to effectively set it up from scratch than I'm willing to put in a simple practitioner's guide. As a result, rather than giving a detailed description and step by step instructions, we are going to briefly discuss how to view the **iptables** configuration and provide a baseline script to get the reader started. Our "baseline" script has been provided by Robby Workman (<http://www.rlworkman.net>).

First, we need to make sure we understand the differences between the protections provided by **hosts.allow** and **hosts.deny** vs. **iptables**. As we mentioned earlier, the **hosts*** files block access at the *application* level, whereas **iptables** blocks network traffic at the specified physical network interface. This is an important distinction. **iptables** essentially sits between the network and the applications and accepts or rejects network packets at the kernel level.

In simple terms, **iptables** deals with chains. The **INPUT** chain for incoming traffic, the **OUTPUT** chain for outgoing traffic, and the **FORWARD** chain that handles traffic with neither its origin or destination at the filtered interface. These chains have default policies, to which additional rules can be appended.

Let's have a look at our default **iptables** configuration (in this case "default" means "empty configuration"). To do this we can use **iptables** with the **-S** option to display the rules within

each chain. If you do not provide the chain name (**INPUT**, for example), then the command will list all the chains and their rules, starting with the default policies:

```
root@forensicbox:~# iptables -S
-P INPUT ACCEPT
-P FORWARD ACCEPT
-P OUTPUT ACCEPT
```

The above command lists the policies for each chain along with any rules that may have been added. As you can see from the output here, the default policies are **ACCEPT**, and there are no other rules. None of our network traffic is being filtered.

It is often desirable to hide our systems from all network traffic, including **ping** traffic. With our empty **iptables** configuration, from an external host, we can **ping** our forensic workstation, (192.168.86.83) and the ICMP packets come through:

```
root@hyperion:~# ping 192.168.86.83
PING 192.168.86.83.lan (192.168.86.83) 56(84) bytes of data.
64 bytes from 192.168.86.83.lan (192.168.86.83): icmp_seq=1 ttl=64 time=213 ms
64 bytes from 192.168.86.83.lan (192.168.86.83): icmp_seq=2 ttl=64 time=34.5 ms
64 bytes from 192.168.86.83.lan (192.168.86.83): icmp_seq=3 ttl=64 time=54.2 ms
^C
--- 192.168.86.83.lan ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 3003ms
rtt min/avg/max/mdev = 34.471/95.667/213.198/69.815 ms
```

Now we are going to create an **iptables** script based heavily on the one found at <http://www.rlworkman.net/conf/firewall/rc.firewall.desktop.generic>

Have a look at the following version of the script, edited with **vi** (I'm told this is a good exercise in **vi** editing...), which we'll save as **/etc/rc.d/rc.firewall**. It's important that you get the name right as this is another script that is called from **/etc/rc.d/rc.inet2**, as we discussed earlier. Once the script is created and saved, let's have a look at it.

```
root@forensicbox:~# less /etc/rc.d/rc.firewall
# Define variables

IPT=$(which iptables)      # change if needed
EXT_IF=eth0                # external interface (connected to internet)

# Enable TCP SYN Cookie Protection
if [ -e /proc/sys/net/ipv4/tcp_syncookies ]; then
    echo 1 > /proc/sys/net/ipv4/tcp_syncookies
fi
```

```
# Disable ICMP Redirect Acceptance
echo 0 > /proc/sys/net/ipv4/conf/all/accept_redirects

# Do not send Redirect Messages
echo 0 > /proc/sys/net/ipv4/conf/all/send_redirects

# Set default policy to DROP
$IPT -P INPUT DROP
$IPT -P OUTPUT DROP
$IPT -P FORWARD DROP

# Flush old rules
$IPT -F

# Allow loopback traffic
$IPT -A INPUT -i lo -j ACCEPT
$IPT -A OUTPUT -o lo -j ACCEPT

# Allow packets of established connections and those related to them
$IPT -A INPUT -i $EXT_IF -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT

# Allow all outgoing packets except invalid ones
$IPT -A OUTPUT -o $EXT_IF -m conntrack --ctstate INVALID -j DROP
$IPT -A OUTPUT -o $EXT_IF -j ACCEPT

# Allow incoming ssh (uncomment the line below if needed)
$IPT -A INPUT -i $EXT_IF -p tcp --dport 22 --syn -m conntrack --ctstate NEW -j
    ↪ ACCEPT
```

The file, shown above, starts with variable definitions, followed by a number of lines that set various kernel parameters for better security. We then continue with setting all the default policies for **INPUT**, **OUTPUT** and **FORWARD** to the far more secure **DROP**, rather than simply **ACCEPT** ↪ . Then we define rules that are appended (-A) to the various chains. Also note that I uncommented the last line in the script, referring to TCP traffic (-p tcp) on destination port 22 (--dport 22). This will allow SSH traffic in.

Important Note: Make sure line 4 defines the correct network interface. In the script above we have **EXT_IF=eth0**. For example, if you are using a wireless interface, you might need to change your interface to **wlan0**. You can use the **ifconfig** command to see which interface has an address.

With the file saved to **/etc/rc.d/rc.firewall** we start it by making the file executable. The firewall script, if it exists and has executable permissions, will be called from **/etc/rc.d/rc** ↪ **.inet2**. Check the permissions on the file, change them to executable (using **chmod**) and check again. We can load the rules by simply calling the script explicitly:

```
root@forensicbox:~# ls -l /etc/rc.d/rc.firewall
-rw-r--r-- 1 root root 1080 Jun 18 22:22 /etc/rc.d/rc.firewall

root@forensicbox:~# chmod 755 /etc/rc.d/rc.firewall

root@forensicbox:~# ls -l /etc/rc.d/rc.firewall
-rwxr-xr-x 1 root root 1080 Jun 18 22:22 /etc/rc.d/rc.firewall*

root@forensicbox:~# sh /etc/rc.d/rc.firewall
```

The last command in the session illustrated above shows that we have executed the firewall script. Now when we look at our `iptables` configuration, we see the rules in place:

```
root@forensicbox:~# iptables -S
-P INPUT DROP
-P FORWARD DROP
-P OUTPUT DROP
-A INPUT -i lo -j ACCEPT
-A INPUT -i eth0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A INPUT -i eth0 -p tcp -m tcp --dport 22 --tcp-flags FIN,SYN,RST,ACK SYN -m
    ↪ conntrack --ctstate NEW -j ACCEPT
-A OUTPUT -o lo -j ACCEPT
-A OUTPUT -o eth0 -m conntrack --ctstate INVALID -j DROP
-A OUTPUT -o eth0 -j ACCEPT
```

Here we see the default policies (-P) are now set to **DROP** and we have several rules in each chain. The lines starting with -A signify that we are appending a rule to our chain. Note that since our default policy is to drop all incoming traffic, and there is no explicit rule to allow incoming ICMP traffic, we can no longer ping our forensic workstation from an external host. We can, however, connect with SSH since we have a rule that accepts TCP traffic on destination port 22 (assuming you un-commented the last line of the script and assuming the SSH server is running on the default port).

```
root@hyperion:~# ping 192.168.86.32
PING 192.168.86.32 (192.168.86.32) 56(84) bytes of data.
^C
--- 192.168.86.32 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5153ms
```

Trying to ping the forensic workstation (at IP 192.168.86.32) from a host called hyperion. If you need to allow other traffic, there are many tutorials and examples on the Internet to work from. This is a very simple and generic host based network packet filter. And as with the other subjects in this guide, it is meant to provide a primer for additional learning.

6.2 Updating the Operating System

Keeping the operating system up to date is an important part of workstation security. Most Linux distributions come with some sort of mechanism for keeping the OS up to date with the latest security and stability patches. If you choose to use a distribution other than Slackware, then be sure to check the appropriate documentation.

- Debian and Ubuntu – synaptic aptitude
- Fedora – yum
- Arch Linux – pacman
- Gentoo – portage
- Slackware – pkgtools

From the perspective of a forensic workstation, Slackware takes a particularly conservative (and therefore safe) approach to updating the operating system. Once a Slackware release is considered “stable”, the addition of updated library and binary packages is generally limited to those required for a properly patched OS. Little or no emphasis is placed on running the “latest and greatest” for the simple sake of doing so. I would strongly suggest against continuously updating software without having a good reason (security patches, for example). New versions of critical libraries and system software should always be tested before use in a production forensic environment – this does not imply re-validation with every update. That’s up to your own policies and procedures.


Note that with some distributions, updating the OS on a regular basis, without proper and often complex configuration, can result in a dozen or so new and updated packages every couple of weeks. In the context of a stable, well tested forensic platform, this is less than ideal. Also, Slackware developers tend not to patch upstream code, as is common among some other distributions. Slackware takes the approach of “if it ain’t broke, don’t fix it.”

This information is not meant to disparage other distributions. Far from it. Any properly administered Linux distribution makes a fine forensic platform. These are, however, important considerations if you are running a forensic workstation in any sort of litigious setting. Too often, Linux Forensics beginners trust their platform to numerous untested, desktop oriented updates, without thinking about potential changes in behavior that can, in admittedly limited circumstances, raise questions.

6.2.1 Slackware’s **pkgtools**

Slackware provides a set of utilities for manipulating packages in **pkgtools**. These are **pkgtool** (menu driven), **installpkg**, **removepkg** and **upgradepkg**. These are discussed in detail here: <https://www.slackbook.org/html/package-management-package-utilities.html>.

6.2.2 slackpkg for automated updates

We are going to focus on an automated tool for package management in Slackware - **slackpkg** . It's extremely easy to configure and use. The **man** page provides very clear instructions on using **slackpkg** along with a good description of some of it's capabilities.

We start by picking a single "mirror" (Slackware repository) listed in the **slackpkg** configuration file. Log in as **root** and open `/etc/slackpkg/mirrors` with **vi** (or your editor of choice). Un-comment a single line and you're ready to go (delete the **#** sign from the front of an address). The line you un-comment needs to be for the specific architecture (32 bit vs 64 bit, etc.) and version of Slackware you are running⁷, and should be near your geographic region (US, UK, Poland, etc.).

Take note that "Slackware-current" is a development branch of Slackware and is NOT suitable for our purposes. Do not select a mirror from the Slackware-current list.

The below example shows an edited `/etc/slackpkg/mirrors` file where a single mirror for a server in the USA has been un-commented (bold for emphasis - note the **#** has been removed). The mirror we are selecting is for Slackware64-14.2. Select a mirror appropriate for your location.

```
root@forensicbox:~# vi /etc/slackpkg/mirrors
...
#-----
# Slackware64-14.2
#-----
# USE MIRRORS.SLACKWARE.COM (DO NOT USE FTP - ONLY HTTP FINDS A NEARBY MIRROR)
http://mirrors.slackware.com/slackware/slackware64-14.2/
#
# AUSTRALIA (AU)
# ftp://ftp.cc.swin.edu.au/slackware/slackware64-14.2/
# http://ftp.cc.swin.edu.au/slackware/slackware64-14.2/
# ftp://ftp.iinet.net.au/pub/slackware/slackware64-14.2/
# http://ftp.iinet.net.au/pub/slackware/slackware64-14.2/
# ftp://mirror.aarnet.edu.au/pub/slackware/slackware64-14.2/
...
```

One precaution you may want to take with **slackpkg** is to add several packages to the **blacklist**. The **blacklist** specifies those programs and packages that we do not want upgraded on a regular basis. We do this to avoid having to complicate periodic security updates with changes to our bootloader and other components that add excessive complexity to our

⁷Pay attention to the architecture and version. I made a complete muppet of myself on the **##slackware** IRC channel one day, asking for help when I was trying to upgrade Slackware64 (64 bit OS), not knowing I had selected a 32 bit mirror and therefore destroying my system when I updated.

upgrade process. In particular, we want to avoid (for now) having to go through all the steps required to upgrade our kernel packages.

The `blacklist` file is located at `/etc/slackpkg/blacklist` and there are several lines regarding kernel upgrades that are included but commented out. Un-comment those lines by removing the leading `\#` symbol, and add the additional lines as shown so the file looks like this (in part):

```
root@forensicbox:~# vi /etc/slackpkg/blacklist
...
# Automated upgrade of kernel packages aren't a good idea (and you need to
# run "lilo" after upgrade). If you think the same, uncomment the lines
# below
#
kernel-firmware
kernel-generic
kernel-generic-smp
kernel-headers
kernel-huge
kernel-huge-smp
kernel-modules
kernel-modules-smp
kernel-source
...
```

When a critical update to one of the kernel packages is required, the lines in the blacklist can always be temporarily commented out and the packages updated as usual. If you leave the lines commented out, you will get periodic kernel updates. Just remember that an updated kernel will require you to update your boot loader as well. You will be prompted for this anyway.

We've selected our mirror and adjusted our blacklisted packages, now it is simply a matter of updating our package list...we do this with the simple command **slackpkg update**, which will download the current file list (including patches). Once that is complete, you run **slackpkg ↪ upgrade-all** and you will be presented with a selection of packages to upgrade (minus the blacklisted packages).

The man page for **slackpkg** provides easy to follow instructions. In a nutshell, for our purposes here, usage is simply:

1. un-comment a mirror in `/etc/slackpkg/mirrors`
2. optionally add files (or un-comment entries) in `/etc/slackpkg/blacklist`
3. run `slackpkg update`

4. run `slackpkg upgrade-all`

I would strongly suggest you take a minute to read the change log for the version of Slackware (or whatever distribution) you are using. Understanding what you are updating and why is an important part of understanding your forensic platform. It may seem tedious at first, but it should be part of your common system maintenance tasks. You can read the file `ChangeLog.txt` at the mirror you selected for updating your system, or simply go to: <https://mirror.SlackBuilds.org/slackware/slackware64-14.2/ChangeLog.txt> when updates are available.

Using the **`slackpkg`** method above is the easiest way to keep your OS up to date with the latest stable security fixes and patches. Periodically, you can run **`slackpkg update`** and **`slackpkg upgrade-all`** to keep your system up to date. The first two steps only need to be done once on your system.

Once again, if you are not using Slackware, be sure to check your distribution's documentation to determine how best to keep your workstation properly patched. But please continue to bear in mind that "latest and greatest" does not translate to "properly patched". An important distinction.

6.3 Installing and Updating "External" Software

So we've discussed using **`slackpkg`** for updating the OS packages and keeping the system properly patched and updated. What about "external" software, that is, software that is not included in a default installation, like our forensic utilities? There are a number of ways we can install this "external" software on our system.

1. Compile from source
2. Use a pre-built package (usually distro dependent)
3. Build your own package



6.3.1 Compiling From Source

Compiling from source is the most basic method for installing software on Linux. It is generally distribution agnostic and will work for any given package on most distributions, assuming dependencies are met. Correctly used, compiling from source has the benefit of being tailored more to your environment, with better optimization. The biggest drawback is that compiling from source, without careful manipulation of configuration files, can "litter" your system with executables and libraries placed in less than optimal locations. It can

also result in difficult to manage upgrade paths for installed software, or even just trying to remember what you have previously installed.

The source files (containing source code) normally come in a package commonly referred to as a “tarball”, or a **tar.gz** file (a **gzip**compressed tar archive). The archive is extracted, the source is compiled, and then an install script is executed to place the resulting program files and documentation in the appropriate directories. The following shows a very abbreviated view of a quick source compilation. The normal course of commands used is (usually):

```
tar xzvf packagename.tar.gz
cd packagename
./configure
make
make install
```

First we extract the package and change into the resulting directory. The **./configure**  command ⁸ sets environment variables and enables or disables program features based on available libraries and arguments. The **make** command compiles the program, using the parameters provided by the results of the previous **./configure** command. Finally, the **make**  **install** command moves the compiled executables, libraries and documentation to their respective directories on the computer. Note that **make install** is generally not distribution aware, so the resulting placement of program files might not fit the conventions for a given Linux distro, unless the proper variables are passed during configuration.

Here's a quick illustration:

Once we have a package downloaded, we extract the tarball. After the package has been extracted, we change into the resulting directory and then run a “configure script” to allow the program to ascertain our system configuration and prepare compiler options for our environment. We do this with **./configure**

```
root@forensicbox:~# tar xzvf package.tar.gz
...
root@forensicbox:~# cd package

root@forensicbox:~# ./configure
...
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
configure: autobuild project... package
...
```

Assuming no errors, we type **make** and watch the compiler go to work. Finally, we run the

⁸The **./** indicates that the **configure** command is run from the current directory

command that properly installs both the tools to the proper path, and any required libraries to the proper directories. This is generally accomplished with **make install**

```
root@forensicbox:~# make
Making all in lib
make[1]: Entering directory '/root/package/lib'
...

root@forensicbox:~# make install
Making install in lib
make[1]: Entering directory '/root/package'
make install-am
make[2]: Entering directory
...
```

Our program is now installed and ready to use. Knowing how to use source packages for software installation is important part of understanding how Linux works...just keep in mind that it's generally a better idea to use distribution packages (or create your own). Note that the example shown above is for source packages built with autoconf/automake. You may also run across software that is Python or Perl based, etc. These will differ in how they are built and installed. Most source packages will include a **README** or **INSTALL.txt** file when extracted. Read them.

6.3.2 Using Distribution Packages

As we've already mentioned, just about every Linux distribution has some sort of "package manager" for installing and updating packages. For updating and adding official Slackware software (included in the distribution), we've introduced using **slackpkg**. **slackpkg** is actually a front end to the **pkgtool** group of utilities which handle the work of adding and removing software packages from your system. For an excellent overview of **pkgtool** and its various commands, have a look at <http://www.slackware.com/config/packages.php>.

Slackware packages are really just compressed archives that, when installed, place the package files in the proper place. To install a Slackware package, when we are not using the **slackpkg** front end, we use the **pkgtool** command **installpkg**.

Our example here will be a pretend Slackware package called **software**. Slackware packages are generally named with the extension **tgz** or **txz** (since they are really just compressed archives). Once you've downloaded or prepared your package (our example is called **software** → **.tgz**), you install it with the following command:

```
root@forensicbox:~# installpkg software.tgz
Verifying package software.tgz.
Installing package software.tgz:
```

```
PACKAGE DESCRIPTION:
# software
#
# This is where you will find a description
# of the software package you are installing
#
#
# Homepage:  http://www.software.homepage.com
#
Executing install script for software.tgz.
Package software.tgz installed.
```

Packages can be similarly removed or upgraded with **removepkg** or **upgradepkg**, respectively.

You can find pre-made packages for all sorts of software for many distributions all over the Internet. The problem with many of them is that they do not come from trusted sources and you often have no idea what configuration options were used to build them.

As a general rule of thumb, I always like to build my own packages for software that is not part of the Slackware full installation. This allows me to build the software with the options I need (or without ones I don't), optimized for my particular system, and it further allows me to control how the software is eventually installed. Luckily Slackware provides a relatively easy way to create packages from source code. *SlackBuilds*.

6.3.3 Building Packages with SlackBuilds

In short, a SlackBuild is a script that (normally) takes source code and compiles and packages it into a Slackware **.tgz** (or **.ttx**) file that we can install using **installpkg**.

The SlackBuild script handles the configure options and optimizations that the script author decides on (but are visible and editable by you), and then installs the software and related files into a package that follows Slackware software conventions for executable and libraries, where applicable, and assuming the build author follows the template. The scripts are easily editable if you want to change some of the options or the target version, and provide for an easy, human readable way to control the build process. SlackBuilds for a large selection of software are available at <http://www.SlackBuilds.org>.

The SlackBuild itself comes as a **.tar.gz** file that you extract with the **tar** command. The resulting directory contains the build script itself. The script is named **software.SlackBuild** ↪ , with software being the name of the program we are creating a package for. There are normally four files included in the SlackBuild package:

- **software.info** gives information about where to obtain the source code, the version of

the software the script is written for, the hash of the source code, required dependencies, and more.

- **README** contains useful information about the package, potential pitfalls, and optional dependencies.
- **software.SlackBuild** is the actual build script.
- **slack-desc** is a brief description of the file displayed during install.

To build a Slackware compatible package, you simply drop the source code for the software into the same directory the SlackBuild is in (no need to extract the source tarball) and execute the SlackBuild script. The package is created and (normally) placed in the **/tmp** directory ready for installation via **installpkg**.

Of course, there are automated tools to handle building Slackware packages for you. You can check out **sbopkg**, **sbotools**, **slpkg** and a few others.

A WORD OF CAUTION: Be careful about relying solely on automated tools for package management. Regardless of the platform you choose to run on, I would urge you to learn how to build packages yourself, or at the very least learn how to determine how to change package options or determine what build options were used before running software. This is not to say automated tools are bad...but one of the strengths of Linux that we often talk about is the control it gives us over our system. Controlling your system software is one aspect of that. You can use automated tools and still maintain control...you just need to be careful. We will use that approach here.

We will talk specifically about one of the package tools you can use with Slackware to automate some of the more mundane steps we take when installing software. To illustrate the build process, we will install **sbotools** via a manual SlackBuild process, and then use **sbotools** to assist us in building and installing the remainder of the software we'll use in this guide.

First, we'll grab the SlackBuild from <https://www.SlackBuilds.org>. You can go to the website search and browse the packages there, but since we know the package we want, we'll use the **wget** tool to download it directly. In the next set of commands we'll accomplish the following:

- download the SlackBuild tarball for **sbotools** with **wget**
- extract the contents of the tarball with the **tar** command
- change (**cd**) to the resulting **sbotools** directory and list the files (**ls**)

```
root@forensicbox:~# wget https://www.SlackBuilds.org/SlackBuilds/14.2/system/
sbotoools.tar.gz
```

```
--2019-06-20 16:58:35-- https://www.slackbuilds.org/slackbuilds/
14.2/system/sbotoools.tar.gz
Resolving www.slackbuilds.org (www.slackbuilds.org)... 208.94.237.149
Connecting to www.slackbuilds.org (www.slackbuilds.org)|208.94.237.149|
:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2038 (2.0K) [application/x-gzip]
Saving to: 'sbotoools.tar.gz'
```

```
sbotoools.tar.gz      100%[=====>]    1.99K  --.-KB/s    in 0s
```

```
2019-06-20 16:58:36 (90.5 MB/s) - 'sbotoools.tar.gz' saved [2038/2038]
```

```
root@forensicbox:~# ls
sbotoools.tar.gz
```

```
root@forensicbox:~# tar xzvf sbotoools.tar.gz
sbotoools/
sbotoools/sbotoools.info
sbotoools/slack-desc
sbotoools/README
sbotoools/sbotoools.SlackBuild
```

```
root@forensicbox:~# cd sbotoools
```

```
...
```

```
root@forensicbox:~# ls
README  sbotoools.SlackBuild*  sbotoools.info  slack-desc
```

So what we've done up to this point is just download and extract the SlackBuild package. Now we need to get the **sbotoools** source package in the same directory.

The **sbotoools.info** file will help with this. We'll view that file and then use the information contained therein to download the source code and check the MD5 hash. The MD5 hash is a value that lets us know the file we download is what we expect. Using **wget** and the URL provided in the **DOWNLOAD** field, the source code for **sbotoools** will end up in the same directory.

```
root@forensicbox:~# cat sbotoools.info
PRGNAM="sbotoools"
VERSION="2.7"
HOMEPAGE="https://pink-mist.github.io/sbotoools/"
DOWNLOAD="https://pink-mist.github.io/sbotoools/downloads/
```

```
sbotoools-2.7.tar.gz"
MD5SUM="ddf4b174fa29839564d7e784ff142581"
DOWNLOAD_x86_64=""
MD5SUM_x86_64=""
REQUIRES=""
MAINTAINER="Andreas Guldstrand"
EMAIL="andreas.guldstrand@gmail.com"

root@forensicbox:~# wget https://pink-mist.github.io/sbotoools/downloads/
sbotoools-2.7.tar.gz

--2019-06-21 08:10:10-- https://pink-mist.github.io/sbotoools/downloads/
sbotoools-2.7.tar.gz

Resolving pink-mist.github.io... 185.199.108.153, 185.199.111.153,
185.199.110.153, ...
Connecting to pink-mist.github.io|185.199.108.153|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 45833 (45K) [application/gzip]
Saving to: 'sbotoools-2.7.tar.gz'

sbotoools-2.7.tar 100%[=====>] 44.76K --.-KB/s in 0.1s

2019-06-21 08:10:10 (384 KB/s) - 'sbotoools-2.7.tar.gz' saved [45833/45833]

root@forensicbox:~# ls
README sbotoools-2.7.tar.gz sbotoools.SlackBuild* sbotoools.info slack-desc

root@forensicbox:~# md5sum sbotoools-2.7.tar.gz
ddf4b174fa29839564d7e784ff142581 sbotoools-2.7.tar.gz
```

The output from our **md5sum** command on the downloaded source matches the **MD5SUM** field in the **sbotoools.info** file, so we know our download is good.

This is where, if we have not already done so, we need to read the **README** file (using **cat** ↪ or **less**)...understand the caveats and possible optional dependencies...and then compile our source code and make our Slackware **.tgz** package. The latter two steps are simply accomplished by calling the SlackBuild file itself with **./sbotoools.Slackbuild**:

```
root@forensicbox:~# ./sbotoools.SlackBuild
sbotoools-2.7/
sbotoools-2.7/sboclean
sbotoools-2.7/man5/
sbotoools-2.7/man5/sbotoools.conf.5
...
Checking if your kit is complete...
Looks good
```

Generating a Unix-style Makefile

```
...
Creating Slackware package: /tmp/sbotoools-2.7-noarch-1_SBo.tgz

./
usr/
usr/share/
...
Slackware package /tmp/sbotoools-2.7-noarch-1_SBo.tgz created.
```

And looking at the last line of the output, we see that we have a usable `.tgz` Slackware package created for us in `/tmp`. All we need to do now is install the package with **installpkg** from **pkgtools**:

```
root@forensicbox:~# installpkg sbotoools-2.7-noarch-1_SBo.tgz
Verifying package sbotoools-2.7-noarch-1_SBo.tgz.
Installing package sbotoools-2.7-noarch-1_SBo.tgz:
PACKAGE DESCRIPTION:
# sbotoools (ports-like interface to slackbuilds.org)
#
# sbotoools is a set of perl scripts providing a ports-like automation
# interface to slackbuilds.org. Its features include requirement
# handling and the ability to handle 32-bit and compat32 builds on
# multilib x86_64 systems.
#
# https://pink-mist.github.io/sbotoools/
#
Package sbotoools-2.7-noarch-1_SBo.tgz installed.
```

6.3.4 Using the automated package tool **sbotoools**

So, now we've installed **sbotoools**, and we are going to use it in lieu of all the downloading, MD5 checks, extracting and building. It is extremely important that we remain mindful of the **README** files and ensure that we don't allow the automation to make us complacent. Read the documentation for each package you are installing and be familiar with what it is doing to your system along with what options you may want to enable or disable.

sbotoools is actually a collection of utilities. The very first time we call **sbotoools**, we need to initialize the SlackBuild repository. By default, **sbotoools** (via **sbosnap**) will pull the entire SlackBuilds tree (from [SlackBuilds.org](https://slackbuilds.org) and place it in `/usr/sbo/repo`.

```
root@forensicbox:~# sbosnap fetch
Pulling SlackBuilds tree...
 102,492,492 100%    1.30MB/s    0:01:15 (xfr#53203, to-chk=0/61135)
```

...

```

root@forensicbox:~# ls -l /usr/sbo/repo
total 10792
-rw-r--r-- 1 root root 3591376 Jun 14 22:00 CHECKSUMS.md5
-rw-r--r-- 1 root root 195 Jun 14 22:03 CHECKSUMS.md5.asc
-rw-r--r-- 1 root root 1176734 Jun 14 21:27 ChangeLog.txt
-rw-r--r-- 1 root root 329 Jun 14 2018 README
-rw-r--r-- 1 root root 4021715 Jun 14 21:59 SLACKBUILDS.TXT
-rw-r--r-- 1 root root 810115 Jun 14 21:59 SLACKBUILDS.TXT.gz
-rw-r--r-- 1 root root 358076 Jun 14 21:47 TAGS.txt
-rw-r--r-- 1 root root 131691 Jun 14 21:47 TAGS.txt.gz
drwxr-xr-x 296 root root 36864 Jun 8 18:40 academic/
drwxr-xr-x 20 root root 4096 Feb 22 20:34 accessibility/

```

Once this is done, you can search, install and upgrade packages and their initial dependencies all from single commands using the following:

```

sbofind      : search for packages based on names and keywords
sbocheck     : update the repository and identify packages that need upgrading
sboinstall   : install a package (and it's dependencies)
sboupgrade   : upgrade an already installed package

```

We'll be using **sbotools** to install software throughout the remainder of this document (if you are using Slackware). But let's start with a quick example of a simple installation for some anti-virus/malware detection software that we'll cover later.

I have a clean Slackware install with a single external software package, **sbotools**, installed. Now I want to install more software.

To illustrate a more automated install - but one that still requires user intervention - we'll install ClamAV (**clamav**), a virus/malware scanner. We first use **sbofind** to search for **clamav** → . We then narrow our search and take a quick look at the **README** file. Then we simply run **sboinstall** to download, check, build and install the package for us. The shortcut to all this is to simply type **sboinstall clamav** and we're done. But I prefer a more cautious approach.

First, let's search for available packages that match **clamav**:

```

root@forensicbox:~# sbofind clamav
SBo:    thunar-sendto-clamtk 0.06
Path:   /usr/sbo/repo/desktop/thunar-sendto-clamtk

SBo:    clamav-unofficial-sigs 5.6.2
Path:   /usr/sbo/repo/network/clamav-unofficial-sigs

```

```
SBo:    clamav 0.101.2
Path:   /usr/sbo/repo/system/clamav

SBo:    clamsmtp 1.10
Path:   /usr/sbo/repo/system/clamsmtp

SBo:    clamtk 5.26
Path:   /usr/sbo/repo/system/clamtk

SBo:    squidclamav 6.16
Path:   /usr/sbo/repo/system/squidclamav
```

The clamav package is the third one down. Now I'm going to run **sbofind** again, but this time limit the output to an exact match for clamav (**-e**) with no tags (**-t**) and view the README file for the package (**-r**).

```
root@forensicbox:~# sbofind -t -e -r clamav
```

```
SBo:    clamav 0.101.2
Path:   /usr/sbo/repo/system/clamav
```

```
README:
```

```
Clam AntiVirus is a GPL anti-virus toolkit for UNIX. The main purpose
of this software is the integration with mail servers (attachment
scanning). The package provides a flexible and scalable multi-threaded
daemon, a command line scanner, and a tool for automatic updating via
Internet.
```

```
This build script should build a package that "just works" after install.
You will need to specify a two-letter country code (such as "us") as an
argument to the COUNTRY variable when running the build script (this will
default to "us" if nothing is specified). For example:
```

```
COUNTRY=nl ./clamav.SlackBuild
```

Groupname and Username

You must have the 'clamav' group and user to run this script,
for example:

```
groupadd -g 210 clamav
useradd -u 210 -d /dev/null -s /bin/false -g clamav clamav
```

Configuration

See README.SLACKWARE for configuration help.

And what we have here is a perfect example of why we read the **README** files prior to installing software. In order to make it run correctly, we need to make sure we have a group and user called **clamav**. The commands we need to accomplish this are provided right in the **README**. So we run those and then we are ready to install the software. You can even allow **sbotools** to run the commands for you, but I would suggest you run them yourself and decline the prompt in the **sboinstall** command. **clamav** also has a secondary **README.Slackware** file with additional instructions for running the program as a mail scanner. You can elect to read that as well, if you like, though we won't be setting that up in this example.

```
root@forensicbox:~# groupadd -g 210 clamav
```

```
root@forensicbox:~# useradd -u 210 -d /dev/null -s /bin/false -g clamav clamav
```

```
root@forensicbox:~# sboinstall clamav
```

Now **sbotools** will download, check, unpack, configure, build and finally install the package for us. We'll continue to use this method to install software through the rest of this guide. We will cover ClamAV usage later in this document.

Remember we can periodically use **sbocheck** to see if we have any external software that needs updating (recall that **slackpkg update** is used for official Slackware packages).

We'll also install another package we talked about previously. Back when we did our initial system inventory, we described the **lshw** command. We can install that easily from SlackBuilds.org using **sboinstall**.

First, let's make sure we can find **lshw**, and then read the **README** file:

```
root@forensicbox:~# sbofind lshw
```

output

```
SBo:    lshw B.02.18
```

```
Path:   /usr/sbo/repo/system/lshw
```

```
root@forensicbox:~# sbofind lshw -r
```

```
SBo:    lshw B.02.18
```

```
Path:   /usr/sbo/repo/system/lshw
```

README:

```
lshw (Hardware Lister) is a small tool to provide detailed information on
the hardware configuration of the machine. It can report exact memory
configuration, firmware version, mainboard configuration, CPU version and
speed, cache configuration, bus speed, etc. on DMI-capable x86 or EFI
(IA-64) systems and on some PowerPC machines (PowerMac G4 is known to work).
```

```
Information can be output in plain text, XML, or HTML.
```

```
It currently supports DMI (x86 and EFI only), OpenFirmware device tree
```

(PowerPC only), PCI/AGP, ISA PnP (x86), CPUID (x86), IDE/ATA/ATAPI, PCMCIA (only tested on x86), USB, and SCSI.

On x86, `lshw` needs to be run as root to be able to access DMI information from the BIOS. Running `lshw` as a non-root user usually gives much less detailed information.

When we actually run the **`sboinstall`** command, the **README** is displayed by default anyway, but we show it above for explicitness. I prefer to read the **README** before the install command so I know what to expect and what caveats to prepare for. And now we simply install the build:

```
root@forensicbox:~# sboinstall lshw
```

```
Proceed with lshw? [y]
```

```
Install queue: lshw
```

```
Are you sure you wish to continue? [y]
```

```
Executing install script for lshw-B.02.18-x86_64-1_SBo.tgz.  
Package lshw-B.02.18-x86_64-1_SBo.tgz installed.
```

```
Cleaning for lshw-B.02.18...
```

One final note on package management. A complete list of packages installed on your system is maintained in `/var/log/packages`. You can browse that directory to see what you have installed, as well as view the files themselves to see what was installed with the package. One nice thing about using SlackBuilds is that an **SBo** tag is added to the package name. We can **grep** for this tag in `/var/log/packages` and see exactly which external packages we have installed via SlackBuilds. This is one of the great advantages of using a package manager vs. simply compiling and installing software from source directly...the ability to track what versions of which packages are installed.

We have just installed three packages using build scripts from [SlackBuilds.org](https://slackbuilds.org). One via manual download (`sbotools`), and two via `sbotools` (`clamav` and `lshw`). We can use **grep** to see this within the `/var/log/packages` directory (assuming this is a clean Slackware system and you've installed no other `.tgz` or `.txz` Slackware packages):

```
root@forensicbox:~# ls /var/log/packages/ | grep SBo  
clamav-0.101.2-x86_64-1_SBo  
lshw-B.02.18-x86_64-1_SBo  
sbotools-2.7-noarch-1_SBo
```

When it comes time to upgrade (or check for updates to) software we've installed via SlackBuilds and **sbotools**, you can use **sbocheck**. Running this command will fetch a fresh SlackBuilds tree from SlackBuilds.org and compare your installed packages to those currently available.

```
root@forensicbox:~# sbocheck
Updating SlackBuilds tree...
    10,564,128  10%  906.15kB/s    0:00:11 (xfr#280, to-chk=0/61163)
Checking for updated SlackBuilds...

No updates available.
```

In this case, we run **sbocheck** and there are no updates. If an update were listed in the output, we would simply run **sboupgrade <packagename>** and the upgrade would be downloaded, compiled, and properly upgraded. This provides us an easy way to install and upgrade external packages, in a Slackware friendly format, with minimal fuss.

The earlier caution still stands. Make sure you understand what you are installing and always always read the **README** file.

7 Linux and Forensics

Here we begin our study of the actual tools and some of the associated processes we'll use for forensics. The process is simplified here, focusing on the basics:

- Organizing output
- Acquiring disk based evidence
- Obtaining a hash
- Mounting acquired evidence volumes
- Scanning for malware (limited to certain evidence types)
- Basic data review using the command line

This is meant to introduce tools, concepts, and analytical approaches to forensics while using Linux. We've mentioned it previously, but there have been significant changes to hardware in the past couple of years that will require some additional research by the reader.

As with the majority of this guide, we'll be concentrating on using command line tools for the above steps. These are the basics, and should not be considered as a template for real life examinations. This is a 'walk before you run' approach.

7.1 Evidence Acquisition

In this section we'll run through a few of the acquisition tools that are available to us. We'll cover some of the collection issues, device information, image verification and more advanced mounting options. Obviously, the first thing we need to do is make sure we have a proper place to output the results of our imaging and analysis.

As we go through the following sections, try and use an old (smaller) hard drive to follow along. Find an old SATA drive (the one I'm using is 40GB) and attach it to your computer, either to the SATA bus directly, or through a USB (preferably 3.x) bridge. That way you can follow along with the commands and compare the output with what we have here (which will differ depending on your hardware). You can even use a USB thumb drive, but then the output for some of the media information collection sections will not provide comparable output. The best way to learn this material is to actually do it and experiment with options.

7.2 Analysis Organization

Before we start collecting evidence images and information that might become useful in a court or an administrative hearing, we might want to make sure we store all this data in an organized fashion. Obviously this is not something specific to Linux, but we need to make sure we have several file system locations ready to store and retrieve data:

1. Case specific directories or volumes used to store forensic images for a given case.
2. Case specific directories for storing forensic software output and subject media information.
3. Specific directories to be used as mount points for evidence images.
4. A log file of our actions. Documentation and note taking are an imperative part of proper forensics.

Wherever you might store your case data, you'll want to keep it organized. In most cases, when conducting an analysis, you'll want to make sure you are using "working copies" rather than the actual image files. This should be common practice. Practitioners will often collect images or other data directly as evidence. Copies will then be made of that evidence, with the originals being placed in some sort of controlled storage. Additional copies (perhaps multiple additional copies) are then made as "working copies". We will discuss the simple creation of directories to store these files as we move through the upcoming pages. The following is just an example of how you might organize the various directories in which you are storing data. Obviously nothing will be written to the subject disk (the disk we are analyzing). Then in the section following that we will describe how to identify the correct disks so you don't confuse the subject disk with the disk or volume you will use to write your images to.

NOTE: All of these preparation steps should be taken before you connect a subject disk to your workstation to minimize the chances of writing to the wrong drive. Proper lab setup (dedicated imaging workstations or images storage, etc.) is outside the scope of this document. For simplicity and illustration, we'll assume you have a single workstation and will be collecting an image from one drive (subject) and writing the image files on a mounted volume or local directory.

You may also want to prepare your evidence drive by wiping and verifying. We'll also cover that later once we've had a better introduction to imaging tools.

On the evidence drive (where evidence images are to be stored) you might want to create a top level directory with a case number or other unique identifier for images. Depending on the tool you use to acquire, an acquisition log might be placed in this directory (or specified location). The only other files that might normally be kept with the original evidence images would be the acquisition log (more on that later) and perhaps the media information files

(more on that later as well). Pay attention to the prompts in the following examples to ensure you have root permissions when needed (like when writing to the `/mnt` directory). Some distributions will have you use **sudo** rather than logging in as **root**. In those cases, just precede each command with **sudo**.

First, in order to make sure you have enough room on your target storage, you can run the **df -h** command. This “disk free” command will show you the free space on each of your mount points. For example, If you have a 1TB evidence partition on an external disk, you confirm that it is detected and properly identified. Then mount it to your evidence directory and then check the free space:

```
root@forensicbox:~# lsscsi
...
[0:0:0:0]    disk    ATA      INTEL SSDSC2CT12 300i  /dev/sda
[2:0:0:0]    disk    ATA      Hitachi HDS72302 A5C0  /dev/sdb
[3:0:0:0]    cd/dvd  HL-DT-ST DVDROM GH24NS90 IN01  /dev/sr0
[8:0:0:0]    disk    Seagate  BUP Slim BK      0304  /dev/sdc
...

root@forensicbox:~# mkdir /mnt/evidence

root@forensicbox:~# lsblk | grep sdc
sdc      8:32    0    1.8T    0 disk
├─sdc1    8:33    0      1T    0 part
└─sdc2    8:34    0    839G    0 part

root@forensicbox:~# mount /dev/sdc1 /mnt/evidence

root@forensicbox:~# df -h /mnt/evidence
Filesystem      Size  Used Avail Use% Mounted on
/dev/sdc1       1008G  375G  582G   40% /mnt/evidence
```

From this output, I can see that the file system mounted on `/mnt/evidence` has 582GB of free space. The **df** command is used with **-h** to give “human readable” output, and the mount point is passed as an argument to limit the output. If given without arguments, **df -h** will show the free space on all mounted file systems.

For illustration in the following examples we will be writing our output to a case directory in `/mnt/evidence`. In the command below, we are creating the `case1` directory in `/mnt/evidence`

```
root@forensicbox:~# mkdir /mnt/evidence/case1

root@forensicbox:~# ls /mnt/evidence
case1/
```

Once you've prepared the evidence drive and destination directory, you can connect the subject disk. Keep in mind our previous discussion regarding write blocking. It's always a good idea to use a write blocker or ensure that you are absolutely certain you are not writing to your evidence. There is some misconception in the community now on what constitutes an effective "write blocker". In the case of modern storage media, physical write blocking might not be possible in the sense of the normally acceptable context. Modern solid state media does not physically operate the same as the spinning disks many examiners are used to. More on this in the following section.

A default install of Slackware (using the XFCE desktop, at least) will not attempt to auto mount attached devices. But you should thoroughly test your system before relying on this (or any other operating system).

7.3 Write Blocking

Let's discuss this in some more detail. When we talk about "write blocking", in most cases we are referring to "preventing unwanted writes to media". What we cannot control (directly) are writes to media by onboard controllers. In this guide we are referring to write blocking as "preventing users from making changes to attached media via user actions". Discussions of "wear leveling" and "trim operations" are outside the scope of our discussion.

In the past, much was made about the ability to mount volumes as "read only" in Linux. This should never be trusted other than to provide the very minimum of accidental changes to a working copy, or when no other options exist (and always document those instances). This guide is about using tools, so while covering acquisition policies is not our purpose, it bears mentioning that write protection is something that should always be kept in mind. Modern computing environments are extremely complex, and unless you've tested every function in every possible setting, there's no way to be completely certain that some underlying kernel mechanism is not making unknown or unexpected writes to poorly protected evidence drives through some previously untested interface or other mechanism.

Write blocking can be as simple as the physical switch on removable media, or as exotic (and expensive) as purpose built forensic write blockers. There are also methods available for "software" write blocking (various kernel patches and other scripts) that will let you set devices as read only, as with **blockdev**, but again, your mileage may vary on those techniques. Many kernel level hardware settings take place after the kernel has already had access to target media. Specific changes to operating system functions and system calls to try and prevent such access are outside the scope of this document.

7.4 Examining Physical Media Information

Back to our acquisition: with the subject disk connected, it's time for use to collect information about the drive, its capabilities, and specific identification - information we'll need to effectively acquire evidence from source media. One of the first things we'll need to do is re-inventory our system's connected devices to ensure that we identify the correct subject disk. Normally you would have taken notes on the physical markings of the hard drive (or other media) as you removed it from the subject computer, etc. Some suggest an enlarged photocopy of the disk label as part of the acquisition notes, providing a reliable record of disk identification.

Figure 4: Hardware details on a drive label



For this exercise I will be using a USB to SATA bridge. Because there is some translation going on here, I want to make sure I can identify the bridge as well as the disk attached to it. So once the bridge is attached and powered on, I can run **lsusb** to see its information (bold for emphasis). If you are using a directly attached SATA drive, you will not need to run this command (this is just to see the USB bridge):

```
root@forensicbox:~# lsusb
```

```
...
```

```
Bus 006 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 005: ID 0bc2:ab24 Seagate RSS LLC Backup Plus Portable Drive
Bus 003 Device 003: ID 174c:5106 ASMedia Technology Inc.ASM1051 SATA 3Gb/s bridge
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

...

If you run the **lsusb** command before and after you attach the USB bridge, you will be able to easily identify the device for your notes. So knowing we are dealing with a Western Digital 80GB hard disk (WD800HLFS from the label) in a USB/SATA bridge (ASMedia Technology), we can identify its device node better with **lsscsi**:

```
root@forensicbox:~# lsscsi
[0:0:0:0]    disk      ATA          INTEL SSDSC2CT12 300i  /dev/sda
[2:0:0:0]    disk      ATA          Hitachi HDS72302 A5C0  /dev/sdb
[3:0:0:0]    cd/dvd   HL-DT-ST    DVDROM GH24NS90  IN01  /dev/sr0
[8:0:0:0]    disk      Seagate     BUP Slim BK      0304  /dev/sdc
[9:0:0:0]    disk      ASMT       2105              0      /dev/sdd
```

Now we can query the disk attached to the host using **hdparm**. In this case, the USB bridge supports SATA translation, so commands “pass through” the bridge to the drive itself. This tool can provide both detailed information as well as powerful commands to set options on a disk. Some of these options are useful for forensic examiners.

First, however, we are looking for information. For that we can use the simple **hdparm** command with the **-I** option on our subject disk, **/dev/sdb**. This gives detailed information about the disk that we can redirect to a file for our records.

```
root@forensicbox:~# hdparm -I /dev/sdd
```

```
/dev/sdd:
```

```
ATA device, with non-removable media
```

```
Model Number: WDC WD800HLFS-75G6U1
```

```
Serial Number: WD-WXD0CB928540
```

```
Firmware Revision: 04.04V03
```

```
Transport: Serial, SATA 1.0a, SATA II Extensions, SATA Rev 2.5
```

```
Standards:
```

```
Supported: 8 7 6 5
```

```
Likely used: 8
```

```
Configuration:
```

```
Logical max current
```

```
cylinders 16383 16383
```

```
heads 16 16
```

```
sectors/track 63 63
```

```
--
```

```
CHS current addressable sectors: 16514064
```

```
LBA user addressable sectors: 156250000
```

```
LBA48 user addressable sectors: 156250000
```

```
Logical/Physical Sector size: 512 bytes
```

device size with M = 1024*1024: 76293 MBytes
device size with M = 1000*1000: 80000 MBytes (80 GB)
cache/buffer size = 16384 KBytes
Nominal Media Rotation Rate: 10000

Capabilities:

LBA, IORDY(can be disabled)
Queue depth: 32
Standby timer values: spec'd by Standard, with device specific minimum
R/W multiple sector transfer: Max = 16 Current = 0
Recommended acoustic management value: 128, current value: 254
DMA: mdma0 mdma1 mdma2 udma0 udma1 udma2 udma3 udma4 udma5 *udma6
Cycle time: min=120ns recommended=120ns
PIO: pio0 pio1 pio2 pio3 pio4
Cycle time: no flow control=120ns IORDY flow control=120ns

Commands/features:

Enabled Supported:

- * SMART feature set
- Security Mode feature set
- * Power Management feature set
- * Write cache
- * Look-ahead
- * Host Protected Area feature set
- * WRITE_BUFFER command
- * READ_BUFFER command
- * NOP cmd
- * DOWNLOAD_MICROCODE
- SET_MAX security extension
- * Automatic Acoustic Management feature set
- * 48-bit Address feature set
- * Device Configuration Overlay feature set
- * Mandatory FLUSH_CACHE
- * FLUSH_CACHE_EXT
- * SMART error logging
- * SMART self-test
- * General Purpose Logging feature set
- * WRITE_{DMA|MULTIPLE}_FUA_EXT
- * 64-bit World wide name
- * WRITE_UNCORRECTABLE_EXT command
- * {READ,WRITE}_DMA_EXT_GPL commands
- * Segmented DOWNLOAD_MICROCODE
- * Gen1 signaling speed (1.5Gb/s)
- * Gen2 signaling speed (3.0Gb/s)
- * Native Command Queueing (NCQ)
- * Phy event counters
- DMA Setup Auto-Activate optimization
- * Software settings preservation
- * SMART Command Transport (SCT) feature set

- * SCT Read/Write Long (AC1), obsolete
- * SCT Write Same (AC2)
- * SCT Error Recovery Control (AC3)
- * SCT Features Control (AC4)
- * SCT Data Tables (AC5)
- unknown 206[12] (vendor specific)
- unknown 206[13] (vendor specific)

Security:

Master password revision code = 65534
supported

not enabled

not locked

not frozen

not expired: security count

supported: enhanced erase

14min for SECURITY ERASE UNIT. 14min for ENHANCED SECURITY ERASE UNIT.

Logical Unit WWN Device Identifier: 50014ee001ffdba8

NAA : 5

IEEE OUI : 0014ee

Unique ID : 001ffdba8

Checksum: correct

There's a lot of information laid out for us by **hdparm**. By comparing the first few lines (bold for emphasis) to the photograph of the disk label shown previously, we've again confirmed we are collecting information from the correct disk. This command can be redirected to a file and saved to our case folder:

```
root@forensicbox:~# cd /mnt/evidence/case1
```

```
root@forensicbox:case1:~# pwd
/mnt/evidence/case1
```

```
root@forensicbox:case1:~# hdparm -I /dev/sdd > case1.disk1.hdparm.txt
```

```
root@forensicbox:case1:~# ls
case1.disk1.hdparm.txt
```

In the second command above, we've redirected (>) the output of **hdparm -I /dev/sdd** to a file in the **case1** directory⁹. The file is called **case1.disk1.hdparm.txt**. The last command lists the contents of the **case1** directory. If we had multiple disks, then we could have output for disk2, disk3, etc. The file naming here is arbitrary. This is just an example.

You can keep a running log of things that you do by using a double redirect (>>) symbol

⁹You'll notice the command prompt in my example is only showing **case1** (the base directory) rather than the entire path. I've adjusted the prompt to keep the command line examples smaller. Your prompt might show the entire path

to add all the case info to a single log. I would suggest not taking this approach as you learn, though. If you mistakenly use a single redirect (>) you risk clobbering an entire log file (recall that we can use our previously discussed **chattr +a** command to prevent this, setting the file to append only).

We can also use the **hdparm** tool to help identify disk configuration overlays or host protected areas (DCO or HPA, respectively). Manufacturers use these to change the number of sectors available to the user, sometimes to make differing drives match in size for marketing (DCO), and sometimes for hiding things like “restore” or “recovery” partitions (HPA). The history and specifics of these areas are well documented on the Internet. If you have not heard of them, do some research. As forensic examiners, we are always interested in acquiring the entire disk (or at least those areas we can nominally access through kernel tools). There are even deeper areas on disks that we will not address here if imaging only the kernel accessible areas are insufficient.

hdparm can tell us if there is a DCO (and the changes actually implemented by the DCO). These can be manipulated using **hdparm** as well, but I will leave those advanced topics to your own research (hint: read **man hdparm**).

In this case we see we have no HPA:

```
root@forensicbox:~# hdparm -N /dev/sdd
/dev/sdd:
max sectors    = 156250000/156250000, HPA is disabled
```

Output from **hdparm** would be different if an HPA is present (shown here on a Seagate disk):

```
root@forensicbox:~# hdparm -N /dev/sdf
/dev/sdd:
max sectors    = 41943040/78125000, HPA is enabled
```

And the output from **hdparm -I** run against **/dev/sdi** would show only **41943040** (partial output for brevity):

```
root@forensicbox:~# hdparm -I /dev/sdf

/dev/sdf:

ATA device, with non-removable media
Model Number:      ST340014AS
Serial Number:     5MQ0QS22
Firmware Revision: 8.12
...
LBA      user addressable sectors: 41943040
```

```
LBA48  user addressable sectors:    41943040
Logical/Physical Sector size:      512 bytes
...
```

Read the **hdparm** man page carefully and be aware of the options and conditions under which a DCO or HPA can be detected and removed. For example, restoring the full number of sectors on `/dev/sdf` would look like this.

```
root@forensicbox:~# hdparm N78125000 /dev/sdf
/dev/sdf:
  setting max visible sectors to 78125000 (temporary)
  max sectors   = 78125000/78125000, HPA is disabled
```

Should you come across a disk with an HPA or DCO, I would suggest, as the safest course of action, acquiring an image as the disk sits. Once an image of the disk is obtained, you can pass commands to remove protected areas and re-image.

7.5 Hashing Media

One important step in any evidence collection is verifying the integrity of your data both before and after the acquisition is complete. You can get a hash (MD5, or SHA) of the physical device in a number of different ways.

There are a number of hash algorithms and tools that implement them, including :

- **md5sum** - 128 bit checksum
- **sha1sum** - 160 bit checksum
- **sha224sum** - 224 bit checksum
- **sha256sum** - 256 bit checksum
- **sha384sum** - 384 bit checksum
- **sha512sum** - 512 bit checksum

In this example, we will use the SHA1 hash. SHA1 is a hash signature generator that supplies a 160 bit “fingerprint” of a file or disk (which is represented by a file-like device node). It is not feasible for someone to computationally recreate a file based on the SHA1 hash. This means that matching SHA1 signatures mean identical files. There has been a lot of talk in the digital forensic community over the years of (even recent) proof of “collisions” that render certain hash algorithms “obsolete”. This guide is about learning the tools. Do your

research and check your agency or community guidelines for additional information on which algorithm to select.

We can collect a SHA1 hash of a disk by running the following command (the following commands can be replaced with **md5sum** if you prefer to use the MD5 hash algorithm, or any of the other above listed checksum tools):

```
root@forensicbox:casel:~# sha1sum /dev/sdd
ddddd4252d1adeffa267636b1ae0fbf40c9d3b3  /dev/sdd
```

or

```
root@forensicbox:casel:~# sha1sum /dev/sdd > case1.disk1.sha1.txt

root@forensicbox:casel:~# cat case1.disk1.sha1.txt
ddddd4252d1adeffa267636b1ae0fbf40c9d3b3  /dev/sdd
```

The redirection in the second command allows us to store the signature in a file and use it for verification later on. To get a hash of a raw disk (**/dev/sdc**, **/dev/sdd**, etc.) the disk does NOT have to be mounted. We are hashing the device (the disk) not the contents. As we discussed earlier, Linux treats all objects, including physical disks, as files. So whether you are hashing a file or a hard drive, the command is the same.

7.6 Collecting a Forensic Image with **dd**

Now that we have collected information on our subject media and obtained a hash of the physical disk for verification purposes, we can begin our acquisition.

dd is the very basic data copying utility that comes with a standard GNU/Linux distribution. There are, no doubt, some better imaging tools out there for use with Linux, but **dd** is the old standby. We'll be covering some of the more forensic oriented imaging tools in the following sections, but learning **dd** is important for much the same reason as learning **vi**. Like **vi**, you are bound to find **dd** on just about any Unix machine you might come across. In some cases, the best imaging tool you have available might just be the one you will almost always have access to.

This is your standard forensic image of a suspect disk. The **dd** command will copy every bit from the kernel accessible areas of the media to the destination of your choice (a physical device or file). There are a couple of concepts to keep in mind when using **dd**. Some of these concepts also apply to the other forensic imaging tools we will cover. In very basic form, the **dd** command looks like this:

```
dd if=/dev/sdd of=/path/to/evidence.raw bs=512
```

- Input file (**if=**): this is the source media. What we are imaging.
 - **if=/dev/sdd**
 - * Disk image (**/dev/sdx**): We can use the name for the entire device node.
 - * Partition image (**/dev/sdx#**): We can use the device name and the partition number to image a single partition/file system. **#** is the partition number (as returned by **fdisk -l**, for example).
- Output file (**of=**): this is the destination. Where we are placing the image/copy.
 - **of=/path/to/evidence.raw**
 - Output can be a file (as above). This is most common.
 - Output can be a physical device. This is often referred to as a “clone”.
- Block size (**bs=**): The block size of the device being imaged. The kernel usually handles this. This may become a future issue as block sizes change with the evolution of storage media. For our current subject disk (**/dev/sdd**), the **hdparm -I** output is showing 512 bytes per sector (Logical/Physical Sector Size). Be aware of some newer devices that are using 2048 bytes per sector.
- Progress indication.
 - **status=progress** This option provides a nice updating status line that shows the progress of your imaging. A relatively recent addition to the **dd** options.
- There are also options that are often used to avoid problems in case there are bad sectors on the disk.
 - **conv=noerror,sync** This option instructs **dd** to by pass copying sectors with errors AND pad those matching sectors in the destination with zeros. The padding keeps offsets correct in any file system data and perhaps still result in a usable image (more on this later). I’m not a fan of using this option.

As part of our case organization, we’ll make a new directory called **images** in our **case1** directory. This is where we will keep working copies of our images. Normally, you would create images directly to either a larger drive that has been sanitized, or to a network storage volume that is used to maintain original copies. That will depend on your specific policies.

In this case, for illustration, we will image directly to our **case1/images** directory. I prefer keeping images separate as it allows protecting the directory with attributions that prevent changes or deletions to our working copy image files, once we’ve completed the imaging process. This is personal preference, though.

To keep our **dd** command line shorter, we’ll change into our **case1/images** directory and write our output file here. Without needing to specify the directory (we are writing to the current directory), we keep the command line shorter and easier to read.

```
root@forensicbox:case1# mkdir images

root@forensicbox:case1# cd images

root@forensicbox:images# pwd
/mnt/evidence/case1/images

root@forensicbox:images# dd if=/dev/sdd of=case1.disk1.raw bs=512
79975662080 bytes (80 GB, 74 GiB) copied, 2168 s, 36.9 MB/sk^[^[
156250000+0 records in
156250000+0 records out
80000000000 bytes (80 GB, 75 GiB) copied, 2169.67 s, 36.9 MB/s
```

This takes your disk device `/dev/sdd` as the input file if and writes the output file called `case1.disk1.raw` in the current directory `/mnt/evidence/case1/images`. The **bs** option specifies the block size. This is really not needed for most block devices (hard drives, etc.) as the Linux kernel handles the actual block size. It's added here for illustration, as it can be a useful option in many situations (discussed later). Try the above command again with **status=progress** to watch for updates on how near to completion the imaging is.

Using **dd** creates an exact duplicate of the physical device file. This includes all the file slack and unallocated space. We are not simply copying the logical file structure. Unlike many forensic imaging tools, **dd** does not fill the image with any proprietary data or information. It is a simple bit stream copy from start to end. This has a number of advantages, as we will see later.

You can see from our output above that **dd** read in the same number of records (512 byte blocks, in this case) as the number of sectors for this disk previously reported by **hdparm -I**, 156250000. To verify your image, we can do the following: We want to recall the hash we obtained from the original device (`/dev/sdd`), which we stored in the file `case1.disk1.sha1`. ↪ `txt` and compare that to the hash of the image file we just obtained.

```
root@forensicbox:images# cat ../case1.disk1.sha1.txt
ddddd4252d1adeffa267636b1ae0fbf40c9d3b3  /dev/sdd

root@forensicbox:images# shasum case1.disk1.raw
ddddd4252d1adeffa267636b1ae0fbf40c9d3b3  case1.disk1.raw
```

You can see the two hashes match, verifying our image as a true copy of the original drive. Take note of the first command. Remember that we are currently in the `/mnt/evidence/case1` ↪ `/images` directory. The hash file `case1.disk1.sha1.txt` is stored in the parent directory, `/mnt/evidence/case1`. When we issue our **cat** command (stream the contents of a file), we use the `../` notation to indicate that the file we are calling is in the parent directory (`..`).

This is the simplest use case for **dd**.

7.6.1 dd and Splitting Images

It has become common practice in digital forensics to split the output of our imaging. This is done for a number of reasons, either for archiving or for use in another program. We will first discuss using `split` on its own, then in conjunction with `dd` for “on the fly” splitting.

For example, we have our 80GB image and we now want to split it into 4GB parts so they can be written to other media. Or, if you wish to store the files on a file system with file size limits and need a particular size, you might want to split the image into smaller sections.

For this we use the `split` command.

The `split` command normally works on lines of input (i.e. from a text file). But if we use the `-b` option, we force `split` to treat the file as binary input and lines are ignored. We can specify the size of the files we want along with the prefix we want for the output files. `split` can also use the `-d` option to give us numerical numbering (`*.01`, `*.02`, `*.03`, etc.) for the output files as opposed to alphabetical (`*.aa`, `*.ab`, `*.ac`, etc.). The `-a` option specifies the suffix length. The command looks like:

```
split -d -aN -bXG <file to be split> <prefix of output files>
```

where **N** is the length of the extension (or suffix) we will use and **X** is the size of the resulting files with a unit modifier (K, M, G, etc.). With our image of `/dev/sdc`, we can split it into 4GB files using the following command (The last file will be sized for the remainder of the volume if it's not an exact multiple of your chosen size.):

```
root@forensicbox:images# split -d -a3 -b4G case1.disk1.raw case1.disk1.split.
```

This would result in a group of files (4GB in size) each named with the prefix `case1.split1` ↪ as specified in the command, followed by `.000`, `.001`, `.002`, and so on. The `-a` option with `3` specifies that we want the extension to be at least 3 digits long. Without `-a 3`, our files would be named `*.01`, `*.02`, `*.03`, etc. Using 3 digits maintains consistency with other tools¹⁰. Note the trailing dot in our output file name. We do this so the suffix is added as a file extension rather than as a suffix string appended to the end of the name string.

```
root@forensicbox:images# ls -lh case1.disk1.split.*  
-rw-r--r-- 1 root root 4.0G Jul  8 07:00 case1.disk1.split.000  
-rw-r--r-- 1 root root 4.0G Jul  8 07:01 case1.disk1.split.001  
-rw-r--r-- 1 root root 4.0G Jul  8 07:02 case1.disk1.split.002  
-rw-r--r-- 1 root root 4.0G Jul  8 07:04 case1.disk1.split.003  
-rw-r--r-- 1 root root 4.0G Jul  8 07:05 case1.disk1.split.004
```

¹⁰some forensic software suites will not recognize split images that are named with other than a three character extension.

```
-rw-r--r-- 1 root root 4.0G Jul  8 07:06 case1.disk1.split.005
-rw-r--r-- 1 root root 4.0G Jul  8 07:07 case1.disk1.split.006
-rw-r--r-- 1 root root 4.0G Jul  8 07:08 case1.disk1.split.007
-rw-r--r-- 1 root root 4.0G Jul  8 07:10 case1.disk1.split.008
-rw-r--r-- 1 root root 4.0G Jul  8 07:11 case1.disk1.split.009
-rw-r--r-- 1 root root 4.0G Jul  8 07:12 case1.disk1.split.010
-rw-r--r-- 1 root root 4.0G Jul  8 07:13 case1.disk1.split.011
-rw-r--r-- 1 root root 4.0G Jul  8 07:14 case1.disk1.split.012
-rw-r--r-- 1 root root 4.0G Jul  8 07:15 case1.disk1.split.013
-rw-r--r-- 1 root root 4.0G Jul  8 07:17 case1.disk1.split.014
-rw-r--r-- 1 root root 4.0G Jul  8 07:18 case1.disk1.split.015
-rw-r--r-- 1 root root 4.0G Jul  8 07:19 case1.disk1.split.016
-rw-r--r-- 1 root root 4.0G Jul  8 07:20 case1.disk1.split.017
-rw-r--r-- 1 root root 2.6G Jul  8 07:21 case1.disk1.split.018
```

The process can be reversed. If we want to reassemble the image from the split parts, we can use the **cat** command and redirect the output to a new file. Remember **cat** simply streams the specified files to standard output. If you redirect this output, the files are assembled into one.

```
root@forensicbox:images# cat case1.disk1.split*> case1.disk1.new.raw
```

In the above command we've re-assembled the split parts into a new 80GB image file. The original split files are not removed, so the above command will essentially double your space requirements if you are writing to the same mounted device/directory.

The same **cat** command can be used to check the hash of the resulting image sections by streaming all the parts of the image through a pipe to our hash command:

```
root@forensicbox:images# cat case1.disk1.split*| shasum
ddddd4252d1adeffa267636b1ae0fbf40c9d3b3  -
```

Once again, we see that the hash remains unchanged. The **-** at the end of the output denotes that we took our input from **stdin**, not from a file or device. In the command above, **shasum** receives its input directly from the **cat** command through the pipe.

Another way of accomplishing multi-segment images would be to split the image as we create it (directly from a **dd** command). This is essentially the “on the fly” splitting we mentioned earlier. We do this by piping the output of the **dd** command straight to **split**, omitting the **of=** portion of the **dd** command. Assuming our subject drive is **/dev/sdd**, we would use the command:

```
root@forensicbox:images# dd if=/dev/sdd | split -d -a3 -b4G - case1.disk1.split.
156250000+0 records in
```

```
156250000+0 records out
80000000000 bytes (80 GB, 75 GiB) copied, 1146.87 s, 69.8 MB/s
```

Here, instead of giving the name of the file to be split in the **split** command, we give a simple - (after the 4G, where we had the input name in our previous example). The single dash is a descriptor that means “standard input”. In other words, the command is taking its input from the data pipe provided by the standard output of **dd** instead of from a file. Any options you want to pass to **dd** (block size, count, etc. go before the pipe). The output above shows the familiar number of sectors is correct for the disk we are imaging (156250000).

Once we have the image, the same technique using **cat** will allow us to reassemble it for hashing or analysis as we did with the split images above.

For practice, you can use a small USB thumb drive if you have one available and try this method on that device, splitting it into a reasonable number of parts. You can use any sample drive, being sure to replace our device node in the following command with **/dev/sdx** (where x is your thumb drive or other media). Obtain a hash first, so that you can compare the split files and the original and make sure that the splitting changes nothing.

This next example uses a 2GB USB drive that is arbitrarily split into 512M section. Follow along with the commands, and experiment with options while watching changes in the resulting output. It's the best way to learn. We'll start by identifying the thumb disk with **lsscsi** as soon as it's plugged in (output is abbreviated for readability):

```
root@forensicbox:~# lsscsi
...
[3:0:0:0]    disk    Generic  USB Flash Drive  1.00  /dev/sdb
...

root@forensicbox:~# shasum /dev/sdb
b4531adb315a48329c9b05361bf66794dd50ca27  /dev/sdb

root@forensicbox:~# dd if=/dev/sdb | split -d -a3 -b512M - thumb.split.
4156416+0 records in
4156416+0 records out
2128084992 bytes (2.1 GB, 2.0 GiB) copied, 153.338 s, 13.9 MB/s

root@forensicbox:~# ls -lh thumb.split.*| shasum
total 2.0G
-rw-r--r-- 1 root root 512M Jul  8 06:03 thumb.split.000
-rw-r--r-- 1 root root 512M Jul  8 06:03 thumb.split.001
-rw-r--r-- 1 root root 512M Jul  8 06:04 thumb.split.002
-rw-r--r-- 1 root root 494M Jul  8 06:05 thumb.split.003

root@forensicbox:~# cat thumb.split.*| shasum
b4531adb315a48329c9b05361bf66794dd50ca27  -
```

Looking at the output of the above commands, we firsts see that the thumb drive that was plugged in is identified as a 'Generic USB Flash Drive'. We then hash the device, image and split on the fly with **dd**, and check the hash. We find the same hash for the disk, for the split images "cat-ed" together, and for the newly reassembled image.

We'll have some more fun with this command later on. It is more than just an imaging tool.

7.7 Alternative Imaging Tools

Standard Linux **dd** is a fine imaging tool. It is robust, well tested, and has a proven track record.

As good as **dd** is as an imaging tool, it has one simple, perceived flaw: It was never actually designed to be used for forensic acquisitions. While the word "flaw" is a little harsh, we need to know that as much as the digital forensics community refer to **dd** as an "imaging" tool, that is not what it was designed for. It is very capable, but some practitioners prefer full featured, dedicated imaging tools that do not require external programs to accomplish logging, hashing, and imaging error documentation. Additionally, **dd** is not the best solution for obtaining evidence from damaged or failing media.

There are a number of forensic specific tools out there for Linux users that wish to acquire evidence. Some of these tools include:

- **dc3dd** – enhanced **dd** – program for forensic use (based on **dd** code).
- **dcfldd** – enhanced **dd** program for forensic use (fork of **dd** code).
- **ewfacquire** – Provided as part of the **libewf** project, this tool is used to acquire Expert Witness Format (EWF) images. We will cover it in some detail later.
- **ddrescue** – An imaging tool specifically designed to recover data from media exhibiting errors (not to be confused with **dd_rescue**).
- **aimage** – forensic imaging tool provided primarily to create images in the Advanced Forensic Format (AFF).

This is not an exhaustive list. These, however, are some of the more commonly used (as far as I know). We will cover **dc3dd**, **ewfacquire**, and **ddrescue** in this document. There are other common imaging tools that run in a GUI as well (Guymager, for example ¹¹), but I will leave the GUI programs to the reader's own research (for the most part).

¹¹<https://guymager.sourceforge.io/>

7.7.1 dc3dd

The first alternative imaging tool we will cover is **dc3dd**. This imaging tool is based on original (patched) code from **dd**. It is very similar to the popular **dcfldd** but provides a slightly different feature set. My choice of whether to cover either **dcfldd** or **dc3dd** is largely arbitrary. **dc3dd** is maintained by the DoD (Department of Defense) Cyber Crime Center (otherwise known as Dc3)¹² Regardless of which (**dc3dd** or **dcfldd**) you prefer, familiarity with one of these tools will translate very nicely to the other with some reading and experimentation, as they are very similar. While there are some significant differences, many of the features we discuss in this section are common to both **dc3dd** and **dcfldd**.

The source package and more information for **dc3dd** can be found at <https://sourceforge.net/projects/dc3dd/>.

dc3dd is installed by default on recent versions of Slackware. If you are using a different distribution, check your package manager's repository.

The **man** page for **dc3dd** is concise and easy to read. All the information you need to use the advanced features of this imaging tool are neatly laid out for you.

Let's have a look at the basic usage of **dc3dd**. As you read through the usage section of the **man** page, you'll notice a number of additions to regular **dd** for the forensic examiner. Let's concentrate on these notable additions:

hof=FILE or DEVICE	<ul style="list-style-type: none"> • similar to the of= parameter of dd • hashes the input bytes • hashes the output bytes • writes the output to the specified destination
ofs=BASE.FMT	<ul style="list-style-type: none"> • similar to the of= parameter of dd • split the output file • use the name BASE for the output files • use the extension FMT for the output files • FMT is numerical or alphabetical • more on this below
hofs=BASE.FMT	<ul style="list-style-type: none"> • hash and split the output file. This is essentially a combination of the two parameters above.

¹²**dcfldd** is also named for the Defence Computer Forensics Lab

ofsz=BYTES	<ul style="list-style-type: none"> • output file size • when using either ofs or hofs, this parameter sets the size of each split file. • see the man page for proper usage
hash=ALGORITHM	<ul style="list-style-type: none"> • specify the algorithm we will use to hash input/output bytes (md5, sha1, sha256, etc.) when we use hof or hofs.
log=FILE	<ul style="list-style-type: none"> • write an acquisition log to FILE
hlog=FILE	<ul style="list-style-type: none"> • write a hash log of the image and any split files to FILE

If we redo our imaging of `/dev/sdd` (our original 80GB disk) using **dc3dd** with simple **if=** and **of=** parameters, as we used with **dd**, the session would look something like this. We are still in our `~/case1/images` directory.

```

root@forensicbox:images# dc3dd if=/dev/sdc of=case1.disk1.dc3dd.raw
dc3dd 7.2.646 started at 2019-07-08 08:31:35 -0400
compiled options:
command line: dc3dd if=/dev/sdd of=case1.disk1.dc3dd.raw
device size: 156250000 sectors (probed),  80,000,000,000 bytes
sector size: 512 bytes (probed)
 80000000000 bytes ( 75 G ) copied ( 100% ),  855 s, 89 M/s

input results for device '/dev/sdd':
 156250000 sectors in
  0 bad sectors replaced by zeros

output results for file 'case1.disk1.dc3dd.raw':
 156250000 sectors out

dc3dd completed at 2019-07-08 08:45:50 -0400

```

Our input file is still the disk **sdd** (**if=/dev/sdd**), our output file is now **case1.disk1.dc3dd** **→ .raw** (**of=case.disk1.dc3dd.raw**). One of the first things you notice right away is that **dc3dd** returns more usable information while the program is running. It gives you a very nice

progress indicator. We also see immediately that the correct number of sectors for `/dev/sdd` were captured (`156250000`), and that there were no "bad" sectors detected. The start and stop time stamps are also added by default. If you specify a log file, this information is all captured very nicely. We will look at the hashing options and logging in more detail in just a moment.

This verbose output and the availability of simple logging is one of the things that makes **dc3dd** a better candidate for general forensic imaging when compared to **dd**.

As mentioned, **dc3dd** can incorporate the hashing, splitting and logging of an acquisition into a single command. All of this can be done with regular **dd** and external tools (with pipes, redirection or scripting) but many practitioners prefer an integrated approach. Additionally, the standard options available to the regular **dd** command still are readily available in **dc3dd** (`bs`, `skip`, etc.).

More than just incorporating the other steps into a single command, **dc3dd** extends the functionality. For example, using a regular **split** command with **dd** as we did in a previous exercise, we can either allow the default alphabetic naming convention of **split**, or pass the **-d** option to provide us with decimal extensions on our files. In contrast, **dc3dd** allows us to not only define the size of each split as an option to the imaging command (using **ofsz**) without need for a piped command, but it also allows more granular control over the format of the extensions each split will have as part of its filename. So, to our 80GB disk into 4GB sections, I would simply use:

```
ofs=BASENAME.FMT ofsz=4G
```

The **ofs** parameter is essentially "output file split". The extension following the output files names is directly formatted in the command itself. According to the **dc3dd** man page:

...

4. FMT is a pattern for a sequence of file extensions that can be numerical

```
starting at zero, numerical starting at one, or alphabetical.
Specify FMT by using a series of zeros, ones, or a's,
respectively. The number of characters used indicates the
desired length of the extensions. For example, a FMT
specifier of 0000 indicates four character numerical
extensions starting with 0000.
```

...

So if I issue the base command as something like this:

```
dc3dd if=/dev/sdd ofs=filename.FMT ofsz=32M
```

I can adjust the values for **FMT** and my split file extensions would change accordingly:

FMT=aa	<ul style="list-style-type: none"> • filename.aa • filename.ab • filename.ac
FMT=aaa	<ul style="list-style-type: none"> • filename.aaa • filename.aab • filename.aac
FMT=00	<ul style="list-style-type: none"> • filename.00 • filename.01 • filename.02
FMT=000	<ul style="list-style-type: none"> • filename.000 • filename.001 • filename.002

In addition, when using regular GNU **dd**, our hashing functions are performed external to the imaging, by either the **md5sum** or **sha1sum** commands, depending on the analyst preference for algorithm. **dc3dd** allows the user to run BOTH hashes concurrently on an acquisition and log the hashes. Before we run our split images with **dc3dd**, lets look at the hashing options a little closer.

We select our hash algorithm with the option **hash=**, specifying any of **md5**, **sha1**, **sha256**, **sha512**, or a comma separated list of algorithms. In this way you can select multiple hash methods for a single image file. These will be written to a log file we indicate, a special hash log, or to standard output if no log is specified.

dc3dd also provides **hof** and **hofs** parameters. The **hof** option acts much like **of**, but hashes the output, compares it to the input and records it. You must select a hash algorithm. Likewise, **hofs** acts much like **ofs**, splitting the output into chunk sizes specified by **ofsz**. The **hofs** option differs in that it also hashes each of the input/output streams and compares and logs them for each chunk.

You can pass the **log=filename** parameter to log all output in a single place, or you can log hashes separately using the **hlog=filename** option.

Let us redo our **dd** example with the 2G thumb drive. This time we will use **dc3dd**. We will discuss the options and output below. We are running these commands in root's home

directory (or any directory of your choosing - so we don't get the resulting image files confused with our `case1` data).

```

root@forensicbox:~# ls SCSI
...
[5:0:0:0]    disk    Generic  USB Flash Drive  1.00  /dev/sdb
...

root@forensicbox:~# sha1sum /dev/sdb
b4531adb315a48329c9b05361bf66794dd50ca27  /dev/sdb

root@forensicbox:~# dc3dd if=/dev/sdb hofs=thumb.dc3dd.000 ofsz=512M hash=sha1
hash=md5 log=thumb.dc3dd.log
dc3dd 7.2.646 started at 2019-07-08 10:03:24 -0400
compiled options:
command line: dc3dd if=/dev/sdb hofs=thumb.dc3dd.000 ofsz=512M hash=sha1 hash=md5
    ↪ log=thumb.dc3dd.log
device size: 4156416 sectors (probed),    2,128,084,992 bytes
sector size: 512 bytes (probed)
    2128084992 bytes ( 2 G ) copied ( 100% ),   183 s, 11 M/s
    2128084992 bytes ( 2 G ) hashed ( 100% ),    7 s, 282 M/s

input results for device '/dev/sdb':
4156416 sectors in
0 bad sectors replaced by zeros
6662cd15f59767e5eb1378b71dc20f68 (md5)
    4ae688f36ccef38b3cee374d8d9f79f5, sectors 0 - 1048575
    5077e2575359eecd7782b0c2215b4ab, sectors 1048576 - 2097151
    ae6dc7d9832550de29965e4c90286fac, sectors 2097152 - 3145727
    248e734b964c3cfcac8ce88017ffb1c2, sectors 3145728 - 4156415
b4531adb315a48329c9b05361bf66794dd50ca27 (sha1)
    8a60b96ef1e46272f3c9de0becd93768074918e4, sectors 0 - 1048575
    c7bc3a35ff023c47c69ac037cff60bb0e055fd0f, sectors 1048576 - 2097151
    c2ea65f9b27e11a4657950239f7fba918e350aa7, sectors 2097152 - 3145727
    6d83e285369cab5d8bda105fd8fdca29e8210f69, sectors 3145728 - 4156415

output results for files 'thumb.dc3dd.000':
4156416 sectors out
[ok] 6662cd15f59767e5eb1378b71dc20f68 (md5)
[ok] 4ae688f36ccef38b3cee374d8d9f79f5, sectors 0 - 1048575, 'thumb.dc3dd.000'
[ok] 5077e2575359eecd7782b0c2215b4ab, sectors 1048576 - 2097151, 'thumb.dc3dd.001'
[ok] ae6dc7d9832550de29965e4c90286fac, sectors 2097152 - 3145727, 'thumb.dc3dd.002'
[ok] 248e734b964c3cfcac8ce88017ffb1c2, sectors 3145728 - 4156415, 'thumb.dc3dd.003'
] b4531adb315a48329c9b05361bf66794dd50ca27 (sha1)
[ok] 8a60b96ef1e46272f3c9de0becd93768074918e4, sectors 0 - 1048575, 'thumb.dc3dd
    ↪ .000'
[ok] c7bc3a35ff023c47c69ac037cff60bb0e055fd0f, sectors 1048576 - 2097151, 'thumb.
    ↪ dc3dd.001'

```

```
[ok] c2ea65f9b27e11a4657950239f7fba918e350aa7, sectors 2097152 - 3145727, 'thumb.
    ↪ dc3dd.002'
[ok] 6d83e285369cab5d8bda105fd8fdca29e8210f69, sectors 3145728 - 4156415, 'thumb.
    ↪ dc3dd.003'
```

dc3dd completed at 2019-07-08 10:06:27 -0400

The options used above are:

if=/dev/sdb	: Our source device, as indicated by the output of lsccsi (or lsblk)
hofs=thumb.dc3dd.000	: Our output file BASE and FMT . Using the hofs option indicates that we want hashes and splits of the output file. The BASE is thumb.dc3dd. and the format of our extension (FMT) will be numerical, three digits.
ofsz=512M	: Since we indicated split files (using hofs or ofs), we need to specify an output file size.
hash=sha1, hash=md5	: Since we indicated hash the input and output files (hofs or hof), we need to provide the algorithm we want. In this case we are illustrating that we can use TWO algorithms, and both will be calculated and recorded.
log=thumb.dc3dd.log	: Indicates that we want the output of dc3dd logged to a file that we specify. Note that you can log hashes separately using hlog .

The resulting output (shown by our **ls** command below) gives us 4 split image files, with numerical extensions starting with 000. We also have a log file of our hashes and any error messages, which we can view with **less** or **cat**:

```
root@forensicbox:~# ls -lh thumb.dc3dd.*
total 2.0G
-rw-r--r-- 1 root root 512M Jul  8 10:04 thumb.dc3dd.000
-rw-r--r-- 1 root root 512M Jul  8 10:04 thumb.dc3dd.001
-rw-r--r-- 1 root root 512M Jul  8 10:05 thumb.dc3dd.002
-rw-r--r-- 1 root root 494M Jul  8 10:06 thumb.dc3dd.003
-rw-r--r-- 1 root root 2.1K Jul  8 10:06 thumb.dc3dd.log
```

As previously discussed, the log file contains our hashes and our error messages. For the hashes, the input hash from the imaged device are displayed first (for each hash algorithm we requested). Then the output hashes are displayed for each of the output files. If the input hash matches the output hash for a given range (or the whole device), the output hash is preceded with **[ok]** so you do not have to manually compare the output.

The log file ends with a time stamp for your documentation.

Another useful feature of **dc3dd** when compared to regular **dd** is the ability (without the use of external pipes or piping programs) to collect multiple images at the same time. If logistics allows, and there is a need to collect multiple copies on location to distribute to multiple parties, we can create additional output files:

```
root@forensicbox:~# dc3dd if=/dev/sdb hof=thumbcopy.dc3ddhof=duplicate.dc3dd
hash=md5

dc3dd 7.2.646 started at 2019-07-08 12:15:15 -0400
compiled options:
command line: dc3dd if=/dev/sdb hof=thumbcopy.dc3dd hof=duplicate.dc3dd hash=md5
device size: 4156416 sectors (probed),    2,128,084,992 bytes
sector size: 512 bytes (probed)
  2128084992 bytes ( 2 G ) copied ( 100% ),   179 s, 11 M/s
  2128084992 bytes ( 2 G ) hashed ( 100% ),    3 s, 614 M/s

input results for device '/dev/sdb':
  4156416 sectors in
  0 bad sectors replaced by zeros
  6662cd15f59767e5eb1378b71dc20f68 (md5)

output results for file 'thumbcopy.dc3dd':
  4156416 sectors out
  [ok] 6662cd15f59767e5eb1378b71dc20f68 (md5)

output results for file 'duplicate.dc3dd':
  4156416 sectors out
  [ok] 6662cd15f59767e5eb1378b71dc20f68 (md5)

dc3dd completed at 2019-07-08 12:18:14 -0400

root@forensicbox:~# ls -lh *.dc3dd
-rw-r--r-- 1 root root 2.0G Jul  8 12:18 duplicate.dc3dd
-rw-r--r-- 1 root root 2.0G Jul  8 12:18 thumbcopy.dc3dd
```

The demonstration above illustrates collecting two images simultaneously. You can see we selected to hash the output files (**hof**) using the **md5** algorithm (**hash=md5**). The output shows the single input stream was hashed, but there are two output streams, and each was hashed and verified separately. This can be a very useful feature of **dc3dd**.

Remember that **dc3dd** outputs raw images. They can be hashed exactly the same as **dd** output: Directly hashed with your hashing algorithm of choice (**sha1sum**, **md5sum**, etc.), or in the case of split files, using the **cat** command to stream the output of multiple files to the hash program.

Now we'll continue our look at alternative imaging tools with a utility that is used to collect and manipulate Expert Witness (E01 or EWF) files, one of the more ubiquitous formats used in computer forensics today.

7.7.2 libewf and ewfacquire

There may be times when you are asked to perform examinations collected by someone else, or perhaps your organization has elected to standardize on a given format for forensic images. In any case, chances are you will eventually come across Expert Witness format files (EWF, commonly referred to as “EnCase” format). There are many tools that can read, convert or work with these images. In this section we will learn to acquire and manipulate evidence in the EWF format.

We will explore a set of tools here belonging to the **libewf** project. These tools provide the ability to create, view, convert and work with expert witness evidence containers.

One of the benefits of covering **libewf** before other advanced forensic utilities is because it needs to be installed first in order to supply the required libraries for other packages to support EWF image formats. The **libewf** tools and detailed project information can be found at <https://github.com/libyal/libewf/>

We will start by installing **libewf** using **sbotools**. Check your distribution documentation, or the install instructions at the website shown above if you are using a distribution other than Slackware. The installation is simple. **libewf** has no additional requirements (you can view the info file with **sbofind -tei libewf**). When you start the installation process be sure to take the time to read the **README**file that displays.

```
root@forensicbox:~# sboinstall libewf
```

```
libewf (libYAL Expert Witness Compression library)
```

```
libewf allows you to read media information of EWF
files in the SMART (EWF-S01) format and the EnCase (EWF-E01) format.
libewf allows reading files created by EnCase 1 to 6, linen and FTK
Imager.
```

```
Proceed with libewf? [y]
```

```
Are you sure you wish to continue? [y]
```

```
...
```

```
Executing install script for libewf-20140806-x86_64-1_SBo.tgz.
Package libewf-20140806-x86_64-1_SBo.tgz installed.
```

```
Cleaning for libewf-20140806...
```

Once the download, compile, build, and installation of the resulting package is complete, the actual tools are placed in `/usr/bin`. We will have a closer look at the following:

- **ewfacquire**
- **ewfverify**
- **ewfinfo**
- **ewfexport**
- **ewfacquirestream** (in a later section)

We'll start with the **ewfacquire** command used to create EWF files that can be used in other programs. The easiest way to describe how **ewfacquire** works is to watch it run. There are a number of options available. To get a list of options (there are many, just run **ewfacquire -h**. To obtain an image, simply issue the command with the name of the file or physical device you wish to image. Unless you memorize or script the options, this is the easiest way to run the program. You are prompted for required information, to be stored with the data in the EWF format (the below output is interactive):

```
root@forensicbox:~# ls SCSI
[5:0:0:0]    disk    Generic  USB Flash Drive  1.00  /dev/sdb
```

```
root@forensicbox:~# ewfacquire /dev/sdb
ewfacquire 20140806
```

Device information:

```
Bus type:          USB
Vendor:
Model:
Serial:
```

Storage media information:

```
Type:              Device
Media type:        Fixed
Media size:        2.1 GB (2128084992 bytes)
Bytes per sector:  512
```

Acquiry parameters required, please provide the necessary input

Image path and filename without extension: **case1.disk2**

Case number: **2019-0001**

Description: **Thumb drive seized from bad guy**

Evidence number: **2019-001-002**

Examiner name: **Barry J. Grundy**

Notes:

Media type (fixed, removable, optical, memory) [fixed]:
Media characteristics (logical, physical) [physical]:
Use EWF file format (ewf, smart, ftk, encase1, encase2, encase3, encase4, encase5,
 ↪ encase6, linen5, linen6, ewfx) [encase6]:
Compression method (deflate) [deflate]:
Compression level (none, empty-block, fast, best) [none]:
Start to acquire at offset (0 <= value <= 2128084992) [0]:
The number of bytes to acquire (0 <= value <= 2128084992) [2128084992]:
Evidence segment file size in bytes (1.0 MiB <= value <= 7.9 EiB) [1.4 GiB]: 512M
The number of bytes per sector (1 <= value <= 4294967295) [512]:
The number of sectors to read at once (16, 32, 64, 128, 256, 512, 1024, 2048, 4096,
 ↪ 8192, 16384, 32768) [64]:
The number of sectors to be used as error granularity (1 <= value <= 64) [64]:
The number of retries when a read error occurs (0 <= value <= 255) [2]:
Wipe sectors on read error (mimic EnCase like behavior) (yes, no) [no]:

The following acquiry parameters were provided:

Image path and filename: case1.disk2.E01
Case number: 2019-0001
Description: Thumb drive seized from bad guy
Evidence number: 2019-001-002
Examiner name: Barry J. Grundy
Notes:
Media type: fixed disk
Is physical: yes
EWF file format: EnCase 6 (.E01)
Compression method: deflate
Compression level: none
Acquiry start offset: 0
Number of bytes to acquire: 1.9 GiB (2128084992 bytes)
Evidence segment file size: 512 MiB (536870912 bytes)
Bytes per sector: 512
Block size: 64 sectors
Error granularity: 64 sectors
Retries on read error: 2
Zero sectors on read error: no

Continue acquiry with these values (yes, no) [yes]:

...

Status: at 44%.

 acquired 907 MiB (952074240 bytes) of total 1.9 GiB (2128084992 bytes).
 completion in 1 minute(s) and 26 second(s) with 13 MiB/s
 (13818733 bytes/second).

...

```
Status: at 99%.
        acquired 1.9 GiB (2121498624 bytes) of total 1.9 GiB (2128084992 bytes).
        completion in 1 second(s) with 13 MiB/s (13909052 bytes/second).
```

```
Acquiry completed at: Jul 08, 2019 14:46:35
```

```
Written: 1.9 GiB (2128085180 bytes) in 2 minute(s) and 32 second(s) with 13 MiB/s
        ↪ (14000560 bytes/second).
```

```
MD5 hash calculated over data:      6662cd15f59767e5eb1378b71dc20f68
ewfacquire: SUCCESS
```

```
root@forensicbox:~# ls -lh
total 2.0G
-rw-r--r-- 1 root root 512M Jul  8 14:44 case1.disk2.E01
-rw-r--r-- 1 root root 512M Jul  8 14:45 case1.disk2.E02
-rw-r--r-- 1 root root 512M Jul  8 14:45 case1.disk2.E03
-rw-r--r-- 1 root root 495M Jul  8 14:46 case1.disk2.E04
```

In the above command session, user input is shown in bold. In places where there is no input provided by the user, the defaults (shown in brackets) are used. Notice that **ewfacquire** gives you several options for image formats that can be specified. The file(s) specified by the user is given an **E**** extension and placed in the path directed by the user. Finally, an MD5 hash is provided at the end of the output for verification. As with **dc3dd**, you also get a time stamp for documentation.

You can also issue a single command and specify those options we used above on the command line. For example, to get similar results, we can issue the following command:

```
root@forensicbox:~# ewfacquire -C "2019-001" -d sha1 -D "Thumb drive seized from
        bad guy" -e "Barry J. Grundy" -E "2019-001-002" -m removable -M physical -S 512M
        -t case1.disk2 -u /dev/sdb
```

```
ewfacquire 20140806
```

```
...
```

```
Status: at 98%.
        acquired 1.9 GiB (2091614208 bytes) of total 1.9 GiB (2128084992 bytes).
        completion in 3 second(s) with 12 MiB/s (13217919 bytes/second).
```

```
Acquiry completed at: Jul 08, 2019 15:13:07
```

```
Written: 1.9 GiB (2128085336 bytes) in 2 minute(s) and 40 second(s) with 12 MiB/s
        ↪ (13300533 bytes/second).
```

```
MD5 hash calculated over data:      6662cd15f59767e5eb1378b71dc20f68
SHA1 hash calculated over data:     b4531adb315a48329c9b05361bf66794dd50ca27
ewfacquire: SUCCESS
```

You can look at the individual options provided in the command above

by viewing **man ewfacquire**. Essentially this command allows us to run **ewfacquire** without having to answer any prompts. The important options to note here are **-d** that allows us to specify an additional checksum algorithm and **-u** (unattended mode) that forces **ewfacquire** to use the defaults for options not specified. Make sure you know what you are doing before running the command unattended.

Once acquired, the resulting files from **ewfacquire** are compatible with any software that will read EWF format images. We'll be using some forensic utilities later to do just that.

Let's look now at **ewfinfo** and **ewfverify**. These two tools, also included with **libewf**, provide information on any properly formatted EWF files you may come across.

ewfinfo simply reads the image metadata that was entered during the imaging process. It will work with image files acquired using other EWF software as well, as long as it is in a proper EWF format. For the files we just collected, using **ewfacquire**, the output would look like this (Note the Operating system used and the Software version used):

```
root@forensicbox:~# ewfinfo case1.disk2.E01
ewfinfo 20140806
```

Acquiry information

```
Case number:      2019-001
Description:      Thumb drive seized from bad guy
Examiner name:    Barry J. Grundy
Evidence number:  2019-001-002
Acquisition date: Mon Jul  8 15:10:27 2019
System date:      Mon Jul  8 15:10:27 2019
Operating system used: Linux
Software version used: 20140806
Password:         N/A
```

EWF information

```
File format:      EnCase 6
Sectors per chunk: 64
Error granularity: 64
Compression method: deflate
Compression level: no compression
```

Media information

```
Media type:       removable disk
Is physical:      yes
Bytes per sector: 512
Number of sectors: 4156416
Media size:       1.9 GiB (2128084992 bytes)
```

Digest hash information

```
MD5:              6662cd15f59767e5eb1378b71dc20f68
```

SHA1: b4531adb315a48329c9b05361bf66794dd50ca27

If you run **ewfinfo** on files collected using tools other than **ewfacquire** (EnCase under Windows, for example), the output might look like this. Note the Operating system used and Software version used fields. These give some hint as to how the files were created (EnCase version 7 on Windows 7).

```
root@forensicbox:~# ewfinfo EnCaseImageSample.E01
ewfinfo 20140806
```

Acquiry information

```
Description:      TestImage
Examiner name:    Susan B. Analyst
Acquisition date:  Fri Feb 17 13:59:50 2017
System date:      Fri Jan 13 16:10:42 2017
Operating system used: Windows 7
Software version used: 7.10.05
Password:         N/A
Model:            ST2500
Serial number:    03-016831-C
Device label:     WT055 12
Extents:          0
```

EWF information

```
File format:      unknown
Sectors per chunk: 64
Error granularity: 64
Compression method: deflate
Compression level: best compression
Set identifier:    ff582a89-3aba-cf46-a634-75edf9c15a97
```

Media information

```
Media type:       physical
Is physical:      yes
Bytes per sector: 512
Number of sectors: 250044416
Media size:       119 GiB (128022740992 bytes)
```

Digest hash information

```
MD5: 46c4d29a3ba96fffb8d7690949ddea1b
```

Also note that the MD5 value shown is the value of the data, NOT the image files themselves. Hashing the image files does will not allow you to verify against the hash of the original media – the E0* files contain meta data and so do not represent an exact copy of the source media. If you want to verify the hash of the data after it's been moved, you need to use a tool like **ewfverify**.

Hashing the data in EWF files requires a tool that recognizes the metadata associated with an EWF file and can parse and hash the original data. For this we use **ewfverify**.

```
root@forensicbox:~# ewfverify case1.disk2.E01
ewfverify 20140806
```

```
Verify started at: Jul 08, 2019 15:24:11
This could take a while.
```

```
Status: at 83%.
        verified 1.6 GiB (1785036800 bytes) of total 1.9 GiB (2128084992 bytes).
        completion in 0 second(s) with 507 MiB/s (532021248 bytes/second).
```

```
Verify completed at: Jul 08, 2019 15:24:15
```

```
Read: 1.9 GiB (2128084992 bytes) in 4 second(s) with 507 MiB/s (532021248 bytes/
    ↪ second).
```

```
MD5 hash stored in file:      6662cd15f59767e5eb1378b71dc20f68
MD5 hash calculated over data: 6662cd15f59767e5eb1378b71dc20f68
```

```
Additional hash values:
SHA1:   b4531adb315a48329c9b05361bf66794dd50ca27
```

```
ewfverify: SUCCESS
```

This command simply rehashed the data and compared it to the hash already stored within the file's metadata. Every time you move data between volumes, it's always good practice to check that the data is still intact. **ewfverify** allows you to accomplish this integrity check quickly and efficiently with EWF files.

Now for one last command in the **libewf** suite of tools. Let's talk about those situations where you've been provided a set of image files (or file) that were obtained using a popular Windows forensic tool. There will be times where you would like read the meta-data included with the images, verify the contents of the images, or export or convert the images to a bit stream (or raw) format. Once again, the **libewf** tools come in handy for this. They operate at the Linux command line, don't require any other special software, license, or dongle and are very fast. We will use a copy of an NTFS practical exercise image we will see more of later in our upcoming advanced exercises. The EWF files we'll be working on can be downloaded using **wget**, as we have done previously. Once downloaded, check the hash and compare:

```
root@forensicbox:~# wget http://www.linuxleo.com/Files/NTFS_Pract_2017_E01.tar.gz
...
root@forensicbox:~# shasum NTFS_Pract_2017_E01.tar.gz
246c144896c5288369992acc721c95968d2fe9ef  NTFS\_Pract\_2017\_E01.tar.gz
```

As we've seen previously with our software downloads, this file has a **tar.gz** extension. That means it is a compressed TAR archive. To review, the **tar** part of the extension indicates that the file was created using the **tar** command (see **man tar** for more info). The **gz** extension indicates that the file was compressed (commonly with **gzip**). When you first download a tar archive, particularly from un-trusted sources, you should always have a look at the contents of the archive before decompressing, extracting and haphazardly writing the contents to your drive. View the contents of the archive with the following command:

```
root@forensicbox:~# tar tzvf NTFS_Pract_2017_E01.tar.gz
NTFS_Pract_2017/
NTFS_Pract_2017/NTFS_Pract_2017.E04
NTFS_Pract_2017/NTFS_Pract_2017.E02
NTFS_Pract_2017/NTFS_Pract_2017.E01
NTFS_Pract_2017/NTFS_Pract_2017.E03
```

The above **tar** command will list (**t**) and decompress (**z**) the file (**f**) **NTFS_Pract_2017_E01.tar.gz**. This allows you to see where the file will be extracted, and as the output shows, there are five files that will be extracted to a new directory, **NTFS_Pract_2017/**, in the current directory. We will use the **tar** command extensively throughout this document for downloaded files.

Now we actually untar the images with the **tar x** option and change into the resulting directory:

```
root@forensicbox:~# tar xzvf NTFS_Pract_2017_E01.tar.gz
NTFS_Pract_2017/
NTFS_Pract_2017/NTFS_Pract_2017.E04
NTFS_Pract_2017/NTFS_Pract_2017.E02
NTFS_Pract_2017/NTFS_Pract_2017.E01
NTFS_Pract_2017/NTFS_Pract_2017.E03
```

```
root@forensicbox:~# cd NTFS_Pract_2017/
```

```
root@forensicbox:NTFS_Pract_2017/ #
```

The first thing we can do is run the **ewfinfo** command on the image the first file of the image set. This will return the meta-data that includes acquisition and media information, as we've seen previously. We learn the version of the software that the images were created with, along with the collection platform, date of acquisition, name of the examiner that created the image with the description and notes. Have a look at the output of **ewfinfo** on our file set (you only need provide the first file in the set as an argument to the command):

```
root@forensicbox:NTFS_Pract_2017/# ewfinfo NTFS_Pract_2017.E01
ewfinfo 20140806
```

Acquiry information

Case number: 11-1111-2017
Description: Practical Exercise Image
Examiner name: Barry J. Grundy
Evidence number: 11-1111-2017-001
Notes: This image is for artifact recovery.
Acquisition date: Mon May 1 18:19:14 2017
System date: Mon May 1 18:19:14 2017
Operating system used: Linux
Software version used: 20140608
Password: N/A

EWF information

File format: EnCase 6
Sectors per chunk: 64
Error granularity: 64
Compression method: deflate
Compression level: no compression
Set identifier: f9f1b88f-9ac9-e04f-bfe5-195039426d7c

Media information

Media type: fixed disk
Is physical: yes
Bytes per sector: 512
Number of sectors: 1024000
Media size: 500 MiB (524288000 bytes)

Digest hash information

MD5: eb4393cfcc4fca856e0edbf772b2aa7d

Notice that the last line in the output provides us with an MD5 hash of the data in the file set. Again, don't confuse this with the hash of the file itself. A file in EWF format stores the original data from the media that was imaged along with a series of CRC checks and meta-data. The hash of the E01 file(s) itself will NOT match the hash of the original media imaged. The hash of the original media and therefore the data collected is recorded in the metadata of the EWF file for later verification.

You can see from our output below that the NTFS_Pract_2017.E0* file set verifies without error. The hash obtained during the verification matches that stored within the file:

```
root@forensicbox:NTFS_Pract_2017/# ewfverify NTFS_Pract_2017.E01
ewfverify 20140806
```

```
Verify started at: Jul 08, 2019 16:51:06
This could take a while.
```

Verify completed at: Jul 08, 2019 16:51:07

Read: 500 MiB (524288000 bytes) in 1 second(s) with 500 MiB/s (524288000 bytes/
 ↪ second).

MD5 hash stored in file: eb4393cfcc4fca856e0edbf772b2aa7d

MD5 hash calculated over data: eb4393cfcc4fca856e0edbf772b2aa7d

ewfverify: SUCCESS

Now we'll look at **ewfexport**. This tool allows you to take an EWF file set and convert it to a bit stream image file, essentially removing the meta-data and leaving us with the data in raw format, as with **dd**. It is interesting to note that **ewfexport** can actually write to standard output, making it suitable for piping to other commands. Here, we issue the command with several options that result in the EWF file being exported to a raw image.

```
root@forensicbox:NTFS_Pract_2017/# ewfexport -t NTFS_Pract_21017 -f raw -u
NTFS_Pract_2017.E01
ewfexport 20140806
```

Export started at: Jul 08, 2019 16:52:49

This could take a while.

Export completed at: Jul 08, 2019 16:52:51

Written: 500 MiB (524288000 bytes) in 2 second(s) with 250 MiB/s (262144000 bytes/
 ↪ second).

MD5 hash calculated over data: eb4393cfcc4fca856e0edbf772b2aa7d

ewfexport: SUCCESS

We use the **-t** option (“target”) to write to a file. The **-f** option with **raw** indicates that the file format we are writing to is raw, as with **dd** output. We use **-u** to accept the remaining defaults and prevent an interactive session. This results in a single raw file that has the same hash as the original media (see the output of the **md5sum** command). We also see an XML formatted **.info** file that contains the hash value.¹³

```
root@forensicbox:NTFS_Pract_2017/# ls -lh NTFS_Pract_2017.raw*
-rw-r--r-- 1 barry users 500M Jul  8 16:59 NTFS_Pract_2017.raw
-rw-r--r-- 1 barry users  158 Jul  8 16:59 NTFS_Pract_2017.raw.info
```

```
root@forensicbox:NTFS_Pract_2017/# md5sum NTFS_Pract_2017.raw
eb4393cfcc4fca856e0edbf772b2aa7d NTFS_Pract_2017.raw
```

¹³The output of this command might differ greatly depending on the version of **libewf** you install. Some repositories might use versions that do not append the **.raw** extension or provide an **.info** file.

At this point, we've covered **dd**, **dc3dd**, **ewfacquire** and common methods for checking the integrity of and exporting the collected images.

All of the tools we've covered so far are great for ideal situations, where our media behaves as we expect. In addition, they all have options or built in mechanisms that would allow our acquisition to read past (or more accurately "around") any non-fatal disk errors while syncing the output so that the resulting image might still be usable. While many practitioners suggest these options as a default for running **dd** related commands, I tend to urge against it. Some of the reasons for this will become more apparent in the following section.

7.7.3 Media Errors - **ddrescue**

Now that we have a basic understanding of media acquisition and the collection of evidence images, what do we do if we run into an error? Suppose you are creating a disk image with **dd** and the command exits halfway through the process with a read error?

We can instruct **dd** to attempt to read past the errors using the **conv=noerror** option. In basic terms, this is telling the **dd** command to ignore the errors that it finds, and attempt to read past them. When we specify the **noerror** option it is a good idea to include the **sync** option along with it. This will "pad" the **dd** output wherever errors are found and ensure that the output will be "synchronized" with the original disk. This may allow file system access and file recovery where errors are not fatal. Assuming that our subject drive is **/dev/sdc**, the command will look something like:

```
root@forensicbox:~# dd if=/dev/sdc of=image.raw conv=noerror, sync
```

I would like to caution forensic examiners against using the **conv=noerror, sync** option, however. While **dd** is capable of reading past errors in many cases, it is not designed to actually recover any data from those areas. There are a number of tools out there that are designed specifically for this purpose. If you need to use **conv=noerror, sync**, then you are using the wrong tool. That is not to say it will not work as advertised (with some caveats), only that there are better options, or at least important considerations.

Which brings us to **ddrescue**.

Testing has shown that standard **dd** based tools are simply inadequate for acquiring disks that have actual errors. This is NOT to say that **dd**, **dc3dd** or **dcfldd** are useless...far from it. They are just not optimal for error recovery. You may be forced to use **dd** or **dc3dd** because of limits to external tool access or considerations of time. We teach **dd** in this guide because there are instances where it may be the only tool available to you. In those cases, understanding the use of command line options to optimize the recovery of the disk regardless of errors is important for evidence preservation. However, if there are options, then perhaps a different tool would make sense.

This section is not meant to provide an education on disk errors, media failure, or types of failure. Nor is it meant to imply that any tool is better or worse than any other. I will simply describe the basic functionality and leave it to the reader to pursue the details.

First, let's start with some of the issues that arise with the use of common **dd** based tools. For the most part, these tools take a "linear" approach to imaging, meaning that they start at the beginning of the input file and read block by block until the end of the file is reached. When an error is encountered, the tool will either fail with an "input/output" error, or if a parameter such as **conv=noerror** is passed, will ignore the errors and attempt to read through (or skip) them, continuing to read block by block until it comes across readable data again. Here is a simple **dd** command on a disk with errors. The disk is 41943040 sectors:

```
root@forensicbox:~# blockdev -getsz /dev/sdd
41943040

root@forensicbox:~# dd if=/dev/sdd of=diskimage.raw
dd: error reading '/dev/sdd': Input/output error
12840+0 records in
12840+0 records out
6574080 bytes (6.6 MB, 6.3 MiB) copied, 8.48712 s, 775 kB/s
```

The **dd** command above was only able to read 12840 sectors (which is 6574080 bytes, as the **dd** output shows). The same command, this time using **conv=noerror,sync** will ignore the error, pad the error sectors with null bytes, and continue on:

```
root@forensicbox:~# dd if=/dev/sdd of=diskimage.raw conv=noerror,sync
dd: error reading '/dev/sdd': Input/output error
12840+0 records in
12840+0 records out
6574080 bytes (6.6 MB, 6.3 MiB) copied, 7.83969 s, 839 kB/s
dd: error reading '/dev/sdd': Input/output error
12840+1 records in
12841+0 records out
6574592 bytes (6.6 MB, 6.3 MiB) copied, 11.6881 s, 563 kB/s
dd: error reading '/dev/sdd': Input/output error
12840+2 records in
12842+0 records out
6575104 bytes (6.6 MB, 6.3 MiB) copied, 15.5426 s, 423 kB/s
dd: error reading '/dev/sdd': Input/output error
12840+3 records in
12843+0 records out
6575616 bytes (6.6 MB, 6.3 MiB) copied, 19.4103 s, 339 kB/s
dd: error reading '/dev/sdd': Input/output error
12840+4 records in
12844+0 records out
6576128 bytes (6.6 MB, 6.3 MiB) copied, 23.2758 s, 283 kB/s
```

```
dd: error reading '/dev/sdd': Input/output error
12840+5 records in
12845+0 records out
6576640 bytes (6.6 MB, 6.3 MiB) copied, 27.1286 s, 242 kB/s
dd: error reading '/dev/sdd': Input/output error
12840+6 records in
12846+0 records out
6577152 bytes (6.6 MB, 6.3 MiB) copied, 30.9714 s, 212 kB/s
dd: error reading '/dev/sdd': Input/output error
12840+7 records in
12847+0 records out
6577664 bytes (6.6 MB, 6.3 MiB) copied, 34.8038 s, 189 kB/s
41943032+8 records in
41943040+0 records out
21474836480 bytes (21 GB, 20 GiB) copied, 1112.08 s, 19.3 MB/s
```

What you end up with at the end of this command is an image of the entire disk, but with the error sectors filled in (sync'd) with zeros. This done to maintain correct offsets within file systems, etc.

Obviously, simple failure (“giving up” when errors are encountered) is not good. Any data in readable areas beyond the errors will be missed. The problem with ignoring errors and attempting to read through them (using options like **conv=noerror**) is that we are further stressing a disk that is already possibly on the verge of complete failure. The fact of the matter is that you may get few chances at reading a disk that has recorded “bad sectors”. If there is an actual physical defect, the simple act of reading the bad areas may make matters worse, leading to disk failure before other viable areas of the disk are collected. All of this applies, of course, to disks with “physical” storage. Solid state storage is another matter entirely.

So, when we pass **conv=noerror** to an imaging command, we are actually asking our imaging tools to “grind through” the bad areas. Why not initially skip over the bad sections altogether, since in many cases recovery may be unlikely? Instead we should concentrate on recovering data from areas of the disk that are good. Once the “good” data is acquired, we can go back and attempt to collect data from the error areas, preferably with a recovery algorithm designed with purpose.

In a nutshell, that is the philosophy behind **ddrescue**. Used properly, **ddrescue** will read the “healthy” portions of a disk first, and then fall back to recovery mode – trying to read data from bad sectors. It does this through the use of some very robust logging, referred to as a ‘map file’. This allows **ddrescue** to resume any imaging job at any point, given a map file to work from. This is an important (perhaps the most important) point about using **ddrescue** - that is, with a map file you never need to re-read already successfully recovered sectors. When **ddrescue** references the map file on successive runs, it fills in the gaps, it does not “redo” work already finished.

ddrescue is installed by default in Slackware Linux (for full installations), but check with your distribution of choice to determine availability.

The documentation for **ddrescue** is excellent. The detailed manual is in an **info** page. The command **info ddrescue** will give you a great start to understanding how this program works, including examples and the ideas behind the algorithm used. I'll run through the process here, but I strongly advise that you read the info page for **ddrescue** before attempting to use it on a case.

The first consideration when using any recovery software, is that the disk must be accessible by the Linux kernel. If the drive does not show up in the **/dev** structure, then there's no way to get tools like **ddrescue** to work.

Next, we have to have a plan to recover as much data as we can from a bad drive. The prevailing philosophy of **ddrescue** is that we should attempt to get all the good data first. This differs from normal **dd** based tools, which simply attempt to get all the data at one time in a linear fashion. **ddrescue** uses the concept of "splitting the errors". In other words, when an area of bad sectors is encountered, the errors are split until the "good" areas are properly imaged and the unreadable areas marked as bad. Finally, **ddrescue** attempts to retry the bad areas by re-reading them until we either get data or fail after a certain number of specified attempts.

There are a number of ingenious options to **ddrescue** that allow the user to try and obtain the most important part of the disk first, then move on until as much of the disk is obtained as possible. Areas that are imaged successfully need not be read more than once. As mentioned previously, this is made possible by a robust map file. The map file is written periodically during the imaging process, so that even in the event of any interruption, the session can be restarted, keeping duplicate imaging efforts, and therefore disk access, to a minimum.

Given that we are addressing forensic acquisition here, we will concentrate all our efforts on obtaining the entire disk, even if it means multiple runs. The following examples will be used to illustrate how the most important options to **ddrescue** work for the forensic examiner. We will concentrate on detailing the map file used by **ddrescue** so that the user can see what is going on with the tool, and how it operates.

Let's look at a simple example of using **ddrescue** on a small drive without errors, to start. The simplest way to run **ddrescue** is by providing the input file, output file and a name for our map file. Note that there is no **if=** or **of=**. In order to get a good look at how the map file works, we'll interrupt our imaging process halfway through, check the map file to illustrate how an interruption is handled, and then resume the imaging.

```
root@forensicbox:~# ddrescue /dev/sdb ddres_image.raw ddres_map.txt
GNU ddrescue 1.24
Press Ctrl-C to interrupt
  ipos:    1091 MB, non-trimmed:      0 B,   current rate:   4718 kB/s
  opos:    1091 MB, non-scraped:     0 B,   average rate:  13473 kB/s
```

```

non-tried:    1036 MB,  bad-sector:    0 B,    error rate:    0 B/s
  rescued:    1091 MB,  bad areas:    0,      run time:    1m 20s
pct rescued:  51.28%, read errors:    0,    remaining time:  1m 19s
                                time since last successful read:    n/a
Copying non-tried blocks... Pass 1 (forwards)^C
Interrupted by user

```

Here we used `/dev/sdb` as our input file, wrote the image to `ddres_image.raw`, and wrote the map file to `ddres_map.txt`. Note the output shows the progress of the imaging by default, giving us a running count of the amount of data copied or "rescued", along with a count of the number of errors encountered (in this case zero), and the imaging speed. The process was interrupted right at about 50% completion, with the `<ctrl-c>` key combo.

Now let's have a look at our map file:

```

root@forensicbox:~# cat ddres_map.txt
# Mapfile. Created by GNU ddrescue version 1.24
# Command line: ddrescue /dev/sdb ddres_image.raw ddres_map.txt
# Start time:   2019-07-09 08:32:18
# Current time: 2019-07-09 08:33:38
# Copying non-tried blocks... Pass 1 (forwards)
# current_pos  current_status  current_pass
0x410D0000    ?                1
#      pos      size  status
0x00000000  0x410D0000  +
0x410D0000  0x3DCB0000  ?

```

The map file shows us the current status of the acquisition ¹⁴. Lines starting with a `#` are comments. There are two sections of note. The first non comment line shows the current status of the imaging while the second section (two lines, in this case) shows the status of various blocks of data. The values are in hexadecimal, and are used by **ddrescue** to keep track of those areas of the target device that have marked errors, those areas that have already been successfully read and written, and those that remain to be read. The status symbols we will discuss here (taken from the **info** page) are as follows:

<u>Character</u>	<u>Status</u>
?	non-tried
*	bad area - non trimmed
/	bad area - non scraped
-	bad hardware block(s)
+	finished

In this case we are concerned only with the `?` and the `+`. Essentially, when the copying

¹⁴The **ddrescue info** page has a very detailed explanation of the map file structure.

process is interrupted, the log is used to tell **ddrescue** where the copying left off, and what has already been copied (or otherwise marked). The first section (status) alone may be sufficient in this case, since **ddrescue** need only pickup where it left off, but in the case of a disk with errors, the block section is required so **ddrescue** can keep track of what areas still need to be retried as good data is sought among the bad.

Translated, our log would tell us the following (narrowed down to the position and status):

```
#current position  current status
0x410D0000        ?
```

The status shows that the current imaging process is copying data at byte offset **1091371008** (0x410D0000). In our first pass, this indicates the “non-tried” blocks.

```
#      pos      size  status
0x00000000  0x410D0000  +
0x410D0000  0x3DCB0000  ?
```

- The data blocks from byte offset 0 (0x00000000) of size **1091371008** bytes (0x410D0000) are finished.
- The data blocks from offset **1091371008** (0x410D0000) of size 1036713984 bytes (0x3DCB0000) are still not tried.

The size of our partial image file matches the size of the block of data marked “finished” with the + symbol in our log file (size bold for emphasis):

```
root@forensicbox:~# ls -l ddres_image.raw
-rw-r--r-- 1 root root 1091371008 Jul  9 08:33 ddres_image.raw
```

We can continue and complete the copy operation now by simply invoking the same command. By specifying the same input and output files, and by providing the map file, we tell **ddrescue** to continue where it left off:

```
root@forensicbox:~# ddrescue /dev/sdb ddresi_image.raw ddres_map.txt
GNU ddrescue 1.24
Press Ctrl-C to interrupt
Initial status (read from mapfile)
rescued: 1091 MB, tried: 0 B, bad-sector: 0 B, bad areas: 0
```

Current status

```
ipos:      2128 MB, non-trimmed:      0 B,  current rate:  2555 kB/s
opos:      2128 MB, non-scraped:      0 B,  average rate: 13640 kB/s
```

```
non-tried:      0 B,  bad-sector:      0 B,   error rate:      0 B/s
  rescued:    2128 MB,  bad areas:      0,   run time:      1m 15s
pct rescued: 100.00%, read errors:      0,   remaining time:    n/a
                                   time since last successful read:    n/a
```

Finished

```
root@forensicbox:~# cat ddres_map.txt
# Mapfile. Created by GNU ddrescue version 1.24
# Command line: ddrescue /dev/sdb ddres_image.raw ddres_map.txt
# Start time: 2019-07-09 12:41:25
# Current time: 2019-07-09 12:42:40
# Finished
# current_pos  current_status  current_pass
0x7ED70000    +                1
#      pos      size  status
0x00000000  0x7ED80000  +

root@forensicbox:~# echo "ibase=16;7ED80000" | bc
2128084992

root@forensicbox:~# ls -l ddres_image.raw
-rw-r--r-- 1 root root 2128084992 Jul  9 12:42 ddres_image.raw
```

The above session shows the output of the completed **ddrescue** command followed by the contents of the map file. The **ddrescue** command shows the initial status line indicating where we left off, and then current status through image completion. The **echo** command converts our hexadecimal value to decimal, just so we can illustrate that the total rescued is equal in size to the size of the image.

The real power of the map file lies in the fact that we can start and stop the imaging process as needed and potentially attack the recovery from different directions (using the **-R** option to read the disk in reverse) until you've scraped together as much of the original data as you can. For example, if you had two identical disks, with mirrored data, and both had bad or failing sectors, you could probably reconstruct a complete image by imaging both with **ddrescue** and using the same map file (and output file). Once recovered and recorded as such in the map file, sectors are not accessed again. This limits the stress to the disk.

Using a disk with known errors we'll invoke **ddrescue** with some additional options. In this case, I may have started imaging a subject disk using a common tool like **dd** or **dc3dd**, and found that the copy failed with errors. Knowing this, I'll switch to using **ddrescue**. The options in the below command are **-i0** to indicate starting at offset 0. Offset 0 is the default, but I'm being explicit here. There are situations where you might want to start at a different offset and then go back...the map file allows for this easily. The **-d** option means that we are going to directly access the disk, bypassing the kernel cache. Next, the **-N** option is provided to prevent **ddrescue** from "trimming" the bad areas that are found. This option allows **ddrescue** to start the recovery process by collecting good data first, disturbing the

error areas as little as possible.

```

root@forensicbox:~# ddrescue -i0 -d -N /dev/sdd bad_disk.raw bad_log.txt
GNU ddrescue 1.24
Press Ctrl-C to interrupt
      ipos:    6619 kB, non-trimmed:    65536 B,  current rate:    851 kB/s
      opos:    6619 kB, non-scraped:      0 B,  average rate:   52634 kB/s
non-tried:      0 B,  bad-sector:      0 B,  error rate:      0 B/s
      rescued: 21474 MB,  bad areas:      0,  run time:      6m 47s
pct rescued:   99.99%, read errors:      1,  remaining time:    n/a
                                time since last successful read:  n/a
Finished
                                [1

root@forensicbox:~# cat bad_log.txt

# Mapfile. Created by GNU ddrescue version 1.24
# Command line: ddrescue -i0 -d -N /dev/sdd bad_disk.raw bad_log.txt
# Start time:   2019-07-09 12:58:21
# Current time: 2019-07-09 13:05:08
# Finished
# current_pos  current_status  current_pass
0x00660000    +                2
#      pos      size  status
0x00000000  0x00640000  +
0x00640000  0x00010000  *
0x00650000  0x4FF9B0000  +

root@forensicbox:~# echo "ibase=16;00010000" | bc
65536

```

The output above shows a couple of things (highlights for emphasis). We have the completed initial run with the **-N** option, and the output shows that we have **65536** bytes “non-trimmed”, indicating an area of errors. The map file shows the position of un-copied area of the disk (offset **0x00640000**) and a size of **0x00010000** (65536 bytes). The status of this area is indicated with an asterisk. Note that the 65536 bytes is exactly 128 sectors, and this is the default “cluster” size used by **ddrescue**. This does not mean that there are 128 sectors that cannot be read. It simply means that the entire cluster could not be read, and the **-N** option prevented “trimming”, or paring the sectors down to smaller readable chunks. The cluster size can be controlled with the **--cluster-size=X** option, where **X** is the number of sectors in a cluster. We now have a partial image.

Now we can continue the imaging with the same input and output file, and the same map file, but this time we remove the **-N** option, allowing error areas to be trimmed, and we add the **-r** option to specify the number of retries when a bad sector is encountered, which is three in this case.


```
root@forensicbox:~# ddrescue -r3 -d /dev/sdd bad_disk.raw bad_log.txt
GNU ddrescue 1.24
Press Ctrl-C to interrupt
Initial status (read from mapfile)
rescued: 21474 MB, tried: 65536 B, bad-sector: 0 B, bad areas: 0

Current status
      ipos:      6576 kB, non-trimmed:      0 B,  current rate:      0 B/s
      opos:      6576 kB, non-scraped:      0 B,  average rate:     643 B/s
non-tried:      0 B,  bad-sector:     3072 B,  error rate:     128 B/s
      rescued:    21474 MB,  bad areas:      1,      run time:      1m 37s
pct rescued:    99.99%, read errors:      24, remaining time:      n/a
                        time since last successful read:      1m 25s

root@forensicbox:~# cat bad_log.txt
# Mapfile. Created by GNU ddrescue version 1.24
# Command line: ddrescue -r3 -d /dev/sdd bad_disk.raw bad_log.txt
# Start time: 2019-07-09 14:09:06
# Current time: 2019-07-09 14:10:43
# Finished
# current_pos  current_status  current_pass
0x00645A00      +              3
#      pos      size  status
0x00000000  0x00645000  +
0x00645000  0x00000C00  -
0x00645C00  0x4FF9BA400  +

root@forensicbox:~# echo "ibase=16;00000C00" | bc
3072

root@forensicbox:~# echo "3072/512" | bc
6
```

The output shows that our “non-trimmed” areas are now 0, and the error size is 3072 bytes. Looking at the map file, we see that there is a section of the disk that is marked with the “-”, indicating bad hardware blocks, which in this case are unrecoverable. The size in the map file (0x00000C00) matches the bad-sector in the output (3072). This means we have 6 bad sectors (512 bytes each).

While we were not able to obtain the entire disk in this example, hopefully you recognize the benefits of the approach we take using **ddrescue** to get the good data first while recovering as much as we can before accessing and potentially causing additional damage to bad areas of the disk.

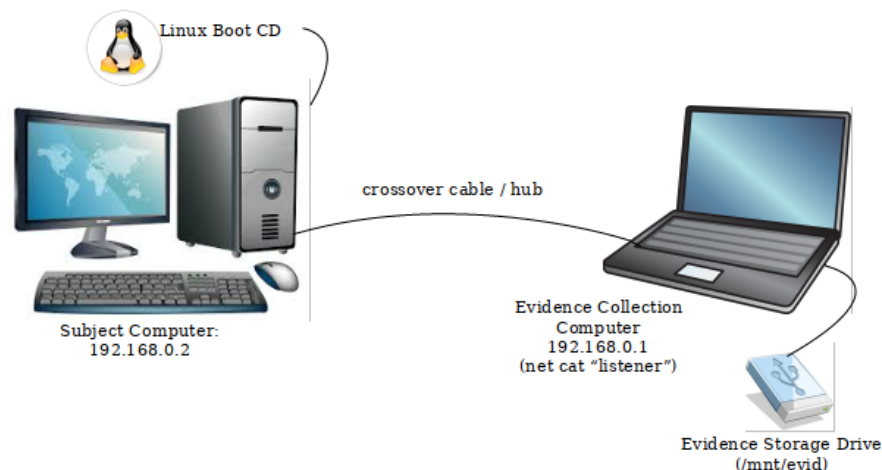
7.8 Imaging Over the Wire

There may occasions where you want or need to acquire an image of a computer using a boot disk and network connectivity. Most often, this approach is used with a Linux boot disk on the subject machine (the machine you are going to image). Another computer, the imaging collection platform, is connected either via a network hub or switch; or through a crossover cable. These sorts of acquisitions can even take place across the country or anywhere around the world. The reasons and applications of this approach range from level of physical access to the hardware and interface issues to local resources. As an example, you might come across a machine that has a drive interface that is incompatible with your equipment. If there are no external ports (USB for example), then you might need to resort to the network interface to transfer data. So the drive is left in place, and your collection platform is attached through a hub, switch, or via crossover cable. Obviously the most secure path between the subject and collection platform is most desirable. Where possible, I would use either a crossover cable and a small hub. Consider the security and integrity of your data if you attempt to transfer evidence across an enterprise or an external network. We will concentrate on the mechanics here, and the very basic commands required. As always, I urge you to follow along.

First, let's clarify some terminology for the purpose of our discussion here. In this instance, the computer we want to image will be referred to as the subject computer. The computer to which we are writing the image will be referred to as the collection workstation.

In order to accomplish imaging across the network, we will need to setup our collection workstation to "listen" for data from our subject computer. We do this using *netcat*, the **nc** command. The basic setup looks like this image: Once you have the subject computer

Figure 5: Example network acquisition diagram



booted with a Linux Boot CD (preferably one that is set up with forensics in mind). You'll

need to ensure the two computers are configured on the same network, and can communicate.

Checking and configuring network interfaces is accomplished with the **ifconfig** command (interface configure). If you run **ifconfig -a**, you will get a list of interfaces and their current (if any) settings. On my collection workstation, to shorten the output, I'll run the command on the network interface (**eth0**) directly:

```
root@forensicbox:~# ifconfig eth0
eth0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    ether a4:4c:c8:14:b7:cb txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
    device interrupt 16 memory 0xed400000-ed420000
```

Right now, the output is showing no IPv4 address and the **eth0** interface is down. I can give it a simple address with the **ifconfig** command again, this time specifying some simple settings:

```
root@forensicbox:~# ifconfig eth0 192.168.0.1 netmask 255.255.255.0

root@forensicbox:~# ifconfig eth0
eth0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 192.168.0.1 netmask 255.255.255.0 broadcast 192.168.0.255
    ether a4:4c:c8:14:b7:cb txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
    device interrupt 16 memory 0xed400000-ed420000
```

Now the output above shows the interface is “up”, and the address is now **192.168.0.1**, and the netmask and broadcast address are also set. For now that's all for our collection workstation as far as simple configuration goes.

On our subject workstation, we'll need to boot it with a suitable boot disk (or bootable USB thumb drive assuming you have access to a USB port and the ability to boot from it). I carry several with me, and just about any of them will work as long as they have a robust tool set. Once you boot the subject system, repeat the steps above to setup a simple network interface, making sure that the two computers are physically connected via crossover cable, hub, or some other means. Note the prompt change here to illustrate we are working on the SUBJECT computer now (hostname is **bootdisk**), and not our collection system:

```
root@bootdisk:~# ifconfig eth0 192.168.0.2 netmask 255.255.255.0
```

```
root@bootdisk:~# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.0.2 netmask 255.255.255.0 broadcast 192.168.0.255
    ether 08:00:27:99:d6:30 txqueuelen 1000 (Ethernet)
    RX packets 73 bytes 11716 (11.4 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 16 bytes 1392 (1.3 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Now we have the assigned IP addresses of our systems on a local network:

Subject Computer: 192.168.0.2
Collection Workstation: 192.168.0.1

We can then see if we can communicate with our evidence collection workstation (192.168.0.1 ↪) using the **ping** command (which we interrupt with **ctrl-c**)¹⁵:

```
root@bootdisk:~# ping 192.168.0.1
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=39.5 ms
64 bytes from 192.168.0.1: icmp_seq=2 ttl=64 time=3.05 ms
64 bytes from 192.168.0.1: icmp_seq=3 ttl=64 time=282 ms
64 bytes from 192.168.0.1: icmp_seq=4 ttl=64 time=203 ms
64 bytes from 192.168.0.1: icmp_seq=5 ttl=64 time=25.8 ms
64 bytes from 192.168.0.1: icmp_seq=6 ttl=64 time=350 ms
^C
--- 192.168.0.1 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 6009ms
rtt min/avg/max/mdev = 3.049/167.451/349.541/131.698 ms
```

Now that we have both computers talking, we can begin our imaging. Check the hash of the subject disk:

```
root@bootdisk:~# sha1sum/dev/sda
0dec26535e6264544488c08a65cefd a22ced0f66
```

7.8.1 Over the wire - dd

The next step is to open a "listening" port on the collection computer. We will do this on our evidence collection system with **nc** (our **netcat** utility), making sure we have a mounted

¹⁵if your **ping** command fails, be sure to check and see if ICMP is disabled via **rc.firewall** as described in our workstation configuration section.

file system to store the image on. In this case we are using an external USB drive mounted on `/mnt/evidence` (on the collection workstation) to store our image:

```
root@forensicbox:~# nc -l -p 2525 | dd of=/mnt/evidence/net_dd.raw
```

...the command returns nothing until the collection process starts on the subject computer

This command opens a netcat (**nc**) listening session (**-l**) on TCP port 2525 (**-p 2525**) and pipes any traffic that comes across that port to the **dd** command (with only the **of=** flag), which writes the file `/mnt/evidence/net_dd.raw`.

Now on the subject computer (note the command prompt with the hostname `bootdisk`), we issue the **dd** command. Instead of giving the command an output file parameter using **of=**, we pipe the **dd** command output to netcat (**nc**) and send it to our listening port (2525) on the collection computer at IP address `192.166.0.1`.

```
root@bootdisk~# dd if=/dev/sda | nc 192.168.0.1 2525
```

...again the command returns nothing

This command pipes the output of **dd** straight to **nc**, directing the image over the network to TCP port 2525 on the host `192.168.0.1` (our collection box's IP address). If you want to use **dd** options like **conv=noerror**, **sync** or **bs=x**, then you do that on the **dd** side of the pipe.

Once the imaging is complete¹⁶, we will see that the commands at both ends appear to “hang”. After we receive our completion output from **dd** on the subject computer (records in / records out), we can kill the **nc** listening on our collection box with a simple **<ctrl-c>**.

```
root@bootdisk:~#
41943040+0 records in
41943040+0 records out
21474836480 bytes (21 GB, 20 GiB) copied, 3081.68 s, 7.0 MB/s
```

When you see this on the subject computer, you would use **<ctrl-c>**.

```
root@forensicbox:~# nc -l -p 2525 | dd of=/mnt/evidence/net_dd.raw
^C <- We've used <ctrl-c> to finish the process
34410182+9694042 records in
41943040+0 records out
21474836480 bytes (21 GB, 20 GiB) copied, 24497 s, 877 kB/s
```

¹⁶You can add the status option to **dd** on the subject side if you want to watch the progress.

This should return our prompts on both sides of the connections. You should then check the resulting image on the collection workstation to see if they match.

```
root@forensicbox:~# sha1sum /mnt/evidence/net_dd.raw
0dec26535e6264544488c08a65cefd2ced0f66  /mnt/evidence/net_dd.raw
```

Our hashes match and our network acquisition was successful.

7.8.2 Over the Wire - **dc3dd**

As we discussed previously, there are a number of tools we can use for imaging that provide a more forensic oriented approach. **dc3dd** is as good a choice for over the wire imaging as it is on local disks. You also have some flexibility with **dc3dd** in that even if your boot disk does not come with it installed, you are still able to use all its features on the evidence collection computer.

dc3dd does all its magic on the output side of the acquisition process (unless you are acquiring from file sets or some other non-standard source). This means we can use plain **dd** on our subject computer (using the boot disk) to acquire the disk and stream the contents across our netcat pipe, and still allow **dc3dd** on our collection machine to handle hashing, splitting and logging. Most of **dc3dd**'s options and parameters work on the output stream. So, while our listening process on the collection system will use **dc3dd** commands, the subject system can use the same **dd** commands we used before.

On the collection system, let's set up a listening process that uses **dc3dd** to split the incoming data stream into 2GB chunks and logs the output to **nc.dc3dd.raw**. As soon as we initiate our command, **dc3dd** will start and sit waiting for input from the listening port (2525):

```
root@forensicbox:~# nc-l -p 2525 | dc3dd ofs=/mnt/evidence/net_dc3dd.000 ofsz=4G
log=/mnt/evidence/net_dc3dd.log
dc3dd 7.2.646 started at 2019-07-13 17:06:22 -0400
compiled options:
command line: dc3dd ofs=/mnt/evidence/net_dc3dd.000 ofsz=4G log=/mnt/evidence/
↪ net_dc3dd.log
sector size: 512 bytes (assumed)
1324389536 bytes ( 1.2 G ) copied (??%), 207 s, 6.1 M/s
...imaging continues
```

The **dc3dd** output will start immediately, but stay at 0 bytes until it receives input through the pipe. As soon as you start the imaging process on the subject machine, you'll see the **dc3dd** command on the listening machine start to process the incoming data. Again notice we are using plain **dd** on the subject box to simply stream bytes over the pipe. **dc3dd** takes over on the collection machine to implement our **dc3dd** options and logging.

```
root@bootdisk:~# dd if=/dev/sda | nc 192.168.0.1 2525
```

When the transfer is complete, we can look at the resulting files and the **dc3dd** log on our collection machine.

```
root@forensicbox:~# ls -lh /mnt/evidence/net_dc3dd.*
-rw-r--r-- 1 root root 4.0G Jul 13 17:16 /mnt/evidence/net_dc3dd.000
-rw-r--r-- 1 root root 4.0G Jul 13 17:28 /mnt/evidence/net_dc3dd.001
-rw-r--r-- 1 root root 4.0G Jul 13 17:36 /mnt/evidence/net_dc3dd.002
-rw-r--r-- 1 root root 4.0G Jul 13 17:45 /mnt/evidence/net_dc3dd.003
-rw-r--r-- 1 root root 4.0G Jul 13 17:55 /mnt/evidence/net_dc3dd.004
-rw-r--r-- 1 root root 438 Jul 13 17:57 /mnt/evidence/net_dc3dd.log

root@forensicbox:~# cat /mnt/evidence/net_dc3dd.log
dc3dd 7.2.646 started at 2019-07-13 17:06:22 -0400
compiled options:
command line: dc3dd ofs=/mnt/evidence/net_dc3dd.000 ofsz=4G log=/mnt/evidence/
↳ net_dc3dd.log
sector size: 512 bytes (assumed)
21474836480 bytes ( 20 G ) copied (??%), 3045.31 s, 6.7 M/s

input results for file 'stdin':
41943040 sectors in

output results for files '/mnt/evidence/net_dc3dd.000':
41943040 sectors out

dc3dd aborted at 2019-07-13 17:57:07 -0400
```

We can see that the **dc3dd** log ended with an “aborted” message because we had to manually stop the listening process (with **ctrl-c**) since in this case **dc3dd** is not handling the input itself, but just accepting the stream through **netcat** – you need to manually tell it when the stream is complete. Again, when completed, you should check the resulting hashes against our original hash of **/dev/sda** on the subject machine.

```
root@forensicbox:~# cat /mnt/evidence/net_dc3dd.00*| shasum
0dec26535e6264544488c08a65cefd2ced0f66 -
```

7.8.3 Over the Wire - **ewfacquirestream**

Last, but not least, we will cover a tool that will allow us to take a stream of input (with the same **netcat** pipe) and create an EWF file from it. **ewfacquirestream** acts much like

ewfacquire (and is part of the same **libewf** package we installed previously), but allows for data to be gathered via standard input. The most obvious use for this is taking data passed by our netcat pipe.

In previous examples, once the data reached the destination collection computer, the listening netcat process piped the output to the **dd** or **dc3dd** command output string, and the file was written exactly as it came across, as a bitstream image.

But by using **ewfacquirestream**, we can create EWF files instead of a bitstream image. We simply pipe the output stream from netcat to **ewfacquirestream**. If we do not wish to have the program use default values, then we issue the command with options that define how we want the image made (sectors, hash algorithms, error handling, etc.) and what information we want stored. The command on the subject machine remains the same. The command on the collection system would look something like this (utilizing many of the command defaults):

```
root@forensicbox:~# nc -l -p 2524 | ewfacquirestream -c 2019-001 -D"Subject Disk"
-e "BGrundy" -E '1' -f encase6 -m fixed -M physical -N "Imaged via network
connection" -t /mnt/evidence/net_ewfstream
```

```
ewfacquirestream 20140806
```

Unsupported compression values defaulting to method: deflate with level: none.

Using the following acquiry parameters:

```
Image path and filename:      /mnt/evidence/net_ewfstream.E01
Case number:                  case_number
Description:                  Subject Disk
Evidence number:              1
Examiner name:                BGrundy
Notes:                        Imaged via network connection
Media type:                   fixed disk
Is physical:                  yes
EWF file format:              EnCase 6 (.E01)
Compression method:           deflate
Compression level:            none
Acquiry start offset:         0
Number of bytes to acquire:    0 (until end of input)
Evidence segment file size:    1.4 GiB (1572864000 bytes)
Bytes per sector:              512
Block size:                   64 sectors
Error granularity:            64 sectors
Retries on read error:        2
Zero sectors on read error:    no
```

Acquiry started at: Jul 13, 2019 21:44:46

This could take a while.

...output pauses until the acquisition is started on the subject computer.

This command takes the output from **netcat** (**nc**) and pipes it to **ewfacquirestream**.

- the case number is specified with **-C**
- the evidence description is given with **-D**
- the examiner given with **-e**
- evidence number with **-E**
- encase6 format is specified with **-f encase6**
- the media type is given with **-m**
- the media flags are given with **-M**
- notes are provided with **-N**
- the target path and file name is specified with **-t /path/file**.

No extension is given, and **ewfacquirestream** automatically appends an **E0*** extension to the resulting file. To get a complete list of options, look at the **man** pages, or run the command with the **-h** option.

Back on the subject system we use our standard “send the data across the wire from a subject computer” command...

```
root@bootdisk:~# dd if=/dev/sda | nc 192.168.0.1 2525
```

Once the acquisition completes (you'll need to stop the **ewfacquirestream** process on the collection system when the **dd** command completes on the subject system - again you will know when you see the **dd** input/output message on the subject side), you can look at the resulting files, and compare the hashes. Since we used **shasum** previously, we'll re-run with **md5sum** so we can compare the hash against the **ewfverify** output (which uses MD5):

```
root@bootdisk:~# md5sum /dev/sda
e663f5fd97a73a8b37a942a58dde5496 /dev/sda
```

```
root@forensicbox:~# ls -lh /mnt/evidence/net_ewfstream.*
-rw-r--r-- 1 root root 1.5G Jul 14 09:52 /mnt/evidence/net_ewfstream.E01
-rw-r--r-- 1 root root 1.5G Jul 14 09:52 /mnt/evidence/net_ewfstream.E02
-rw-r--r-- 1 root root 1.5G Jul 14 09:52 /mnt/evidence/net_ewfstream.E03
-rw-r--r-- 1 root root 1.5G Jul 14 09:52 /mnt/evidence/net_ewfstream.E04
-rw-r--r-- 1 root root 1.5G Jul 14 09:52 /mnt/evidence/net_ewfstream.E05
-rw-r--r-- 1 root root 1.5G Jul 14 09:52 /mnt/evidence/net_ewfstream.E06
-rw-r--r-- 1 root root 1.5G Jul 14 09:52 /mnt/evidence/net_ewfstream.E07
```

```
-rw-r--r-- 1 root root 1.5G Jul 14 09:52 /mnt/evidence/net_ewfstream.E08
-rw-r--r-- 1 root root 1.5G Jul 14 09:52 /mnt/evidence/net_ewfstream.E09
-rw-r--r-- 1 root root 1.5G Jul 14 09:52 /mnt/evidence/net_ewfstream.E10
-rw-r--r-- 1 root root 1.5G Jul 14 09:52 /mnt/evidence/net_ewfstream.E11
-rw-r--r-- 1 root root 1.5G Jul 14 09:52 /mnt/evidence/net_ewfstream.E12
-rw-r--r-- 1 root root 1.5G Jul 14 09:52 /mnt/evidence/net_ewfstream.E13
-rw-r--r-- 1 root root 988M Jul 14 09:52 /mnt/evidence/net_ewfstream.E14
```

```
root@forensicbox:~# ewfverify net_ewfstream.E01
ewfverify 20140806
```

Verify started at: Jul 14, 2019 09:56:03
This could take a while.

Status: at 6%.
verified 1.2 GiB (1370980352 bytes) of total 20 GiB (21474836480 bytes).
completion in 1 minute(s) and 2 second(s) with 310 MiB/s (325376310 bytes/
↪ second).

...

Status: at 99%.
verified 19 GiB (21454585856 bytes) of total 20 GiB (21474836480 bytes).
completion in 0 second(s) with 341 MiB/s (357913941 bytes/second).

Verify completed at: Jul 14, 2019 09:57:03

Read: 20 GiB (21474836480 bytes) in 1 minute(s) and 0 second(s) with 341 MiB/s
↪ (357913941 bytes/second).

MD5 hash stored in file: e663f5fd97a73a8b37a942a58dde5496
MD5 hash calculated over data: e663f5fd97a73a8b37a942a58dde5496

ewfverify: SUCCESS

...compare the **md5sum** output from `/dev/sda` on the subject computer and the end of the **ewfverify** output and we see our verification is successful.

7.9 Compression - Local and Over the Wire

Here we've covered the very basics and mechanics of imaging media over a network pipe. **netcat** is not your only solution to this, though it is one of the simpler options and is usually available on most boot disks and Linux systems.

In reality, you might want to consider whether you want your data encrypted as it traverses the network. In our example above, we may have been connected via a crossover cable (interface to interface) or through a standalone network hub. But what if you are in a

situation where the only means of collection is remote? Or over enterprise network hardware? In that case you would perhaps want encryption. For that you could use **cryptcat**, or even **ssh**. Now that you understand the basic mechanics of this technique, you are urged to explore other tools and methods. There are projects out there like **rdd** (<https://sourceforge.net/projects/rdd/>) and **air** (<https://sourceforge.net/projects/air-imager/>) you might want to explore and test on your own.

7.9.1 Compression on the Fly with dd

Another useful technique while imaging is compression. Considering our concern for forensic application here, we will be sure to manage our compression technique so that we can verify our hashes without having to decompress and write our images out before checking them.

For this exercise, we'll use the GNU **gzip** utility. **gzip** is a command line utility that allows us some fairly granular control over the compression process. There are other compression utilities (**lzip**, **xz**, etc.), but we'll concentrate on **gzip** for the same reasons we learned **dd** and **vi**...almost always available, and fine starting place to learn the foundations of command line concepts. Most source packages for software is minimally available in a **gz** compressed format, but I urge you to explore other compression options on your own.

First, for the sake of familiarity, let's look at the simple use of **gzip** on a single file and explore some of the options at our disposal. I have created a directory called **testcomp** and I've copied the image file **NTFS_Pract_2017.raw** (from our ewfexport exercise in the section on **libewf**) into that directory to practice on. Copying the image file into an empty directory gives me an uncluttered place to experiment. You can use any file you want to practice this particular section on. First, let's double check the hash of the image:

```
root@forensicbox:~# mkdir testcomp

root@forensicbox:~# cp NTFS_Pract_2017.raw testcomp/.

root@forensicbox:~# cd testcomp

root@forensicbox:testcomp# ls -lh
total 501M
-rw-r--r-- 1 root root 500M Jul 14 10:54 NTFS_Pract_2017.raw

root@forensicbox:testcomp# shasum NTFS_Pract_2017.raw
094123df4792b18a1f0f64f1e2fc609028695f85 NTFS_Pract_2017.raw
```

Now, in its most simple form, we can call **gzip** and simply provide the name of the file we want compressed. This will replace the original file with a compressed file that has a **.gz** suffix appended.

```
root@forensicbox:testcomp# gzip NTFS_Pract_2017.raw

root@forensicbox:testcomp# ls -lh
total 61M
-rw-r--r-- 1 root root 61M Jul 14 10:54 NTFS_Pract_2017.raw.gz
```

So now we see that we have replaced our original 500M file with a 61M file that has a .gz extension. To decompress the resulting .gz file:

```
root@forensicbox:testcomp# gzip -d NTFS_Pract_2017.raw.gz
root@forensicbox:testcomp# ls -lh
total 501M
-rw-r--r-- 1 root root 500M Jul 14 10:54 NTFS_Pract_2017.raw

root@forensicbox:testcomp# shasum NTFS_Pract_2017.raw
094123df4792b18a1f0f64f1e2fc609028695f85  NTFS_Pract_2017.raw
```

We've decompressed the file and replaced the .gz file with the original image. A check of the hash shows that all is in order.

Suppose we would like to compress a file but leave the original intact. We can use the **gzip** command with the **-c** option. This writes to standard output instead of a replacement file. When using this option we need to redirect the output to a filename of our choosing so that the compressed file is not simply streamed to our terminal. Here is a sample session using this technique:

```
root@forensicbox:testcomp# ls -lh
total 501M
-rw-r--r-- 1 root root 500M Jul 14 10:54 NTFS_Pract_2017.raw

root@forensicbox:testcomp# gzip -c NTFS_Pract_2017.raw > NewImage.raw.gz

root@forensicbox:testcomp# ls -lh
total 561M
-rw-r--r-- 1 root root 500M Jul 14 10:54 NTFS_Pract_2017.raw
-rw-r--r-- 1 root root 61M Jul 14 11:51 NewImage.raw.gz

root@forensicbox:testcomp# gzip -cd NewImage.raw.gz > NewUncompressed.raw

root@forensicbox:testcomp# ls -lh
total 1.1G
-rw-r--r-- 1 root root 500M Jul 14 10:54 NTFS_Pract_2017.raw
-rw-r--r-- 1 root root 61M Jul 14 11:51 NewImage.raw.gz
-rw-r--r-- 1 root root 500M Jul 14 11:52 NewUncompressed.raw
```

```
root@forensicbox:testcomp# shasum NewUncompressed.raw
094123df4792b18a1f0f64f1e2fc609028695f85  NewUncompressed.raw
```

In the above output, we see that the first directory listing shows the single image file. We then compress using **gzip -c** which writes to standard output. We redirect that output to a new file (name of our choice). The second listing shows that the original file remains, and the compressed file is created. We then use **gzip -cd** to decompress the file, redirecting the output to a new file and this time preserving the compressed file.

These are very basic options for the use of **gzip**. The reason we learn the **-c** option is to allow us to decompress a file and pipe the output to a hash algorithm. In a more practical sense, this allows us to create a compressed image and check the hash of that image without writing the file twice.

If we go back to a single image file in our directory, we can see this in action. Remove all the files we just created (using the **rm** command) and leave the single original **dd** image. Now we will create a single compressed file from that original image and then check the hash of the compressed file to ensure it's validity:

```
root@forensicbox:testcomp# rm -f NewImage.raw.gz NewUncompressed.raw
```

```
root@forensicbox:testcomp# ls -lh
total 501M
-rw-r--r-- 1 root root 500M Jul 14 10:54 NTFS_Pract_2017.raw
```

```
root@forensicbox:testcomp# gzip NTFS_Pract_2017.raw
```

```
root@forensicbox:testcomp# ls -lh
total 61M
-rw-r--r-- 1 root root 61M Jul 14 10:54 NTFS_Pract_2017.raw.gz
```

```
root@forensicbox:testcomp# gzip -cd NTFS_Pract_2017.raw.gz | shasum
094123df4792b18a1f0f64f1e2fc609028695f85  -
```

```
root@forensicbox:testcomp# ls -lh
total 61M
-rw-r--r-- 1 root root 61M Jul 14 10:54 NTFS_Pract_2017.raw.gz
```

First we see that we have the correct hash. Then we compress the image with a simple **gzip** command that replaces the original file. Now, all we want to do next is check the hash of our compressed image without having to write out a new image. We do this by using **gzip -c** (to standard out) **-d** (decompress), passing the name of our compressed file but piping the output to our hash algorithm (in this case **shasum**). The result shows the correct hash of the output stream, where the output stream is signified by the **-**.

Okay, so now that we have a basic grasp of using **gzip** to compress, decompress, and verify hashes, let's put it to work "on the fly" using **dd** to create a compressed image. We will then check the compressed image's hash value against an original hash.

Find a small thumb drive or other removable media to image. I'll be using a small 8GB USB stick. Clear out the **testcomp** directory - or create a new directory - so that we have a clean place to write our image to (or where ever you have the space to write).

Obtaining a compressed **dd** image on the fly is simply a matter of streaming our **dd** output through a pipe to the **gzip** command and redirecting that output to a file. Our resulting image's hash can then be checked using the same method we used above. Consider the following session. The physical device we have as an example in this case is **/dev/sdi** (if you are using a device to follow along, remember to use **lsscsi** or **lsblk** to find the correct device file).

```
root@forensicbox:~# lsblk
NAME        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
...
sdc          8:32    1    15G  0 disk
└─sdc1       8:33    1    15G  0 part

root@forensicbox:~# shasum /dev/sdc
b8a8a99b3f9bf1ffbf942eeb0729e34e5ae7f36f  /dev/sdc

root@forensicbox:~# dd if=/dev/sdc | gzip -c > sdc_img.raw.gz
31326208+0 records in
31326208+0 records out
16039018496 bytes (16 GB, 15 GiB) copied, 1512.94 s, 10.6 MB/s

root@forensicbox:~# ls -lh
total 6.8G
-rw-r--r-- 1 root root 6.8G Jul 14 17:40 sdc_img.raw.gz

root@forensicbox:~# gzip -cd sdc_img.raw.gz | shasum
b8a8a99b3f9bf1ffbf942eeb0729e34e5ae7f36f  -
```

In the above **dd** command there is no "output file" specified, just as when we piped the output to **netcat** in our "over the wire" section. The output is simply piped straight to **gzip** for redirection into a new file. We then follow up with our integrity check by decompressing the file to standard output and hashing the stream. The hashes match, so we can see that we used **dd** to acquire a compressed image (6.8G compressed image vs. the 16G device), and verified our acquisition without the need to decompress (and write to disk) first.

Now let's go one step further in our on the fly compression demonstration. How about putting a few of these steps altogether? Recall our imaging over the network through **netcat**. If you look at the different sizes of our compressed vs. un-compressed images, you'll

see there's quite a difference in size (which will, of course, depend on the compress-ability of the data on the volume being imaged). Do you think it might be faster to compress data before sending over the network? Let's find out.

Going back to our simple network setup, let's do the same imaging, but this time we'll pipe to **gzip -c** on one side of the network and **gzip -cd** on the other, effectively sending compressed data across the wire. The resulting image is NOT compressed. We decompress it before it reaches the imaging tool. You can elect to leave that out if you like and simply write a compressed image.

We'll start by hashing the subject hard drive again from our boot disk. Assuming the network settings are all correct, and then opening our **netcat** listener and **dc3dd** process on the collection box:

```
root@bootdisk~# sha1sum/dev/sda
0dec26535e6264544488c08a65cefd22ced0f66  /dev/sda
```

Open the listening process, redirecting the output to a file. I'm using **dc3dd** with **hof=** to collect input and output hash to compare with the SHA1 hash above.

```
root@forensicbox:~# nc -l -p 2525 | gzip -cd | dc3dd hash=sha1 hof=netCompress.raw
log=netCompress.log
dc3dd 7.2.646 started at 2019-07-15 10:07:56 -0400
compiled options:
command line: dc3dd hash=sha1 hof=netCompress.raw log=netCompress.log
sector size: 512 bytes (assumed)
21474836480 bytes ( 20 G ) copied (??%), 12716 s, 1.6 M/s
21474836480 bytes ( 20 G ) hashed ( 100% ), 36 s, 570 M/s

input results for file 'stdin':
41943040 sectors in
0dec26535e6264544488c08a65cefd22ced0f66 (sha1)

output results for file 'netCompress.raw':
41943040 sectors out
[ok] 0dec26535e6264544488c08a65cefd22ced0f66 (sha1)

dc3dd completed at 2019-07-15 13:39:52 -0400
```

And now start the imaging on the subject computer:

```
root@bootdisk:~# dd if=/dev/sda | gzip -c | nc 192.168.0.1 2525
41943040+0 records in
41943040+0 records out
21474836480 bytes (21 GB, 20 GiB) copied, 12564.8 s, 1.7 MB/s
^C
```

When the imaging is complete, we can check the resulting **dc3dd** log, **netCompress.log**, to verify the integrity of the resulting image:

```
root@forensicbox:~# ls -lh netCompress.raw
-rw-r--r-- 1 root root 20G Jul 15 13:39 netCompress.raw

root@forensicbox:~# cat netCompress.log
dc3dd 7.2.646 started at 2019-07-15 10:07:56 -0400
compiled options:
command line: dc3dd hash=sha1 hof=netCompress.raw log=netCompress.log
sector size: 512 bytes (assumed)
 21474836480 bytes ( 20 G ) copied (??%), 12716 s, 1.6 M/s
 21474836480 bytes ( 20 G ) hashed ( 100% ), 35.9389 s, 570 M/s

input results for file 'stdin':
 41943040 sectors in
 0dec26535e6264544488c08a65cefd22ced0f66 (sha1)

output results for file 'netCompress.raw':
 41943040 sectors out
 [ok] 0dec26535e6264544488c08a65cefd22ced0f66 (sha1)

dc3dd completed at 2019-07-15 13:39:52 -0400
```

A couple of things to notice here. First, our hashes match. We successfully read a device, compressed the data, piped it over a network, decompressed the data and wrote an image file. The reason we do this is to save some time. You can test this yourself, imaging over a network and timing the entire process with and without compression.

Keep in mind that the usefulness of this is dependent on where your particular bottlenecks are. On a local network, via crossover cable, and writing to a USB drive, compressing across the network may have little impact. But if you are imaging over an enterprise network, or remotely, you may see quite a performance gain from compression. Your results may vary, but be aware of the technique.

7.10 Preparing a disk for the Suspect Image - Wiping

One common practice in forensic disk analysis is to sanitize or "wipe" a disk prior to restoring or copying a forensic image to it. This ensures that any data found on the wiped disk is from the image and not from "residual" data. That is, data left behind from a previous case or image. In technical terms, residual data should never be an issue unless your operating system or forensic software is drastically broken. Though there has been some concern over whether an examiner accidentally physically searches a device rather than an image file on

the device. In legal terms it's an important step to ensure compliance with best practices that have been around for a long while.

We've already covered simple acquisitions, and media sanitization is a step that is normally performed before you conduct evidence collection. It is being introduced here because it makes more sense to cover the subject of imaging when introducing imaging tools rather than introducing drive wiping before we've covered the tools we'll use.

On to wiping. We can use a special device, `/dev/zero` as a source of zeros. This can be used to create empty files and wipe portions of disks. You can write zeros to an entire disk (or at least to those areas accessible to the kernel and user space) using the following command (assuming `/dev/sdb` is the disk you want to wipe - in this case a small 2G thumb drive):

```
root@forensicbox:~# dd if=/dev/zero of=/dev/sdb bs=4k
dd: error writing '/dev/sdb': No space left on device
519553+0 records in
519552+0 records out
2128084992 bytes (2.1 GB, 2.0 GiB) copied, 1682.24 s, 1.3 MB/s
```

This starts at the beginning of the drive (`/dev/sdb`) and writes zeros (the input file) to every sector on `/dev/sdb` (the output file) in 4 kilobyte chunks (**bs =<block size>**). Specifying larger block sizes can speed the writing process (default is 512 bytes). Experiment with different block sizes and see what effect it has on the writing speed (i.e. 32k, 64k, etc.). Be careful of missing partial blocks at the end of the output if your block size is not a proper multiple of the device size. The error **No space left on device** indicates that the device has been filled with zeros. And, of course, be very sure that the target disk is in fact the disk you intend to wipe. Check and double check.

dc3dd makes the wiping process even easier and provides options to wipe with specific patterns. In it's simplest form, **dc3dd** can wipe a disk with a simple:

```
root@forensicbox:~# dc3dd wipe=/dev/sdb
dc3dd 7.2.646 started at 2019-07-15 15:21:10 -0400
compiled options:
command line: dc3dd wipe=/dev/sdb
device size: 4156416 sectors (probed),    2,128,084,992 bytes
sector size: 512 bytes (probed)
    28084992 bytes ( 2 G ) copied ( 100% ),    1 s, 3.3 G/s
    2128084992 bytes ( 2 G ) copied ( 100% ),  318 s, 6.4 M/s

input results for pattern '00':
    4156416 sectors in

output results for device '/dev/sdb':
    4156416 sectors out
```

```
dc3dd completed at 2019-07-15 15:26:27 -0400
```

So how do we verify that our command to write zeros to a whole disk was a success? You could check random sectors with a hex editor, but that's not realistic for a large drive. One of the best methods would be to use the **xxd** command (command line hexdump) with the "autoskip" option. The output of this command on a zero'd drive would give just three lines. The first line, starting at offset zero with a row of zeros in the data area, followed by an asterisk (*) to indicate identical lines, and finally the last line, with the final offset followed by the remaining zeros in the data area. Here's an example of the command on a zero'd drive and its output.

```
root@forensicbox:~# xxd -a /dev/sdb
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ....
*
7ed7fff0: 0000 0000 0000 0000 0000 0000 0000 0000  ....
```

Using **dc3dd** with the **hwipec** option (hash the wipe), the confirmation would look like this (and is far quicker than the **dd/xxd** combination):

```
root@forensicbox:~# dc3dd hwipec=/dev/sdb hash=sha1
dc3dd hwipec=/dev/sdb hash=sha1

dc3dd 7.2.646 started at 2019-07-15 15:28:13 -0400
compiled options:
command line: dc3dd hwipec=/dev/sdb hash=sha1
device size: 4156416 sectors (probed),    2,128,084,992 bytes
sector size: 512 bytes (probed)
  2128084992 bytes ( 2 G ) copied ( 100% ),  471 s, 4.3 M/s
  2128084992 bytes ( 2 G ) hashed ( 100% ),  151 s, 13 M/s

input results for pattern '00':
  4156416 sectors in
  4c9b7786abd51a554b35193dd1805476859903f4 (sha1)

output results for device '/dev/sdb':
  4156416 sectors out
  [ok] 4c9b7786abd51a554b35193dd1805476859903f4 (sha1)

dc3dd completed at 2019-07-15 15:36:04 -0400
```

7.11 Final Words on Imaging

Anyone who has worked in the field of digital forensics for any length of time can tell you that the acquisition process is the foundation of our business. Everything else we do can be cross verified and validated after the fact. But you often only get one shot at a proper acquisition. You may have a limited amount of time on site, or one shot at recovering data from a failing drive. Make sure you understand how the tools work, and what the options actually do. Validating your approach prior to using it in live field work is essential.

This section has introduced a number of basic tools and a rough technical process. Requirements and a procedures vary from jurisdiction to jurisdiction and across organizations. Know the requirements of your particular governing body, and adhere to them.

Technological advances will change much of how we do acquisitions. Solid state media and storage technologies are more than just changes to the interface that may require an adapter. The way data is physically being stored and data blocks manipulated is constantly evolving. The entire approach to "obtaining an exact duplicate" is changing as storage methods and technologies advance. Don't get too comfortable!

7.12 Mounting Evidence

We've already discussed the **mount** command and using it to access file systems on external devices. Now that we are working with forensic images, we'll need to access those as well. There are two ways we do this: through forensic software "physically"; or through volume mounting "logically".

When we access the image with forensic software, we are accessing the entire physical image including unallocated blocks and otherwise inaccessible file system and volume management artifacts that were successfully recovered and copied by our imaging software (or hardware). We'll cover some forensic software in later sections.

For now we are going to look at some tools and techniques we can use to view the contents of an image as a logically mounted file system.

7.12.1 Structure of the Image

The first step in all this is to determine what volumes and file systems are available for logical mounting within our image. "Structure", in this case, refers to the partitioning scheme and identification of volumes and file systems within the image.

Given that our images have been of physical disks, they should all likely have some sort of partition table in them. We can detect this partition table using **fdisk** or **gdisk**. We will

cover more “forensically” oriented software for this later (**mmfs** from the Sleuth Kit), but for now, **fdisk** and **gdisk** should be available on any modern Linux system.

We will cover **fdisk** first, as it was previously discussed, earlier using the **-l** option. We can get the partition information on **/dev/sda**, for example, with:

```
root@forensicbox:~# fdisk -l /dev/sda
Disk /dev/sda: 20 GiB, 21474836480 bytes, 41943040 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 6139EC4A-2EB3-4AF1-9A09-FE44CDEF22A3

Device            Start      End  Sectors Size Type
/dev/sda1          34    131105    131072 64M Linux filesystem
/dev/sda2        131106    8519713    8388608  4G Linux swap
/dev/sda3        8519714   33685537   25165824 12G Linux filesystem
/dev/sda4       33685538   41943006    8257469  4G Microsoft basic data
```

So the output of **fdisk** shows that the partition label is of type **GPT**. What if we run the **gdisk** command on the same disk? Here's the output:

```
root@forensicbox:~# gdisk -l /dev/sda
GPT fdisk (gdisk) version 1.0.4

Partition table scan:
  MBR: protective
  BSD: not present
  APM: not present
  GPT: present

Found valid GPT with protective MBR; using GPT.
Disk /dev/sda: 41943040 sectors, 20.0 GiB
Sector size (logical/physical): 512/512 bytes
Disk identifier (GUID): 6139EC4A-2EB3-4AF1-9A09-FE44CDEF22A3
Partition table holds up to 128 entries
Main partition table begins at sector 2 and ends at sector 33
First usable sector is 34, last usable sector is 41943006
Partitions will be aligned on 2-sector boundaries
Total free space is 0 sectors (0 bytes)

Number  Start (sector)    End (sector)  Size      Code  Name
   1            34           131105    64.0 MiB   8300   Linux filesystem
   2          131106          8519713    4.0 GiB   8200   Linux swap
   3          8519714         33685537   12.0 GiB   8300   Linux filesystem
```

4	33685538	41943006	3.9 GiB	0700	Microsoft basic data
---	----------	----------	---------	------	----------------------

The important takeaway here is that the output of the two is *functionally the same*. When recording the output of these commands for a forensic examination, however, I would urge you to utilize the tool specifically designed for the system you are currently dealing with. Use **fdisk** for DOS partition schemes, and **gdisk** for GPT partitioning schemes when recording your output. Our output above shows that `/dev/sda` has four partitions. Partitions 1 and 3 are of type “Linux”, and partition two is identified as a Linux swap partition (roughly virtual memory or “swap file” for the Linux OS). The last partition is “Microsoft basic data”, normally the code for a volume containing an NTFS file system.

It is especially important to note that the file system code and name do not necessarily identify the actual file system on that volume. In our example above, the file systems could be ext2, ext3, ext4, reiserfs, etc.

Recording the output for an examination is a simple matter of redirecting the output of either command (**fdisk** or **gdisk**) to a file. Using **gdisk** as an example (assuming a GPT layout):

```
root@forensicbox:~# gdisk -l /dev/sda > /mnt/evidence/sda.gdisk.txt
```

A couple of things to note here: The name of the output file (`sda.gdisk.txt`) is completely arbitrary. There are no rules for extensions. Name the file anything you want. I would suggest you stick to a convention and make it descriptive. Also note that since we identified an explicit path for the file name, `sda.gdisk.txt` will be created in `/mnt/evidence`. Had we not given the path, the file would be created in the current directory (`/root`, as indicated by the `~`).

Once you have determined the partition layout of the disk, it's time to see if we can identify the file system and mount the volumes to review the contents.

7.12.2 Identifying File Systems

Before we jump straight to mounting a volume for analysis or review, you might want to identify the file system contained in that volume. There are a number of ways to do this. The **mount** command is actually very good at identifying file systems when mounting, so giving a **-t <fstype>** option is not always needed (and is often not used). But it is still good practice to check and record the file system prior to mounting, assuming you will be doing a manual review of the logical volume contents.

For a simple file system example, download the following file, and check the hash¹⁷:

¹⁷This image is identical to the one used in previous versions of this guide. We will continue to use it here because it is small, simple, and provides a good practice set for commands in the following sections.

```
root@forensicbox:~# wget https://www.linuxleo.com/Files/fat_fs.raw
--2019-07-16 12:26:38-- https://www.linuxleo.com/Files/fat_fs.raw
Resolving www.linuxleo.com (www.linuxleo.com)... 74.208.236.144
Connecting to www.linuxleo.com (www.linuxleo.com)|74.208.236.144|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1474560 (1.4M)
Saving to: 'fat_fs.raw'

fat_fs.raw          100%[=====>]   1.41M  3.52MB/s   in
    ↪ 0.4s

2019-07-16 12:26:38 (3.52 MB/s) - 'fat_fs.raw' saved [1474560/1474560]

root@forensicbox:~# shasum fat_fs.raw
f5ee9cf56f23e5f5773e2a4854360404a62015cf  fat_fs.raw
```

We can use the **file** command to give us an idea of what is contained in the image. Remember that the output of file is dependent on the magic files for your given Linux distribution. Running the file command on my system gives this:

```
root@forensicbox:~# file fat_fs.raw
fat_fs.raw: DOS/MBR boot sector, code offset 0x3e+2, OEM-ID "(wA~PIHC" cached by
    ↪ Windows 9M, root entries 224, sectors 2880 (volumes <=32 MB), sectors/FAT 9,
    ↪ sectors/track 18, serial number 0x16e42d6d, unlabeled, FAT (12 bit), followed
    ↪ by FAT
```

There's a lot of information provided in the file output. We get the IDs, number of sectors (this is read from meta data, not from the image size itself), serial number, and other identifiers. We'll cover images with separate partitions in a moment. This is a simple image of a file system that is not part of multi-partition media. You may see images like this where USB thumb drives or other removable media have been imaged and contain a single un-partitioned volume.

A quick word on using the **file** command directly on devices. The **file** command will provide a response on exactly the object you reference. If I run file on **/dev/sda**, for example, I get notified that it is a block special file. If you want to know more about the device rather than the device file, then use the **-s** option to **file** to specify that we want to know about the volume being referenced by the **/dev** block device. Try this on your own system:

```
root@forensicbox:~# file /dev/sda
/dev/sda: block special (8/0)

root@forensicbox:~# file -s /dev/sda
/dev/sda: DOS/MBR boot sector, extended partition table (last)
```

So we know what file system we are dealing with, now we need to mount the image as a device so we can see the contents. For that we can use a loop device.

7.12.3 The Loop Device

We can mount the file system(s) within the image using the **loop** interface. Basically, this allows you to “mount” a file system within an image file (instead of a disk) to a mount point and browse the contents. In simple terms, the **loop** device acts as a “proxy disk” to serve up the file system as if it were on actual media.

7.12.4 Loop option to the mount command

For a simple file system image (where there are not multiple partitions in the image), we can use the same **mount** command and the same options as any other file system on a device, but this time we include the option **loop** to indicate that we want to use the **loop** device to mount the file system within the image file. Change to the directory where placed the **fat_fs.raw**, and type the following (skip the **mkdir** command if you already created this directory in our earlier section on mounting external file systems):

```
root@forensicbox:~# mkdir /mnt/analysis

root@forensicbox:~# mount -t vfat -o ro,loop fat_fs.raw /mnt/analysis

root@forensicbox:~# ls /mnt/analysis
ARP.EXE*
Docs/
FTP.EXE*
Pics/
loveletter.virus*
ouchy.dat*
snoof.gz*
```

Now you can change to the **/mnt/analysis** directory and browse the image as if it were a mounted disk. Use the **mount** command by itself to double check the mounted options (we pipe it through **grep** here to isolate our mount point):

```
root@forensicbox:~# mount | grep analysis
/root/fat_fs.raw on /mnt/analysis type vfat (ro,relatime,fmask=0022,dmask=0022,
↪ codepage=437,iocharset=iso8859-1,shortname=mixed,utf8,errors=remount-ro)
```

When you are finished browsing, unmount the image file (again, note the command is **umount**, not “**unmount**”):

```
root@forensicbox:~# umount /mnt/analysis
```

So what happens with that **loop** option? When you pass the **loop** option in the **mount** ↪ command, you are actually calling a shortcut to creating loop devices with a special command, **losetup**. It is important that we understand the background here.

7.12.5 losetup

Creating loop devices is an important skill. Rather than letting the **mount** command take charge of that process, let's have a look at what is actually going on.

Loop devices are created by the Linux kernel in the **/dev** directory, just like other devices.

```
root@forensicbox:~# ls /dev/loop*
/dev/loop-control
/dev/loop0
/dev/loop1
/dev/loop2
/dev/loop3
/dev/loop4
/dev/loop5
/dev/loop6
/dev/loop7
```

These are devices that can be utilized to associate files with a device. The **/dev/loop-control** device is an interface to allow applications to associate loop devices. The command we use to manage our loop devices is **losetup**. Invoked by itself, **losetup** will list associated loop devices (it will return nothing if not loop devices are in use). In simplest form, you simply call **losetup** with the devices name (**/dev/loopX**) and the file you wish to associate it with:

```
root@forensicbox:~# losetup /dev/loop0 fat_fs.raw
```

```
root@forensicbox:~# losetup -l
```

NAME	SIZELIMIT	OFFSET	AUTOCLEAR	R0	BACK-FILE	DIO	LOG-SEC
/dev/loop0	0	0	0	0	/root/fat_fs.raw	0	512

```
root@forensicbox:~# file -s /dev/loop0
```

```
/dev/loop0: DOS/MBR boot sector, code offset 0x3e+2, OEM-ID "(wA~PIHC" cached by
  ↪ Windows 9M, root entries 224, sectors 2880 (volumes <=32 MB), sectors/FAT 9,
  ↪ sectors/track 18, serial number 0x16e42d6d, unlabeled, FAT (12 bit), followed
  ↪ by FAT
```

```
root@forensicbox:~# shasum fat_fs.raw
```

```
f5ee9cf56f23e5f5773e2a4854360404a62015cf /dev/loop0
```

```
root@forensicbox:~# sha1sum /dev/loop0
f5ee9cf56f23e5f5773e2a4854360404a62015cf fat_fs.raw
```

In the above commands, we associate the loop device `/dev/loop0` with the file `fat_fs.raw`. We follow by using the **losetup** command with the **-l** option to list the `/dev/loop` associations. This is essentially what occurs in the background when you issue the **mount** command with the **-o loop** option we used previously. When we identify the device file contents using **file** \rightarrow **-s**, we get the same as when we ran **file** on `fat_fs.raw`. We also see that the hash of `fat_fs.raw` matches the now associated loop device, indicating it is an exact duplicate. With the loop device associated, you can issue the **mount** command as if `fat_fs.raw` were a volume called `/dev/loop0`:

```
root@forensicbox:~# mount -t vfat -o ro /dev/loop0 /mnt/analysis/
```

```
root@forensicbox:~# mount | grep /mnt/analysis
/dev/loop0 on /mnt/analysis type vfat (ro,relatime,fmask=0022,dmask=0022,codepage
    \rightarrow =437,iocharset=iso8859-1,shortname=mixed,utf8,errors=remount-ro)
```

```
root@forensicbox:~# ls /mnt/analysis
ARP.EXE*
Docs/
FTP.EXE*
Pics/
loveletter.virus*
ouchy.dat*
snoof.gz*
```

```
root@forensicbox:~# umount /mnt/analysis
```

In the above command session, we mount the file system associated with `/dev/loop0`, using the read-only option (**-o ro**) on the mount point `/mnt/analysis`. We check the mount with the **mount** command displaying only lines that contain `/mnt/analysis` (using **grep**) and then list the contents of the mount point with **ls**. We unmount the file system with **umount**.

Finally, we can remove the loop association with **losetup -d**:

```
root@forensicbox:~# losetup
NAME          SIZELIMIT OFFSET AUTOCLEAR R0 BACK-FILE          DIO LOG-SEC
/dev/loop0    0          0          0 0 /root/fat_fs.raw    0      512
```

```
root@forensicbox:~# losetup -d /dev/loop0
```

```
root@forensicbox:~# losetup
```

```
root@forensicbox:~#
```

Not all media images are this simple, however...

7.12.6 Mounting Full Disk Images with **losetup**

The example used in the previous exercise utilizes a simple stand alone file system. What happens when you are dealing with boot sectors and multi partition disk images? When you create a raw image of media with **dd** or similar commands you usually end up with a number of components to the image. These components can include a boot sector, partition table, and the various partitions.

If you attempt to mount a full disk image with a loop device, you find that the **mount** command is unable to identify the file system. This is because mount does not know how to “recognize” the partition table. Remember, the **mount** command handles file systems, not disks (or disk images). The easy way around this (although it is not very efficient for large disks) would be to create separate images for each disk partition that you want to analyze. For a simple hard drive with a single large partition, you could create two images.

One for the entire disk:

```
root@forensicbox:~# dd if=/dev/sda of=fulldiskimage.raw
```

And one for the partition:

```
root@forensicbox:~# dd if=/dev/sda1 of=firstpartitionimage.raw
```

The first command gets you a full image of the entire disk (**/dev/sda**) for backup purposes, including the boot sector and partition table. The second command gets you the first partition (**/dev/sda1**). The resulting image from the second command can be mounted via the loop device, just as with our **fat_fs.raw**, because it is a simple file system.

Although both of the above images will contain the same file system with the same data, the hashes will obviously not match. Making separate images for each partition is very inefficient.

One method for handling full disks when using the loop device is to send the **mount** command a message to skip the boot sector of the image and find the partition. These sectors are used to contain information (like the MBR) that is not part of a normal file system. We can look at the offset to a partition, normally given in sectors (using the **fdisk** command), and multiply by 512 (the sector size). This gives us the byte offset from the start of our image to the first partition we want to mount. This is then passed to the **mount** command as an option, which

essentially triggers the use of an available loop device to mount the specified file system. We can illustrate this by looking at the raw image of the file we exported with **ewfexport** in our earlier acquisitions exercise, the **NTFS_Pract_2017.raw** file. Navigate to where you have the file saved.

Very quickly, let's run through the steps we need to mount this image. First time round, we'll determine the structure with **fdisk**, obtain the offset to the actual file system using math expansion, and then mount the file system using the **mount** command with the **-o loop** option.

```

root@forensicbox:~# fdisk -l NTFS_Pract_2017.raw
Disk NTFS_Pract_2017.raw: 500 MiB, 524288000 bytes, 1024000 sectors
Units: sectors of 1 *512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xe8dd21ee

Device                Boot Start      End Sectors  Size Id Type
NTFS_Pract_2017.raw1   2048 1023999 1021952   499M  7 HPFS/NTFS/exFAT

root@forensicbox:~# echo $((2048*512))
1048576

root@forensicbox:~# mount -o ro,loop,offset=1048576 NTFS_Pract_2017.raw /mnt/tmp

root@forensicbox:~# ls /mnt/tmp
ProxyLog1.log*
System\ Volume\ Information\
Users\
Windows\

```

So here we have a full disk image. We run **fdisk** on the image (an image file is no different than a device file) and find that the offset to the partition is **2048** bytes (in red for emphasis). We use arithmetic expansion to calculate the byte offset ($2048 * 512 = 1048576$) and pass that as the offset in our **mount** command. This effectively “jumps over” the boot sector and goes straight to the first partition, allowing the **mount** command to work properly. We will explore this in further detail later.

Note that you can do the calculations for the offset using arithmetic expansion directly in the **mount** command if you choose:

```

root@forensicbox:~# mount -o ro,loop,offset=$((2048*512)) NTFS_Pract_2017.raw
/mnt/tmp/

```

Let's look again and what is going on in the background here with the loop device. We'll run through the same **mount** exercise, but this time using **losetup**.

First, be sure the file system is unmounted:

```
root@forensicbox:~# umount /mnt/tmp
```

Now let's recreate the **mount** command using a loop device rather than an offset passed to mount. In this case we'll use arithmetic expansion directly in the commands:

```
root@forensicbox:~# fdisk -l NTFS_Pract_2017.raw
Disk NTFS_Pract_2017.raw: 500 MiB, 524288000 bytes, 1024000 sectors
Units: sectors of 1 *512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xe8dd21ee
```

Device	Boot Start	End Sectors	Size	Id	Type
NTFS_Pract_2017.raw1	2048	1023999	1021952	499M	7 HPFS/NTFS/exFAT

```
root@forensicbox:~# losetup -o $((2048*512)) -sizelimit
$((1021952*512))/dev/loop0 NTFS_Pract_2017.raw
```

```
root@forensicbox:~# losetup -l
```

NAME	SIZELIMIT	OFFSET	AUTOCLEAR	R0	BACK-FILE	DIO	LOG-SEC
/dev/loop0	523239424	1048576		0	0 /root/NTFS_Pract_2017.raw	0	512

```
root@forensicbox:~# mount /dev/loop0 /mnt/tmp
```

```
root@forensicbox:~# ls /mnt/tmp
ProxyLog1.log*
System\ Volume\ Information\
Users\
Windows\
```

```
root@forensicbox:~# umount /mnt/tmp
```

So here we are using the **losetup** command on an image, but this time we pass it an offset to a file system inside the image (**-o \$((2048*512))**) and we also let the loop device know the exact size of the partition we are associating (**--sizelimit \$((1021952*512))**). Once we've associated the loop device, we mount it to **/mnt/tmp** and list the contents of the image with **ls**¹⁸. Finally, we unmount the file system with **umount**. For a multi partition image,

¹⁸Note the slashes in the output of **ls**. The backslash is an escape character to allow the spaces within the directory name **System Volume Information/**, and the trailing forward slash identifies a directory. We see there are three directories in the output

you could repeat the steps above for each partition you wanted to mount, or you could use a tool set up to do exactly that. For single partition images like we have here, the **--sizelimit** option is actually not required.

7.12.7 Mounting Multi Partition Images with **kpartx**

Up to this point we've mounted a simple file system image with the **mount** command, we've mounted a file system from a full disk image with a single partition, and we've learned about the loop device and how to specify its association with a specific partition.

Let's look now at a disk image that has multiple partitions. Our previous method of identifying each partition by offset and size, and passing those parameters to the **losetup** command would work fine to mount multiple file systems within a disk image (using different loop devices for each partition), but wouldn't it be nice if we had a tool that could do all of that for us? **kpartx** is that tool.

In simple terms, **kpartx** maps partitions within an image to separate loop devices that can then be mounted the same as any other volume (assuming a mountable file system). It is part of the **multipath-tools** package available at slackbuilds.org.

The download URL for the **multipath-tools** source code in the current SlackBuild **.info** file does not work well with automated tools, so you can download the source code manually, install the dependency with **sboinstall**, and then build and install the package - all easier than it sounds, and shown below:

- change to the **multipath-tools** SlackBuild directory
 - `/usr/sbo/repo/system/multipath-tools`
 - or download the **multipath-tools** SlackBuild from slackbuilds.org.
- download the **multipath-tools** source code (to the SlackBuild) directory
- install the dependency (listed in the **.info** file) build and install the **multipath-tools** package

```
root@forensicbox:~# sbofind multipath-tools
SBo:    multipath-tools 0.7.8
Path:   /usr/sbo/repo/system/multipath-tools

root@forensicbox:~# cd /usr/sbo/repo/system/multipath-tools/

root@forensicbox:multipath-tools# cat multipath-tools.info
PRGNAME="multipath-tools"
VERSION="0.7.8"
```

```

HOMEPAGE="http://christophe.varoqui.free.fr/"
DOWNLOAD=
    ↪ "https://git.opensvc.com/?p=multipath-tools/.git;a=snapshot;sf=tgz;h=refs/tags/0.7.8"
MD5SUM="f8d0faed2913bc725c107b4f84f22a3a"
DOWNLOAD_x86_64=""
MD5SUM_x86_64=""
REQUIRES="liburcu"
MAINTAINER="Nikos Giotis"
EMAIL="nikos.giotis@gmail.com"

root@forensicbox:multipath-tools# wget
    "https://git.opensvc.com/?p=multipath-tools/.git;a=snapshot;sf=tgz;h=refs/tags/0.7.8"
...
Notice the url is quoted

2019-07-17 09:57:11 (953 KB/s) -
'index.html?p=multipath-tools%2F.git;a=snapshot;sf=tgz;h=refs%2Ftags%2F0.7.8' saved
    ↪ [415806]

root@forensicbox:multipath-tools# sbinstall liburcu
...
Package liburcu-0.11.1-x86_64-1_SBo.tgz installed.
Cleaning for liburcu-0.11.1...

root@forensicbox:multipath-tools# sh multipath-tools.SlackBuild
...
Slackware package /tmp/multipath-tools-0.7.8-x86_64-1_SBo.tgz created.

root@forensicbox:multipath-tools# installpkg
/tmp/multipath-tools-0.7.8-x86_64-1_SBo.tgz
...
Executing install script for multipath-tools-0.7.8-x86_64-1_SBo.tgz.
Package multipath-tools-0.7.8-x86_64-1_SBo.tgz installed.

```

Once the `multipath-tools` package is installed, you can have a look through the man page for **kpartx** with `man kpartx`. Usage is very simple. There is a very simple multi partition image you can download and use to practice. The partitions are empty for maximum compressability. Download the file with **wget** and check the hash to ensure it matches the one below:

```

root@forensicbox:~# wget http://www.linuxleo.com/Files/gptimage.raw.gz
--2019-07-17 10:18:44-- http://www.linuxleo.com/Files/gptimage.raw.gz
Resolving www.linuxleo.com (www.linuxleo.com)... 74.208.236.144
Connecting to www.linuxleo.com (www.linuxleo.com)|74.208.236.144|:80... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://linuxleo.com/Files/gptimage.raw.gz [following]
--2019-07-17 10:18:45-- https://linuxleo.com/Files/gptimage.raw.gz

```

```

Resolving linuxleo.com (linuxleo.com)... 74.208.236.144
Connecting to linuxleo.com (linuxleo.com)|74.208.236.144|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4181657 (4.0M) [application/gzip]
Saving to: 'gptimage.raw.gz'

gptimage.raw.gz      100%[=====>]    3.99M  2.43MB/s   in 1.6s

2019-07-17 10:18:48 (2.43 MB/s) - 'gptimage.raw.gz' saved [4181657/4181657]

root@forensicbox:~# shasum gptimage.raw.gz
b7dde25864b9686aafe78a3d4c77406c3117d30c  gptimage.raw.gz

```

Decompress the gzip'd file with **gzip -d** and check the hash of the resulting raw image file:

```

root@forensicbox:~# gzip -d gptimage.raw.gz

root@forensicbox:~# shasum gptimage.raw
99b7519cecb9a48d2fd57c673cbf462746627a84  gptimage.raw

```

Now we can run **kpartx** on the image file to check the partitions, and then to map them, read only, to loop device nodes. To see the available options, run **kpartx** by itself:

```

root@forensicbox:~# kpartx
usage : kpartx [-a|-d|-l] [-f] [-v] wholedisk
    -a add partition devmappings
    -r devmappings will be readonly
    -d del partition devmappings
    -u update partition devmappings
    -l list partitions devmappings that would be added by -a
    -p set device name-partition number delimiter
    -g force GUID partition table (GPT)
    -f force devmap create
    -v verbose
    -n nosync mode. Return before the partitions are created
    -s sync mode. Don't return until the partitions are created. Default.

```

The first command below will list the partitions as they will appear (**-l**). After that we add the mappings in the second command with (**-a**) and create them with the read only option as well (**-r**):

```

root@forensicbox:~# kpartx -l gptimage.raw
loop0p1 : 0 204800 /dev/loop0 2048
loop0p2 : 0 2097152 /dev/loop0 206848
loop0p3 : 0 6084575 /dev/loop0 2304000

```

```
root@forensicbox:~# kpartx -r -a gptimage.raw
```

Once we execute the command above, our mappings are created and we can now access each partition through the `/dev/mapper/loop0pX` device, where X is the number of the partition.

```
root@forensicbox:~# ls -l /dev/mapper
total 0
crw----- 1 root root 10, 236 Jul 17 06:16 control
lrwxrwxrwx 1 root root      7 Jul 17 10:27 loop0p1 -> ../dm-0
lrwxrwxrwx 1 root root      7 Jul 17 10:27 loop0p2 -> ../dm-1
lrwxrwxrwx 1 root root      7 Jul 17 10:27 loop0p3 -> ../dm-2
```

One thing to keep in mind is that the `/dev/mapper` nodes are actually symbolic links to `/dev/dm-*` nodes¹⁹, so if you run the `file` command to try and detect the file system type, it will simply say symbolic link. To use the `file` command, run it against the `dm` device. All other operations that we use here can be done on `/dev/mapper/loop0pX`.

```
root@forensicbox:~# file -s /dev/mapper/loop0p1
/dev/mapper/loop0p1: symbolic link to ../dm-1

root@forensicbox:~# file -s /dev/dm-*
/dev/dm-0: Linux rev 1.0 ext4 filesystem data, UUID=cd5213b1-e674-41b2-8f7f-
    ↪ d6f6e97fbdee (extents) (large files) (huge files)
/dev/dm-1: Linux rev 1.0 ext4 filesystem data, UUID=7f7be41c-4b0d-41d4-8c94-
    ↪ ff84a121e542 (extents) (large files) (huge files)
/dev/dm-2: Linux rev 1.0 ext4 filesystem data, UUID=837a55a6-39f1-433b-bf1a-34538
    ↪ feee7e8 (extents) (large files) (huge files)
```

We can now mount and browse these mapped volumes as we would any other.

```
root@forensicbox:~# mount /dev/mapper/loop0p1 /mnt/tmp
mount: /mnt/tmp: WARNING: device write-protected, mounted read-only.

root@forensicbox:~# ls /mnt/tmp
lost+found/

root@forensicbox:~# mount | grep mapper
/dev/mapper/loop0p1 on /mnt/tmp type ext4 (ro,relatime)

root@forensicbox:~# umount /mnt/tmp
```

¹⁹Remember the `../` notation indicates the `dm-*` nodes are in the current directory's parent directory.

Once you are finished and the file system is unmounted with the **umount** command as shown above, you can delete the mappings with **kpartx -d**:

```
root@forensicbox:~# kpartx -d gptimage.raw
loop deleted : /dev/loop0
```

7.12.8 Mounting Split Image Files with **affuse**

We are going to continue our exploration of mounting options for image files by addressing those occasions where you might want to mount and browse an image file that has been split with **dd/split** or **dc3dd**, etc. For that we can use **affuse** from the **afflib** package.

The Advanced Forensic Format (AFF) is an open format for forensic imaging, and the **afflib** package provides a number of utilities to create and manipulate images in the AFF format. We won't cover those tools, or the AFF format in this document (at least not in this version), so all we are interested in right now is the **affuse** program.

affuse provides virtual access to a number of image formats, split files among them. It does this through the File System in User Space software interface. Commonly referred to as “fuse file systems”, fuse utilities allow us to create application level file system access mechanisms that can bridge to the kernel and the normal file system drivers.

The **afflib** package is available as a SlackBuild for Slackware, and can be simply installed with **sboinstall**:

```
root@forensicbox:~# sboinstall afflib
afflib is library and set of tools used to support of Advanced Forensic
Format (AFF).
...
```

Package afflib-3.7.7-x86_64-1_SBo.tgz installed.

Cleaning for afflib-3.7.7...

The following exercise assumes that the split image you are working with is in raw format (like a **dd** image) when reassembled. A file that we will use for a number of exercises later on is in split format and can be downloaded so you can follow along here. Again, use **wget** and check your hash against the one below:

```
root@forensicbox:~# wget http://www.linuxleo.com/Files/able_3.tar.gz
--2019-07-18 08:28:29-- https://linuxleo.com/Files/able_3.tar.gz
Resolving linuxleo.com... 74.208.236.144
```

```
Connecting to linuxleo.com[74.208.236.144]:443... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 526734961 (502M) [application/gzip]  
Saving to: 'able_3.tar.gz'
```

```
able_3.tar.gz      100%[=====>] 502.33M  3.11MB/s   in 3m 31s
```

```
2019-07-18 08:32:02 (2.38 MB/s) - 'able_3.tar.gz' saved [526734961/526734961]
```

```
root@forensicbox:~# sha1sum able_3.tar.gz  
6d8de5017336028d3c221678b483a81e341a9220  able_3.tar.gz
```

View the contents of the archive with the following command:

```
root@forensicbox:~# tar tzf able_3.tar.gz  
able_3/  
able_3/able_3.000  
able_3/able_3.001  
able_3/able_3.log  
able_3/able_3.003  
able_3/able_3.002
```

Now we can extract the archive using the **tar** command with the extract option (**x**) rather than the option to list contents (**t**):

```
root@forensicbox:~# tar xzvf able_3.tar.gz  
able_3/  
able_3/able_3.000  
able_3/able_3.001  
able_3/able_3.log  
able_3/able_3.003  
able_3/able_3.002
```

First, change to the **able_3** directory with **cd**. Note our command prompt changed to reflect our working directory. We now have 4 image files (**.000-.003**) and a log file. The input section of the log file shows that this image is a 4G image taken with **dc3dd** and split into 4 parts.

```
root@forensicbox:~# cd able_3  
  
root@forensicbox:able_3# cat able_3.log  
dc3dd 7.2.646 started at 2017-05-25 15:51:04 +0000  
compiled options:  
command line: dc3dd if=/dev/sda hofs=able_3.000 ofsz=1G hash=sha1 log=able_3.log  
device size: 8388608 sectors (probed),    4,294,967,296 bytes
```

```
sector size: 512 bytes (probed)
4294967296 bytes ( 4 G ) copied ( 100% ), 1037.42 s, 3.9 M/s
4294967296 bytes ( 4 G ) hashed ( 100% ), 506.481 s, 8.1 M/s

input results for device '/dev/sda':
8388608 sectors in
0 bad sectors replaced by zeros
2eddbfe3d00cc7376172ec320df88f61afda3502 (sha1)
4ef834ce95ec545722370ace5a5738865d45df9e, sectors 0 - 2097151
ca848143cca181b112b82c3d20acde6bdaf37506, sectors 2097152 - 4194303
3d63f2724304205b6f7fe5cadcbc39c05f18cf30, sectors 4194304 - 6291455
9e8607df22e24750df7d35549d205c3bd69adfe3, sectors 6291456 - 8388607

output results for files 'able_3.000':
8388608 sectors out
[ok] 2eddbfe3d00cc7376172ec320df88f61afda3502 (sha1)
[ok] 4ef834ce95ec545722370ace5a5738865d45df9e, sectors 0 - 2097151, 'able_3
↪ .000'
[ok] ca848143cca181b112b82c3d20acde6bdaf37506, sectors 2097152 - 4194303, '
↪ able_3.001'
[ok] 3d63f2724304205b6f7fe5cadcbc39c05f18cf30, sectors 4194304 - 6291455, '
↪ able_3.002'
[ok] 9e8607df22e24750df7d35549d205c3bd69adfe3, sectors 6291456 - 8388607, '
↪ able_3.003'

dc3dd completed at 2017-05-25 16:08:22 +0000
```

Let's check the hash of the image parts combined and compare to the log:

```
root@forensicbox:able_3# cat able_3.00* | shasum
2eddbfe3d00cc7376172ec320df88f61afda3502 -
```

Remember that the **cat** command simply streams the files one after the other and sends them through standard out. The **shasum** command takes the data from the pipe and hashes it. As we mentioned earlier, the ' - ' in the hash output indicates standard input was hashed, not a file. The hashes match and our image is good.

Now suppose we want to mount the images to see the file systems and browse or search them for specific files. One solution would be to use the **cat** command like we did above and redirect the output to a new file made up of all the segments.

```
root@forensicbox:able_3# cat able_3.0*> able_3.raw

root@forensicbox:able_3# ls -lh able_3.raw
-rw-r--r-- 1 root root 4.0G Jul 18 09:13 able_3.raw
```

```
root@forensicbox:able_3# shasum able_3.raw
2eddbfe3d00cc7376172ec320df88f61afda3502  able_3.raw
```

The problem with this approach is that it takes up twice the space as we are essentially duplicating the entire acquired disk, but in a single image rather than split. Not very efficient for resource management.

We need a way to take the split images and create a virtual "whole disk" that we can mount using techniques we've learned already. We'll use **affuse** and the fuse file system it provides. All we need to do is call **affuse** with the name of the first segment of our split image and provide a mount point where we can access the virtual disk image:

```
root@forensicbox:able_3# mkdir /mnt/aff

root@forensicbox:able_3# affuse able3.000 /mnt/aff

root@forensicbox:able_3# ls -lh /mnt/aff
total 0
-r--r--r-- 1 root root 4.0G Dec 31 1969 able_3.000.raw

root@forensicbox:able_3# shasum /mnt/aff/able_3.000.raw
2eddbfe3d00cc7376172ec320df88f61afda3502  /mnt/aff/able_3.000.raw
```

In the above session, we create a mount point for our fuse image (the name here is arbitrary) with the **mkdir** command. Then we use **affuse** with the first segment of our four part image and fuse mount it to `/mnt/aff`. **affuse** creates our single virtual image file for us in `/mnt/aff` and names it with the image name and the `.raw` extension. Finally, we check the hash of this new virtual image and find it's the same as the hash for the input and output bytes (for the total disk) in our log file.

Now we can run **gdisk** or **fdisk** on the image to identify the partition layout of the disk; we can use **kpartx** to map the partitions to loop devices we can mount; and we can run the **file** command to identify the file systems for further investigation. All this as we have learned in preceding sections when working on complete image files:

```
root@forensicbox:able_3# fdisk -l /mnt/aff/able_3.000.raw
Disk /mnt/aff/able_3.000.raw: 4 GiB, 4294967296 bytes, 8388608 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: B94F8C48-CE81-43F4-A062-AA2E55C2C833
```

Device	Start	End	Sectors	Size	Type
/mnt/aff/able_3.000.raw1	2048	104447	102400	50M	Linux filesystem

```
/mnt/aff/able_3.000.raw2 104448 309247 204800 100M Linux filesystem
/mnt/aff/able_3.000.raw3 571392 8388574 7817183 3.7G Linux filesystem
```

```
root@forensicbox:able_3# kpartx -a -r /mnt/aff/able_3.000.raw
```

```
root@forensicbox:able_3# ls -l /dev/mapper/loop0p*
lrwxrwxrwx 1 root root 7 Jul 18 09:28 /dev/mapper/loop0p1 -> ../dm-0
lrwxrwxrwx 1 root root 7 Jul 18 09:28 /dev/mapper/loop0p2 -> ../dm-1
lrwxrwxrwx 1 root root 7 Jul 18 09:28 /dev/mapper/loop0p3 -> ../dm-2
```

```
root@forensicbox:able_3# file -s /dev/dm-*
/dev/dm-0: Linux rev 1.0 ext4 filesystem data, UUID=ca05157e-f7b3-4c6a-9b63-235
    ↪ c4cad7b73 (extents) (large files) (huge files)
/dev/dm-1: Linux rev 1.0 ext4 filesystem data, UUID=c4ac4c0f-d9de-4d26-9e16-10583
    ↪ b607372 (extents) (large files) (huge files)
/dev/dm-2: Linux rev 1.0 ext4 filesystem data, UUID=c7f748b2-3a38-44e9-aa43-
    ↪ f924955b9fdd (extents) (large files) (huge files)
```

So without having to reassemble the split images to a single image, we were able to map the partitions and identify the file systems ready for mounting.

```
root@forensicbox:able_3# mount -o ro -t ext4 /dev/mapper/loop0p1 /mnt/analysis/
```

```
root@forensicbox:able_3# ls /mnt/analysis
README.initrd@      config-huge-4.4.14  onlyblue.dat
System.map@          elilo-ia32.efi*     slack.bmp
System.map-generic-4.4.14  elilo-x86_64.efi*  tuxlogo.bmp
System.map-huge-4.4.14    grub/              tuxlogo.dat
boot.0800            inside.bmp          vmlinuz@
boot_message.txt      inside.dat          vmlinuz-generic@
coffee.dat           lost+found/         vmlinuz-generic-4.4.14
config@              map                 vmlinuz-huge@
config-generic-4.4.14  onlyblue.bmp        vmlinuz-huge-4.4.14
```

When we have finished with the **affuse** mount point, we remove it with the **fusermount -u** command. This removes our virtual disk image from the mount point. REMEMBER we must unmount any mounted file systems from the image, and then delete our loop associations with **kpartx -d** prior to our fuse unmount.

```
root@forensicbox:able_3# umount /mnt/analysis
```

```
root@forensicbox:able_3# kpartx -d /mnt/aff/able_3.000.raw
loop deleted : /dev/loop0
```

```
root@forensicbox:able_3# fusermount -u /mnt/aff
```

7.12.9 Mounting EWF files with **ewfmount**

Just as we are bound to come across split images we want to browse, we are also likely to come across Expert Witness (**E01** or **EWF**) files that we want to peak into without having to restore them and take up much more space than we need to.

We've already installed **libewf** as part of our acquisition lessons earlier. If you have not done so already, you can install **libewf** with **sboinstall** on Slackware or using whichever method your distribution of choice allows. For this section we are interested in the **ewfmount** utility that comes with **libewf**.

Like **affuse**, **ewfmount** provides a fuse file system. It is called in the same way, and results in the same virtual raw disk image that can be parsed for partitions and loop mounted for browsing. If you read the prior section on **affuse**, this will all be very familiar. We will use the **EWF** version of **NTFS_Pract_2017.E0*** files we used in our earlier exercises.

It might be a good idea to run **ewfverify** (also from the **libewf** package – recall we used it in the acquisitions section) to ensure the integrity of the **E01** set is still intact.

```
root@forensicbox:~# cd NTFS_Pract_2017
```

```
root@forensicbox:NTFS_Pract_2017# ewfverify NTFS_Pract_2017.E01
ewfverify 20140806
```

```
Verify started at: Jul 18, 2019 09:56:58
This could take a while.
```

```
Verify completed at: Jul 18, 2019 09:56:59
```

```
Read: 500 MiB (524288000 bytes) in 1 second(s) with 500 MiB/s (524288000 bytes/
↪ second).
```

```
MD5 hash stored in file:      eb4393cfcc4fca856e0edbf772b2aa7d
MD5 hash calculated over data: eb4393cfcc4fca856e0edbf772b2aa7d
```

```
ewfverify: SUCCESS
```

Make note of the MD5 hash from our **ewfverify** output.

Now we create our **EWF** mount point (again this is an arbitrary name). Use the **ewfmount** command to fuse mount the image files. You only need to provide the first file name for the image set. **ewfmount** will find the rest of the segments. We can use the **ls** command on our mount point to see the fuse mounted disk image that resulted:

```
root@forensicbox:NTFS_Pract_2017# mkdir /mnt/ewf
```

```
root@forensicbox:NTFS_Pract_2017# ewfmount NTFS_Pract_2017.E01 /mnt/ewf
ewfmount 20140806
```

```
root@forensicbox:NTFS_Pract_2017# ls /mnt/ewf
ewf1
```

Our virtual disk image is **ewf1**. Let's hash that and compare it to our **ewfverify** output above. As you can see, we get a match:

```
root@forensicbox:NTFS_Pract_2017# md5sum /mnt/ewf/ewf1
eb4393cfcc4fca856e0edbf772b2aa7d /mnt/ewf/ewf1
```

And now once again we are ready to parse and mount our fuse disk image using the techniques we've already learned. Remember, the fuse mounted image (**/mnt/ewf/ewf1**) is a raw image:

```
root@forensicbox:NTFS_Pract_2017# fdisk -l /mnt/ewf/ewf1
```

```
Units: sectors of 1 * 512 = 512 bytes
```

```
Sector size (logical/physical): 512 bytes / 512 bytes
```

```
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

```
Disklabel type: dos
```

```
Disk identifier: 0xe8dd21ee
```

Device	Boot	Start	End	Sectors	Size	Id	Type
/mnt/ewf/ewf1p1		2048	1023999	1021952	499M	7	HPFS/NTFS/exFAT

```
root@forensicbox:NTFS_Pract_2017# kpartx -a -r /mnt/ewf/ewf1
```

```
root@forensicbox:NTFS_Pract_2017# file -s /dev/dm-0
```

```
/dev/dm-0: DOS/MBR boot sector, code offset 0x52+2, OEM-ID "NTFS  ", sectors/
↳ cluster 8, Media descriptor 0xf8, sectors/track 63, heads 255, hidden sectors
↳ 2048, dos < 4.0 BootSector (0x0), FAT (1Y bit by descriptor); NTFS, sectors/
↳ track 63, physical drive 0x80, sectors 1021951, $MFT start cluster 42581,
↳ $MFTMirror start cluster 2, bytes/RecordSegment 2^(-1*246), clusters/index
↳ block 1, serial number 0cae0dfd2e0dfc2bd
```

```
root@forensicbox:NTFS_Pract_2017# mount -o ro -t ntfs-3g
/dev/mapper/loop0p1/mnt/analysis
```

```
root@forensicbox:NTFS_Pract_2017# ls /mnt/analysis
ProxyLog1.log* System\ Volume\ Information\ Users\ Windows\
```

Using **fdisk -l**, we see the structure of the image. We use **kpartx** with the read-only option (**-r**) to add the loop mapping (**-a**) for the partition. We check the file system type with

file -s and confirm it is NTFS. Finally we mount the volume with the **mount** command. In this case we use the **ntfs-3g**²⁰ file system driver (**-t ntfs-3g**).

And, as before, when we are finished we need to unmount the volume, delete the mappings, and the unmount the fuse file system. This is the same set of steps (forward and backward) we did with **affuse** and the split image.

```
root@forensicbox:NTFS_Pract_2017# umount /mnt/analysis

root@forensicbox:NTFS_Pract_2017# kpartx -d /mnt/ewf/ewf1
loop deleted : /dev/loop0

root@forensicbox:NTFS_Pract_2017# fusermount -u /mnt/ewf
```

And that covers our section on mounting evidence. As with everything in this guide, we've left a lot of detail out. Experiment and read the man pages. Make sure you know what you are doing when dealing with real evidence. Mounting and browsing images should always be done on working copies when possible.

7.13 Basic Analysis

In this section we'll begin our exploration of basic command line functionality to review mounted evidence. So far we've covered tools that allow us to *access* our evidence, now it's time to explore it.

Most of what we cover here will be applicable in actual examinations. Some of it, you will find, can better be accomplished using other more focused tools. But as with everything else, we are building a foundation for Linux platform knowledge, not just a subset of forensic tools. Once again, there may be times where you are left with only those tools that come with a majority of distributions. It is helpful to know your way around some of these basic utilities.

We'll start with something that *is* useful, even on a fully equipped forensic platform: Virus scanning.

7.13.1 Anti Virus - Scanning the Evidence with **clamav**

Part of our approach to understanding and deploying Linux as a computer forensic platform is making the entire process "stand alone". The goal of this guide is to enable you to conduct

²⁰There are a number of useful options when mounting with **ntfs-3g**, like **show_sys_files** or **streams_interface=windows**. We don't cover them here, but you might want to look at **man mount** → **.ntfs-3g** for more information.

an exam – from analysis through reporting – within the Linux (and preferably command line) environment. One of those steps we should consider taking in almost all examinations we are tasked with is to scan our acquired data with some sort of anti-virus tool.

We've all heard the ever famous "Trojan Horse Defense", where we worry that malicious activity will be blamed on an infected computer that the defendant "had no control over". While it's not something I've personally experienced, it has happened and is well documented²¹. Scanning the evidence makes sense for both making your case more solid and an exculpatory point of view. If there are circumstances where malware may have played a role, we will certainly want to know that.

There are other considerations that warrant a virus/malware scan, and it can very specifically depend on the type of case you are investigating. Simply making it part of some check list routine for analysis is fine, but you must still have an understanding of why the scan is done, and how it applies to the current case. For example, if the media being examined is the victim of compromise, then a virus scan can provide a starting point for additional analysis. The starting point can be as simple as identifying a vector, and utilizing file dates and times to drive additional analysis. Alternatively, we may find ourselves examining the computer in a child exploitation case. Negative results, while presumptive, can still help to combat the previously discussed Trojan Horse Defense. The bottom line is that a simple virus scan should always be included as standard practice. And while there are plenty of tools out there compatible with Linux, we will focus on ClamAV.

ClamAV is open source and freely available. It is well supported and is quite comparable to other anti-virus with respect to identifying infections and artifacts. If you are deploying Linux in a laboratory environment, it also provides excellent backup and cross-verification to anti-virus results provided in other operating systems.

We already installed the ClamAV package earlier in the section covering external software. If you have not done so, either install **clamav** via the SlackBuild (using **sboinstall**) or via your distribution's package management method.

ClamAV has far more uses and configuration options than we will cover here. It can be used to scan "on use" volumes, email servers, and has options and uses for "safe browsing". There are tools installed with the ClamAV package that allow for byte code review, submission of samples, and to assist with daemon mode configuration. We will be using it to scan acquired evidence. This assumes we will update it as needed and run it on targeted image files, volumes or mount points. With our simplified use case, we will concentrate our use on two specific ClamAV tools: **freshclam** and **clamscan**.

Once ClamAV is installed, we will need to download the definition files. We do this with **freshclam**. This command will download the appropriate files with the initial **main.cvd**, as well as the **daily.cvd** containing the most recent signatures:

²¹<http://digitalcommons.law.scu.edu/cgi/viewcontent.cgi?article=1370&context=chtlj>

```
root@forensicbox:~# freshclam
ClamAV update process started at Thu Jul 18 13:07:15 2019
WARNING: [LibClamAV] cl_cvdhead: Can't read CVD header in main.cvd
Downloading main.cvd [100%]
main.cvd updated (version: 58, sigs: 4566249, f-level: 60, builder: sigmgr)
WARNING: [LibClamAV] cl_cvdhead: Can't read CVD header in daily.cvd
Downloading daily.cvd [100%]
daily.cvd updated (version: 25514, sigs: 1660366, f-level: 63, builder: raynman)
Downloading bytecode.cvd [100%]
bytecode.cvd updated (version: 330, sigs: 94, f-level: 63, builder: neo)
Database updated (6226709 signatures) from database.clamav.net (IP: 104.16.218.84)
WARNING: Clamd was NOT notified: Can't connect to clamd through /var/run/clamav/
↳ clamd.socket: No such file or directory
```

Here we see the download of `main.cvd`, `daily.cvd` and `bytecode.cvd`. There are a couple of warnings issued, and this is because the files do not exist and **freshclam** attempts to read the version headers before updating. Subsequent updates will not show these warnings. We also get a warning that Clamd was NOT notified. This is because we are not running a scanning daemon (common for mail servers). You can run **freshclam --no-warnings** if you wish to suppress those.

We are now ready to run **clamscan** on our target. ClamAV supports direct scanning of files, and can recurse through many different file types and archive, including zip files, PDF files, mount points and forensic image files (GPT and MBR partition types). The most reliable way of running **clamscan** is to run it on a mounted file system.

There are options within **clamscan** to copy or move infected files to alternative directories. Normally we do not do this with infected files or malware during a forensic examination, preferring to examine the files in place, or extract them with forensic tools. Check **man** `clamscan` for additional details if you are interested. The output of **clamscan** can be logged with the **--log=logfile** option, useful for keeping complete examination notes.

We will try out **clamscan** on our NTFS EWF files we downloaded previously. Change into the directory the files are located in, use **ewfmount** to mount the images, and then **kpartx** to map the NTFS partition. If you do not already have the destination mount points in `/mnt` created, then use **mkdir** to create them now. We will scan the NTFS partition.

```
root@forensicbox:~# cd NTFS_Pract_2017

root@forensicbox:NTFS_Pract_2017# ewfmount NTFS_Pract_2017.E01 /mnt/ewf
ewfmount 20140806

root@forensicbox:NTFS_Pract_2017# kpartx -a -r /mnt/ewf/ewf1
```

```
root@forensicbox:NTFS_Pract_2017# mount -o ro -t ntfs-3g /dev/mapper/loop0p1  
/mnt/analysis
```

```
root@forensicbox:NTFS_Pract_2017# clamscan -r -i /mnt/analysis -log=NTFS_AV.txt
```

```
/mnt/analysis/Windows/System32/eicar.com: Eicar-Test-Signature FOUND
```

```
----- SCAN SUMMARY -----
```

```
Known viruses: 6217146  
Engine version: 0.101.2  
Scanned directories: 26  
Scanned files: 187  
Infected files: 1  
Data scanned: 304.03 MB  
Data read: 95.12 MB (ratio 3.20:1)  
Time: 88.004 sec (1 m 28 s)
```

Using **ewfmount**, we fuse mount the EWF files to `/mnt/ewf`, and then we use **kpartx** to map the NTFS partition, which is then mounted on `/mnt/analysis`. The command **clamscan** is then run with the **-r** option for recursive (scan sub directories), and **-i** to only show infected files. The **-i** option prevents overly cluttered output (lists of “OK” files). Finally, we use **--log** to document our output. The virus signature found is for a common anti-virus test file.

View the resulting log with **cat**:

```
root@forensicbox:NTFS_Pract_2017# cat NTFS_AV.txt
```

```
-----  
/mnt/analysis/Windows/System32/eicar.com: Eicar-Test-Signature FOUND
```

```
----- SCAN SUMMARY -----
```

```
Known viruses: 6217146  
Engine version: 0.101.2  
Scanned directories: 26  
Scanned files: 187  
Infected files: 1  
Data scanned: 304.03 MB  
Data read: 95.12 MB (ratio 3.20:1)  
Time: 88.004 sec (1 m 28 s)
```

When you are finished, unmount the NTFS file system, remove the **kpartx** mapping, and unmount the fuse mounted image.

```
root@forensicbox:NTFS_Pract_2017# umount /mnt/analysis
```

```
root@forensicbox:NTFS_Pract_2017# kpartx -d /mnt/ewf/ewf1
loop deleted : /dev/loop0
```

```
root@forensicbox:NTFS_Pract_2017# fusermount -u /mnt/ewf
```

This is a very simple example of virus scanning evidence with ClamAV. This is an exceptionally powerful tool, and you should explore the **man** page and the online documentation.

7.13.2 Basic Data Review on the Command Line

Linux comes with a number of simple utilities that make imaging and basic review of suspect disks and drives comparatively easy. We've already covered **dd**, **fdisk**, limited **grep** commands, hashing and file identification with the **file** command. We'll continue to use those tools, and also cover some additional utilities in some hands on exercises.

Following is a very simple series of steps to allow you to perform an easy practice data review using the simple tools mentioned above. All of the commands can be further explored with **man [command]**. Again, this is just an introduction to the basic commands. Our focus here is on the commands themselves, NOT on the file system we are reviewing. These steps can be far more powerful with some command line tweaking.

Having already said that this is just an introduction, most of the work you will do here can be applied to actual casework. The tools are standard GNU/Linux tools, and although the example shown here is very simple, it can be extended with some practice and a little (OK, a lot) of reading. The practice file system we'll use here is a simple old raw image of a FAT file system produced by the **dd** command²² We used this image in some previous exercises. If you have not already, download it now. You can do this as a normal user with **wget**:

```
barry@forensicbox:~$ wget https://www.linuxleo.com/Files/fat_fs.raw
--2019-07-18 21:13:10-- https://www.linuxleo.com/Files/fat_fs.raw
Resolving www.linuxleo.com... 74.208.236.144
Connecting to www.linuxleo.com[74.208.236.144]:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1474560 (1.4M)
Saving to: 'fat_fs.raw'

fat_fs.raw          100%[=====>]   1.41M  4.29MB/s   in 0.3s

2019-07-18 21:13:11 (4.29 MB/s) - 'fat_fs.raw' saved [1474560/1474560]

barry@forensicbox:~$ shasum fat_fs.raw
f5ee9cf56f23e5f5773e2a4854360404a62015cf  fat_fs.raw
```

²²This is the exact same image as the previously named **practical.floppy.dd**.

The output of various commands and the amount of searching we will do here is limited by the scope of this example and the amount of data in this very small image.

As we previously mentioned, when you actually do an analysis on larger media, you will want to have it organized. When you issue a command that results in an output file, that file will end up in your current directory, unless you specify a path for it.

One way of organizing your data would be to create a directory in your **home** directory for evidence and then a sub directory for different cases. You can create your output directory on any media or volume you like. For the sake of simplicity here, we'll use our home directory. We'll go ahead and do this as a regular user, so we can get used to running commands needed for root access. It's never a good idea to do all your work logged in as root. Here's our command to create an output directory for analysis results. We are executing the command in the directory where we placed our image file above. The `./` in front of the directory name we are creating indicates "in the current directory":

```
barry@forensicbox:~$ mkdir ./analysis

barry@forensicbox:~$ ls
analysis/  fat_fs.raw
```

Directing all of our analysis output to this directory will keep our output files separated from everything else and helps maintain case organization. You may wish to have a separate drive mounted as `/mnt/analysis` to hold your analysis output. How you organize it is up to you.

An additional step you might want to take is to create a special mount point for all subject file system analysis. This is another way of separating common system use with evidence processing. To create a mount point in the `/mnt` directory you will need to be temporarily logged in as root. In this case we'll log in as root, create a mount point, and then mount the `fat_fs.raw` image for further examination. Recall our discussion on the "super user" (root). We use the command `su` to become root:

```
barry@forensicbox:~# su -
Password: <enter root password>

root@forensicbox:~# mkdir /mnt/evid
```

Still using our root login, we'll go ahead and mount the `fat_fs.raw` image on `/mnt/evid`:

```
root@forensicbox:~# mount -t vfat -o ro,loop ~barry/fat_fs.raw /mnt/evid

root@forensicbox:~# losetup
```

NAME	SIZELIMIT	OFFSET	AUTOCLEAR	R0	BACK-FILE
/dev/loop0	0	0	1	1	/home/barry/fat_fs.raw

The first command above is our **mount** command with the file system type set to **vfat** (**-t vfat**) and the options (**-o**) read only (**ro**) and using the loop device (**loop**). The file system we are mounting, **fat_fs.raw**, is located in **/home/barry** (as indicated by the tilde in front of the user's name) mounting it on **/mnt/evid**. For illustration, we use the **loop** command to show the loop association. There are other useful mount options as well, such as **noatime** and **noexec**. See **man mount** for more details.

With the image mounted, we can exit our root login.

```
root@forensicbox:~# exit
```

```
barry@forensicbox:~$
```

You can now view the contents of the read-only, loop mounted image. This is where you may find the command line a powerful tool, allowing file redirection and other methods to permanently record your analysis.

Let's assume you are issuing the following commands from the proper mount point (**/mnt/evid**). If you want to save a copy of each command's output, be sure to re-direct the output to your evidence directory (**~/analysis**) using an explicit path. Again, if you are logged in as user **timmy**, then the tilde is a shortcut to **/home/timmy**. So in my case, typing **~/analysis** is the same as typing **/home/barry/analysis**.

Navigate through the directories and see what you can find. Use the **ls** command. Again, you should be in the directory **/mnt/evid**, where the image is mounted.

- The evidence is mounted on **/mnt/evid**
- We will be writing our results to **~/analysis**

You can begin with a simple listing of the contents of the volume:

```
barry@forensicbox:evid$ ls -l
total 107
-rwxr-xr-x 1 root root 19536 Aug 24 1996 ARP.EXE*
drwxr-xr-x 3 root root 512 Sep 23 2000 Docs/
-rwxr-xr-x 1 root root 37520 Aug 24 1996 FTP.EXE*
drwxr-xr-x 2 root root 512 Sep 23 2000 Pics/
-r-xr-xr-x 1 root root 16161 Sep 21 2000 loveletter.virus*
-rwxr-xr-x 1 root root 21271 Mar 19 2000 ouchy.dat*
-rwxr-xr-x 1 root root 12384 Aug 2 2000 snoof.gz*
```

This will list the files in long format to identify permission, date, etc. (**-l**). You can also use the **-R** option to list recursively through directories. You might want to pipe that through **less**.

```
barry@forensicbox:evvid$ ls -lR | less
.:
total 107
-rwxr-xr-x 1 root root 19536 Aug 24 1996 ARP.EXE*
drwxr-xr-x 3 root root 512 Sep 23 2000 Docs/
-rwxr-xr-x 1 root root 37520 Aug 24 1996 FTP.EXE*
drwxr-xr-x 2 root root 512 Sep 23 2000 Pics/
-r-xr-xr-x 1 root root 16161 Sep 21 2000 loveletter.virus*
-rwxr-xr-x 1 root root 21271 Mar 19 2000 ouchy.dat*
-rwxr-xr-x 1 root root 12384 Aug 2 2000 snoof.gz*

./Docs:
total 57
-rwxr-xr-x 1 root root 17920 Sep 21 2000 Benchmarks.xls*
-rwxr-xr-x 1 root root 2061 Sep 21 2000 Computer_Build.xml*
-rwxr-xr-x 1 root root 32768 Sep 21 2000 Law.doc*
drwxr-xr-x 2 root root 512 Sep 23 2000 Private/
-rwxr-xr-x 1 root root 3928 Sep 21 2000 whyhack*

./Docs/Private:
total 0

./Pics:
total 1130
-rwxr-xr-x 1 root root 94426 Mar 19 2000 C800x600.jpg*
-rwxr-xr-x 1 root root 243245 Sep 21 2000 Stoppie.gif*
-rwxr-xr-x 1 root root 183654 Sep 21 2000 bike2.jpg*
-rwxr-xr-x 1 root root 187598 Sep 21 2000 bike3.jpg*
-rwxr-xr-x 1 root root 27990 Sep 21 2000 matrixs3.jpg*
-rwxr-xr-x 1 root root 418582 Sep 21 2000 mulewheelie.gif*
```

We are looking at files on a FAT partition using Linux tools. Things like permissions can be a little misleading depending on the file system. This is where some of our more advanced forensic tools come in later.

Use the space bar to scroll through the recursive list of files. Remember that the letter q will quit a paging (**less**) session.

One important step in any analysis is verifying the integrity of your data both before after the analysis is complete. We've already covered integrity checks on disks and images. The same commands work on individual files. You can get a hash (CRC, MD5, or SHA) of each file in a number of different ways. In this example, we will use the SHA1 hash. We can get an SHA1 hash of an individual file by changing to our evidence directory (/mnt/evvid) and running the following command on one of the files(these commands can be replaced with **md5sum** if you prefer to use the MD5 hash algorithm).

```
barry@forensicbox:evld$ shasum ARP.EXE
49f0405267a653bac165795ee2f8d934fb1650a9  ARP.EXE

barry@forensicbox:evld$ shasum ARP.EXE > /analysis/ARP.sha1.txt

barry@forensicbox:evld$ cat /analysis/ARP.sha1.txt
49f0405267a653bac165795ee2f8d934fb1650a9  ARP.EXE
```

The redirection in the second command, using the `>` allows us to store the signature in the file `~/analysis/ARP.sha1.txt` and use it later on. Having hashes of individual files can serve a number of purposes, including matching the hashes against lists of known bad files (contraband files or malware, for example), or for eliminating known good files from an examination. There are also times you might simply be asked to provide a list of all files on a volume, and including a hash of each file is simply good practice. Doing this as an individual command for each file on a disk would be tedious at best.

We can get a hash of every file on the disk using the **find** command and an option that allows us to execute an additional command on each file found. We can get a very useful list of SHA1 hashes for every file in our mount point by using **find** to identify all the regular files on the file system and run a hash on all those files:

```
barry@forensicbox:evld$ find .-type f -exec shasum {} \; >
~/analysis/sha1.filelist.txt

barry@forensicbox:evld$ cat ~/analysis/sha1.filelist.txt
86082e288fea4a0f5c5ed3c7c40b3e7947afec11  ./Docs/Benchmarks.xls
81e62f9f73633e85b91e7064655b0ed190228108  ./Docs/Computer_Build.xml
0950fb83dd03714d0c15622fa4c5efe719869e48  ./Docs/Law.doc
7a1d5170911a87a74ffff8569f85861bc2d2462d  ./Docs/whyhack
63ddc7bca46f08caa51e1d64a12885e1b4c33cc9  ./Pics/C800x600.jpg
8844614b5c2f90fd9df6f8c8766109573ae1b923  ./Pics/bike2.jpg
4cf18c44023c05fad0de98ed6b669dc4645f130b  ./Pics/bike3.jpg
aeb0151e67ff4dd5c00a19ee351801b5a6f11438  ./Pics/matrixs3.jpg
d252ac06995c1a6215ca5e7df7c3e02c79c24488  ./Pics/mulewheelie.gif
f6f8586ee5fb5f163eac2bd8ec09053d70cae000e  ./Pics/Stoppie.gif
49f0405267a653bac165795ee2f8d934fb1650a9  ./ARP.EXE
9a886c8e8ad376fc53d6398cdcf8aab9e93eda27  ./FTP.EXE
4c703ee9802aa110b0673d7ae80468e6418bf74c  ./loveletter.virus
7191c24f0f15cca6a5ef9a4db0aee7b40789d6c0  ./ouchy.dat
6666d9b50508360f4a2362e7fd74c91fcb68d2e8  ./snoof.gz
```

This command says “find, starting in the current directory (signified by the `."`), any regular file (**-type f**) and execute (**-exec**) the command **shasum** on all files found (**{}**). Redirect the output to `sha.filelist.txt` in the `~/analysis` directory (where we are storing all of our evidence files). The `\;` is an escape sequence that ends the **-exec** command. The result is a

list of files from our analysis mount point and their SHA hashes. Again, you can substitute the **md5sum** command if you prefer.

We then look at the hashes by using the **cat** command to stream the file to standard output (in this case, our terminal screen), as in the second command above.

You can also do your verification (or hash matching) using a hash list created with one of our hashing programs (**shasum**, **md5sum**, etc.), you can use the **-c** option. If the files match those in the hash list, the command will return **OK**. Making sure you are in a directory where the relative paths provided in the list will target the correct files, use the following command:

```
barry@forensicbox:evid$ shasum -c ~/analysis/sha1.filelist.txt
./Docs/Benchmarks.xls: OK
./Docs/Computer_Build.xml: OK
./Docs/Law.doc: OK
./Docs/whyhack: OK
./Pics/C800x600.jpg: OK
./Pics/bike2.jpg: OK
./Pics/bike3.jpg: OK
./Pics/matrixs3.jpg: OK
./Pics/mulewheelie.gif: OK
./Pics/Stoppie.gif: OK
./ARP.EXE: OK
./FTP.EXE: OK
./loveletter.virus: OK
./ouchy.dat: OK
./snoof.gz: OK
```

Again, the SHA hashes in the file will be compared with SHA sums taken from the mount point. If anything has changed, the program will give a **FAILED** message. If there are failed hashes, you will get a message summarizing the number of failures at the bottom of the output. This is the fastest way to verify hashes. Note that the filenames start with **./** this indicates a relative path. Meaning that we must be in the same relative directory when we check the hashes, since that's where the command will look for the files.

Let's get creative. Take the **ls** command we used earlier and redirect the output to your **~/analysis** directory. With that you will have a list of all the files and their owners and permissions on the subject file system. Check the man page for various uses and options. For example, you could use the **-i** option to include the inode in the list (for Linux file systems), the **-t** option can be used so that the output will include and sort by modification time.

```
barry@forensicbox:evid$ ls -lRt
```

You could also get a list of the files, one per line, using the **find** command (with **-type f**) and redirecting the output to another file.

```
barry@forensicbox:evvid$ find .-type f > ~/analysis/find.filelist.txt
```

Or a list of just directories (**-type d**)

```
barry@forensicbox:evvid$ find . -type d > ~/analysis/find.dirlist.txt
```

There is also the **tree** command, which prints a recursive listing that is more visual...It indents the entries by directory depth and colorizes the filenames (if the terminal is correctly set).

```
barry@forensicbox:evvid$ tree
```

```
.
|-- ARP.EXE
|-- Docs
|   |-- Benchmarks.xls
|   |-- Computer_Build.xml
|   |-- Law.doc
|   |-- Private
|   '-- whyhack
|-- FTP.EXE
|-- Pics
|   |-- C800x600.jpg
|   |-- Stoppie.gif
|   |-- bike2.jpg
|   |-- bike3.jpg
|   |-- matrixs3.jpg
|   '-- mulewheelie.gif
|-- loveletter.virus
|-- ouchy.dat
'-- snoof.gz
```

3 directories, 15 files

Have a look at the above commands, and compare their output. Which do you like better? Remember the syntax assumes you are issuing the command from the `/mnt/evvid` directory (look at your prompt, or use **pwd** if you don't know where you are). The **find** command is especially powerful for searching for files of a specific date stamp or size (or upper and lower limits).

You can also use the **grep** command on any of the lists created by the commands above for any strings or extensions you want to look for.

```
barry@forensicbox:evvid$ grep -i .jpg ~/analysis/find.filelist.txt
./Pics/C800x600.jpg
```

```
./Pics/bike2.jpg  
./Pics/bike3.jpg  
./Pics/matrixs3.jpg
```

This command looks for the pattern `.jpg` in the list of files, using the filename extension to alert us to a JPEG file. The `-i` makes the **grep** command case insensitive. Once you get a better handle on **grep**, you can make your searches far more targeted. For example, specifying strings at the beginning or end of a line (like file extensions) using `^` or `$`. **man grep** has a whole section on these regular expression terms.

7.13.3 Making a List of File Types

What if you are looking for JPEGs but the name of the file has been changed, or the extension is wrong? You can also run the **file** command on each file and see what it might contain. As we saw in earlier sections when looking at file systems, the **file** command compares each file's header (the first few bytes of a raw file - or whatever defines a file) with the contents of the "magic" file. It then outputs a description of the file.

Remember our use of the **find** command's `-exec` option with **shasum**? Let's do the same thing with **file**:

```
barry@forensicbox:evid$ find . -type f -exec file {} \; >  
~/analysis/filetype.txt
```

This creates a text file with the output of the **file** command for each file that the **find** command returns. The text file is in `~/analysis/filetype.txt`. View the resulting list with **cat** (or **less**). The file entries are separated below for readability:

```
barry@forensicbox:evid$ cat ~/analysis/filetype.txt  
./Docs/Benchmarks.xls: Composite Document File V2 Document, Little Endian,  
Os: Windows, Version 4.10, Code page: 1252, Author: Barry J. Grundy, Last  
Saved By: Barry J. Grundy, Name of Creating Application: Microsoft Excel,  
Create Time/Date: Sat Jan 9 19:53:35 1999, Security: 0  
  
./Docs/Computer_Build.xml: gzip compressed data, from Unix, original size 39880  
  
./Docs/Law.doc: Composite Document File V2 Document, Little Endian,  
Os: Windows, Version 4.0, Code page: 1252, Title: The Long Arm of the Law,  
Author: OAG, Template: Normal.dot, Last Saved By: OAG, Revision Number: 2,  
Name of Creating Application: Microsoft Word 8.0, Total Editing Time: 01:00,  
Create Time/Date: Thu Sep 21 13:16:00 2000, Last Saved Time/Date:  
Thu Sep 21 13:16:00 2000, Number of Pages: 1, Number of Words: 1335,  
Number of Characters: 7610, Security: 0
```

./Docs/whyhack: ASCII text, with very long lines, with CRLF, LF line terminators

...

If you are looking for images in particular, then use **grep** to specify that. The following command would look for the string 'image' using the **grep** command on the file /root/evid/filetype.list

```
barry@forensicbox:evid$ grep 'image' ~/analysis/filetype.txt
./Pics/C800x600.jpg: JPEG image data, JFIF standard 1.02, resolution
(DPI), density 80x80, segment length 16, comment: "File written by
Adobe Photoshop 5.0", progressive, precision 8, 800x600, components 3

./Pics/matrixs3.jpg: JPEG image data, JFIF standard 1.01, aspect ratio, density 1x1
  ↪ , segment length 16, baseline, precision 8, 483x354, components 3

./Pics/Stoppie.gif: GIF image data, version 87a, 1024 x 693

./ouchy.dat: JPEG image data, JFIF standard 1.02, resolution (DPI),
density 74x74, segment length 16, comment: "File written by Adobe
Photoshop 5.0", baseline, precision 8, 440x297, components 3
```

The file `ouchy.dat` does not have the proper extension, but it is still identified as a JPEG image. Also note that some of the images above do not show up in our **grep** list because their descriptions do not contain the word "image". There are two Windows Bitmap images that have `.jpg` extensions that do not end up in the **grep** list. Be aware of this when using the **file** command.

When using **grep** on this list you are looking for strings in the description, not actual file types. Pay attention to the difference.

7.13.4 Viewing Files

For text files, as we've covered, you can use **cat**, **more**, or **less** to view the contents.

cat filename

more filename

less filename

Be aware that if the output is not standard text, then you might corrupt the terminal output (type **reset** or **stty sane** at the prompt and it should clear up). Using the **file** command will give you a good idea of which files will be view-able and what program might best be

used to view the contents of a file. For example, Microsoft Office documents can be opened under Linux using programs like OpenOffice, **catdoc** or **catdocx**.

Perhaps a better alternative for viewing unknown files would be to use the **strings** command. This command can be used to parse regular ASCII text out of any file. It's good for formatted documents, non XML compressed data files (older MS Office files, etc.) and even binaries (unidentified executable files, for example), which might have interesting text strings hidden in them. It might be best to pipe the output through **less**.

Have a look at the mounted image on **/mnt/evid**. There is a file called **ARP.EXE**. What does this file do? We can't execute it, and from using the **file** command we know that it's a DOS/Windows executable. Run the following command (again, assuming you are in the **/mnt/evid** directory) and scroll through the output. Do you find anything of interest (hint: like a usage message)?

```
barry@forensicbox:evid$ strings ARP.EXE | less
!This program cannot be run in DOS mode.
.text
'.data
.rsrc
@.reloc
WSOCK32.dll
CRTDLL.dll
KERNEL32.dll
NTDLL.DLLoutput
...
Displays and modifies the IP-to-Physical address translation tables used by
address resolution protocol (ARP).
ARP -s inet_addr eth_addr [if_addr]
ARP -d inet_addr [if_addr]
ARP -a [inet_addr] [-N if_addr]
    -a          Displays current ARP entries by interrogating the current
                  protocol data. If inet_addr is specified, the IP and Physical
                  addresses for only the specified computer are displayed. If
                  more than one network interface uses ARP, entries for each ARP
                  table are displayed.
    -g          Same as -a.
    inet_addr   Specifies an internet address.
...
```

Using **strings** can help us potentially identify the purpose of an executable in some cases.

Viewing images (picture files) from your evidence mount point can be done on the command line with the **xv** or **display** commands (assuming you are in an X window session). **xv** is installed by default in most modern Linux distributions. Have a look at the **ouchy.dat** file in the root of your **/mnt/evid** mount point. We can see it is a picture file, even though the

extension is wrong by using the **file** command. Without leaving the command line, we can view the file using **xv**:

```
barry@forensicbox:evid$ file ouchy.dat
ouchy.dat: JPEG image data, JFIF standard 1.02, resolution (DPI),
density 74x74, segment length 16, comment: "File written by Adobe
Photoshop 5.0", baseline, precision 8, 440x297, components 3

barry@forensicbox:evid$ xv ouchy.dat
```

Figure 6: Viewing an image with a mis-matched extension



Close the image with your mouse, or use the **<ctrl-c>** key combo from the command line to kill the program.

One neat trick you can do if you have a handful of picture files in a directory you want to view without having to use a separate command for each is to use a **bash** loop. Scripting and **bash** programming are outside the scope of this document (for now), but this is a very simple loop that illustrates some more powerful command line usage. This can be done all on one line, but separating the individual commands with the **<enter>** key makes it a bit more readable.²³

First, let's **cd** into the **Pics/** directory under **/mnt/evid**, do a quick **ls** and see that we have a small directory with a few picture files (you can check this with **file ***). We then type our loop:

```
barry@forensicbox:evid$ cd Pics
```

²³There are, of course, image viewing programs that let you browse thumbnail images, like **ranger**, but we cover this loop for educational purposes

```
barry@forensicbox:Pics$ ls
C800x600.jpg*  Stoppie.gif*  bike2.jpg*
bike3.jpg*    matrixs3.jpg*  mulewheelie.gif*

barry@forensicbox:Pics$ for pic in ./* <enter>
> do <enter>
> xv $pic <enter>
> done <enter>
```

The first line of a bash loop above means “for every file in the current directory (`./*`), assign each file the variable name `pic` as we move through the loop”. The second line is simply the bash keyword `do`. The third line executes `xv` on the value of the `pic` variable (`$pic`) at each iteration of the loop, followed by the bash keyword `done` to close the loop. As you run the loop, each image will display, and the loop will pause until you close `xv`. When you close `xv` the loop continues until all the values of `\$pic` are exhausted (all the files in the directory) and the loop exits. Learn to do this and I promise you will find it useful almost daily.

If you are currently running the X window system, you can use any of the graphics tools that come standard with whichever Linux distribution you are using. **geeqie** is one graphics tool for the XFCE desktop that will display graphic files in a directory. Experiment a little. Other tools, such as **gthumb** for Gnome and Konqueror from the KDE desktop have a feature that will create a very nice html image gallery for you from all images in a directory.

Once you are finished exploring, be sure to unmount the loop mounted disk image. Again, make sure you are not anywhere in the mount point (using that directory in another terminal session) when you try to unmount, or you will get the “busy” error. The following commands will take you back to your home directory (`cd` without arguments takes you to your home directory automatically). We `su` to root, and unmount the loop mounted file system.

```
barry@forensicbox:Pics$ cd

barry@forensicbox:~$ su -
Password:

root@forensicbox:~# umount /mnt/evid

root@forensicbox:~# exit

barry@forensicbox:~$
```

7.13.5 Searching All Areas of the Forensic Image for Text

Now let's go back to the original image. The loop mounted disk image allowed you to check all the files and directories using a logical view of the file system. What about unallocated and slack space (physical view)? We will now analyze the image itself, since it was a bit for bit copy and includes data in the unallocated areas of the disk. We'll do this using rudimentary Linux tools.

Let's assume that we have seized this image from media used by a former employee of a large corporation. The would-be cracker sent a letter to the corporation threatening to unleash a virus in their network. The suspect denies sending the letter. This is a simple matter of finding the text from a deleted file (unallocated space).

First, change back to the directory where you saved the image file `fat_fs.raw`. In this case, the file is in the home directory (which you can see is the present working directory by both the `~` in the prompt, and the output of the `pwd` command).

```
barry@forensicbox:~$ pwd
/home/barry

barry@forensicbox:~$ ls
Desktop/  Downloads/  analysis/  fat_fs.raw*
```

Now we will use the `grep` command to search the image for any instance of an expression or pattern. We will use a number of options to make the output of `grep` more useful. The syntax of `grep` is normally:

grep [-options] <pattern> <file-to-search>

The first thing we will do is create a list of keywords to search for. It's rare we ever want to search evidence for a single keyword, after all. For our example, let's use "ransom", "\$50,000", and "unleash a virus". These are some keywords and a phrase that we have decided to use from the original letter received by the fictitious corporation. Make the list of keywords (using `vi` of course!) and save it as `~/analysis/searchlist.txt`. Ensure that each string you want to search for is on a different line.

```
$50,000 ransom unleash a virus
```

Make sure there are NO BLANK LINES IN THE LIST OR AT THE END OF THE LIST!!
Now we run the `grep` command on our image:

```
barry@forensicbox:~$ grep -abif analysis/searchlist.txt fat_fs.raw >
analysis/hits.txt
```

We are asking **grep** to use the list we created in `./analysis/searchlist.txt` for the patterns we are looking for. This is specified with the **-f <file>** option. We are telling **grep** to search `fat_fs.raw` for these patterns, and redirect the output to a file called `hits.txt` in the `./analysis` directory, so we can record the output. The **-a** option tells **grep** to process the file as if it were text, even if it's binary. The option **-i** tells **grep** to ignore upper and lower case. And the **-b** option tells **grep** to give us the byte offset of each hit so we can find the line in **xxd** (our command line hex viewer). Earlier we mentioned the **grep** manual page and the section it has on regular expressions. Please take the time to read through it and experiment.

Once you run the command above, you should have a new file in your `analysis` directory called `hits.txt`. View this file with **less** or any text viewer. Keep in mind that **strings** might be best for the job. Again, if you use **less**, you run the risk of corrupting your terminal if there are non-ASCII characters. We will simply use **cat** to stream the entire contents of the file to the standard output. The file `hits.txt` should give you a list of lines that contain the words in your `searchlist.txt` file. In front of each line is a number that represents the byte offset for that "hit" in the image file. For illustration purposes, the search terms are underlined, and the byte offsets are bold in the output below:

```
barry@forensicbox:~$ cat analysis/hits.txt
75441:you and your entire business ransom.
75500:I have had enough of your mindless corporate piracy and will
no longer stand for it. You will recieve another letter next week. It will have
a single bank account number and bank name. I want you to deposit $50,000
in the account the day you receive the letter.
75767:Don't try anything, and dont contact the cops. If you do,
I will unleash a virus that will bring down your whole network and destroy
your consumer's confidence.
```

In keeping with our command line philosophy, we will use **xxd** to display the data found at each byte offset. **xxd** is a command line hex dump tool, useful for examining files. Do this for each offset in the list of hits. The **-s** option to **xxd** is so we can "seek" into the file the specified number of bytes. This should yield some interesting results if you scroll above and below the offsets. Here we'll use **xxd** and seek to the first hit at byte offset 75441 with the **-s** option. We'll pipe the output to the **head** command, which will show us the first 10 lines of output. You can view more of the output by piping through **less** instead.

```
barry@forensicbox:~$ xxd -s 75441 fat_fs.raw | head
000126b1: 796f 7520 616e 6420 796f 7572 2065 6e74  you and your ent
000126c1: 6972 6520 6275 7369 6e65 7373 2072 616e  ire business ran
000126d1: 736f 6d2e 0a0a 5468 6973 2069 7320 6e6f  som...This is no
000126e1: 7420 6120 6a6f 6b65 2e0a 0a49 2068 6176  t a joke...I hav
000126f1: 6520 6861 6420 656e 6f75 6768 206f 6620  e had enough of
00012701: 796f 7572 206d 696e 646c 6573 7320 636f  your mindless co
00012711: 7270 6f72 6174 6520 7069 7261 6379 2061  rporate piracy a
```

```
00012721: 6e64 2077 696c 6c20 6e6f 206c 6f6e 6765  nd will no longe
00012731: 7220 7374 616e 6420 666f 7220 6974 2e20  r stand for it.
00012741: 596f 7520 7769 6c6c 2072 6563 6965 7665  You will recieve
```

Please note that the use of **grep** in this manner is fairly limited. There are character sets that the common versions of **grep** (and **strings** as well) do not support. So doing a physical search for a string on an image file is really only useful for what it does show you. In other words, negative results for a **grep** search of an image can be misleading. The strings or keywords may exist in the image in a form not recognizable to **grep** or **strings**. There are tools and methods that address this, and we will discuss some of them later.

In addition to the structure of the images and the issues of image sizes, we also have to be concerned with memory usage and our tools. You might find that **grep**, when used as illustrated in small image analysis example, might not work as expected with larger images and could exit with an error similar to:

```
grep: memory exhausted
```

The most apparent cause for this is that **grep** does its searches line by line. When you are “grepping” a large disk image terabytes in size, you might find that you have a huge number of bytes to read through before **grep** comes across a newline character. What if **grep** had to read several gigabytes of data before coming across a newline? It would “exhaust” itself (the input buffer fills up). There are many variables that will affect this, and the causes are actually far more complex.

One potential solution is to force-feed **grep** some newlines. In our example analysis we are “grepping” for text. We are not concerned with non-text characters at all. If we could take the input stream to **grep** and change the non-text characters to newlines, in most cases **grep** would have no problem. Note that changing the input stream to **grep** does not change the image itself. Also, remember that we are still looking for a byte offset. Luckily, the character sizes remain the same, and so the offset does not change as we feed newlines into the stream (simply replacing one character with another).

Let’s say we want to take all of the control characters streaming into **grep** from the disk image and change them to newlines. We can use the translate command, **tr**, to accomplish this. Check out **man tr** for more information about this powerful command:

```
barry@forensicbox:~$ tr '[:cntrl:]' '\n' < fat_fs.raw | grep -abif
analysis/searchlist.txt
75441:you and your entire business ransom.
75500:I have had enough of your mindless corporate piracy and will
no longer stand for it. You will recieve another letter next week.
It will have a single bank account number and bank name. I want you
to deposit $50,000 in the account the day you receive the letter.
75767:Don't try anything, and dont contact the cops. If you do, I will
unleash a virus that will bring down your whole network and destroy
```

your consumer's confidence.

This command would read: "Translate all the characters contained in the set of control characters [:cntrl:] to newlines \n. Take the input to **tr** from **fat_fs.raw** (we are re-directing in the opposite direction this time) and pipe the output to **grep**, and then to **head**. This effectively changes the stream before it gets to **grep**. Notice the output does not change. The translation occurs in the stream, and it's a character for character swap.

This is only one of many possible problems you could come across. The point here is that when issues such as these arise, you need to be familiar enough with the tools provided to be able to understand why such errors might have been produced, and how you can get around them. Remember, the shell tools and the GNU software that accompany a Linux distribution are extremely powerful, and are capable of tackling nearly any task. Where the standard shell fails, you might look at perl or python as options. These subjects are outside of the scope of the current presentation, but are introduced as fodder for further experimentation.

Be sure to unmount the image when you are finished:

```
barry@forensicbox:~$ su -
Password:

root@forensicbox:~# umount /mnt/evid

root@forensicbox:~# exit

barry@forensicbox:~$
```

8 Advanced (Beginner) Forensics

The following sections are more advanced and detailed. New tools are introduced to help round out some of your knowledge and provide a more solid footing on the capabilities of the GNU/Linux command line. The topics are still at the beginner level, but you should be at least somewhat comfortable with the commands demonstrated here before tackling the exercises. Although I've included the commands and much of the output for those who are reading this without the benefit of a Linux computer at hand, it is important that you follow along on your own system as we go through the practical exercises. Hands on experimentation is the best way to learn.

8.1 Manipulating and Parsing Files

Let's dig a little deeper into the command line. Often there are arguments made about the usefulness of the command line interface (CLI) versus a GUI tool for analysis. I would argue that in the case of large sets of structured data, the CLI can be faster and more flexible than many GUI tools available today when it comes to parsing.

As an example, we will look at a set of log files from a single Unix system. We are not going to analyze them for any sort of evidentiary data. The point here is to illustrate the ability of commands through the CLI to organize and parse data by using pipes to string a series of commands together and obtain the desired output. Follow along with the example, and keep in mind that to get anywhere near proficient with this will require a great deal of reading and practice. The payoff is enormous.

Create a directory called `Logs` and download the file `logs.v3.tar.gz` into that directory:

```
barry@forensicbox:~$    mkdir Logs

barry@forensicbox:~$    cd Logs

barry@forensicbox:Logs$  wget https://www.linuxleo.com/Files/logs.v3.tar.gz
--2019-07-22 08:53:46--  https://www.linuxleo.com/Files/logs.v3.tar.gz
Resolving www.linuxleo.com... 74.208.236.144
Connecting to www.linuxleo.com|74.208.236.144|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5144 (5.0K) [application/gzip]
Saving to: 'logs.v3.tar.gz'

logs.v3.tar.gz      100%[=====>]    5.02K  --.-KB/s    in 0s

2019-07-22 08:53:47 (1.15 GB/s) - 'logs.v3.tar.gz' saved [5144/5144]
```

Once the file is downloaded, check the hash and use the **tar** command to list the contents. Our command below shows that the files in the archive will extract directly to our current directory. There are 5 messages logs.

```
barry@forensicbox:Logs$ sha1sum logs.v3.tar.gz
a66bc61628af6eab8cef780e4c3f60edcedbcf12 logs.v3.tar.gz

barry@forensicbox:Logs$ tar tzvflogs.v3.tar.gz
-rw-r--r-- root/root      8282 2003-10-29 12:45 messages
-rw----- root/root      8302 2003-10-29 16:17 messages.1
-rw----- root/root      8293 2003-10-29 16:19 messages.2
-rw----- root/root      4694 2003-10-29 16:23 messages.3
-rw----- root/root      1215 2003-10-29 16:23 messages.4
```

The **messages** logs contain entries from a variety of sources, including the kernel and other applications. The numbered files result from log rotation. As the logs are filled, they are rotated and eventually deleted. On most Unix systems, the logs are found in **/var/log/** or **/var/adm**. These are from a very old system, but again it's not the contents we are interested in here, it's using the tools.

Extract the logs:

```
barry@forensicbox: Logs$ tar xzvf logs.v3.tar.gz
messages
messages.1
messages.2
messages.3
messages.4
```

Instead of listing the contents with the **t option**, we are extracting it with the **x option**. All the other options remain the same.

Let's have a look at one log entry. We pipe the output of **cat** to the command **head -n 1** so that we only get the 1st line (recall that **head** without additional arguments will give the first 10 lines):

```
barry@forensicbox:Logs$ cat messages | head -n 1
Nov 17 04:02:14 hostname123 syslogd 1.4.1: restart.
```

Each line in the log files begin with a date and time stamp. Next comes the host name followed by the name of the application that generated the log message. Finally, the actual message is printed.

For the sake of our exercise, let's assume these logs are from a victim system, and we want to analyze them and parse out the useful information. We are not going to worry about

what we are actually seeing here (there's nothing nefarious in these logs), our objective is to understand how to boil the information down to something useful.

First of all, rather than parsing each file individually, let's try and analyze all the logs at one time. They are all in the same format, and essentially they comprise one large log. We can use the **cat** command to add all the files together and send them to standard output. If we work on that data stream, then we are essentially making one large log out of all five logs. Can you see a potential problem with this?

```
barry@forensicbox:Logs$ cat messages*| less
Nov 17 04:02:14 hostname123 syslogd 1.4.1: restart.
Nov 17 04:05:46 hostname123 su(pam_unix)[19307]: session opened for user news
Nov 17 04:05:47 hostname123 su(pam_unix)[19307]: session closed for user news
...
Nov 23 18:27:58 hostname123 kernel: hda: hda1 hda2 hda3 hda4 < hda5 hda6 hda7 >
Nov 23 18:27:00 hostname123 rc.sysinit: Mounting proc filesystem: succeeded
Nov 10 04:02:08 hostname123 syslogd 1.4.1: restart.
Nov 10 04:05:55 hostname123 su(pam_unix)[15181]: session opened for user news
Nov 10 04:05:55 hostname123 su(pam_unix)[15181]: session closed for user news
Nov 11 04:06:09 hostname123 su(pam_unix)[32640]: session opened for user news
...
```

If you look at the output (scroll using **less**), you will see that the dates ascend and then jump to an earlier date and then start to ascend again. This is because the later log entries are added to the bottom of each file, so as the files are added together, the dates appear to be out of order. What we really want to do is stream each file backwards so that they get added together with the most recent date in each file at the top instead of at the bottom. In this way, when the files are added together they are in order. In order to accomplish this, we use **tac** (yes, that's **cat** backwards).

```
barry@forensicbox:Logs$ tac messages*| less
Nov 23 18:27:00 hostname123 rc.sysinit: Mounting proc filesystem: succeeded
Nov 23 18:27:58 hostname123 kernel: hda: hda1 hda2 hda3 hda4 < hda5 hda6 hda7 >
Nov 23 18:27:58 hostname123 kernel: Partition check:
Nov 23 18:27:58 hostname123 kernel: ide-floppy driver 0.99.newide
...
```

Beautiful. The dates are now in order. We can now work on the stream of log entries as if they were one large (in order) file. We will continue to work with this **tac** command to create our in-order stream with each command. We could redirect to another single log file that contains all the logs, but there's no need to right now and creating one large log file serves no real purpose.

First, let's gather some information. We might want to know, perhaps for our notes, how many entries are in each file, and how many entries total. Here's a quick way of doing that

from the command line:

```
barry@forensicbox:Logs$  tac messages*| wc -l
374
```

The same command is used to stream all the files together and send the output through the pipe to the **wc** command ("word count"). The **-l** option specifies that we want to count just lines instead of the default output of lines, words and bytes. To get a count for all the files and the total at the same time, use **wc -l** on all the **messages** files at one time:

```
barry@forensicbox:Logs$  wc -l messages*
100 messages
109 messages.1
100 messages.2
 50 messages.3
 15 messages.4
374 total
```

Now we will introduce a new command, **awk** to help us view specific fields from the log entries. In this case we will view the dates. **awk** is an extremely powerful command. The version most often found on Linux systems is **gawk** (GNU **awk**). While we are going to use it as a stand-alone command, **awk** is actually a programming language on its own, and can be used to write scripts for organizing data. Our concentration will be centered on the **awk print** function. See **man awk** for more details.

Sets of repetitive data can often be divided into columns or "fields", depending on the structure of the file. In this case, the fields in the log files are separated by simple white space (the **awk** default field separator). The date is comprised of the first two fields (month and day). So let's have a look at **awk** in action:

```
barry@forensicbox:Logs$  tac messages*| awk '{print $1" "$2}' | less
Nov 23
Nov 23
Nov 23
...
Oct 20
Oct 20
Oct 20
...
```

This command will stream all the log files (each one from bottom to top) and send the output to **awk** which will print the first field, **\$1** (month), followed by a space (" "), followed by the second field, **\$2** (day). This shows just the month and day for every entry. Suppose I just want to see one of each date when an entry was made. I don't need to see repeating dates. I ask to see one of each unique line of output with **uniq**:

```
barry@forensicbox:Logs$    tac messages*| awk '{print $1" "$2}' | uniq | less
Nov 23
Nov 22
Nov 21
...
Oct 22
Oct 21
Oct 20
```

This removes repeated dates, and shows me just those dates with log activity.

CLI Hint: Instead of re-typing the command each time, use the up arrow on your keyboard to scroll through older commands (part of the command history of **bash**). Hit the up arrow once, and you can edit your last command. Very useful when adjusting commands for this sort of parsing.

If a particular date is of interest, I can **grep** the logs for that particular date (note there are *2 spaces* between **Nov** and **4**, one space will not work in our **grep** command):

```
barry@forensicbox:Logs$    tac messages*| grep "Nov 4"
Nov  4 17:41:27 hostname123 sshd(pam_unix)[27630]: session closed for user root
Nov  4 17:41:27 hostname123 sshd[27630]: Received disconnect from 1xx.183.221.214:
    ↪ 11: Disconnect requested by Windows SSH Client.
Nov  4 17:13:07 hostname123 sshd(pam_unix)[27630]: session opened for user root by
    ↪ (uid=0)
Nov  4 17:13:07 hostname123 sshd[27630]: Accepted password for root from 1xx
    ↪ .183.221.214 port 1762 ssh2
Nov  4 17:08:23 hostname123 sshd(pam_unix)[27479]: session closed for user root
...
```

Of course, we have to keep in mind that this would give us any lines where the string **Nov 4** resided, not just in the date field. To be more explicit, we could say that we only want lines that *start* with **Nov 4**, using the **^** (in our case, this gives essentially the same output):

```
barry@forensicbox:Logs$    tac messages*| grep ^"Nov 4" | less
Nov  4 17:41:27 hostname123 sshd(pam_unix)[27630]: session closed for user root
Nov  4 17:41:27 hostname123 sshd[27630]: Received disconnect from 1xx.183.221.214:
    ↪ 11: Disconnect requested by Windows SSH Client.
Nov  4 17:13:07 hostname123 sshd(pam_unix)[27630]: session opened for user root by
    ↪ (uid=0)
Nov  4 17:13:07 hostname123 sshd[27630]: Accepted password for root from 1xx
    ↪ .183.221.214 port 1762 ssh2
Nov  4 17:08:23 hostname123 sshd(pam_unix)[27479]: session closed for user root
...
```

Also, if we don't know that there are two spaces between **Nov** and **4**, we can tell **grep** to look for any number of spaces between the two:

```
barry@forensicbox:Logs$    tac messages*| grep ^"Nov[ ]*4" | less
Nov  4 17:41:27 hostname123 sshd(pam_unix)[27630]: session closed for user root
Nov  4 17:41:27 hostname123 sshd[27630]: Received disconnect from 1xx.183.221.214:
    ↪ 11: Disconnect requested by Windows SSH Client.
...
```

The above **grep** expression translates to "Lines starting (^) with the string **Nov** followed by zero or more (*) of the preceding characters that are between the brackets ([] - in this case, a space) followed by a 4". Obviously, this is a complex issue. Knowing how to use regular expression will give you huge flexibility in sorting through and organizing large sets of data. As mentioned earlier, read **man grep** for a good primer on regular expressions.

As we look through the log files, we may come across entries that appear suspect. Perhaps we need to gather all the entries that we see containing the string
Did not receive identification string from <IP> for further analysis.

```
barry@forensicbox:Logs$    tac messages*| grep "identification string"
Nov 22 23:48:47 hostname123 sshd[19380]: Did not receive identification string from
    ↪ 19x.xx9.220.35
Nov 22 23:48:47 hostname123 sshd[19379]: Did not receive identification string from
    ↪ 19x.xx9.220.35
Nov 20 14:13:11 hostname123 sshd[29854]: Did not receive identification string from
    ↪ 200.xx.114.131
...
```

How many of these entries are there?

```
barry@forensicbox:Logs$    tac messages*| grep "identification string" | wc -l
35
```

There are 35 such entries. Now we just want the date (fields 1 and 2), the time (field 3) and the remote IP address that generated the log entry. The IP address is the last field. Rather than count each word in the entry to get to the field number of the IP, we can simply use the variable **\$NF**, which means "number of fields". Since the IP is the last field, its field number is equal to the number of fields:

```
barry@forensicbox:Logs$    tac messages*| grep "identification string" | awk
' {print $1" "$2" "$3" "$NF}'
Nov 22 23:48:47 19x.xx9.220.35
Nov 22 23:48:47 19x.xx9.220.35
Nov 20 14:13:11 200.xx.114.131
```

Nov 18 18:55:06 6x.x2.248.243

...

We can add some tabs (`\t`) in place of spaces in our `awk` command to make the output more readable (this assumes fixed string length). The following command will place a tab character between the date and the time, and between the time and the IP address:

```
barry@forensicbox:Logs$ tac messages* | grep "identification string" | awk
'{print $1" "$2"\t"$3"\t"$NF}'
Nov 22 23:48:47 19x.xx9.220.35
Nov 22 23:48:47 19x.xx9.220.35
Nov 20 14:13:11 200.xx.114.131
Nov 18 18:55:06 6x.x2.248.243
...
```

This can all be redirected to an analysis log or text file for easy addition to a report. Remember that `> report.txt` creates the report file (overwriting anything there previously), while `>> report.txt` appends to it. You can use `su` to become `root` and set the "append only" attribute on your report file to prevent accidental overwrites²⁴.

The following commands are typed on one line each:

```
barry@forensicbox:Logs$ echo "Localhost123: Log entries from /var/log/messages"
> report.txt

barry@forensicbox:Logs$ echo "\"Did not receive identification string\":" >>
report.txt

barry@forensicbox:Logs$ tac messages* | grep "identification string" | awk
'{print $1" "$2"\t"$3"\t"$NF}' >> report.txt

barry@forensicbox:Logs$ cat report.txt
Localhost123: Log entries from /var/log/messages "Did not receive identification
↪ string":
Nov 22 23:48:47 19x.xx9.220.35
Nov 22 23:48:47 19x.xx9.220.35
Nov 20 14:13:11 200.xx.114.131
Nov 18 18:55:06 6x.x2.248.243
Nov 17 19:26:43 200.xx.72.129
...
```

We can also get a sorted (`sort`) list of the unique (`-u`) IP addresses involved in the same way:

²⁴We covered this earlier in the guide with the `chattr` command.

```
barry@forensicbox:Logs$ echo "Unique IP addresses: " >> report.txt

barry@forensicbox:Logs$ tac messages*| grep "identification string" | awk
'{print $NF}' >> report.txt

barry@forensicbox:Logs$ cat report.txt
Localhost123: Log entries from /var/log/messages
"Did not receive identification string":
Nov 22 23:48:47 19x.xx9.220.35
Nov 22 23:48:47 19x.xx9.220.35
...
Unique IP addresses:
19x.xx9.220.35
200.xx.114.131
200.xx.72.129
212.xx.13.130
2xx.54.67.197
2xx.71.188.192
2xx.x48.210.129
6x.x2.248.243
6x.x44.180.27
xx.192.39.131
```

The command above prints only the last field (`$NF`) of our **grep** output (which is the IP address). The resulting list of IP addresses can also be fed to a script that does **nslookup** or **whois** database queries.

You can view the resulting report (**report.txt**) using the **less** command.

As with all the exercises in this document, we have just sampled the abilities of the Linux command line. It all seems somewhat convoluted to the beginner. After some practice and experience with different sets of data, you will find that you can glance at a file and say “I want that information”, and be able to write a quick piped command to get what you want in a readable format in a matter of seconds. As with all language skills, the Linux command line “language” (actually bash in this case) is perishable. Keep a good reference handy and remember that you might have to look up syntax a few times before it becomes second nature.

8.2 Fun with **dd**

We’ve already done some simple imaging and wiping using **dd**, let’s explore some other uses for this flexible tool. **dd** is sort of like a little forensic Swiss army knife. It has lots of applications, limited only by your imagination.

8.2.1 Data Carving with **dd**

In this next example, we will use **dd** to carve a JPEG picture file from a chunk of raw data. By itself, this is not a real useful exercise. There are lots of tools out there that will "carve" files from forensic images, including a simple cut and paste from a hex editor. The purpose of this exercise is to help you become more familiar with **dd**. In addition, you will get a chance to use a number of other tools in preparation for the "carving". This will help familiarize you further with the Linux toolbox. First you will need to download the raw data chunk and check it's hash:

```
barry@forensicbox:~$ wget https://www.linuxleo.com/Files/image_carve_2017.raw
...

barry@forensicbox:~$ shasum image_carve_2017.raw
ac3dd14e9a84f8dc5b827ba6262c295d28d3cecc image_carve_2017.raw
```

Have a brief look at the file `image_carve_2017.raw` with your wonderful command line hex-dump tool, **xxd**:

```
barry@forensicbox:~$ xxd image_carve_2017.raw | less
00000000: f0d5 0291 431e 41db 5fb9 abce 7240 4543  ....C.A.....r@EC
00000010: 9a71 389a e0f1 4cf7 bfb4 32e2 6fe9 1132  .q8...L...2.o..2
00000020: fc36 ddca eb48 56c1 1501 bcfd e7dd 2631  .6...HV.....&1
00000030: ffa6 bc3e e7bc ddd4 e986 f222 7198 11a9  ...>....."q...
00000040: ee92 a2a1 56c2 22fc 9838 dff4 5d24 8a56  ....V.."..8..]$.V
00000050: da3d 0a2c a91c e2dd 5095 40fd e43a 1208  .=,....P.@:....
00000060: a76d 997e 9daf f4fa 9218 a2e4 6d81 a8ca  .m.~.....m...
00000070: cdf2 5055 12d5 f703 44bd 8d8b 88ed abab  ..PU....D.....
00000080: 9023 ee54 f4f4 77f5 c89e ffdc 7c1a dba3  .#.T..w.....|...
00000090: 42c7 9f07 902e 08c9 778c 67e3 479b 70f4  B.....w.g.G.p.
...
```

It's really just a file full of random characters. Somewhere inside there is a standard JPEG image. Let's go through the steps needed to recover the picture file using **dd** and other Linux tools. We are going to stick with command line tools available in most default Linux installations.

First we need a plan. How would we go about recovering the file? What are the things we need to know to get the image (picture) out, and only the image? Imagine **dd** as a pair of scissors. We need to know where to put the scissors to start cutting, and we need to know where to stop cutting. Finding the start of the JPEG and the end of the JPEG can tell us this. Once we know where we will start and stop, we can calculate the size of the JPEG. We can then tell **dd** where to start cutting, and how much to cut. The output file will be our JPEG image. Easy, right? So here's our plan, and the tools we'll use:

1. Find the start of the JPEG (**xxd** and **grep**)
2. Find the end of the JPEG (**xxd** and **grep**)
3. Calculate the size of the JPEG in bytes (**bc**)
4. Cut from the calculated start - the calculated number of bytes and output to a file (**dd**)

This exercise starts with the assumption that we are familiar with standard file headers. Since we will be searching for a standard JPEG image within the data chunk, we will start with the stipulation that the JPEG header begins with `0xffd8` (the hexadecimal value `ffd8`) with a six-byte offset to the string `JFIF`. The end of the standard JPEG is marked by hex `ffd9`.

Let's go ahead with step 1: Using **xxd**, we pipe the output of our `image_carve.raw` file to **grep** and look for the start of the JPEG²⁵:

```
barry@forensicbox:~$ xxd image_carve_2017.raw | grep ffd8
0000f900: 901d cfe7 8488 ac23 ffd8 24ab 4f4d 1613 .....#..$.OM..
0001bba0: e798 a4b6 d833 9567 af5f ffd8 e5e9 ed24 .....3.g.....$
00033080: 84a5 aeec d7db ffd8 3c37 c52d a80e 6e7e .....<7.-..n~
00036ac0: 1676 761b e3d4 ffd8 ffe0 0010 4a46 4946 .vv.....JFIF
```

The **grep** command found four lines that contain the potential header of our picture file. We know that we are looking for a JPEG image, and we know that following an additional four bytes after the `0xffd8` we should see the `JFIF` string. The last line of our output shows that, meaning this is the correct match. This is shown in red above.

The start of a standard JPEG file header has been found. The offset (in hex) for the beginning of this line of **xxd** output is `0x00036ac0`. Now we can calculate the byte offset in decimal. For this we will use the **bc** command. As we discussed in an earlier section, **bc** is a command line "calculator", useful for conversions and calculations. It can be used either interactively or take piped input. In this case we will **echo** the hex offset to **bc**, first telling it that the value is in base 16. **bc** will return the decimal value.

```
barry@forensicbox:~$ echo "ibase=16;36AC0" | bc
223936
```

It's important that you use uppercase letters in the hex value. This is NOT the start of the JPEG, just the start of the line in the **xxd** output. The `ffd8` string is actually located another six bytes farther into that line of output (each hex pair is a character value, and

²⁵The perceptiveness you will notice that this is a "perfect world" situation. There are a number of variables that can make this operation more difficult. The **grep** command can be adjusted for many situations using a complex regular expression (outside the scope of this document).

there are six pairs before the `ffd8`). So we add 6 to the start of the line. Our offset is now 223942. We have found and calculated the start of the JPEG image in our data chunk.

```
00036ac0: 16 76 76 1b e3 d4 ffd8 ...
(223936) +1 +2 +3 +4 +5 +6 = 223942 offset to the header start
```

Now it's time to find the end of the file.

Since we already know where the JPEG starts, we will start our search for the end of the file from that point. Again using **xxd** and **grep** we search for the footer value `0xffd9` somewhere after the header:

```
barry@forensicbox:~$ xxd -s 223942 image_carve_2017.raw | grep ffd9
0005d3c6: af29 6ae7 06e1 2e48 38a3 ffd9 8303 a138 .)j....H8.....8
```

The `-s 223942` option to **xxd** specifies where to start searching (since we know this value is the start of the JPEG header, there's no reason to search before it and we eliminate false hits from that region). The output shows the first `0xffd9` on the line at hex offset `0x0005d3c6`. Let's convert that to decimal, again noting the uppercase value in our hex:

```
barry@forensicbox:~$ echo "ibase=16;0005D3C6" | bc
381894
```

Because 381894 is the offset for the start of the line, we need to add 12 to the value to include the `0xffd9` (giving us 381906). We do this because the `0xffd9` needs to be included in our carve, so we skip *past it*.

```
0005d3c6: af 29 6a e7 06 e1 2e 48 38 a3 ff d9
(381894) +1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11 +12 = 381906 offset to header end
```

Now that we know the start and the end of the file, we can calculate the size:

```
barry@forensicbox:~$ echo "381906-223942" | bc
157964
```

We now know the file is 157964 bytes in size, and it starts at byte offset 223942. The carving is the easy part! We will use **dd** with three options:

skip= how far into the data chunk we begin “cutting”. **bs=** (block size) the number of bytes we include as a “block”. **count=** the number of blocks we will be “cutting”.

The input file for the **dd** command is `image_carve_2017.raw`. Obviously, the value of **skip** will be the offset to the start of the JPEG. The easiest way to handle the block size is to

specify it as **bs=1** (meaning one byte) and then setting **count** to the size of the file. The name of the output file is arbitrary.

```
barry@forensicbox:~$ dd if=image_carve_2017.raw of=carved.jpg bs=1 skip=223942
count=157964
157964+0 records in
157964+0 records out
157964 bytes (158 kB, 154 KiB) copied, 0.174065 s, 908 kB/s
```

You should now have a file in your current directory called **carved.jpg**. From the terminal (assuming you are in a GUI environment)²⁶, simply use the **xv** command to view the file (or any other image viewer, like **display**) and see what you've got.

```
barry@forensicbox:~$ xv carved.jpg
```

8.2.2 Carving Partitions with **dd**

Now we can try another useful exercise in carving with **dd**. At times you might find it desirable to separate partitions within a disk image. Remember, you cannot simply mount an entire disk image, only the partitions. We've already learned that we can find the structure of an image and mount the partitions within using tools like **kpartx** and the **loop** device with the **mount** command.

We introduce this technique here not to teach it for practical use (though it may have some limited practical use), but to provide another practical exercise using a number of important command line tools. In any event, for the beginning Linux forensics student, I would still consider this an important skill. It's just good practice for a number of common and useful commands.

The method we will use in this exercise entails identifying the partitions within a raw image with **fdisk** or **gdisk**. We will then use **dd** to carve the partitions out of the image.

We will use the same disk image we used previously (**able_3.00***). If you have not downloaded it already, do so now using **wget**. Then check the hash of the downloaded file. It should match mine here:

```
barry@forensicbox:~$ wget https://www.linuxleo.com/Files/able_3.tar.gz
...

barry@forensicbox:~$ shasum able_3.tar.gz
6d8de5017336028d3c221678b483a81e341a9220 able_3.tar.gz
```

²⁶If you are at a console without a GUI, you can use a program like **fbi**

Check the contents of the tar archive (**tar tzvf**), untar the files (**tar xzvf**), and change into the **able_3** directory with **cd**. You can skip all of this if you already have the **able_3** directory from our previous exercise. Just change into the directory.

```
barry@forensicbox:~$ tar tzvf able_3.tar.gz
drwxr-xr-x barry/users      0 2017-05-25 12:42 able_3/
-rw-r--r-- barry/users 1073741824 2017-05-25 12:13 able_3/able_3.000
-rw-r--r-- barry/users 1073741824 2017-05-25 12:13 able_3/able_3.001
-rw-r--r-- barry/users      1339 2017-05-25 12:14 able_3/able_3.log
-rw-r--r-- barry/users 1073741824 2017-05-25 12:14 able_3/able_3.003
-rw-r--r-- barry/users 1073741824 2017-05-25 12:14 able_3/able_3.002
```

```
barry@forensicbox:~$ tar xzvf able_tar.gz
able_3/
able_3/able_3.000
able_3/able_3.001
able_3/able_3.log
able_3/able_3.003
able_3/able_3.002
```

```
barry@forensicbox:~$ cd able_3
```

Now that we are in the **able_3** directory, we can see that we have our 4 split image files and a log file with the acquisition information. This particular log was created by the **dc3dd** command (we covered earlier). View the log and look at the hashes:

```
barry@forensicbox:able_3$ cat able_3.log
dc3dd 7.2.646 started at 2017-05-25 15:51:04 +0000
compiled options:
command line: dc3dd if=/dev/sda hofs=able_3.000 ofsz=1G hash=sha1 log=able_3.log
device size: 8388608 sectors (probed),    4,294,967,296 bytes
sector size: 512 bytes (probed)
  4294967296 bytes ( 4 G ) copied ( 100% ), 1037.42 s, 3.9 M/s
  4294967296 bytes ( 4 G ) hashed ( 100% ), 506.481 s, 8.1 M/s

input results for device '/dev/sda':
  8388608 sectors in
  0 bad sectors replaced by zeros
  2eddbfe3d00cc7376172ec320df88f61afda3502 (sha1)
    4ef834ce95ec545722370ace5a5738865d45df9e, sectors 0 - 2097151
    ca848143cca181b112b82c3d20acde6bdaf37506, sectors 2097152 - 4194303
    3d63f2724304205b6f7fe5cadcbc39c05f18cf30, sectors 4194304 - 6291455
    9e8607df22e24750df7d35549d205c3bd69adfe3, sectors 6291456 - 8388607

output results for files 'able_3.000':
  8388608 sectors out
```

```
[ok] 2eddbfe3d00cc7376172ec320df88f61afda3502 (sha1)
[ok] 4ef834ce95ec545722370ace5a5738865d45df9e, sectors 0 - 2097151, 'able_3
↪ .000'
[ok] ca848143cca181b112b82c3d20acde6bdaf37506, sectors 2097152 - 4194303, '
↪ able_3.001'
[ok] 3d63f2724304205b6f7fe5cadcbc39c05f18cf30, sectors 4194304 - 6291455, '
↪ able_3.002'
[ok] 9e8607df22e24750df7d35549d205c3bd69adfe3, sectors 6291456 - 8388607, '
↪ able_3.003'
```

dc3dd completed at 2017-05-25 16:08:22 +0000

The first hash in the output above is the entire input hash for the device that was imaged (`/dev/sda1` from the subject system). We can verify that by streaming all our split parts together and piping through **shasum**:

```
barry@forensicbox:able_3$ cat able_3.0*| shasum
2eddbfe3d00cc7376172ec320df88f61afda3502 -
```

The next four hashes are for the split image files (and the sector range in each split). We could also verify these individually, although if the previous command works, we've already confirmed our individual hashes will match. Go ahead and check them anyway:

```
barry@forensicbox:able_3$ shasum able_3.0*
4ef834ce95ec545722370ace5a5738865d45df9e able_3.000
ca848143cca181b112b82c3d20acde6bdaf37506 able_3.001
3d63f2724304205b6f7fe5cadcbc39c05f18cf30 able_3.002
9e8607df22e24750df7d35549d205c3bd69adfe3 able_3.003
```

We can see they match the hashes in the log file.

Okay, now we have our image, and we have verified that it is an accurate copy. In order to check the file system and carve the partitions, we'll need to work on a single raw image instead of splits. Working from the assumption that we are executing this on a system with basic tools, we'll forgo using tools like **affuse** and **kpartx**. Instead, we'll simply recreate a raw image by using **cat** to add the files back together and re-direct to the raw image:

```
barry@forensicbox:able_3$ cat able_3.0*> able_3.raw
```

And now we will work on the `able_3.raw` image.

Let's start by exploring the contents of the image with some of our partition parsing tools. To use these tools, you'll need to be **root** (or use **sudo**) and change to the directory where the images are (user's home directory and `able_3` sub directory):

```
barry@forensicbox:able_3$ su -  
Password:  
  
root@forensicbox:~# cd /home/barry/able_3  
  
root@forensicbox:able_3#
```

Starting with **fdisk**:

```
root@forensicbox:able_3# fdisk -l able_3.raw  
Disk able_3.raw: 4 GiB, 4294967296 bytes, 8388608 sectors  
Units: sectors of 1 * 512 = 512 bytes  
Sector size (logical/physical): 512 bytes / 512 bytes  
I/O size (minimum/optimal): 512 bytes / 512 bytes  
Disklabel type: gpt  
Disk identifier: B94F8C48-CE81-43F4-A062-AA2E55C2C833  
  
Device        Start      End Sectors  Size Type  
able_3.raw1    2048      104447 102400    50M Linux filesystem  
able_3.raw2 104448    309247 204800   100M Linux filesystem  
able_3.raw3 571392   8388574 7817183  3.7G Linux filesystem
```

Looking at the output, we see that the disk has a **GPT** partitioning scheme. You could re-run the command using **gdisk** for documentation purposes. Once we've finished with **fdisk**, exit the **root** login and you are back to a normal user:

```
root@forensicbox:able_3# exit  
  
barry@forensicbox:able_3$
```

Now use **dd** to carve each of the partitions. With the output of **fdisk -l** shown above, the job is easy.

```
barry@forensicbox:able_3$ dd if=able_3.raw of=able_3.part1.raw bs=512 skip=2048  
count=102400  
102400+0 records in  
102400+0 records out  
52428800 bytes (52 MB, 50 MiB) copied, 1.742 s, 30.1 MB/s  
  
barry@forensicbox:able_3$ dd if=able_3.raw of=able_3.part2.raw bs=512  
skip=104448 count=204800  
204800+0 records in  
204800+0 records out  
104857600 bytes (105 MB, 100 MiB) copied, 0.731567 s, 143 MB/s
```

```
barry@forensicbox:able_3$ dd if=able_3.raw of=able_3.part3.raw bs=512
    skip=571392 count=7817183
7817183+0 records in
7817183+0 records out
4002397696 bytes (4.0 GB, 3.7 GiB) copied, 35.585 s, 112 MB/s
```

Examine these commands closely. The input file (**if=able_3.raw**) is the full disk image. The output files (**of=able_3.part#.raw**) will contain each of the partitions. The block size that we are using is the sector size (**bs=512**), which matches the output of the **fdisk** command. Each **dd** section needs to start where each partition begins (**skip=X**), and cut as far as the partition goes (**count=Y**). The **count** is from the **Sectors** column of the **fdisk** output and tells us the number of sectors to count.

This will leave you with three **able_3.part*.raw** files in your current directory that can now be loop mounted without the need for special programs. We will explore and work with these partitions in the following section.

8.2.3 Reconstructing a Subject File System (Linux)

Going back to our **able_3** case raw images, we now have the original image along with the partition images that we carved out (plus the original split images).

able_3.part1.raw (1st Partition) **able_3.part2.raw** (2nd Partition) **able_3.part3.raw** (3rd Partition)

The next trick is to mount the partitions in such a way that we reconstruct the original file system. This generally pertains to subject disks that were imaged from Unix hosts, but it still makes for a good command line exercise.

One of the benefits of Linux/Unix systems is the ability to separate the file system across partitions. This can be done for any number of reasons, allowing for flexibility where there are concerns about disk space or security, etc.

For example, a system administrator may decide to keep the directory **/var/log** on its own separate partition. This might be done in an attempt to prevent rampant log files from filling the root (**/** not **/root**) partition and bringing the system down. In the past, finding the **/boot** directory in its own partition was common as well. This allows the kernel image to be placed near "the front" (in terms of cylinders) of a boot volume, an issue in some older boot loaders. There are also a variety of security implications addressed by this setup. Finally, some would consider having **/home** on its own partition a good idea, allowing you to format and re-install without having to touch personal data and files.

So when you have a disk with multiple partitions, how do you find out the structure of the file system? Earlier in this paper we discussed the **/etc/fstab** file. This file maintains the

mounting information for each file system, including the physical partition; mount point, file system type, and options. Once we find this file, reconstructing the system is easy. With experience, you will start to get a feel for how partitions are setup, and where to look for the **fstab**. To make things simple here, just mount each partition (loop, read only) and have a look around.

One thing we might like to know is what sort of file system is on each partition before we try and mount them. We can use the **file** command to do this. Remember from our earlier exercise that the **file** command determines the type of file by looking for "header" information.

```
barry@forensicbox:able_3$ file able_3.part*
able_3.part1.raw: Linux rev 1.0 ext4 filesystem data, UUID=ca05157e-f7b3-4c6a-9b63
    ↪ -235c4cad7b73 (extents) (large files) (huge files)
able_3.part2.raw: Linux rev 1.0 ext4 filesystem data, UUID=c4ac4c0f-d9de-4d26-9e16
    ↪ -10583b607372 (extents) (large files) (huge files)
able_3.part3.raw: Linux rev 1.0 ext4 filesystem data, UUID=c7f748b2-3a38-44e9-aa43-
    ↪ f924955b9fdd (extents) (large files) (huge files)
```

Previously, we were able to determine that the partitions were "Linux" partitions from the output of **fdisk**. Now **file** informs us that the file system type is **ext4**. We can use this information to mount the partitions. Remember that you will need to be **root** to mount the partitions, so **su** to root first (or use **sudo**), **mount** and **umount** each partition, until you find the **/etc** directory containing the **fstab**:

```
barry@forensicbox:~$ su -
Password:

root@forensicbox:~# mount -t ext4 -o ro,loop/home/barry/able_3/able_3.part1.raw
/mnt/evid

root@forensicbox:~# ls /mnt/evid
README.initrd@      config-huge-4.4.14  onlyblue.dat
System.map@         elilo-ia32.efi*    slack.bmp
System.map-generic-4.4.14  elilo-x86_64.efi*  tuxlogo.bmp
System.map-huge-4.4.14    grub/             tuxlogo.dat
boot.0800           inside.bmp         vmlinuz@
boot_message.txt     inside.dat         vmlinuz-generic@
coffee.dat          lost+found/        vmlinuz-generic-4.4.14
config@             map               vmlinuz-huge@
config-generic-4.4.14  onlyblue.bmp      vmlinuz-huge-4.4.14
```

(We are looking for **/etc/fstab**, and it's not here...)

```
root@forensicbox:~# umount /mnt/evid
```

If you do this for each partition in turn (either un-mounting between partitions, or mounting to a different mount point), you will eventually find the `/etc` directory containing the `fstab` file in `able_3.part3.raw` with the following important entries:

```
barry@forensicbox:able_3$ cat /mnt/evvid/etc/fstab
/dev/sda3      /          ext4      defaults    1    1
/dev/sda1      /boot      ext4      defaults    1    2
/dev/sda2      /home      ext4      defaults    1    2
```

So now we see that the logical file system was constructed from three separate partitions (note that `/dev/sda` here refers to the disk when it is mounted in the original system, and not to the disk in the *forensic workstation*):

```
(root directory) mounted from /dev/sda3 (able_3.part3)
|-- bin
|-- boot mounted from /dev/sda1 (able_3.part1)
|-- dev
|-- etc
|-- home mounted from /dev/sda2 (able_3.part2)
|-- lib
|-- lib64
|-- lost+found
|-- media
|-- mnt
|-- opt
|-- proc
|-- root
|-- run
|-- sbin
|-- srv
|-- sys
|-- tmp
|-- usr
'-- var
```

Now we can create the original file system at our evidence mount point. The mount point `/mnt/evvid` already exists. When you mount the root partition of `able_3.raw` on `/mnt/evvid`, you will note that the directories `/mnt/evvid/boot` and `/mnt/evvid/home` already exist, but are empty. That is because we have to mount those volumes to access the contents of those directories. We mount the root file system first, and the others are mounted to that. Again, we will be executing as `root` for this:

```
root@forensicbox:able_3# mount -t ext4 -o ro,loop able_3.part3.raw /mnt/evvid

root@forensicbox:able_3# mount -t ext4 -o ro,loop able_3.part1.raw /mnt/evvid/boot
```

```
root@forensicbox:able_3# mount -t ext4 -o ro,loop able_3.part2.raw /mnt/evid/home
```

We now have the recreated original file system under `/mnt/evid`:

```
barry@forensicbox:~$ mount | grep evid
/home/barry/able_3/able_3.part3.raw on /mnt/evid type ext4 (ro)
/home/barry/able_3/able_3.part1.raw on /mnt/evid/boot type ext4 (ro)
/home/barry/able_3/able_3.part2.raw on /mnt/evid/home type ext4 (ro)
```

At this point we can run all of our searches and commands just as we did for the previous `fat_fs.raw` exercise on a complete file system “rooted” at `/mnt/evid`.

As always, you should know what you are doing when you mount a complete file system on your forensic workstation. Be aware of options to the **mount** command that you might want to use (check **man mount** for options like **nodev** and **nosuid**, **noatime**, etc.). Take note of where links point to from the subject file system. Note that we have mounted the partitions “read only” (**ro**). Remember to unmount (**umount**) each partition when you are finished exploring.

9 Advanced Analysis Tools

So now you have some experience with using the Linux command line and the powerful tools that are provided with a Linux installation.

However, as forensic examiners, we soon come to find out that time is a valuable commodity. While learning to use the command line tools native to a Linux install is useful for a myriad of tasks in the "real world", it can also be tedious. After all, there are Windows based tools out there that allow you to do much of what we have discussed here in a simple point and click GUI.

The popularity of Linux is growing at a fantastic rate. Not only do we see it in an enterprise environment and in big media, but it continues to grow in popularity within the field of computer forensics. In recent years we've seen the list of available forensic tools for Linux grow with the rest of the industry.

In this section we will cover a number of forensic tools available to make your analysis easier and more efficient.

AUTHOR'S NOTE: Inclusion of tools and packages in this section in no way constitutes an endorsement of those tools. Please test them yourself to ensure that they meet your needs.

Since this is a Linux document, I am covering available Linux tools. This does not mean that the common tools available for other platforms cannot be used to accomplish many of the same results.

Remember, as you work through these exercises, this document is NOT meant to be an education in file system or physical volume analysis. As you work through the exercises you will come across terms like *inode*, *MFT entry*, *allocation status*, *partition tables* and direct and indirect blocks, etc. These exercises are about using the tools, and are not meant to instruct you on basic forensic knowledge, Linux file systems or any other file systems. This is all about the tools.

If you need to learn file system structure as it relates to computer forensics, please read Brian Carrier's book: *File System Forensic Analysis* (Published by Addison-Wesley, 2005). This is not the last time I will suggest this.

To get a quick overview of some file systems, you can do a quick Internet search. There is a ton of information readily available if you need a primer. Here are some simple links to get you started. If you have questions on any of these file systems, or how they work, I would suggest some light reading before diving into these exercises.

NTFS <http://www.ntfs.com>
 <http://en.wikipedia.org/wiki/NTFS>

ext2/3/4 <http://e2fsprogs.sourceforge.net/ext2intro.html>

<http://en.wikipedia.org/wiki/Ext3>

<http://en.wikipedia.org/wiki/Ext4>

FAT http://en.wikipedia.org/wiki/File_allocation_table

Also, once Sleuth Kit (which we cover soon) is installed, you might want to browse around <http://wiki.sleuthkit.org/> for additional information on file systems and implementation.

9.1 The Layer Approach to Analysis

One of the reasons Linux is seen as both extremely efficient by its proponents and excessively complex by its detractors is its focus on modular programming where one tool accomplishes one task rather than the "suite of tools" approach of many commercial tools. The design of Linux tools, both GNU command line utilities and forensic software such as the Sleuth Kit, can appear daunting when a student realizes that they must try to remember multiple tools, outputs, and command parameters in order to execute an effective examination rather than navigating a graphical menu where functions, options, and output are displayed in an organized "one click away" fashion.

Brian Carrier, author of The Sleuth Kit, utilizes a framework for storage device analysis in his book File System Forensic Analysis, which we mentioned earlier. Using this approach, Carrier organizes his tools into a series of virtual layers that define the purpose of each tool with respect to a specific layer. Conveniently, the Sleuth Kit tools are named according to these layers. By introducing tools in a given category and defining their respective applicability to a specific virtual layer, students can better organize their understanding of each tool's function and where it best fits in an analysis.

This approach can easily be extended and expanded to encompass additional tools from outside Sleuth Kit. While some tools do not succinctly fit in this paradigm, they can still be addressed in a sequence that fits the overall analytical approach. The following figure provides a graphical summary of the layers Carrier designates for the analysis of evidence.

Figure 7: An example of layers and their associated content based on Carrier's work

Physical Media			Media Management	File System			Application
Physical Description							
Head	Cyl	Sectors	Partition Table				
			Partition	Boot Sector	FAT	Data Areas	
				...			
				File			File Meta-data
							Content

This has the added benefit of giving students a method of conceptualizing the way tools are employed.

We have a diverse set of tools to work with in Linux, particularly where tackling an analysis from the command line is concerned. Knowing when to use which tools can be better understood by extending this layer approach to our entire analysis.

We've already covered a number of common tools like **dd**, **dc3dd**, **hdparm**, **lsscsi**, **lshw** and others. These are examples of tools that work at the physical media layer – looking directly at physical media and disk information, including serial numbers, disk sector sizes and the physical bus on which the media resides.

We've also looked at tools that act on the media management layer, like **fdisk**, **gdisk**, **kpartx** and others. These tools parse or work on information provided at the partition table level, but without specifically acting on the file systems themselves.

As we progress through the rest of this guide, be aware that often a tool's place in the layer approach is not defined by the tool itself, but by how you use it. Take **grep** for example. **grep** looks for matching expressions in a file. So we could say it works on the file sub layer of the *file system* layer (refer to the illustration above). However, when we use it against a forensic image of a physical disk (like our **able_3.raw** file), we are not using it at the file layer of that image, but at the physical layer of the image. I can **grep** for an expression in a set of files, or I can **grep** for an expression in a disk image. How I use the tool defines its place, not the tool itself in many cases.

So we need to adjust our thinking on how we approach our analysis, keeping in mind that the tool organization of the Sleuth Kit may not always directly match our analysis approach. But we can simply summarize it like this :

1. Analyze the physical device layer:

- **lshw**
- **lsscsi**
- **hdparm**

2. Analyze the media management layer:

- **fdisk**
- **gdisk**
- **file -s**
- **mmls** (we will cover soon)

3. Analyze the file system layer:

- **file**
- **fsstat**

- **fls**

4. Analyze the file layer:

- **file**
- **find**
- common tools like **ls**, etc.

5. Application layer:

- View file content with **less** and **cat**, etc.
- Locate specific content with **grep**.
- Use external applications to view binary content: **xv**, **display**, **regfmount** for registry files, etc.

The commands shown here are only a few examples. There are many more that work at each given layer. So you can see from the list above that we can have tools apply to several different layers. This speaks to the simplicity of the Unix development approach that has been around for decades. The tools generally do one thing, do it well, but can be versatile in their employment.

In summary, this all means that instead of taking the approach that we might normally take with multi-functional Windows forensic software:

- Open a program
- Open (or acquire) an image file with that program
- "Index" the image file within the program
- Navigate the menus, collecting data and reporting it

...we can now sit at a command prompt and step through the various layers of our examination, collecting and redirecting information as we go, peeling through layer by layer of our analysis until we reach our conclusion. Instead of fumbling around the command line, we target our commands to the layer we are currently examining.

9.2 The Sleuth Kit

The first of the advanced external tools we will cover here is a collection of command line tools called the Sleuth Kit (TSK). This is a suite of tools written by Brian Carrier and maintained at <http://www.sleuthkit.org>. It is partially based on The Coroner's Toolkit (TCT) originally written by Dan Farmer and Wietse Venema. TSK adds additional file system support and allows you to analyze various file system types regardless of the platform you are currently working on. The current version, as of this writing is 4.6.x.

Let's start with a discussion of the tools first. Most of this information is readily available in the Sleuth Kit documentation or on the Sleuth Kit website.

We've already discussed the TSK's organization of tool function by layers. Here's a list of some of the tools and where they fit in.

- Media management layer: **mmls**, **mmcat**, **mmstat**
- File system layer: **fsstat**
- File name layer ("Human Interface") : **fls**, **ffind**
- Meta data (inode) layer: **icat**, **ils**, **ifind**, **istat**
- Content (data) layer: **blkcalc**, **blkcat**, **blkls**, **blkstat**

We also have tools that address physical disks and tools that address the "journals" of some file systems.

- Journal tools: **jcat**, **jls**
- File content tools: **hfind**, **fcatt**

Notice the commands that correspond to the analysis of a given layer generally begin with a common letter. For example, the file system commands start with **fs** and the inode (meta-data) layer commands start with **i** and so on.

If the "layer" approach referenced above seems a little confusing to you, you should take the time to read TSK's tool overview at:

http://wiki.sleuthkit.org/index.php?title=TSK_Tool_Overview

The author does a fine job of defining and describing these layers and how they fit together for a forensic analysis. Understanding that TSK tools operate at different layers is extremely important.

When running through the following exercises, pay attention to the fact that the output of each tool is specifically tailored to the file system being analyzed. For example, the **fsstat** command is used to print file system details (*fs layer*). The structure of the output and the descriptive fields change depending on the target file system. This will become apparent throughout the exercises.

In addition to the tools already mentioned, there are some miscellaneous tools included with the Sleuth Kit that don't fall into the above categories:

- **tsk_recover**: recovers unallocated (or all) files from a file system.
- **tsk_gettimes**: creates a body file for timelines (file activity only)

- **sorter**: categorizes allocated and unallocated files based on type (images, executables, etc). Extremely flexible and configurable.
- **img_cat**: allows for the separation of meta-data and original data from image files (media duplication, not pictures).
- **img_stat**: provides information about a forensic image. The information it provides is dependent on the image format (**aff**, **ewf**, etc.).
- **hfind**: hash lookup tool. Creates and searches an indexed database.
- **sigfind**: searches a given file (forensic image, disk, etc.) for a hex signature at any specified offset (sector boundary). Used for finding data structures
- **mactime**: creates a time line of file activity. Useful for intrusion investigations where temporal relationships are critical.
- **srch_strings**: like standard BSD **strings** command, but with the ability to parse different encodings.

9.2.1 Sleuth Kit Installation

On Slackware, we can install TSK simply with **sboinstall**:

```
root@forensicbox:~# sboinstall sleuthkit
...
Proceed with sleuthkit? [y]
...
Cleaning for sleuthkit-4.6.6...
```

When we install TSK using the SlackBuild through **sboinstall**, or if you install it manually from source, you can watch the build process. It should be noted that in order for Sleuth Kit tools to have built-in support for Expert Witness format images (EWF images), we need to have **libewf** installed first. This is why we covered **libewf** and installed it earlier in the guide. While the Sleuth Kit is configuring its installation process, it searches the system for libraries that it supports. Unless it's told not to include specific capabilities, it will compile itself accordingly. In this case, since we have **libewf** and **afflib** already installed, TSK will be built with those formats supported. This will allow us to work directly on EWF and AFF images.²⁷

When the installation is finished, you will find the Sleuth Kit tools located in **/usr/bin**.

Remember, you can view a list of what was installed (and other package information) by viewing the file at **/var/log/packages/sleuthkit-<version>**:

²⁷TSK also supports VMDK and VHDI formats which we do not cover in this guide. Install **libvmdk** and **libvhdi** prior to installing TSK if you want to try them out.

```
root@forensicbox:~# less /var/log/packages/sleuthkit-4.6.6-x86_64-1_SBo
PACKAGE NAME:      sleuthkit-4.6.6-x86_64-1_SBo
COMPRESSED PACKAGE SIZE:  656K
...
usr/
usr/bin/
usr/bin/blkcalc
usr/bin/blkcat
usr/bin/blkls
usr/bin/blkstat
usr/bin/fcat
usr/bin/ffind
usr/bin/fiwalk
...
```

Use the **man** page for each of the utilities installed to `/usr/bin` shown in the output to get an idea of the complete capabilities of TSK.

9.3 Sleuth Kit Exercises

This section remains one of the most popular sections of this guide, providing hands on exercises for TSK and a sample of its tools.

Like all of the other exercises introduced here, it is strongly suggested you follow along if you can. Using these commands on your own is the only way to really learn the techniques. Read the included **man** pages and play with the options to obtain novel output. The image files used in the following examples are available for download, and some have already been downloaded and used earlier in the guide.

There are a number of ways to tackle the following problems. In some cases we'll use **affuse** or **ewfmount** to provide fuse mounted images from EWF files or split files. We'll do it for practice here, but feel free to run the tools directly on the image files themselves (there will be demonstrations of both). Practice and experiment.

We'll also use some of the older image files that were used in previous versions of the guide. While the images are old and the file systems somewhat deprecated, we use them here because they provide a perfect vehicle for demonstrating tool usage. You'll understand this a bit more as we progress. We can compare output on some of the newer images and you'll understand some of the limitations.

For the following set of exercises, we'll use the **able2** image, one of the older but more educational images we've used. Create a directory for the **able2** image and then **cd** into the directory. As usual, download with **wget** and check the hash, making sure it matches what we have here:

```
barry@forensicbox:~$ mkdir able2
```

```
barry@forensicbox:~$ cd able2
```

```
barry@forensicbox:able2$ wget https://www.linuxleo.com/Files/able2.tar.gz
```

```
barry@forensicbox:able2$ shasum able2.tar.gz
a093ec9aed6054665b89aa82140803790be97a6e  able2.tar.gz
```

Extract the archive to decompress the image and then let's get started. Get your hands on the keyboard and follow along.

```
barry@forensicbox:able2$ tar tzf able2.tar.gz <- list contents first
```

```
able2.dd
able2.log
md5.dd
md5.hdd
```

```
barry@forensicbox:able2$ tar xzvf able2.tar.gz <- then extract the contents
```

```
able2.dd
able2.log
md5.dd
md5.hdd
```

Recall that by listing the contents of the archive first with the `t` option, we can see that it will extract to our current directory (no leading paths in the listing) instead of somewhere unexpected.

9.3.1 Sleuth Kit Exercise 1A: Deleted File Identification and Recovery (ext2)

We will start with a look at a couple of the file system and file name layer tools, **fsstat** and **fls**, running them against our **able2** image.

Part of the TSK suite of tools, **mmls**, provides access to the partition table within an image, and gives the partition offsets in sector units. **mmls** provides much the same information as we get from **fdisk** or **gdisk**.

```
barry@forensicbox:able2$ mmls able2.dd
```

```
DOS Partition Table
```

```
Offset Sector: 0
```

```
Units are in 512-byte sectors
```

Slot	Start	End	Length	Description
------	-------	-----	--------	-------------

000:	Meta	0000000000	0000000000	0000000001	Primary Table (#0)
001:	-----	0000000000	0000000056	0000000057	Unallocated
002:	000:000	0000000057	0000010259	0000010203	Linux (0x83)
003:	000:001	0000010260	0000112859	0000102600	Linux (0x83)
004:	000:002	0000112860	0000178694	0000065835	Linux Swap / Solaris x86 (0 → x82)
005:	000:003	0000178695	0000675449	0000496755	Linux (0x83)

For the sake of this analysis, the information we are looking for is located on the root partition (file system) of our image. The root (/) file system is located on the second partition. Looking at our **mmls** output, we can see that that partition starts at sector **10260** (actually numbered **03** in the **mmls** output, or slot **000:001**).

So, we run the Sleuth Kit **fsstat** command with **-o 10260** to gather file system information at that offset. By using the sector offset provided by **mmls**, we tell our TSK tools where the volume begins. Pipe the output through **less** to page through:

```
barry@forensicbox:able2$ fsstat -o 10260 able2.dd | less
```

```
FILE SYSTEM INFORMATION
```

```
-----
```

```
File System Type: Ext2
```

```
Volume Name:
```

```
Volume ID: 906e777080e09488d0116064da18c0c4
```

```
Last Written at: 2003-08-10 14:50:03 (EDT)
```

```
Last Checked at: 1997-02-11 00:20:09 (EST)
```

```
Last Mounted at: 1997-02-13 02:33:02 (EST)
```

```
Unmounted Improperly
```

```
Source OS: Linux
```

```
Dynamic Structure
```

```
InCompat Features: Filetype,
```

```
Read Only Compat Features: Sparse Super,
```

```
METADATA INFORMATION
```

```
-----
```

```
Inode Range: 1 - 12881
```

```
Root Directory: 2
```

```
Free Inodes: 5807
```

```
CONTENT INFORMATION
```

```
-----
```

```
Block Range: 0 - 51299
```

```
Block Size: 1024
```

```
Reserved Blocks Before Block Groups: 1
```

```
Free Blocks: 9512
```

BLOCK GROUP INFORMATION

```
-----  
Number of Block Groups: 7  
Inodes per group: 1840  
Blocks per group: 8192  
  
Group: 0:  
  Inode Range: 1 - 1840  
  Block Range: 1 - 8192  
  Layout:  
    Super Block: 1 - 1  
    Group Descriptor Table: 2 - 2  
    Data bitmap: 3 - 3  
    Inode bitmap: 4 - 4  
    Inode Table: 5 - 234  
    Data Blocks: 235 - 8192  
  Free Inodes: 789 (42%)  
  Free Blocks: 4601 (56%)  
  Total Directories: 16  
...
```

The **fsstat** command provides type specific information about the file system in a volume. As previously noted, we ran the **fsstat** command above with the option **-o 10260**. This specifies that we want information from the file system residing on the partition that starts at sector offset **10260**.

We can get more information using the **fls** command. **fls** lists the file names and directories contained in a file system or in a directory if the meta-data identifier for a particular directory is passed. The output can be adjusted with a number of options, to include gathering information about deleted files. If you type **fls** on its own, you will see the available options (view the **man** page for a more complete explanation).

If you run the **fls** command with only the **-o** option to specify the file system, then by default it will run on the file system's root directory. This is inode 2 on an EXT file system and MFT entry 5 on an NTFS file system. In other words, on an EXT file system, running:

```
barry@forensicbox:able2$ fls -o 10260 able2.dd
```

And ...

```
barry@forensicbox:able2$ fls -o 10260 able2.dd 2
```

...will result in the same output. In the second command, the 2 passed at the end of the command means "root directory" (for EXT), which is the default in the first command.

So, in the following command, we run **fls** and only pass **-o 10260**. This results in a listing of the contents of the root directory:

```
barry@forensicbox:able2$ fls -o 10260 able2.dd
d/d 11: lost+found
d/d 3681: boot
d/d 7361: usr
d/d 3682: proc
d/d 7362: var
d/d 5521: tmp
d/d 7363: dev
d/d 9201: etc
d/d 1843: bin
d/d 1844: home
d/d 7368: lib
d/d 7369: mnt
d/d 7370: opt
d/d 1848: root
d/d 1849: sbin
r/r 1042: .bash_history
d/d 11105: .001
V/V 12881: $OrphanFiles
```

There are several points we want to take note of before we continue. Let's take a few lines of output and describe what the tool is telling us. Have a look at the last three lines from the above **fls** command.

```
r/r 1042: .bash_history
d/d 11105: .001
V/V 12881: $OrphanFiles
```

Each line of output starts with two characters separated by a slash. This field indicates the file type as described by the file's directory entry, and the file's meta-data (in this case, the inode because we are looking at an EXT file system). For example, the first file listed in the snippet above, **.bash_history**, is identified as a regular file in both the file's directory and inode entry. This is noted by the **r/r** designation. Conversely, the following entry (**.001** and is identified as a directory.

The last line of the output, **\$OrphanFiles** is a virtual folder created by TSK and assigned a virtual inode. This folder contains virtual file entries that represent unallocated meta data entries where there are no corresponding file names. These are commonly referred to as "orphan files", which can be accessed by specifying the meta data address, but not through

any file name path.

The next field is the meta-data entry number (inode, MFT entry, etc.) followed by the filename. In the case of the file `.bash_history` the inode is listed as **1042**.

We can continue to run **fls** on directory entries to dig deeper into the file system structure (or use **-r** for a recursive listing). By passing the meta data entry number of a directory, we can view its contents. Read man **fls** for a look at some useful features. For example, have a look at the **.001** directory in the listing above. This is an unusual directory and would cause some suspicion. It is hidden (starts with a "."), and no such directory is common in the root of the file system. So to see the contents of the **.001** directory, we would pass its inode to **fls**:

```
barry@forensicbox:able2$ fls -o 10260 able2.dd 11105
r/r 2138: lolit_pics.tar.gz
r/r 11107: lolitaz1
r/r 11108: lolitaz10
r/r 11109: lolitaz11
r/r 11110: lolitaz12
r/r 11111: lolitaz13
r/r 11112: lolitaz2
r/r 11113: lolitaz3
r/r 11114: lolitaz4
r/r 11115: lolitaz5
r/r 11116: lolitaz6
r/r 11117: lolitaz7
r/r 11118: lolitaz8
r/r 11119: lolitaz9
```

The contents of the directory are listed. We will cover commands to view and analyze the individual files later on.

fls can also be useful for uncovering deleted files. By default, **fls** will show both allocated and unallocated files. We can change this behavior by passing other options. For example, if we wanted to see only deleted entries that are listed as files (rather than directories), and we want the listing to be recursive, we could use the following command:

```
barry@forensicbox:able2$ fls -o 10260 -Frd able2.dd
r/r * 11120(realloc): var/lib/slocate/slocate.db.tmp
r/r * 10063: var/log/xferlog.5
r/r * 10063: var/lock/makewhatis.lock
r/r * 6613: var/run/shutdown.pid
r/r * 1046: var/tmp/rpm-tmp.64655
r/r * 6609(realloc): var/catman/cat1/rdate.1.gz
r/r * 6613: var/catman/cat1/rdate.1.gz
r/r * 6616: tmp/logrot2V6Q1J
```

```

r/r * 2139: dev/ttYZ0/lrkn.tgz
d/r * 10071(realloc): dev/ttYZ0/lrk3
r/r * 6572(realloc): etc/X11/fs/config-
l/r * 1041(realloc): etc/rc.d/rc0.d/K83ypbind
l/r * 1042(realloc): etc/rc.d/rc1.d/K83ypbind
l/r * 6583(realloc): etc/rc.d/rc2.d/K83ypbind
l/r * 6584(realloc): etc/rc.d/rc4.d/K83ypbind
l/r * 1044: etc/rc.d/rc5.d/K83ypbind
l/r * 6585(realloc): etc/rc.d/rc6.d/K83ypbind
r/r * 1044: etc/rc.d/rc.firewall~
r/r * 6544(realloc): etc/pam.d/passwd-
r/r * 10055(realloc): etc/mtab.tmp
r/r * 10047(realloc): etc/mtab~
r/- * 0: etc/inetd.conf.swx
r/r * 2138(realloc): root/lolita_pics.tar.gz
r/r * 2139: root/lrkn.tgz
-/r * 1055: $OrphanFiles/OrphanFile-1055
-/r * 1056: $OrphanFiles/OrphanFile-1056
-/r * 1057: $OrphanFiles/OrphanFile-1057
-/r * 2141: $OrphanFiles/OrphanFile-2141
-/r * 2142: $OrphanFiles/OrphanFile-2142
-/r * 2143: $OrphanFiles/OrphanFile-2143
...

```

In the above command, we run the **fls** command against the partition in **able2.dd** starting at sector offset **10260** (**-o 10260**), showing only file entries (**-F**), descending into directories recursively (**-r**), and displaying deleted (unallocated) entries (**-d**).

Notice that all of the files listed have an asterisk (*) before the inode. This indicates the file is deleted or unallocated, which we expect in the above output since we specified the **-d** option to **fls**. We are then presented with the meta-data entry number (inode, MFT entry, etc.) followed by the filename.

Have a look at the line of output for inode number **2138** (**root/lolita_pics.tar.gz**). The inode is followed by **realloc**. Keep in mind that **fls** describes the file name layer. The **realloc** means that the file name listed is marked as unallocated, even though the meta data entry (**2138**) is marked as allocated. In other words...the inode from our deleted file may have been “reallocated” to a new file.

According to Brian Carrier:

The difference comes about because there is a file name layer and a metadata layer. Every file has an entry in both layers and each entry has its own allocation status.

If a file is marked as "deleted" then this means that both the file name and metadata entries are marked as unallocated. If a file is marked as "realloc" then

this means that its file name is unallocated and its metadata is allocated.

The latter occurs if:

- *The file was renamed and a new file name entry was created for the file, but the metadata stayed the same.*
- *NTFS resorted the names and the old copies of the name will be "unallocated" even though the file still exists (we are on an EXT file system here, so this does not apply)*
- *The file was deleted, but the metadata has been reallocated to a new file.*

In the first two cases, the metadata correctly corresponds to the deleted file name.

In the last case, the metadata may not correspond to the name because it may instead correspond to a new file.

In the case of inode 2138, it looks as though the **realloc** was caused by the file being moved to the directory **.001** (see the **fls** listing of **.001** on the previous page - inode 11105). This causes it to be deleted from its current directory entry (**root/lolitt_pics.tar.gz**) and a new file name created (**.001/lolitt_pics.tar.gz**). The inode and the data blocks that it points to remain unchanged and in "allocated status", but it has been "reallocated" to the new name.

Let's continue our analysis exercise using a couple of meta data (inode) layer tools included with the Sleuth Kit. In a Linux EXT type file system, an inode has a unique number and is assigned to a file. The number corresponds to the inode table, allocated when a partition is formatted. The inode contains all the meta data available for a file, including the modified/accessed/changed (mac) times and a list of all the data blocks allocated to that file.

If you look at the output of our last **fls** command, you will see a deleted file called **lrkn.tgz** located in the **/root** directory (the last file in the output of our **fls** command, before the list of orphan files -recall that the asterisk indicates it is deleted):

```
...
r/r * 2139: root/lrkn.tgz
...
```

The inode displayed by **fls** for this file is 2139. This same inode also points to another deleted file in **/dev** earlier in the output (same file, different location). We can find all the file names associated with a particular meta data entry by using the **ffind** command:

```
barry@forensicbox:able2$ ffind -o 10260 -a able2.dd 2139
* /dev/ttYZ0/lrkn.tgz
* /root/lrkn.tgz
```

Here we see that there are two file names associated with inode 2139, and both are deleted, as noted again by the asterisk (the **-a** ensures that we get all the inode associations).

Continuing on, we are going to use **istat**. Remember that **fsstat** took a file system as an argument and reported statistics about that file system. **istat** does the same thing; only it works on a specified inode or meta data entry. In NTFS, this would be an MFT entry, for example. Here we are working on an EXT file system, so the **istat** command works on EXT inodes.

We use **istat** to gather information about inode 2139:

```
barry@forensicbox:able2$ istat -o 10260 able2.dd 2139 | less
```

```
Not Allocated
```

```
Group: 1
```

```
Generation Id: 3534950564
```

```
uid / gid: 0 / 0
```

```
mode: rrw-r--r--
```

```
size: 3639016
```

```
num of links: 0
```

```
Inode Times:
```

```
Accessed: 2003-08-10 00:18:38 (EDT)
```

```
File Modified: 2003-08-10 00:08:32 (EDT)
```

```
Inode Modified: 2003-08-10 00:29:58 (EDT)
```

```
Deleted: 2003-08-10 00:29:58 (EDT)
```

```
Direct Blocks:
```

```
22811 22812 22813 22814 22815 22816 22817 22818
```

```
22819 22820 22821 22822 22824 22825 22826 22827
```

```
...
```

```
32233 32234
```

```
Indirect Blocks:
```

```
22823 23080 23081 23338 23595 23852 24109 24366
```

```
30478 30735 30992 31249 31506 31763 32020
```

This reads the inode statistics (**istat**), on the file system located in the **able2.dd** image in the partition at sector offset **10260** (**-o 10260**), from inode **2139** found in our **fls** command. There is a large amount of output here, showing all the inode information and the file system blocks ("Direct Blocks") that contain all of the file's data. We can either pipe the output of **istat** to a file for logging, or we can send it to **less** for viewing.

The Sleuth Kit supports a number of different file systems. **istat** (along with many of the Sleuth Kit commands) will work on more than just an EXT file system. The descriptive output will change to match the file system **istat** is being used on. We will see more of this a little later. You can see the supported file systems by running **istat** with **-f list**.

```
barry@forensicbox:able2$ istat -f list
```

```
Supported file system types:
```

```
ntfs (NTFS)
fat (FAT (Auto Detection))
ext (ExtX (Auto Detection))
iso9660 (ISO9660 CD)
hfs (HFS+)
ufs (UFS (Auto Detection))
raw (Raw Data)
swap (Swap Space)
fat12 (FAT12)
fat16 (FAT16)
fat32 (FAT32)
exfat (exFAT)
ext2 (Ext2)
ext3 (Ext3)
ext4 (Ext4)
ufs1 (UFS1)
ufs2 (UFS2)
yaffs2 (YAFFS2)
```

We now have the name of a deleted file of interest (from **fls**) and the inode information, including where the data is stored (from **istat**).

Now we are going to use the **icat** command from TSK to grab the actual data contained in the data blocks referenced from the inode. **icat** also takes the inode as an argument and reads the content of the data blocks that are assigned to that inode, sending it to standard output. Remember, this is a deleted file that we are recovering here.

We are going to send the contents of the data blocks assigned to inode 2139 to a file for closer examination.

```
barry@forensicbox:able2$ icat -o 10260 able2.dd 2139 > lrkn.tgz.2139
```

This runs the **icat** command on the file system in our **able2.dd** image at sector offset **10260** (**-o 10260**) and streams the contents of the data blocks associated with inode **2139** to the file **lrkn.tgz.2139**. The filename is arbitrary; I simply took the name of the file from **fls** and appended the inode number to indicate that it was recovered. Normally this output should be directed to some results or specified evidence directory.

Now that we have what we hope is a recovered file, what do we do with it? Look at the resulting file with the **file** command:

```
barry@forensicbox:able2$ file lrkn.tgz.2139
lrkn.tgz.2139: gzip compressed data, was "lrkn.tar", last modified: Sat Oct  3
  ↪ 09:04:08 1998, from Unix, original size 10106880
```

Have a look at the contents of the recovered archive (pipe the output through **less**...it's long). Remember that the **t** option to the **tar** command lists the contents of the archive.

```
barry@forensicbox:able2$ tar tzvf lrkn.tgz.2139 | less
drwxr-xr-x lp/lp          0 1998-10-01 18:48 lrk3/
-rwxr-xr-x lp/lp          742 1998-06-27 11:30 lrk3/1
-rw-r--r-- lp/lp          716 1996-11-02 16:38 lrk3/MCONFIG
-rw-r--r-- lp/lp        6833 1998-10-03 05:02 lrk3/Makefile
-rw-r--r-- lp/lp        6364 1996-12-27 22:01 lrk3/README
-rwxr-xr-x lp/lp           90 1998-06-27 12:53 lrk3/RUN
...
-rwxr-xr-x lp/lp        5526 1998-08-06 06:36 lrk3/z2
-rw-r--r-- lp/lp        1996 1996-11-02 16:39 lrk3/z2.c
```

We have not yet extracted the archive, we've just listed its contents. Notice that there is a **README** file included in the archive. If we are curious about the contents of the archive, perhaps reading the **README** file would be a good idea, yes? Rather than extract the entire contents of the archive, we will go for just the **README** using the following **tar** command:

```
barry@forensicbox:able2$ tar xzvf0 lrkn.tgz.2139 lrk3/README > lrkn.2139.README
lrk3/README
```

The difference with this **tar** command is that we specify that we want the output sent to stdout (**0** -capital letter 'oh') so we can redirect it. We also specify the name of the file that we want extracted from the archive (**lrk3/README**). This is all redirected to a new file called **lrkn.2139.README**.

If you read that file (use **less**), you will find that we have uncovered a "rootkit", full of programs used to hide a hacker's activity.

Briefly, let's look at a different type of file recovered by **icat**. The concept is the same, but instead of extracting a file, you can stream its contents to stdout for viewing. Recall our previous directory listing of the **.001** directory at inode **11105**:

```
barry@forensicbox:able2$ fls -o 10260 able2.dd 11105
r/r 2138:  lolit_pics.tar.gz
r/r 11107:  lolitaz1
r/r 11108:  lolitaz10
r/r 11109:  lolitaz11
r/r 11110:  lolitaz12
r/r 11111:  lolitaz13
r/r 11112:  lolitaz2
r/r 11113:  lolitaz3
r/r 11114:  lolitaz4
r/r 11115:  lolitaz5
```

```
r/r 11116: lolitaz6
r/r 11117: lolitaz7
r/r 11118: lolitaz8
r/r 11119: lolitaz9
```

We can determine the contents of the (allocated) file at inode **11108**, for example, by using **icat** to stream the inode's data blocks through a pipe to the **file** command. We use the **'-'** to indicate that **file** is getting its input from the pipe:

```
barry@forensicbox:able2$ icat -o 10260 able2.dd 11108 | file -
/dev/stdin: GIF image data, version 89a, 233 x 220
```

The output shows that we are dealing with a picture file. So we decide to use the **display** command to show us the contents. **display** is a useful program as it will take input from stdin (from a pipe). This is particularly useful with the **icat** command.

```
barry@forensicbox:able2$ icat -o 10260 able2.dd 11108 | display
```

This results in an image opening in a window, assuming you are running in a graphical environment and have ImageMagick installed, which provides the **d;isplay** utility.

Figure 8: The image produced by **display** when used directly from **icat**



9.3.2 Sleuth Kit Exercise 1B: Deleted File Identification and Recovery (ext4)

The previous exercise is a good primer for learning how to run TSK commands against a forensic image and identify and extract files. We use an older forensic image of an ext2

file system because it allows us to run the full course of identification and extraction tools provided by TSK. We can do this because ext2 files that are deleted still have enough information in their associated file system metadata ("inode" for EXT file systems) to be able to recover the file. As you will see in the coming pages, this has changed for the ext4 file system. As it has been made clear in the past, this is not meant to be an education on file systems in general. Rather, the purpose here is to highlight the tools and how you can expect different output based on the file system being examined. We also want to ensure that the limitations of our tools are known. Were you to learn TSK on an ext2 file system alone, you might expect it to work in exactly the same way on ext4. This is not the case, and this exercise illustrates that. It is one of the primary reasons why the **able_3** image was added to our problem set.

So now we are going to roughly replicate the same analysis as the previous exercise, but this time examining an ext4 file system in the **able_3** image. We'll be brief in the explanation of the commands, since they are largely the same as those we ran in exercise 1A. Review that exercise and make sure you are familiar with the commands used before proceeding here. The files being recovered are the same, but their placement differs a bit from the **able2** image.

First we need to decide how we want to access our image file. The **able_3** disk image, as it was downloaded, is a set of four split images. As we've done before, you could use **affuse** to mount the splits as a single image and even use **kpartx** to separate the partitions. But since the Sleuth Kit supports analysis of split image files, we'll go ahead and just leave them as is. You can use the **img_stat** command from TSK to document this.

Start by changing into the **able_3** directory we created previously for our image files, run **img_stat** to see the split file support and run **mmls** to identify the partitions. When using TSK on split images, we only need to provide the first image file in the set (the same rule holds for EWF files – you only provide the first file name in the set):

```
barry@forensicbox:~$ cd able_3

barry@forensicbox:able_3$ img_stat able_3.000
IMAGE FILE INFORMATION
-----
Image Type: raw

Size in bytes: 4294967296
Sector size:    512

-----
Split Information:
able_3.000  (0 to 1073741823)
able_3.001  (1073741824 to 2147483647)
able_3.002  (2147483648 to 3221225471)
able_3.003  (3221225472 to 4294967295)
```

```
barry@forensicbox:able_3$ mmls able_3.000
```

```
GUID Partition Table (EFI)
```

```
Offset Sector: 0
```

```
Units are in 512-byte sectors
```

	Slot	Start	End	Length	Description
000:	Meta	0000000000	0000000000	0000000001	Safety Table
001:	-----	0000000000	0000002047	0000002048	Unallocated
002:	Meta	0000000001	0000000001	0000000001	GPT Header
003:	Meta	0000000002	0000000033	0000000032	Partition Table
004:	000	0000002048	0000104447	0000102400	Linux filesystem
005:	001	0000 104448	0000309247	0000204800	Linux filesystem
006:	-----	0000309248	0000571391	0000262144	Unallocated
007:	002	0000571392	0000838574	00007817183	Linux filesystem
008:	-----	0000838575	00008388607	00000000033	Unallocated

Since our purpose here it to highlight the differences between the examination of this image set vs. the **able2** image, rather than search each partition individually we will just focus on the **/home** partition. Recall from our file system reconstruction exercise that the partition used for the **/home** directory on the **able_3** image is the partition at offset **104448** (bold for emphasis above).

Run **fsstat** on that partition to identify the file system type and information. You might want to pipe the output through **less** for easier viewing:

```
barry@forensicbox:able_3$ fsstat -o 104448 able_3.000
```

```
FILE SYSTEM INFORMATION
```

```
-----
```

```
File System Type: Ext4
```

```
Volume Name:
```

```
Volume ID: 7273603b5810169e264dded90f4cacc4
```

```
Last Written at: 2017-05-25 15:20:50 (EDT)
```

```
Last Checked at: 2017-05-06 16:49:45 (EDT)
```

```
Last Mounted at: 2017-05-25 15:10:23 (EDT)
```

```
Unmounted properly
```

```
Last mounted on: /home
```

```
...
```

Here we see we are examining an ext4 file system that was mounted on **/home**. Run a quick **fls** command to view the contents of this partition:

```
barry@forensicbox:able_3$ fls -o 104448 able_3.000
```

```
d/d 11: lost+found
```

```
d/d 12: ftp
d/d 13: albert
V/V 25689: $OrphanFiles
```

You can see there are few entries here. You could start digging down by providing the inode to the **fls** command for the contents of individual directories, but instead we'll simply do a recursive **fls**.

```
barry@forensicbox:able_3$ fls -o 104448 -r able_3.000
d/d 11: lost+found
d/d 12: ftp
d/d 13: albert
+ d/d 14: .h
++ r/d * 15(realloc): lolit_pics.tar.gz
++ r/r * 16(realloc): lolitaz1
++ r/r * 17: lolitaz10
++ r/r * 18: lolitaz11
++ r/r * 19: lolitaz12
++ r/r 20: lolitaz13
++ r/r * 21: lolitaz2
++ r/r * 22: lolitaz3
++ r/r * 23: lolitaz4
++ r/r * 24: lolitaz5
++ r/r * 25: lolitaz6
++ r/r * 26: lolitaz7
++ r/r * 27: lolitaz8
++ r/r * 28: lolitaz9
+ d/d 15: Download
++ r/r 16: index.html
++ r/r * 17: lrkn.tar.gz
V/V 25689: $OrphanFiles
```

You can see some familiar files in this output. We see the **lolitaz** files we saw in the **.001** directory on **able2**, and we also see the **lrkn.tar.gz** file we recovered and extracted the **README** from. For this exercise, we will be interested in the **lolitaz** files. The **lrkn.tar.gz** contents will come later. You'll notice that the majority of the files reside in an allocated (not deleted) directory called **.h** and are unallocated files (signified by the asterisk *****). There is a single allocated file in that directory called **lolitaz13**. Compare the output of **istat** and a follow-up **icat** command between the allocated file **lolitaz13** (inode 20), and one of the deleted files - we'll use **lolitaz2** (inode 21). For the **icat** command, we'll pipe the output to our hex viewer **xxd** and look at the first five lines with **head -n 5**. Here's the output of both:

```
barry@forensicbox:able_3$ istat -o 104448 able3.000 20
inode: 20
```

Allocated

```
Group: 0
Generation Id: 1815721463
uid / gid: 1000 / 100
mode: rrw-r--r--
Flags: Extents,
size: 15045
num of links: 1
```

Inode Times:

```
Accessed: 2017-05-08 00:18:16 (EDT)
File Modified: 2003-08-03 19:15:07 (EDT)
Inode Modified: 2017-05-08 00:18:16 (EDT)
```

Direct Blocks:

```
9921 9922 9923 9924 9925 9926 9927 9928
9929 9930 9931 9932 9933 9934 9935
```

```
barry@forensicbox:able_3$ icat -o 104448 able3.000 20 | xxd | head -n 5
00000000: ffd8 ffe0 0010 4a46 4946 0001 0100 0001  ....JFIF.....
00000010: 0001 0000 ffdb 0043 0008 0606 0706 0508  ....C.....
00000020: 0707 0709 0908 0a0c 140d 0c0b 0b0c 1912  .....
00000030: 130f 141d 1a1f 1e1d 1a1c 1c20 242e 2720  .... $. '
00000040: 222c 231c 1c28 3729 2c30 3134 3434 1f27  ",#..(7),01444.'
```

The interesting output of **istat** is highlighted in red. We can see that the inode is allocated and the data can be found in the direct blocks specified at the bottom. When viewed with **xxd** and **head** we see the expected signature of a JPEG image.

...and now for unallocated inode 21:

```
barry@forensicbox:able_3$ istat -o 104448 able3.000 21
inode: 21
Not Allocated
Group: 0
Generation Id: 1815721464
uid / gid: 1000 / 100
mode: rrw-r--r--
Flags: Extents,
size: 0
num of links: 0

Inode Times:
Accessed: 2017-05-08 00:18:16 (EDT)
File Modified: 2017-05-08 00:22:58 (EDT)
Inode Modified: 2017-05-08 00:22:58 (EDT)
Deleted: 2017-05-08 00:22:58 (EDT)
```

Direct Blocks:

```
barry@forensicbox:able_3$  icat -o 104448 able3.000 21 | xxd | head -n 5
<no output>
```

Here we have a different outcome. Inode 21 points to an unallocated file. On an ext4 file system, when an inode is unallocated the entry for the **Direct Blocks** is cleared. There are no longer pointers to the data, so commands like **icat** will not work. Remember that **icat** works at the inode (file meta-data) layer. The **icat** command uses the information found in the inode to recover the file. In this case there is none.

This does not mean we cannot recover the data that was there. On the contrary, there are a number of techniques we can use to attempt to recover the deleted files. But in this case it becomes far more difficult to recover the data and associate it with a particular file name and inode information. While this sort of forensic analysis is outside the scope of our exercise, it does highlight the difference between using these tools on two different file systems. And that is the point: Know your tools, their capabilities, and their limits.

When we test tools for forensic use, it is not enough to say "X tool does not work on Y file system". You should understand why. In this case it would be accurate to say that "icat works as expected on an ext4 file system, but is of limited use on deleted entries". Be sure to understand the difference, and test your tools!

9.3.3 Sleuth Kit Exercise 2A: Physical String Search & Allocation Status (ext2)

We did a very basic recovery of a physical string search on our **fat_fs.raw** file system image earlier in this document. This exercise is meant to take some of what we learned there and apply it to a more complex disk image with additional challenges. In a normal examination you are going to want to find out (if possible) what file a positive string search result belonged to and whether or not that file is allocated or unallocated. That is the purpose of this exercise.

Exercises like this highlight very clearly the benefit of learning digital forensics with tools like the Sleuth Kit. Unlike most GUI forensic tools with menus and multiple windows, TSK forces you to understand these concepts behind the tools. You cannot use TSK without understanding which tools to use and when. Without knowing the concepts behind the tools, you don't get very far.

Back to our **able2** image. This time we are going to do a search for a single string in **able2.dd**. In this case we will search our image for the keyword **Cybernetik**. Change to the directory containing our **able2.dd** image and use **grep** to search for the string:

```
barry@forensicbox:~$  cd able2
```

```
barry@forensicbox:able2$ grep -abi cybernetik able2.dd
10561603: * updated by Cybernetik for linux rootkit
55306929: Cybernetik proudly presents...
55312943: Email: cybernetik@nym.alias.net
55312975: Finger: cybernetik@nym.alias.net
```

Recall that our **grep** command is taking the file **able2.dd** treating it as a text file (**-a**) and searching for the string **cybernetik**. The search is case-insensitive (**-i**) and will output the byte offset of any matches (**-b**).

Our output shows that the first match comes at byte offset **10561603**. Like we did in our first string search exercise, we are going to quickly view the match by using our hex viewer **xxd** and using the **-s** option to provide the offset given by **grep**. We will also use the **head** command to indicate that we only want to see a specific number of lines, in this case just 5 (**-n 5**). We just want to get a quick look at the context of the match before proceeding.

```
barry@forensicbox:able2$ xxd -s 10561603 able2.dd | head -n 5
00a12843: 202a 0975 7064 6174 6564 2062 7920 4379  *.updated by Cy
00a12853: 6265 726e 6574 696b 2066 6f72 206c 696e  bernetik for lin
00a12863: 7578 2072 6f6f 746b 6974 0a20 2a2f 0a0a  ux rootkit. */..
00a12873: 2369 6e63 6c75 6465 203c 7379 732f 7479  #include <sys/ty
00a12883: 7065 732e 683e 0a23 696e 636c 7564 6520  pes.h>.#include
```

We also have to keep in mind that what we have found is the offset to the match in the entire disk (**able2.dd** is a full disk image), not in a specific file system. In order to use the Sleuth Kit tools, we need to have a file system to target.

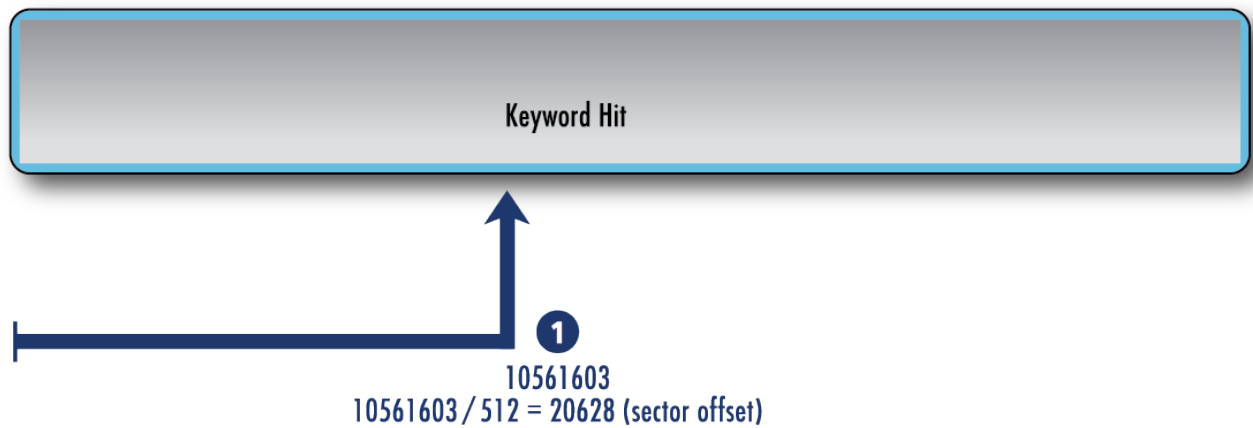
Let's figure out which partition (and file system) the match is in. Use **bc** to calculate which sector of the image and therefore the original disk the keyword is in. Each sector is 512 bytes, so dividing the byte offset by 512 tells us which sector:

```
barry@forensicbox:able2$ echo "10561603/512" | bc
```

The Sleuth Kit's **mmls** command gives us the offset to each partition in the image:

```
barry@forensicbox:able2$ mmls able2.dd
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors
```

	Slot	Start	End	Length	Description
000:	Meta	0000000000	0000000000	0000000001	Primary Table (#0)
001:	-----	0000000000	0000000056	0000000057	Unallocated
002:	000:000	0000000057	0000010259	0000010203	Linux (0x83)

Figure 9: The sector offset in `able2.dd` where the search hit was found.**Able2.dd (entire image)**

003:	000:001	0000010260	0000112859	0000102600	Linux (0x83)
004:	000:002	0000112860	0000178694	0000065835	Linux Swap / Solaris x86 (0
					→ x82)
005:	000:003	0000178695	0000675449	0000496755	Linux (0x83)

From the output of `mmfs` above, we see that our calculated sector, **20628**, falls in the second partition (between 10260 and 112859). The offset to our file system for the Sleuth Kit commands will be 10260.

The problem is that the offset that we have is the keyword's offset in the disk image, not in the file system (which is what the volume data block is associated with). So we have to calculate the offset to the file AND the offset to the partition that contains the file. The offset to the partition is simply a matter of multiplying the sector offset by the size of the sector for our file system:

```
barry@forensicbox:able2$ echo "10260*512" | bc
5253120
```

The difference between the two is the *volume offset* of the keyword hit, instead of the physical disk (or image) offset.

```
barry@forensicbox:able2$ echo "10561603-5253120" | bc
5308483
```

Figure 10: The volume offset in `able2.dd` where the search hit was found.

Able2.dd (entire image)

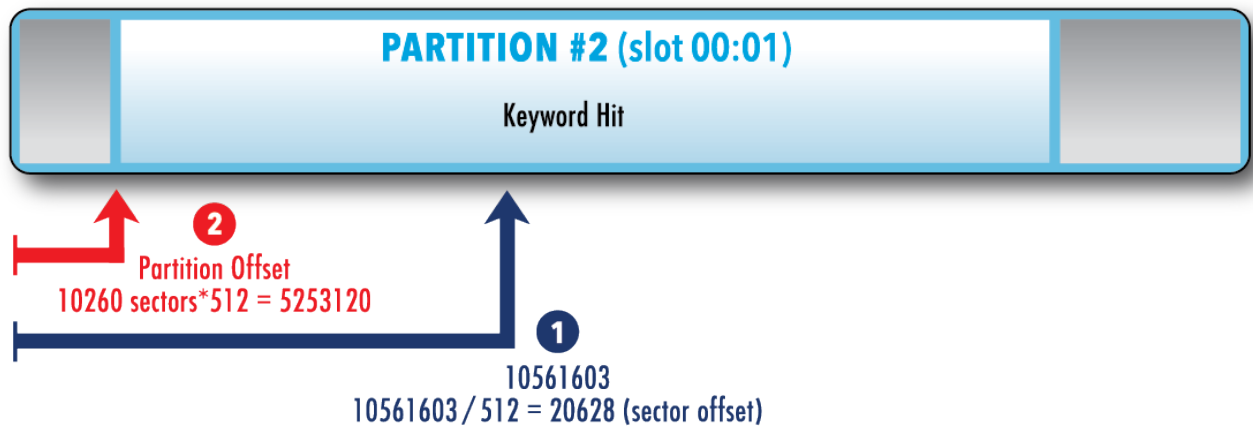
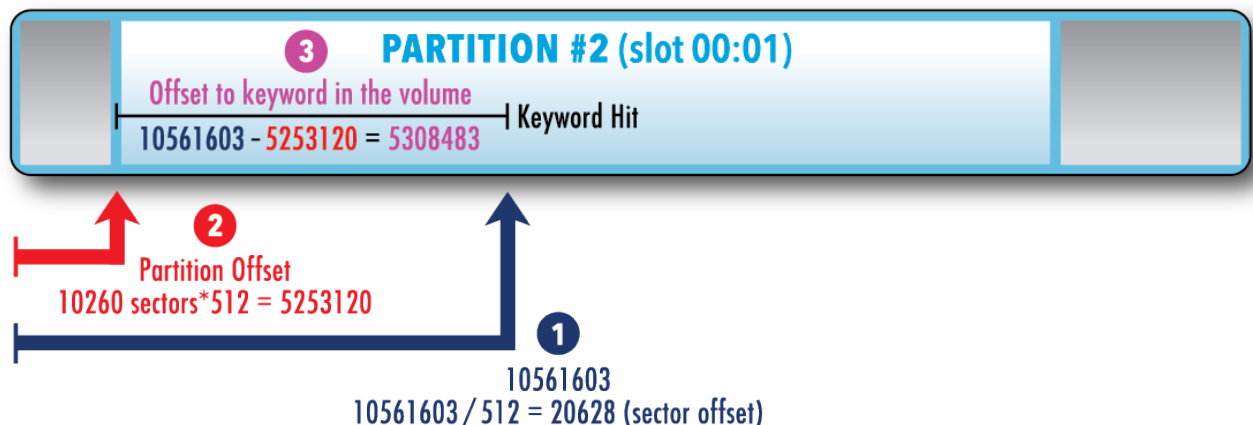


Figure 11: The offset to the keyword in the volume

Able2.dd (entire image)



Now we know the offset to the keyword within the actual volume, rather than the entire image. Let's find out what inode (meta-data unit) points to the volume data block at that offset. To find which inode this belongs to, we first have to calculate the volume data block address. Look at the Sleuth Kit's **fsstat** output to see the number of bytes per block. We need to run **fsstat** on the file system at sector offset **10260**:

```
barry@forensicbox:able2$ fsstat -o 10260 able2.dd
```

```
FILE SYSTEM INFORMATION
```

```
-----
```

```
File System Type: Ext2
```

```
Volume Name:
```

```
Volume ID: 906e777080e09488d0116064da18c0c4
```

...

CONTENT INFORMATION

Block Range: 0 - 51299

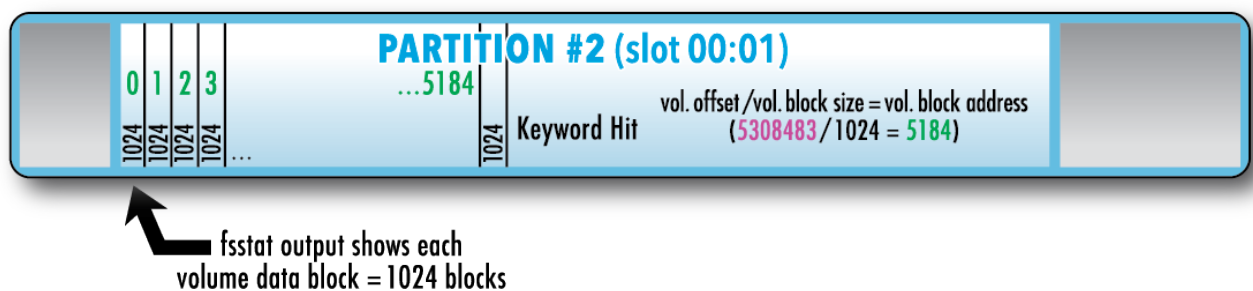
Block Size: 1024

The abbreviated **fsstat** output above shows us (highlighted in bold) that the data blocks within the volume are **1024** bytes each. If we divide the volume offset by **1024**, we identify the data block that holds the keyword hit.

```
barry@forensicbox:able2$ echo "5308483/1024" | bc
5184
```

Figure 12: Identifying the data block of the keyword

Able2.dd (entire image)



Here are our calculations, summarized:

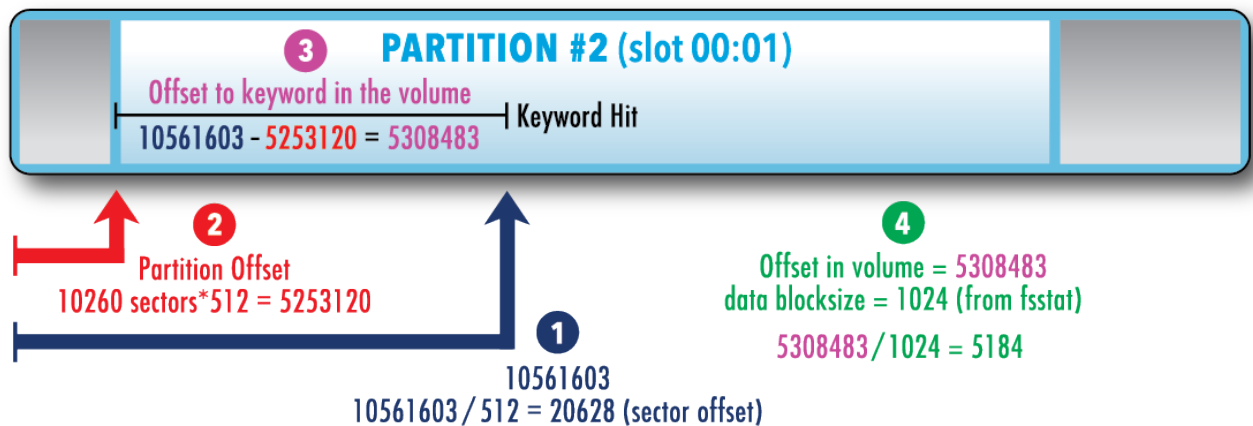
- offset to the string in the disk image (from our **grep** output): **10561603**
- offset to the partition that contains the file: **10260** sectors * **512** bytes per sector
- offset to the string in the partition is the difference between the two above numbers.
- the data block is the offset in the file system divided by the block size, (data unit size) **1024**, from our **fsstat** output.

In short, our calculation, taking into account all the illustrations above, is simply:

```
barry@forensicbox:able2$ echo "(10561603-(10260*512))/1024" | bc
5184
```

Note that we use parentheses to group our calculations. We find the byte offset to the file system first (**10260*512**), subtract that from the offset to the string (**10561603**) and then divide the whole thing by the data unit size (**1024**) obtained from **fsstat**. This (**5184**) is our

Figure 13: Single calculation to identify the data block of the keyword
Able2.dd (entire image)



data unit (not the inode!) that contains the string we found with **grep**. Very quickly, we can ascertain its allocation status with the Sleuth Kit command **blkstat**

```
barry@forensicbox:able2$ blkstat -o 10260 able2.dd 5184
Fragment: 5184
Not Allocated
Group: 0
```

The command **blkstat** takes a data block from a file system and tells us what it can about its status and where it belongs. We'll cover the TSK **blk** tools in more detail later. So in this case, **blkstat** tells us that our key word search for the string **cybernetik** resulted in a match in an unallocated block. Now we use **ifind** to tell us which inode (meta-data structure) points to data block 5184 in the second partition of our image:

```
barry@forensicbox:able2$ ifind -o 10260 -d 5184 able2.dd
10090
```

Excellent! The inode that holds the keyword match is **10090**. Now we use **istat** to give us the statistics of that inode:

```
barry@forensicbox:able2$ istat -o 10260 able2.dd 10090
inode: 10090
Not Allocated
Group: 5
Generation Id: 3534950782
uid / gid: 4 / 7
mode: rrw-r--r--
size: 3591
num of links: 0
```

Inode Times:

```
Accessed: 2003-08-10 00:18:36 (EDT)
File Modified: 1996-12-25 16:27:43 (EST)
Inode Modified: 2003-08-10 00:29:58 (EDT)
Deleted: 2003-08-10 00:29:58 (EDT)
```

Direct Blocks:

```
5184 5185 5186 5187
```

From the **istat** output we see that inode **10090** is unallocated (same as **blkstat** told us about the data unit). Note also that the first direct block indicated by our **istat** output is **5184**, just as we calculated.

We can get the data from the direct blocks of the original file by using **icat -r**. Pipe the output through **less** so that we can read it easier. Note that our keyword is right there at the top:

```
barry@forensicbox:able2$ icat -o 10260 able2.dd 10090 | less
/*
 * fixer.c
 * by Idefix
 * inspired on sum.c and SaintStat 2.0
 * updated by Cybernetik for linux rootkit
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <stdio.h>
...
```

At this point, we have recovered the data we were looking for. We can run our **icat** command as above again, this time directing the output to a file (as we did with the rootkit file from our previous recovery exercise). We'll do that here for possible later reference:

```
barry@forensicbox:able2$ cat -o 10260 able2.dd 10090 > 10090.recover

barry@forensicbox:able2$ ls -l 10090.recover
-rw-r--r-- 1 barry users 3591 Jul 30 06:47 10090.recover

barry@forensicbox:able2$ md5sum 10090.recover
c3b01f91d3fa72b1b951e6d6d45c7d9a 10090.recover
```

One additional note: the Sleuth Kit provides a virtual directory that contains entries for

orphan files. As we previously noted, in our discussion of the **fls** command, these files are the result of an inode containing file data having no file name (directory entry) associated with it. Sleuth Kit organizes these in the virtual `$OrphanFiles` directory. This is a useful feature because it allows us to identify and access orphan files from the output of the **fls** command.

In this exercise, we determined through our calculations that we were looking for the contents of inode `10090`. The Sleuth Kit command **ffind** can tell us the file name associated with an inode. Here, we are provided with the `$OrphanFiles` entry:

```
barry@forensicbox:able2$ ffind -o 10260 able2.dd 10090
* /$OrphanFiles/OrphanFile-10090
```

Remember that various file systems act very differently. We'll continue to explore the differences between ext2 and ext4 here in the next exercise. Much like TSK exercise #1, we are going to do the same set of steps on the `able_3` image and see what we get.

9.3.4 Sleuth Kit Exercise 2B: Physical String Search & Allocation Status (ext4)

Much like TSK exercise #1, we are going to repeat our steps here for the ext4 image in `able_3.000`. Again, we are illustrating the differences in output for our tools based on the type of file system being analyzed so that we can recognize the difference file system behavior makes in our output. No diagrams this time. You should be familiar with the commands we are going to use here. The goal is to show the output we can expect at the end, and how we can perhaps deal with it.

Change back into the `able_3/` directory where the `able_3` image set is stored. In the `able2` exercise we did a full disk search for the term `cybernetik`. In this case we have a set of split images. We know the Sleuth Kit tools work on the split files, but how do I **grep** the entire disk when I have split images? As we mentioned in our previous `able_3` exercise, we can use **affuse** to provide a fuse mounted full disk image for us. In this case, however, I don't need a full disk image except for the **grep** command. And since **grep** will take input from stdin (through a pipe), why not just stream the images through a pipe to **grep** so they appear as a single image - without a preliminary step? That is what we do here, searching for the same term as we did before:

```
barry@forensicbox:able_3$ cat able_3.00*| grep -abi cybernetik
429415089:Cybernetik proudly presents...
429422127:Email: cybernetik@nym.alias.net
429422159:Finger: cybernetik@nym.alias.net
1632788547: *updated by Cybernetik for linux rootkit
2934551933:23140 Cybernetik.net
```

We use the **cat** command to stream our split files to **grep** for our search. This is no different than reconstructing the file (creating a single image with **cat >**), but instead we just pass the output of **cat** straight to **grep**. The command may take longer, and the keyword offsets are obviously different than in **able2** because this is a larger and different image. We are going to concentrate on the same match we used for our **able2** ext2 exercise. That would be the keyword hit at **1632788547**.

Remember our steps from here. We need to calculate the offset in sectors (divide by **512**), then calculate the offset to the volume we found the keyword in, and then subtract the volume offset from the keyword offset to find the offset to the string in the volume. Make sure we calculate using the correct block size for the file system. Remember we are working with data blocks here. The **ffstat** command will give you the proper size for this file system.

We end up with the numbers below. Review the previous exercise if you have any questions on the steps taken:

```
barry@forensicbox:able_3$ echo $((1632788547/512))
3189040
```

```
barry@forensicbox:able_3$ mmls able_3.000
GUID Partition Table (EFI)
Offset Sector: 0
Units are in 512-byte sectors
```

	Slot	Start	End	Length	Description
000:	Meta	0000000000	0000000000	0000000001	Safety Table
001:	-----	0000000000	0000002047	0000002048	Unallocated
002:	Meta	0000000001	0000000001	0000000001	GPT Header
003:	Meta	0000000002	0000000033	0000000032	Partition Table
004:	000	0000002048	0000104447	0000102400	Linux filesystem
005:	001	0000104448	0000309247	0000204800	Linux filesystem
006:	-----	0000309248	0000571391	0000262144	Unallocated
007:	002	0000571392	0008388574	0007817183	Linux filesystem
008:	-----	0008388575	0008388607	0000000033	Unallocated

```
barry@forensicbox:able_3$ fsstat -o 571392 able_3.000 | less
```

```
FILE SYSTEM INFORMATION
```

```
-----
```

```
File System Type: Ext4
```

```
Volume Name:
```

```
Volume ID: dd9f5b9524f943aae944383ab248f7c7
```

```
...
```

```
CONTENT INFORMATION
```

```
-----
```

```
Block Groups Per Flex Group: 16
```

```
Block Range: 0 - 977146
```

```
Block Size: 4096
```

Free Blocks: 557639

...

```
barry@forensicbox:able_3$ echo "(1632788547-(571392*512))/4096" | bc
327206
```

We're ready to run our **blkstat** command to find out if our keyword hit is in a block assigned to an allocated inode:

```
barry@forensicbox:able_3$ blkstat -o 571392 able_3.000 327206
Fragment: 327206
Not Allocated
Group: 9
```

So the block is unallocated. Let's now see if we can find what inode this unallocated block belonged to:

```
barry@forensicbox:able_3$ ifind -o 571392 -d 327206 able_3.000
Inode not found
```

And there's our answer. The inode cannot be found. Again this is because the inodes in ext4 that are unallocated have the direct block pointers deleted. The **ifind** command is searching for a pointer to the data unit (**-d 327206**) in the inode table and cannot find one.

All is not lost, though. Instead of using **icat** to extract the data blocks pointed to by an inode, we can instead use **blkcat** to directly stream the contents of a data block. Have a look below. We'll use **blkcat** and redirect to a file:

```
barry@forensicbox:able_3$ blkcat -o 571392 able_3.000 327206 > blk.327206

barry@forensicbox:able_3$ ls -l blk.327206
-rw-r--r-- 1 barry users 4096 Jul 30 07:31 blk.327206
```

Look at the file with **cat** or **less**. You'll see it is the same file as the one we recovered from **able2**. It has some garbage at the end, though. Why is that? Remember when we recovered this same file from **able2** with **icat**? **icat** had the information it needed to do a complete recovery of the correct data. We don't have that here, and all we did was stream ("block cat") a single block of data (that we know is 4096 bytes from our **fsstat** output) and save the whole thing. Remember our output from the **able2** exercise prior to this (the below commands are run in the **able2** directory - note the command prompt):

```
barry@forensicbox:able2$ ls -l 10090.recover
-rw-r--r-- 1 barry users 3591 Jul 30 06:47 10090.recover
```

```
barry@forensicbox:able2$ md5sum 10090.recover
c3b01f91d3fa72b1b951e6d6d45c7d9a 10090.recover
```

The above is from the **able2** disk image. Look at the size of the file. **3591** bytes. Now, realistically we would not have this information available for us in a real exam, but just for fun, let us see if we can make the files match using the size of the file from our **able2** recovery as a go-by. Since the file from **able_3** is bigger, we can use **dd** to cut the correct data from it. The file is currently **4096** bytes in size. We need it to be **3591** bytes:

```
barry@forensicbox:able_3$ dd if=blk.327206 bs=1 count=3591 > 327206.recover
3591+0 records in
3591+0 records out
3591 bytes (3.6 kB, 3.5 KiB) copied, 0.00413113 s, 869 kB/s
```

```
barry@forensicbox:able_3$ md5sum 327206.recover
c3b01f91d3fa72b1b951e6d6d45c7d9a 327206.recover
```

Compare the output of **md5sum** from **327206.recover** with that of **able2/10090.recover**. Look at that! The **md5sum** of the file we recovered from **able2** with **icat** now matches the file we recovered using **blkcat** in **able_3**. Again, not quite realistic, but it serves to illustrate exactly what data we are getting and why. Hopefully there is some educational value for you there.

9.3.5 Sleuth Kit Exercise 3: Unallocated Extraction & Examination

As the size of media being examined continues to grow, it is becoming apparent to many investigators that data reduction techniques are more important than ever. These techniques take on several forms, including hash analysis (removing known "good" files from a data set, for example) and separating allocated space in an image from unallocated space, allowing them to be searched separately with specialized tools. We will be doing the latter in this exercise.

The **blkcat** command we used earlier is a member of the Sleuth Kit set of tools for handling information at the "block" layer of the analysis model. The block layer consists of the actual file system blocks that hold the information we are seeking. They are not specific to unallocated data only, but are especially useful for working on unallocated blocks that have been extracted from an image. The tools that manipulate this layer, as you would expect, start with **blk** and include:

```
blkls
blkcalc
blkstat
blkcat
```

We will be focusing on **blkls**, **blkcalc** and **blkstat** for the next couple of exercises.

The tool that starts us off here is **blkls**. This command "lists all the data blocks". If you were to use the **-e** option, the output would be the same as the output of **dd** for that volume, since **-e** tells **blkls** to copy /emph every block. However, by default, **blkls** will only copy out the *unallocated blocks* of an image.

This allows us to separate allocated and unallocated blocks in our file system. We can use logical tools (**find**, **ls**, etc.) on the "live" files in a mounted file system, and concentrate data recovery efforts on only those blocks that may contain deleted or otherwise unallocated data. Conversely, when we do a physical search of the output of **blkls**, we can be sure that artifacts found are from unallocated content.

To illustrate what we are talking about here, we'll run the same exercise we did in TSK Exercise #2A, this time extracting the unallocated data from our volume of interest and comparing the output from the whole volume analysis vs. just unallocated analysis. So, we'll be working on the **able2.dd** image. We expect to get the same results we did in Exercise #2A, but this time by analyzing only the unallocated space, and then associating the recovered data with its original location in the full disk image.

First we'll need to change into the directory containing our **able2.dd** image. Then we check the partition table and decide which volume we'll be examining so we know the **-o** (offset) value for our Sleuth Kit commands. To do this, we run the **mmls** command as before:

```
barry@forensicbox:~$ cd able2
```

```
barry@forensicbox:able2$ mmls able2.dd
```

```
DOS Partition Table
```

```
Offset Sector: 0
```

```
Units are in 512-byte sectors
```

	Slot	Start	End	Length	Description
000:	Meta	0000000000	0000000000	0000000001	Primary Table (#0)
001:	-----	0000000000	0000000056	0000000057	Unallocated
002:	000:000	0000000057	0000010259	0000010203	Linux (0x83)
003:	000:001	0000010260	0000112859	0000102600	Linux (0x83)
004:	000:002	0000112860	0000178694	0000065835	Linux Swap / Solaris x86 (0
	↪ x82)				
005:	000:003	0000178695	0000675449	0000496755	Linux (0x83)

As with Exercise #2, we've decided to search the unallocated space in the second Linux partition (at offset **10260**, in bold above).

We run the **blkls** command using the offset option **-o** which indicates what partition's file system we are exporting the unallocated space from. We then redirect the output to a new file that will contain only the unallocated blocks of that particular volume.

```
barry@forensicbox:able2$ blkls -o 10260 able2.dd > able2.blkls
```

```
barry@forensicbox:able2$ ls -lh able2.blkls
-rw-r--r-- 1 barry users 9.3M Jul 30 09:37 able2.blkls
```

In the above command, we are using **blkls** on the second partition (**-o 10260**) within the **able2.dd** image, and redirecting the output to a file called **able2.blkls**. The file **able2.blkls** will contain only the unallocated blocks from the target file system. In this case we end up with a file that is **9.3M** in size.

Now, as we did in our previous analysis of this file system (Exercise #2) we will use **grep**, this time on the extracted unallocated space, our **able2.blkls** file, to search for our text string of interest. Read back through Exercise #2 if you need a refresher on these commands.

```
barry@forensicbox:able2$ grep -abi cybernetik able2.blkls
1631299: * updated by Cybernetik for linux rootkit
9317041:Cybernetik proudly presents...
9323055:Email: cybernetik@nym.alias.net
9323087:Finger: cybernetik@nym.alias.net
```

The **grep** command above now tells us that we have found the string **cybernetik** at four different offsets in the extracted unallocated space. We will concentrate on the first hit. Of course these are different from the offsets we found in Exercise #2 because we are no longer searching the entire original image.

So the next obvious question is "so what?". We found potential evidence in our extracted unallocated space. But how does it relate to the original image? As forensic examiners, merely finding potential evidence is not good enough. We also need to know where it came from: physical location in the original image, what file it belongs or (possibly) belonged to, meta data associated with the file, and context. Finding potential evidence in a big block of aggregate unallocated space is of little use to us if we cannot at least make some effort at attribution in the original file system.

That's where the other block layer tools come in. We can use **blkcalc** to calculate the location (by data block or fragment) in our original image. Once we've done that, we simply use the meta data layer tools to identify and potentially recover the original file, as we did in our previous effort.

First we need to gather a bit of data about the original file system. We run the **fsstat** command to determine the size of the data blocks we are working with. We've done this a number of times already, but the repetition is useful to drive home the importance of this information.

```
barry@forensicbox:able2$ fsstat -o 10260 able2.dd | less
```

FILE SYSTEM INFORMATION

```
-----  
File System Type: Ext2  
Volume Name:  
Volume ID: 906e777080e09488d0116064da18c0c4  
...
```

CONTENT INFORMATION

```
-----  
Block Range: 0 - 51299  
Block Size: 1024  
...
```

In the **fsstat** command above, we see that the block size (in bold) is **1024**. We take the offset from our **grep** output on the **able2.blkls** image and divide that by **1024**. This tells us how many unallocated data blocks into the unallocated image we found our string of interest. As usual, we use the **echo** command to pass the math expression to the command line calculator, **bc**:

```
barry@forensicbox:able2$ echo "1631299/1024" | bc  
1593
```

We now know, from the above output, that the string **cybernetik** is in data block **1593** of our extracted unallocated file, **able2.blkls**.

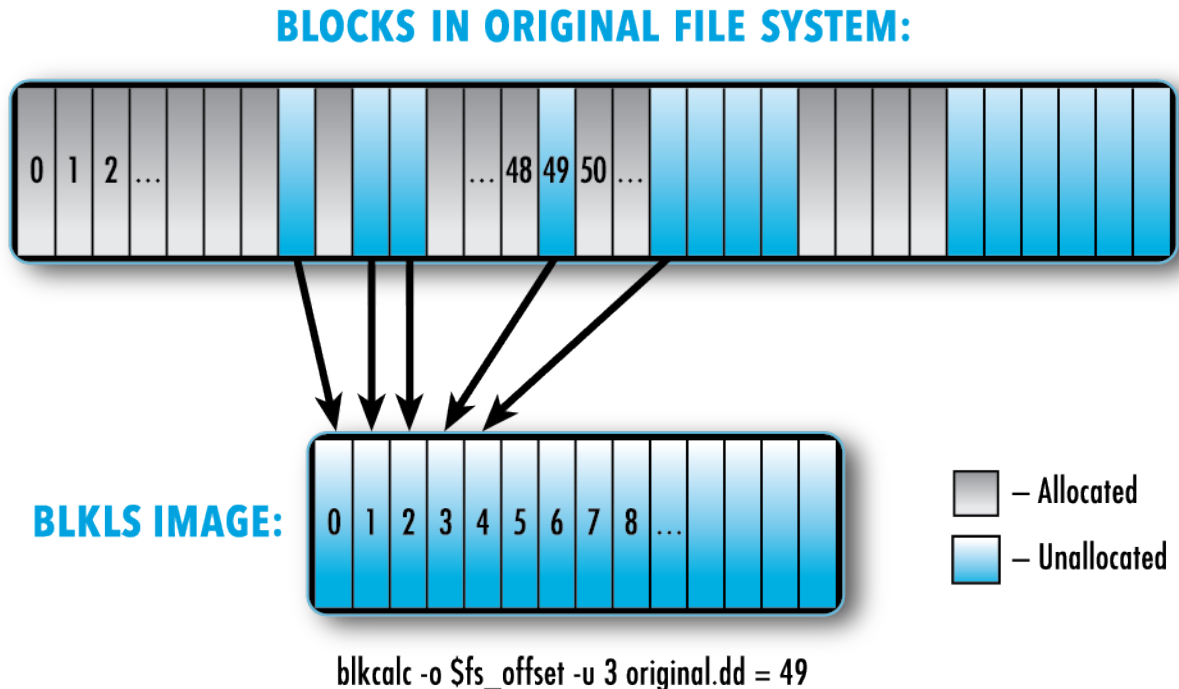
This is where our handy **blkcalc** command comes in. We use **blkcalc** with the **-u** option to specify that we want to calculate the block address from an extracted unallocated image (from **blkls** output). We run the command on the original **dd** image because we are calculating the original data block in that image. The question we are answering here is "What data block in the original image is unallocated block 1593?".

```
barry@forensicbox:able2$ blkcalc -o 10260 -u 1593 able2.dd  
5184
```

The command above is running **blkcalc** on the file system at offset **10260** (**-o 10260**) in the original **able2.dd**, passing the data block we calculated from the **blkls** image **able2.blkls** (**-u 1593**). The result is a familiar block **5184** (see Exercise #2A again). The illustration below gives a visual representation of a simple example: In the illustrated example above, the data in block **3** of the **blkls** image would map to block **49** in the original file system. We would find this with the **blkcalc** command as shown in the figure.

So, in simple terms, we have extracted the unallocated space, found a string of interest in a data block in the unallocated image, and then found the corresponding data block in the original image.

Figure 14: A simple example of how **blkcalc** is used to determine the original address of an unallocated data unit



If we look at the **blkstat** (data block statistics) output for block 5184 in the original image, we see that it is, in fact unallocated, which makes sense, since we found it within our extracted unallocated space (we're back to the same results as in Exercise #2A). Note that we are now running the commands on the original **dd** image. We'll continue on for the sake of completeness. And because it's good practice...

```
barry@forensicbox:able2$ blkstat -o 10260 able2.dd 5184
Fragment: 5184
Not Allocated
Group: 0
```

Using the command **blkcat** we can look at the raw contents of the data block (using **xxd** and **less** as a viewer). If we want to, we can even use **blkcat** to extract the block, redirecting the contents to another file, just as we did in exercise #2B with our ext4 file system image.

If we want to recover the actual file and meta data associated with the identified data block, we use **ifind** to determine which meta data structure (in this case inode since we are working on an EXT file system) holds the data in block 5184. Then **istat** shows us the meta data for the inode:

```
barry@forensicbox:able2$ ifind -o 10260 -d 5184 able2.dd
10090
```

```
barry@forensicbox:able2$ istat -o 10260 able2.dd 10090
```

```
inode: 10090
Not Allocated
Group: 5
Generation Id: 3534950782
uid / gid: 4 / 7
mode: rrw-r--r--
size: 3591
num of links: 0
```

Inode Times:

```
Accessed: 2003-08-10 00:18:36 (EDT)
File Modified: 1996-12-25 16:27:43 (EST)
Inode Modified: 2003-08-10 00:29:58 (EDT)
Deleted: 2003-08-10 00:29:58 (EDT)
```

Direct Blocks:

```
5184 5185 5186 5187
```

Again, as we saw previously, the **istat** command, which shows us the meta data for inode **10090**, indicates that the file with this inode is **Not Allocated**, and its first direct block is **5184**. Just as we expected.

We then use **icat** to recover the file. In this case, we just pipe the first few lines out to see our string of interest, **cybernetik**.

```
barry@forensicbox:able2$ icat -o 10260 able2.dd 10090 | head -n 10
```

```
/*
 * fixer.c
 * by Idefix
 * inspired on sum.c and SaintStat 2.0
 * updated by Cybernetik for linux rootkit
 */
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
```

9.3.6 SleuthKit Exercise 4: NTFS Examination - File Analysis

At this point we've done a couple of intermediate exercises using ext2 and ext4 file systems from Linux disk images. In the following exercises we will do some simple analyses on an

NTFS file system. This is one of the more common file system you are likely to find when it comes to personal and enterprise computers today.

Some might ask, "why?" There are many tools out there capable of analyzing an NTFS file system in its native environment. In my mind there are two very good reasons for learning to apply the Sleuth Kit on Windows file systems. First, the Sleuth Kit is comprised of a number of separate tools with very discrete sets of capabilities. The specialized nature of these tools means that you have to understand their interaction with the file system being analyzed. This makes them especially suited to help learning the ins and outs of file system behavior. The fact that the Sleuth Kit does less of the work for you makes it a great learning tool. Second, an open source tool that operates in an environment other than Windows makes for an excellent cross-verification utility.

The following exercise demonstrates a set of very basic steps useful in most any analysis. Make sure that you follow along at the command line. Experimentation is the best way to learn.

If you have not already done so, I would strongly suggest (again) that you invest in a copy of Brian Carrier's book: File System Forensic Analysis (Published by Addison-Wesley, 2005). This book is the definitive guide to file system behavior for forensic analysts. As a reminder (again), the purpose of these exercises is NOT to teach you file systems or forensic methods, but rather to illustrate and introduce the detailed information TSK can provide on common file systems encountered by field examiners.

For these exercises that follow, we'll be using the `NTFS_Pract_2017.E01` set of files we downloaded and used for our `libewf` sections earlier. Since these are EWF files, and we have support for `libewf` built into TSK, we'll work directly from those files. If you have not already done so, download the NTFS EWF files, extract the archive and let's begin.

```
barry@forensicbox:~$ wget http://www.linuxleo.com/Files/NTFS_Pract_2017_E01.tar.gz
...
barry@forensicbox:~$ tar tzf NTFS_Pract_2017_E01.tar.gz
...
barry@forensicbox:~$ tar xzvf NTFS_Pract_2017_E01.tar.gz
NTFS_Pract_2017/
NTFS_Pract_2017/NTFS_Pract_2017.E04
NTFS_Pract_2017/NTFS_Pract_2017.E02
NTFS_Pract_2017/NTFS_Pract_2017.E01
NTFS_Pract_2017/NTFS_Pract_2017.E03

barry@forensicbox:~$ cd NTFS_Pract_2017
```

We will start by running through a series of basic Sleuth Kit commands as we would in any analysis. The structure of the forensic image is viewed using `mmfs`:

```
barry@forensicbox:NTFS_Pract_2017$ mmls NTFS_Pract_2017.E01
```

```
DOS Partition Table
```

```
Offset Sector: 0
```

```
Units are in 512-byte sectors
```

	Slot	Start	End	Length	Description
000:	Meta	00000000000	00000000000	00000000001	Primary Table (#0)
001:	-----	00000000000	0000002047	0000002048	Unallocated
002:	000:000	0000002048	0001023999	0001021952	NTFS / exFAT (0x07)

The output shows that an NTFS partition (and most likely the file system) begins at sector offset **2048**. This is the offset we will use in all our Sleuth Kit commands. We now use **fsstat** to have a look at the file system statistics inside that partition:

```
barry@forensicbox:NTFS_Pract_2017$ fsstat -o 2048 NTFS_Pract_2017.E01 | less
```

```
FILE SYSTEM INFORMATION
```

```
-----
```

```
File System Type: NTFS
```

```
Volume Serial Number: CAE0DFD2E0DFC2BD
```

```
OEM Name: NTFS
```

```
Volume Name: NTFS_2017d
```

```
Version: Windows XP
```

```
METADATA INFORMATION
```

```
-----
```

```
First Cluster of MFT: 42581
```

```
First Cluster of MFT Mirror: 2
```

```
Size of MFT Entries: 1024 bytes
```

```
Size of Index Records: 4096 bytes
```

```
Range: 0 - 293
```

```
Root Directory: 5
```

```
CONTENT INFORMATION
```

```
-----
```

```
Sector Size: 512
```

```
Cluster Size: 4096
```

```
...
```

Looking at the **fsstat** output on our NTFS file system, we see it differs greatly from the output we saw running on a Linux EXT file system. The tool is designed to provide pertinent information based on the file system being targeted. Notice that when run on an NTFS file system, **fsstat** provides us with information specific to NTFS, including data about the Master File Table (MFT) and specific attribute values.

We will now have a look at how the Sleuth Kit interacts with active and deleted files on an NTFS file system. Let's first run **fls** on just the root level directory of our image:

```
barry@forensicbox:NTFS_Pract_2017$ fls -o 2048 NTFS_Pract_2017.E01
r/r 4-128-4:  $AttrDef
r/r 8-128-2:  $BadClus
r/r 8-128-1:  $BadClus:$Bad
r/r 6-128-4:  $Bitmap
r/r 7-128-1:  $Boot
d/d 11-144-4: $Extend
r/r 2-128-1:  $LogFile
r/r 0-128-6:  $MFT
r/r 1-128-1:  $MFTMirr
r/r 9-128-8:  $Secure:$SDS
r/r 9-144-11: $Secure:$SDH
r/r 9-144-14: $Secure:$SII
r/r 10-128-1: $UpCase
r/r 10-128-4: $UpCase:$Info
r/r 3-128-3:  $Volume
r/r 38-128-1: ProxyLog1.log
d/d 35-144-1: System Volume Information
d/d 64-144-2: Users
d/d 67-144-2: Windows
V/V 293:     $OrphanFiles
```

Note that **fls** displays far more information for us than normal directory listings for NTFS. Included with our regular files and directories are the NTFS system files (starting with the \$), including the \$MFT and \$MFTMIRROR (record numbers 0 and 1). If you look at the MFT numbers, you will see that for some reason record number 5 is missing. MFT record 5 is the root directory, which is what we are displaying here. Just as the default display for EXT file systems with **fls** is inode 2, the default for NTFS is MFT record 5.

You can dig deeper and deeper into the file system by providing **fls** with a directory MFT record and it will display the contents of that directory. For illustration, re run the command (use the up arrow and edit the previous command) with the MFT record 64 (the **Users** directory):

```
barry@forensicbox:NTFS_Pract_2017$ fls -o 2048 NTFS_Pract_2017.E01 64
d/d 65-144-2:  AlbertE
d/d 66-144-2:  ElsaE
```

You can delve deep into each directory this way. This is one way to "browse" the file system with **fls**.

We can also specify that **fls** only show us only "deleted" content on the command line with the **-d** option. We will use **-F** (only file entries) and **-r** (recursive) as well:

```
barry@forensicbox:NTFS_Pract_2017$ fls -o 2048 -Frd NTFS_Pract_2017.E01
```

```

-/r * 40-128-1: Users/AlbertE/Documents/Credit Report.pdf
-/r * 40-128-3: Users/AlbertE/Documents/Credit Report.pdf:Zone.Identifier
r/- * 0:      Users/AlbertE/Documents/ManProj/World's First Atomic Bomb - Manhattan
    ↪ Project Documentary - Films - YouTube.url

-/r * 236-128-2:      Users/AlbertE/Documents/ManProj/MMManhattan Project.docx
-/r * 237-128-2:      Users/AlbertE/Documents/ManProj/The Manhattan Project - YouTube
    ↪ .url

-/r * 238-128-2:      Users/AlbertE/Documents/ManProj/World's First Atomic Bomb -
    ↪ Manhattan Project Documentary - Films - YouTube.url

-/r * 239-128-2:      Users/AlbertE/Documents/ManProj/manhattan_project.zip
-/r * 248-128-2:      Users/AlbertE/Documents/cyberbullying_by_proxy.doc
r/- * 0:      Users/AlbertE/Pictures/Tails/Thumbs.db
r/r * 221-128-2:      Users/AlbertE/Pictures/Tails/Thumbs.db
-/r * 216-128-2:      Users/AlbertE/Pictures/Tails/BigBikeBH1017.jpg
-/r * 217-128-2:      Users/AlbertE/Pictures/Tails/BigBikeSoloCBR900SC33.jpg
-/r * 218-128-2:      Users/AlbertE/Pictures/Tails/BigBikeTailBandit.jpg
-/r * 219-128-2:      Users/AlbertE/Pictures/Tails/GemoTailG4.jpg
-/r * 220-128-2:      Users/AlbertE/Pictures/Tails/GemoTailUniversal.jpg
r/- * 0:      Windows/Prefetch/EXPLORER.EXE-A80E4F97.pf
r/- * 0:      Windows/Prefetch/MAINTENANCESERVICE.EXE-28D2775E.pf
r/- * 0:      Windows/Prefetch/RUNDLL32.EXE-411A328D.pf
d/- * 0:      Windows/System32
-/r * 167-128-2:      Windows/Drop Location 2.kml
-/r * 168-128-2:      Windows/Drop location 1.kml
-/r * 169-128-2:      Windows/Meeting place.kml
-/r * 170-128-2:      Windows/Nums_to_use.txt
-/r * 171-128-2:      Windows/mycase.jpg
-/r * 172-128-2:      Windows/mycase.jpg_original
-/r * 173-128-2:      Windows/pickup location.kml

```

The output above shows that our NTFS example file system holds a number of deleted files in several directories. Let's have a closer look at some NTFS specific information that can be parsed with TSK tools.

Have a look at the deleted file at MFT entry 216. The file is `Users/AlbertE/Pictures/Tails ↪ /BigBikeBH1017.jpg`. We can have a closer look at the file's attributes by examining its MFT entry directly with **istat**. Recall that when we were working on an EXT file system previously, the output of **istat** gave us information directly from the inode of the specified file (see Sleuth Kit Exercise #1). So let's run the command on MFT entry 216 in our current exercise:

```

barry@forensicbox:NTFS_Pract_2017$ istat -o 2048 NTFS_Pract_2017.E01 216
MFT Entry Header Values:

```



```
Entry: 216          Sequence: 2
$LogFile Sequence Number: 4199136
Not Allocated File
Links: 1

$STANDARD_INFORMATION Attribute Values:
Flags: Archive
Owner ID: 0
Security ID: 0  ()
Created:    2017-05-01 09:04:42.810747600 (EDT)
File Modified: 2006-10-14 10:41:41.158486000 (EDT)
MFT Modified: 2017-05-01 09:04:42.818945100 (EDT)
Accessed:    2017-05-01 09:04:42.818865600 (EDT)
```

```
$FILE_NAME Attribute Values:
Flags: Archive
Name: BigBikeBH1017.jpg
Parent MFT Entry: 186   Sequence: 1
Allocated Size: 61440   Actual Size: 59861
Created:    2017-05-01 09:04:42.810747600 (EDT)
File Modified: 2006-10-14 10:41:41.158486000 (EDT)
MFT Modified: 2017-05-01 09:04:42.818865600 (EDT)
Accessed:    2017-05-01 09:04:42.818865600 (EDT)
```

```
Attributes:
Type: $STANDARD_INFORMATION (16-0)   Name: N/A   Resident   size: 48
Type: $FILE_NAME (48-4)              Name: N/A   Resident   size: 100
Type: $SECURITY_DESCRIPTOR (80-1)    Name: N/A   Resident   size: 80
Type: $DATA (128-2)                 Name: N/A   Non-Resident   size: 59861   init_size: 59861
91473 91474 91475 91476 91477 91478 91479 91480
91481 91482 91483 91484 91485 91486 91487
```

The information **istat** provides us from the MFT shows values directly from the **\$STANDARD_INFORMATION** attribute (which contains the basic meta data for a file) as well as the **\$FILE_NAME** attribute and basic information for other attributes that are part of an MFT entry. The data blocks that contain the actual file content are listed at the bottom of the output (for Non-Resident data).

Take note of the fact that there is a separate attribute identifier for the **\$FILE_NAME** attribute, **48-4**. It is interesting to note we can access the contents of each attribute separately using the **icat** command (more on this later).

The **48-4** attribute stores the file name. By piping the output of **icat** to **xxd** we can see the contents of this attribute, allowing us to view individual attributes for each MFT entry. By itself, this may not be of much investigative interest in this particular instance, but you should understand that attributes can be accessed separately by providing the full attribute

identifier.

```
barry@forensicbox:NTFS_Pract_2017$  icat -o 2048 NTFS_Pract_2017.E01 216-48-4 |
    xxd
00000000: ba00 0000 0000 0100 d486 cd7f 7bc2 d201  ....{...
00000010: 5cef 99dc 9eef c601 f0c3 ce7f 7bc2 d201  \.....{...
00000020: f0c3 ce7f 7bc2 d201 00f0 0000 0000 0000  ....{.....
00000030: d5e9 0000 0000 0000 2000 0000 0000 0000  .....
00000040: 1100 4200 6900 6700 4200 6900 6b00 6500  ..B.i.g.B.i.k.e.
00000050: 4200 4800 3100 3000 3100 3700 2e00 6a00  B.H.1.0.1.7...j.
00000060: 7000 6700                                     p.g.
```

The same idea is extended to other attributes of a file, most notably the "alternate data stream" or ADS. By showing us the existence of multiple attribute identifiers for a given file, the Sleuth Kit gives us a way of detecting potentially hidden data. We cover this in our next exercise.

9.3.7 Sleuth Kit Exercise 5: NTFS Examination of ADS

The NTFS file system allows for alternate data streams (ADS) - meaning a file in NTFS can have separate content that is not normally viewable with applications meant to display a given file type.

Obviously, when examining a system, it may be useful to get a look at all of the files contained in an image. We can do this two ways. We could simply loop mount our image and get a file listing. Or, we could use a forensic utility like TSK. We will compare the output here to illustrate an important difference, even with live (allocated) content.

Remember that the **mount** command works on file systems, not disks. The file system in this image starts 2048 sectors into the image, so we mount using an offset. Since we are also examining an EWF image, we'll need to use **ewfmount** to fuse mount the image file. We will accomplish this as root (and if it does not exist already, make sure you create `/mnt/ewf`):

```
barry@forensicbox:NTFS_Pract_2017$  su -
Password:

root@forensicbox:~#  cd  barry/NTFS_Pract_2017

root@forensicbox:NTFS_Pract_2017#  ewfmount NTFS_Pract_2017.E01 /mnt/ewf
ewfmount 20140806

root@forensicbox:NTFS_Pract_2017#  mount -o ro,loop,offset=$((2048*512))
    /mnt/ewf/ewf1 /mnt/evid

root@forensicbox:NTFS_Pract_2017#  ls /mnt/evid
```

```
ProxyLog1.log*  System\ Volume\ Information\  Users\  Windows/
```

```
root@forensicbox:NTFS_Pract_2017#  exit
logout
```

```
barry@forensicbox:NTFS_Pract_2017$
```

In the above set of commands, we **su** to root, use **ewfmount** to mount the EWF image on /mnt/ewf as /mnt/ewf/ewf1²⁸. We then mount the data partition (which we know is at offset 2048 from our previous exercise) and then **exit**.

We can then obtain a simple list of files using the **find** command:

```
barry@forensicbox:NTFS_Pract_2017$  find /mnt/ewf/ -type f
/mnt/ewf/ProxyLog1.log
/mnt/ewf/System Volume Information/IndexerVolumeGuid
/mnt/ewf/System Volume Information/WPSettings.dat
/mnt/ewf/Users/AlbertE/Documents/better_access_unix.txt
/mnt/ewf/Users/AlbertE/Documents/books.txt
/mnt/ewf/Users/AlbertE/Documents/cable.txt
/mnt/ewf/Users/AlbertE/Documents/cabletv.txt
/mnt/ewf/Users/AlbertE/Documents/hackcabl.txt
```

The **find** command, starts at the mount point (/mnt/ewf), looking for all regular files (**type -f**). The result gives us a very long list of all the allocated regular files on the mount point. That's quite a lot of files, so for the sake of this exercise let's just look at the contents of the user Albert's Pictures directory (use the same command, but **grep** for AlbertE/Pictures):

```
barry@forensicbox:NTFS_Pract_2017$  find /mnt/ewf/ -type f | grep
"AlbertE/Pictures"
/mnt/ewf/Users/AlbertE/Pictures/b45ac806a965017dd71e3382581c47f3_refined.jpg
/mnt/ewf/Users/AlbertE/Pictures/bankor1.jpg
/mnt/ewf/Users/AlbertE/Pictures/desktop.ini
/mnt/ewf/Users/AlbertE/Pictures/fighterama2005-ban3.jpg
/mnt/ewf/Users/AlbertE/Pictures/jet.mpg <Pay attention to this one
/mnt/ewf/Users/AlbertE/Pictures/pvannorden2.jpg
...
```

Of particular interest in this output is **jet.mpg**. Take note of this file. Our current method of listing files, however, gives us no indication of why this file is noteworthy.

The output of the **file** command shows us the expected file type. It is an MPEG video. You can play the video with the **mplayer** command from the command line to view it if you

²⁸You *can* use **ewfmount** as a normal user, but in this case we need to be root to loop mount anyway.

like.

```
barry@forensicbox:NTFS_Pract_2017$ file /mnt/evid/Users/AlbertE/Pictures/jet.mpg
/mnt/evid/Users/AlbertE/Pictures/jet.mpg: MPEG sequence, v1, progressive Y'CbCr
    ↪ 4:2:0 video, CIF NTSC, NTSC 4:3, 29.97 fps, Constrained

barry@forensicbox:NTFS_Pract_2017$ mplayer /mnt/evid/Users/AlbertE/Pictures/jet.mpg
<video plays>
Playing /mnt/evid/Users/AlbertE/Pictures/jet.mpg.
libavformat version 56.40.101 (internal)
MPEG-ES file format detected.
...
```

At this point we are finished with the mount point and the fuse mounted image. Keeping track of mounted disks and partitions, and properly unmounting them before you forget is an important part of this process:

```
barry@forensicbox:NTFS_Pract_2017$ su -
Password:

root@forensicbox:~# umount /mnt/evid && fusermount -u /mnt/ewf

root@forensicbox:~# exit
logout
```

We can unmount both the `/mnt/evid` file system and the fuse disk image at `/mnt/ewf` on the same line by separating with the `&&`. This means that the second command (**fusermount**) will only execute if the first **umount** is successful.

Back to our problem...To see why the file **jet.mpg** is interesting, let's try another method of obtaining a file list, the **fls** command. We can use the **-F** option to look only at files, and **-r** to do it recursively. We'll also **grep** for **jet.mpg**. You could use the directory MFT record numbers to browse down to the file, but this is quicker and more efficient:

```
barry@forensicbox:NTFS_Pract_2017$ fls -o 2048 -Fr NTFS_Pract_2017.E01 | grep
jet.mpg
r/r 39-128-1:  Users/AlbertE/Pictures/jet.mpg
r/r 39-128-3:  Users/AlbertE/Pictures/jet.mpg:unixphreak.txt
```

In the output of **fls**, **jet.mpg** has two entries:

```
39-128-1
39-128-3
```

Both entries have the same MFT record number (39) and are identified as file data (39-128) but the attribute identifier increments are different. This is an example of an alternate data stream. Accessing the standard contents (39-128-1) of `jet.mpg` is easy, since it is an allocated file. However, we can access either data stream, the normal data or the ADS, by using the Sleuth Kit command **icat**, much as we did with the files in our previous exercises. We simply call **icat** with the complete MFT record entry, to include the alternate attribute identifier. Here we specify each of the data streams and send them to the **file** command using **icat**:

```
barry@forensicbox:NTFS_Pract_2017$ icat -o 2048 NTFS_Pract_2017.E01 39 | file -  
/dev/stdin: MPEG sequence, v1, progressive Y'CbCr 4:2:0 video, CIF NTSC, NTSC 4:3,  
    ↪ 29.97 fps, Constrained
```

In this first (default) stream, we simply use the MFT record 39 to pass the default data to **file**. For the second stream, we pass the full attribute (39-128-3):

```
barry@forensicbox:NTFS_Pract_2017$ icat -o 2048 NTFS_Pract_2017.E01 39-128-3 |
    file -
/dev/stdin: ASCII text, with CRLF line terminators
```

This time we see it is ASCII text. So now we can just pipe the same command to **less** (or just straight to stdout) to view:

```
barry@forensicbox:NTFS_Pract_2017$ icat -o 2048 NTFS_Pract_2017.E01 39-128-3 | less
```

```
+-----+  
: PHAphaPHAphaPHAphaPHAphaPHAphaPHAphaPHAphaPHAphaPHAphaPHAphaPHA :  
: pha+-----+pha:  
: PHA:                Phreakers/Hackers/Anarchists Present:           :PHA:  
: pha:   ==+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=       :pha:  
: PHA:   +=+      Gaining Better Access On Any Unix System             +=+    :PHA:  
: pha:   ==+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=       :pha:  
: PHA:   Written By Doctor Dissector (doctord@darkside.com) UPDT: 1/8/91 :PHA:  
: pha+-----+pha:  
: PHAphaPHAphaPHAphaPHAphaPHAphaPHAphaPHAphaPHAphaPHAphaPHAphaPHA :  
+-----+
```

```
+-----+  
:[ Disclaimer ]=====:
```

```
+-----+
```

The author and the sponsor group Phreakers/Hackers/Anarchists will not be held responsible for any actions done by anyone reading this material before, during, and after exposure to this document. This document has been released under the notion that the material presented herein is for informational purposes only, and that neither the author nor the group P/H/A encourage the use of this information for any type of illegal purpose. Thank you.

And here we've displayed our NTFS alternate data stream, a text file.

9.3.8 Sleuth Kit Exercise 6: Physical String Search & Allocation Status (NTFS)

We've already done a few string search exercises, but all of them have been on EXT file systems. We make a lot of assumptions when we search for simple strings in an image. We assume the strings will be accessible (not in a container that requires pre-processing), and we assume they will be in a character encoding that our search utility will find. This is not always the case. Most of the string searches we've done thus far have resulted in matches that are found in regular ASCII text files. When we search for strings in documents on Windows systems, for example, that won't always be the case. We'll need to deal with more control characters, and additional application overhead and considerations, like compressed and encoded formats.

This exercise still simplifies some of that, but it also serves to make you aware of some of the more complex issues that may arise when searching larger images with more complex content. It will also introduce us to some basic application level file viewers beyond those we've already seen. The scenario here is the same as previous exercises. We'll pick a keyword, search the entire disk, and then recover and view the associated file. It will be very similar to the EXT exercises we did earlier (2A and 2B). This time, however, NTFS is our target file system.

Once again, we are dealing with the `NTFS_Pract_2017.E01` image set. And, again, since we are doing a physical search using non-EWF aware tools, we'll **ewfmnt** the images and work on the raw fuse mounted disk image. This time we create a mount point in our current directory and use **ewfmnt** with our normal user account...no need for loop devices and root permissions:

```
barry@forensicbox:NTFS_Pract_2017$ mkdir ewfmnt
```

```
barry@forensicbox:NTFS_Pract_2017$ ewfmnt NTFS_Pract_2017.E01 ewfmnt/  
ewfmnt 20140806
```

The **grep** command points to the fuse mounted image in `ewfmnt/`. Since `ewfmnt` is in our current directory (we just created it here), there is no need for a leading `/`.

```
barry@forensicbox:NTFS_Pract_2017$ grep -abi cyberbullying ewfmnt/ewf1  
<unreadable text and characters>
```

When we execute our search, we are greeted with a significant number of non-ASCII characters that seriously impede the readability of the output. When you scroll down the output, you can see the string we are looking for, but the offsets are obscured.

Back in our forensic basics section, early in this document, we discussed using the **tr** command to translate "control characters" to newlines. This has the effect of removing much of the unreadable content from our view as well as from the **grep** search, while the one for one character replacement causes no issue for offset calculations. Use **tr** here:

```
barry@forensicbox:NTFS_Pract_2017$ tr '[:cntrl:]' '\n' < ewfmnt/ewf1 | grep -abi
cyberbullying
```

```
426596865:www.stopcyberbullying.org
426596971:Cyberbullying by proxy
426596995:Cyberbullying by proxy is when a cyberbully gets someone
else to do their dirty work. Most of the time they are unwitting
accomplices and don't know that they are being used by the
cyberbully. Cyberbullying by proxy is the most dangerous kind of
cyberbullying because it often gets adults involve in the harassment
and people who don't know they are dealing with a kid or someone
they know.
...
```

The command above uses **tr** to convert the set of control characters (**'[:cntrl:]'**) to newlines (**'\n'**). The input is taken from **ewfmnt/ewf1**, and then the resulting stream is piped through **grep** to our search with the usual **-abi** options to treat it like a text file (**a**), provide the byte offset (**b**) and make the search case insensitive (**i**). The output shows our offsets and string hits are now much more readable.

Now we run through the same set of commands we did previously. Calculating what sector the keyword is in, the offset within the volume, and finally which data block and meta-data entry is associated with the keyword hit.

We'll work with the first keyword hit (**426596865:www.stopcyberbullying.org** - highlighted above). The sector offset to our hit is found by dividing the byte offset by the sector size (**512**). We already know that there is only one partition in this image, but we'll run **mmls** just to be sure. We also run **fsstat** again to confirm the block size (which we already know from previous exercises is **4096** bytes). Repeating these steps is just good practice:

```
barry@forensicbox:NTFS_Pract_2017$ echo "426596865/512" | bc
833197
```

```
barry@forensicbox:NTFS_Pract_2017$ mmls NTFS_Pract_2017.E01
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors
```

	Slot	Start	End	Length	Description
000:	Meta	0000000000	0000000000	0000000001	Primary Table (#0)
001:	-----	0000000000	0000002047	0000002048	Unallocated

```
002: 000:000 0000002048 0001023999 0001021952 NTFS / exFAT (0x07)
```

```
barry@forensicbox:NTFS_Pract_2017$ fsstat -o 2048 NTFS_Pract_2017.E01
```

```
FILE SYSTEM INFORMATION
```

```
-----
```

```
File System Type: NTFS
```

```
Volume Serial Number: CAE0DFD2E0DFC2BD
```

```
...
```

```
CONTENT INFORMATION
```

```
-----
```

```
Sector Size: 512
```

```
Cluster Size: 4096
```

```
...
```

As expected, the keyword is in the only NTFS partition which resides at offset 2048 (sectors). We can complete the math and determine the block the keyword resides in all at once:

```
barry@forensicbox:NTFS_Pract_2017$ echo "(426596865-(2048*512))/4096" | bc
103893
```

For review, this reads: "Take our offset to the keyword in our disk (426596865), subtract the offset to the start of the partition (2048*512), and divide the resulting value by our file system block size (4096). Our file system block is 103893.

```
barry@forensicbox:NTFS_Pract_2017$ blkstat -o 2048 NTFS_Pract_2017.E01 103893
```

```
Cluster: 103893
```

```
Not Allocated
```

```
barry@forensicbox:NTFS_Pract_2017$ ifind -o 2048 -d 103893 NTFS_Pract_2017.E01
```

```
248-128-2
```

We can see that **blkstat** tells us the cluster (block) is Not Allocated, and **ifind** shows us that the meta-data structure (MFT entry) associated with that data block (**-d 103893**) is 248-128-2.

```
barry@forensicbox:NTFS_Pract_2017$ icat -o 2048 NTFS_Pract_2017.E01 248 | file -
```

```
/dev/stdin: Composite Document File V2 Document, Little Endian, 0s:
```

```
Windows, Version 5.1, Code page: 1252, Template: Normal, Last Saved
```

```
By: buckyball, Revision Number: 2, Name of Creating Application:
```

```
Microsoft Word 10.0, Last Printed: 02:05, Create Time/Date:
```

```
Tue Nov 21 21:41:00 1995, Last Saved Time/Date: Wed Oct 25 23:14:00
```

```
2006, Number of Pages: 2, Number of Words: 822, Number of
```

```
Characters: 4087, Security: 0
```

Piping our **icat** output through the **file** command shows us we have a Microsoft Word document. Note that when we pass the MFT record to **icat**, we use only the record number, **248** rather than the entire attribute since we are looking for the default attribute anyway, which is \$DATA.

If we try and view the document with **cat** or **less**, we again get non-ASCII characters, making reading difficult.

```
barry@forensicbox:NTFS_Pract_2017$ icat -o 2048 NTFS_Pract_2017.E01 248 | less
<unreadable text and characters>
```

We could use **icat** to redirect the contents to a file:

```
barry@forensicbox:NTFS_Pract_2017$ icat -o 2048 NTFS_Pract_2017.E01 248 > ntfs.248
```

From there we could view the file in an MS Word compatible application like LibreOffice for Linux (which is not installed right now, but would look like this): This is fine, but opening

Figure 15: Screenshot of LibreOffice viewing a recovered MS Office Document



and closing GUI programs to view file contents is not ideal for our command line approach. Instead, we can use a simple tool like **catdoc** to read MS Office files (.doc format) from the command line.

catdoc can be installed via **sboinstall** on Slackware:

```
barry@forensicbox:NTFS_Pract_2017$ su -
Password:
```

```
root@forensicbox:~# sboinstall catdoc
catdoc reads MS Word file and prints readable ASCII text to stdout, just
```

like Unix cat command. It also able to produce correct escape sequences if some UNICODE characters have to be represented specially in your typesetting system such as (La)TeX.

Proceed with catdoc? [y]

...

Cleaning for catdoc-0.94.2...

```
root@forensicbox:~# exit
logout
```

Once installed, you can either open the file you exported (ntfs.248) with **catdoc**, or you can simply stream the output of **icat** straight through to **catdoc**, and again through **less** (multiple pipes are just awesome).

```
barry@forensicbox:NTFS_Pract_2017$ icat -o 2048 NTFS_Pract_2017.E01 248-128-2 |
    catdoc | less
```

www.stopcyberbullying.org

Cyberbullying by proxy

Cyberbullying by proxy is when a cyberbully gets someone else to do their dirty work. Most of the time they are unwitting accomplices and

...

This exercise essentially closes the loop on our physical searching of file systems. As we can see there can be a lot more to searching an image than simple **grep** strings.

Let's leave with one more command, and a question. The fuse mounted image should still be available at ewfmnt/ewf1. Do a quick keyword search for "Uranium-235" (sounds ominous, doesn't it?):

```
barry@forensicbox:NTFS_Pract_2017$ grep -abi "Uranium-235" ewfmnt/ewf1
<returns nothing>
```

The pattern "Uranium-235" does not appear to be found. Does this mean I'm free to draw the conclusion that there are no instances of the string "Uranium-235" to be found on the disk? Of course not. We'll address this in our next exercise.

Be sure to unmount the fuse mounted image before you move on.

```
barry@forensicbox:NTFS_Pract_2017$ fusermount -u ewfmnt/
```

9.4 **bulk_extractor** - comprehensive searching

in previous exercises, we discussed issues where simple text based string searches might not be effective depending on character encoding and file formats (compression, etc.). There are a number of character set aware tools out there that we can use to overcome many of these issues, but detailed descriptions of character encoding and searches are not what we are aiming for here. Instead we are going to introduce a tool, **bulk_extractor**, that incorporates some excellent multi-format searching capabilities with some other very useful functions. **bulk_extractor** was created by Simson Garfinkel at the Naval Postgraduate School.

For those of you that have not already heard of or used **bulk_extractor**, it is one of those tools that I very rarely don't use on every case. Even where I have a targeted extraction or analysis to perform, **bulk_extractor** can always find additional information or, at the very least, provide an excellent overview of user activity or media context. It is particularly useful in situations where you have been given (or acquired yourself) a high volume of media and you want to quickly sort out the interesting data. This triage capability is one of the highlights of **bulk_extractor**.

bulk_extractor differs from some other more common tools in that it runs and searches completely independent of the file system. In this case, it's not the files themselves that are interesting, but the content – whether allocated or unallocated, whole or fragmented, or even in compressed containers. **bulk_extractor** reads in the data by blocks, without regard to file system structure, and recursively searches those blocks for interesting features. Recursive in this case means the tool will, for example, decompress an archive to search the contents and extract text from PDF files to be further processed.

A complete user's manual for **bulk_extractor** is available at:

http://downloads.digitalcorpora.org/downloads/bulk_extractor/BEUsersManual.pdf

bulk_extractor also has a GUI tool, **BEviewer**, commonly used to read the feature files and run the program. If you want to see this in action, you'll need java installed. OpenJDK is the easiest to install from **sbotools**, but be sure to read the **README**. OpenJDK will need to be installed prior to **bulk_extractor** for **BEviewer** to be included in the final package. **BEviewer** was written by Bruce Allen.

Let's install **bulk_extractor** now and have a closer look at the options.

```
root@forensicbox:~# sbinstall bulk_extractor
```

```
bulk_extractor is a C++ program that scans a disk image, a file, or a directory  
of files and extracts useful information without parsing the file system or
```

file system structures...

...

The searches performed by **bulk_extractor** are done using specific scanners that can be enabled and disabled depending on what you want to search for and how. It is these scanners that manage the parsing of PDF files or compressed archives and other formats. We can get a look at available scanners by viewing the output of **bulk_extractor** with **-h**. It's a long list of command options, so you might want to pipe the output through **less**:

```
barry@forensicbox:~$ bulk_extractor -h | less
```

```
bulk_extractor version 1.5.5
```

```
Usage: bulk_extractor [options] imagefile
```

```
    runs bulk extractor and outputs to stdout a summary of what was found where
```

Required parameters:

```
    imagefile      - the file to extract
or -R filedir     - recurse through a directory of files
                    HAS SUPPORT FOR E01 FILES
                    HAS SUPPORT FOR AFF FILES
    -o outdir      - specifies output directory. Must not exist.
                    bulk_extractor creates this directory.
```

Options:

```
-i              - INFO mode. Do a quick random sample and print a report.
-b banner.txt  - Add banner.txt contents to the top of every output file.
-r alert_list.txt - a file containing the alert list of features to alert
                  (can be a feature file or a list of globs)
                  (can be repeated.)
-w stop_list.txt - a file containing the stop list of features (white list)
                  (can be a feature file or a list of globs)
                  (can be repeated.)
-F <rfile>     - Read a list of regular expressions from <rfile> to find
-f <regex>     - find occurrences of <regex>; may be repeated.
```

...

These scanners disabled by default; enable with **-e**:

```
-e base16 - enable scanner base16
-e facebook - enable scanner facebook
-e outlook - enable scanner outlook
-e sceadan - enable scanner sceadan
-e wordlist - enable scanner wordlist
-e xor - enable scanner xor
```

These scanners enabled by default; disable with **-x**:

```
-x accts - disable scanner accts
-x aes - disable scanner aes
-x base64 - disable scanner base64
-x elf - disable scanner elf
-x email - disable scanner email
```

```
-x exif - disable scanner exif
```

```
...
```

You can also get a slightly more descriptive output on the scanners by doing the same as above but with **-H** instead of **-h**.

There are a lot of options to go through. Some we'll cover as we go through a sample exercise, and others we'll skip over and allow you to explore on your own. The simplest way to run **bulk_extractor** is to leave everything default and simply provide an output directory for the results. This can take awhile, but it provides the best overall intelligence on the disk contents. For now, we are going to reduce the output by limiting the scanners and providing a single search term. This will allow us to isolate the results and spend some time talking about the output files.

Some of the more important options to remember when running **bulk_extractor** are:

- o <output_dir>** Directory to write the results (**bulk_extractor** will create this)
- e <scanner>** Enable <scanner>
- E <scanner>** Disable ALL scanners except <scanner>
- x <scanner>** Disable <scanner>

The easiest way to explain the options is to run the command and check the output. We ended the last section on NTFS physical string searching by doing a simple **grep** for the term **Uranium-235** in our NTFS **E01** image set. The results returned nothing. Now we'll run the same search again using **bulk_extractor**. Run the command with the following options. Note that **bulk_extractor** can run directly on the EWF files if **libewf** is installed:

```
barry@forensicbox:~$ bulk_extractor -E zip -e find -f "Uranium-235" -o blk_out
NTFS_Pract_2017/NTFS_Pract_2017.E01
bulk_extractor version: 1.5.5
Hostname: forensicbox
Input file: NTFS_Pract_2017/NTFS_Pract_2017.E01
Output directory: blk_out
Disk Size: 524288000
Threads: 2
11:28:34 Offset 67MB (12.80%) Done in 0:00:05 at 11:28:39
11:28:36 Offset 150MB (28.80%) Done in 0:00:05 at 11:28:41
/colorblue...
1:29:58 Offset 486MB (92.80%) Done in 0:00:06 at 11:30:04
All data are read; waiting for threads to finish...
Time elapsed waiting for 2 threads to finish:
    (timeout in 60 min.)
All Threads Finished!
Producer time spent waiting: 86.5672 sec.
Average consumer time spent waiting: 0.064092 sec.
```

```
MD5 of Disk Image: eb4393cfcc4fca856e0edbf772b2aa7d
Phase 2. Shutting down scanners
Phase 3. Creating Histograms
Elapsed time: 90.6774 sec.
Total MB processed: 524
Overall performance: 5.78191 MBytes/sec (2.89095 MBytes/sec/thread)
```

In the above command, we use **-E zip** to disable every default scanner except the **zip** scanner. We then re-enable the **find** scanner with **-e find** (so that we can run our string search). This is followed by the **-f "Uranium-235"** search term. This term can be a string or a regular expression. We can also add additional terms or create a search term file (with a list of keywords or expressions) and run it with the **-F** option. Our output directory is set with **-o blk_out** (which **bulk_extractor** will create for us).

The command provides some fairly self-explanatory information, including the data processed and the hash of the disk image. Change into the output directory and let's have a look at the files that were produced.

```
barry@forensicbox:~$ cd blk_out

barry@forensicbox:blk_out$ ls -l
total 336
-rw-r--r-- 1 barry users      0 Aug  1 11:28 alerts.txt
-rw-r--r-- 1 barry users  263 Aug  1 11:29 find.txt
-rw-r--r-- 1 barry users  206 Aug  1 11:30 find_histogram.txt
-rw-r--r-- 1 barry users 9803 Aug  1 11:30 report.xml
-rw-r--r-- 1 barry users    0 Aug  1 11:28 unzip_carved.txt
-rw-r--r-- 1 barry users 319995 Aug  1 11:30 zip.txt
```

There are basically three different files shown in the output above. These are:

- Feature files: Files that contain the output of each scanner.
- Histogram files: Files that show the frequency that each item in a feature file is encountered. We'll discuss the usefulness of these in more detail later.
- The report file: A DFXML formatted report of the output and environment.

Any files that are **0** size are empty and no features were noted. In this case the **alerts.txt** file is empty because we did not specify an alert file with the **-r** option. The feature file we are concerned with here is the **find.txt**, produced by the **find** scanner. Open and have a look at this file:

```
barry@forensicbox:blk_out$ cat find.txt
# BANNER FILE NOT PROVIDED (-b option)
```

```
# BULK_EXTRACTOR-Version: 1.5.5 ($Rev: 10844 $)
# Feature-Recorder: find
# Filename: NTFS_Pract_2017/NTFS_Pract_2017.E01
# Feature-File-Version: 1.1
445901295-ZIP-9745  Uranium-235 ferece between Uranium-235 and Uranium-238
```

The `find.txt` file has a commented area (lines starting with #), and the actual output of the scanner itself, with each "feature" found on one line. There are three parts to the scanner output for each feature. The first is an offset. This offset can have multiple parts. In **bulk_extractor** this is referred to as the forensic path. This includes a disk offset to the data containing the feature, the scanner(s) that found the object, and then the offset within that data. The forensic path is followed by the feature itself, in this case our "Uranium-235" search term. Finally we are given a small bit of context. In other words, for our example above:

```
445901295-ZIP-9745      Compressed data (ZIP) was found at disk offset
                        445901295. The feature (Uranium-235) was found at off-
                        set 9745 in that compressed data.
```

```
Uranium-235            The feature that was found (our search term)
```

```
ferece between Uranium  The context the feature was found in.
↪ -235 and Uranium-238
```

Using what we've learned previously about physical searching, let's have a quick look at the data found at that offset. Remember our formula for finding the offset in a file system when given a disk offset? We've seen this NTFS image set before, so we already know the file system starts at sector offset 2048, so we'll calculate the file system offset and then run the **ifind** command we've used several times already to find out what MFT entry points to the data block. Finally we'll use the **icat** command and pipe the output to **file** so we can identify the type:

```
barry@forensicbox:bulk_out$ echo "((445901295-(2048*512))/4096)" | bc
108606

barry@forensicbox:bulk_out$ ifind -d 108606 -o 2048
../NTFS_Pract_2017/NTFS_Pract_2017.E01
236-128-2

barry@forensicbox:bulk_out$ icat -o 2048 ../NTFS_Pract_2017/NTFS_Pract_2017.E01
236 | file -
/dev/stdin: Microsoft Word 2007+
```

So we see that the feature was found in a Microsoft Word document in `.docx` format, which is compressed XML. This file can be viewed with the **catdocx** script (remember **catdoc**? **catdocx** is similar, but for the XML compressed `.docx` format). We will do this at the end of the exercise.

After the feature files, we move on to the histogram file. A histogram is simply a file that will list the features along with the number of times that feature was found. This frequency reporting is one of the more useful aspects of **bulk_extractor**. It is the histograms that provide a great deal of context to the contents of a disk image. Particularly where investigations involving fraud or PII are concerned, the frequency of a credit card number or email address can tell an investigator, at a glance, what accounts were used and the most frequently used accounts, or who the closest associates might be, etc. In our case the histogram shows only one instance (n=1) of our search term.

```
barry@forensicbox:bulk_out$ cat find_histogram.txt
# BANNER FILE NOT PROVIDED (-b option)
# BULK_EXTRACTOR-Version: 1.5.5 ($Rev: 10844 $)
# Feature-Recorder: find
# Filename: NTFS_Pract_2017/NTFS_Pract_2017.E01
# Histogram-File-Version: 1.1
n=1 uranium-235
```

Let's run **bulk_extractor** again, but this time we'll leave all the default scanners running and use a list of search terms instead (just two). Change back to your home directory, and using a text editor (**vi!**), create a file with just these two terms:

```
[Uu]ranium-235
262698143
```

...we've turned our first term into regular expression that looks for either an upper or lowercase letter to start the word. The second is a "known victim" social security number²⁹. Save the file as **myterms.txt**.

We'll also create a banner file so that all of our output files have a heading that identifies the case and the examiner/analyst. Again, using a text editor enter information you might want at the top of each file:

```
Office of Investigations
Case of the Century
Case#: 2017-01-0001
Investigator: Barry Grundy
```

...save the file as **mybanner.txt**.

Now we'll re-run **bulk_extractor**, without disabling or enabling scanners, using a banner file (**-b mybanner.txt**) and a file of terms to search for (**-F myterms.txt**). The output directory will be **blk_out_full** (**-o blk_out_full**). With all the scanners running, you will see quite a few more files in the output directory.

²⁹The second term is a social security number. Numbers for this exercise were generated with <http://www.theonegenerator.com/ssngenerator>

```

barry@forensicbox~$ bulk_extractor -b mybanner.txt -F myterms.txt -o blk_out_full
NTFS_Pract_2017/NTFS_Pract_2017.E01
bulk_extractor version: 1.5.5
Hostname: forensicbox
Input file: NTFS_Pract_2017/NTFS_Pract_2017.E01
Output directory: blk_out_full
Disk Size: 524288000
...
Overall performance: 5.3725 MBytes/sec (2.68625 MBytes/sec/thread)
Total email features found: 570

```

This command results in a great deal more output (but keep in mind that files of zero length are empty – nothing found). Look at the contents of the `find.txt` file now:

```

barry@forensicbox:~$ ls - blk_out_full
aes_keys.txt          find_histogram.txt    telephone_histogram.txt
alerts.txt            gps.txt              unrar_carved.txt
ccn.txt              httplogs.txt         unzip_carved.txt
ccn_histogram.txt     ip.txt              url.txt
ccn_track2.txt        ip_histogram.txt     url_facebook-address.txt
ccn_track2_histogram.txt jpeg_carved.txt      url_facebook-id.txt
domain.txt           json.txt             url_histogram.txt
domain_histogram.txt  kml/                url_microsoft-live.txt
elf.txt             kml.txt             url_searches.txt
email.txt            pii.txt             url_services.txt
email_domain_histogram.txt pii_teamviewer.txt  vcard.txt
email_histogram.txt  rar.txt             windirs.txt
ether.txt            report.xml           winlnk.txt
ether_histogram.txt  rfc822.txt          winpe.txt
exif.txt             sqlite_carved.txt    winprefetch.txt
find.txt             telephone.txt        zip.txt

```

```

barry@forensicbox:~$ cat blk_out_full/find.txt
# Office of Investigations
# Case of the Century
# Case#: 2017-01-0001
# Investigator: Barry Grundy
# BULK_EXTRACTOR-Version: 1.5.5 ($Rev: 10844 $)
# Feature-Recorder: find
# Filename: NTFS_Pract_2017/NTFS_Pract_2017.E01
# Feature-File-Version: 1.1
1193351-PDF-92 262698143 629369510 SSN: 262698143
445901295-ZIP-9745 Uranium-235 ferece between Uranium-235 and Uranium-238
445901295-ZIP-0-MSXML-857 Uranium-235 ferece between Uranium-235 and Uranium-238

```

All the output files also have our `mybanner.txt` text at the top. And this time we see that

our `find.txt` contains both the Uranium-235 hit we saw previously but also the "victim" social security number we added to our terms list. We now have features that were found in a zip archive (`.docx` file we identified earlier) and a PDF file (using the `pdf` scanner). The Microsoft Word file we identified earlier is now showing two features instead of one. This is because it was found by two scanners, the `zip` scanner and the `msxml` scanner.

You can browse around the rest of the feature files and histograms to see what else we may have uncovered. There's quite a bit of information there and you can get a general idea of things like the user's browsing activity by looking at `url_histogram.txt`. You certainly can't draw conclusions, but higher frequency domains can provide some context to you investigation.

One thing you may notice is that a large number of the features found by the `email` and `url` scanners (and others) come from known sources. Every operating system and the external software we use has help files, manuals, and other documentation that contain email addresses, telephone numbers, and web addresses that are uninteresting, but will still end up in your `bulk_extractor` feature files and histograms. These false positives can be limited by using stop lists. Much like our `myterms.txt` file, a stop list can be a simple list of terms (or terms with context) that are blocked from the regular scanner feature files (but still reported in special `stopped.txt` files for each scanner).

A final `bulk_extractor` capability that we'll mention briefly here is the `wordlist` scanner. Disabled by default, the `wordlist` scanner creates lists of words that can be used to attempt password cracking. In a normal `bulk_extractor` run, just use `-e wordlist` to enable the scanner, or use `-E wordlist` to run it on its own.

Very quickly lets go back and use our keyword hit on Uranium-235 to learn about a quick command line `.docx` format file viewer, `catdocx`. This is actually a very short script rather than a program, and it simply unzips the file and makes the XML content readable.

```
barry@forensicbox:bulk_out$ su -
Password:

root@forensicbox:~# wget
      https://raw.githubusercontent.com/jncraton/catdocx/master/catdocx.sh -O
      /usr/bin/catdocx && chmod 755 /usr/bin/catdocx
...
2019-08-01 13:54:31 (81.6 MB/s) - '/usr/bin/catdocx' saved [434/434]

root@forensicbox:~# exit
```

This command uses `wget` to download the `catdocx` script from GitHub directly to `/usr/bin/` ↪ `catdocx` (with the `-O` option). The `&&` allows us to run `chmod` immediately after the `wget` completes to change the permissions and make the file executable.

Now we can re-run the `icat` command we used earlier on the MFT entry pointing to the

Uranium-235 keyword. This time we'll re-direct the output of **icat** to a file called **NTFS.236**. Then we use **catdocx** piped through **less** to display the file:

```
barry@forensicbox:~$ icat -o 2048 NTFS_Pract_2017/NTFS_Pract_2017.E01 236 >
NTFS.236
```

```
barry@forensicbox:~$ catdocx NTFS.236 | less
```

```
Watch modern marvels the Manhattan project. You can find it in 5 parts on youtube
```

```
https://www.youtube.com/watch?v=SwHds1any9Y
```

```
https://www.youtube.com/watch?v=VGGAiuc5dWI
```

```
https://www.youtube.com/watch?v=eHvUgtVOP64
```

```
https://www.youtube.com/watch?v=aAXy5V-zRyc
```

```
https://www.youtube.com/watch?v=aJuBHzgLUAw
```

```
Name_____
```

```
Modern Marvels: Manhattan Project
```

```
...
```

```
What is the difference between Uranium-235 and Uranium-238?
```

```
...
```

...And we see the keyword hit in our output along with the expected context from the feature file.

9.5 Physical Carving

We've seen a number of cases in previous exercises where we needed to locate file headers to recover data. We saw a specific need for this with our ext4 exercise where we found out that direct block pointers were no longer available for deleted files, making recovery very difficult. We also did manual recovery in our "Data Carving With **dd**" exercise, locating the header of a JPEG file in hex and using **dd** to physically "carve" out the file. A useful skill, but a bit tedious on a large disk image with potentially dozens, hundreds or even thousands of files that might require recovery. If you are unfamiliar with file carving, or need a refresher, you can start reading here: http://forensicswiki.org/wiki/File_Carving

Since we gained a cursory understanding of the mechanics of carving through our **dd** problem, we can move on to more automated tools that do the work for us. There are a number of tools available to accomplish this. We are going to concentrate on just two. **scalpel**, and **photorec**. The latter is from the **testdisk** package.

9.5.1 **scalpel**

We'll start by installing **scalpel**. Use **sboinstall** to install it, being sure to read the **README** file. If you are not using Slackware, go ahead and use your distribution's package management

tool. You will see that for Slackware, **scalpel** has a single dependency that must be installed first TRE, which is handled automatically by **sboinstall**:

```
root@forensicbox:~# sboinstall scalpel
```

```
TRE is a lightweight, robust, and efficient POSIX compliant regexp
matching library with some exciting features such as approximate
(fuzzy) matching.
```

```
Proceed with tre? [y]
tre added to install queue.
```

Scalpel is a fast file carver that reads a database of header and footer definitions and extracts matching files or data fragments from a set of image files or raw device files. Scalpel is filesystem-independent and will carve files from FATx, NTFS, ext2/3, HFS+, or raw partitions. It is useful for both digital forensics investigation and file recovery.

To use it, you MUST have a conf file that defines the file types you want to recover. Use the example `scalpel.conf` file from `/usr/doc/scalpel`

See the man page for details.

```
Proceed with scalpel? [y]
```

```
...
```

```
Cleaning for scalpel-2.0...
```

If you read the `README` file (which you did, RIGHT?), you will see that we need to copy and edit the `scalpel.conf` file before we can run the program. We can either edit and use it in place, or copy it to our working directory which scalpel uses by default.

For now, we'll copy the `scalpel.conf` file that was installed with our package to a new `carve` sub directory in our `/home` directory, which we'll create now, and edit the config file there.

```
barry@forensicbox:~$ mkdir ~/carve
```

```
barry@forensicbox:~$ cd ~/carve
```

```
barry@forensicbox:~$ cp /usr/share/doc/scalpel-2.0/scalpel.conf .
```

The final `'.'` in the command above signifies the destination, our current directory. `scalpel.conf` starts out completely commented out. We will need to uncomment some file definitions in order to have **scalpel** work. Open `scalpel.conf` with **vi** (or your editor of choice. You should take time to read the file as it explains the structure of the file definitions in useful detail.

```
barry@forensicbox:~$ vi scalpel.conf
```

```
# Scalpel configuration file
```

```
# This configuration file controls the types and sizes of files that
# are carved by Scalpel. NOTE THAT THE FORMAT OF THIS FILE WAS
# EXTENDED in Scalpel 1.90-->!
```

```
# For each file type, the configuration file describes the file's
# extension, whether the header and footer are case sensitive, the
# min/maximum file size, and the header and footer for the file. The
# footer field is optional, but extension, case sensitivity, size, and
# footer are required. Any line that begins with a '#' is considered
# a comment and ignored. Thus, to skip a file type just put a '#' at
# the beginning of the line containing the rule for the file type.
```

```
...
```

Scroll down to where the **# GRAPHICS FILES** section starts (for the purpose of our exercise) and just uncomment every line that describes a file in that section. Be careful not to uncomment lines that should remain comments. To uncomment a line, simply remove the hash (#) symbol at the start of the line. The **# GRAPHICS FILES** section should look like this when you are done (extra hash symbols don't matter, as long as the correct lines are uncommented, and the section lines are still commented):

```
#-----
# GRAPHICS FILES
#-----
#
#
# AOL ART files
  art y   150000  \x4a\x47\x04\x0e  \xcf\xcb\xcb
  art y   150000  \x4a\x47\x03\x0e  \xd0\xcb\x00\x00
#
# GIF and JPG files (very common)
  gif y   5000000  \x47\x49\x46\x38\x37\x61  \x00\x3b
  gif y   5000000  \x47\x49\x46\x38\x39\x61  \x00\x00\x3b
  jpg y   200000000 \xff\xd8\xff\xe0\x00\x10  \xff\xd9
  jpg y   200000000 \xff\xd8\xff\xe1  \xff\xd9

# PNG
  png y   20000000  \x50\x4e\x47?  \xff\xfc\xfd\xfe

# BMP (used by MSWindows, use only if you have reason to think there are
# BMP files worth digging for. This often kicks back a lot of false
# positives

  bmp y   100000  BM??\x00\x00\x00
```

```
#
# TIFF
    tif y    200000000    \x49\x49\x2a\x00
# TIFF
    tif y    200000000    \x4D\x4D\x00\x2A
#
```

If you look at the lines for the **jpg** images, you will see the familiar pattern that we searched for during our **dd** carving exercise. `\xff\xd8` for the header and `\xff\xd9` for the footer. When we run **scalpel** these uncommented lines will be used to search for patterns. When you are finished editing the file (double check!), save and quit with **:wq**

For this exercise, we will use the **able_3** split image as our exercise target. In our Sleuth Kit exercise **#1B** (deleted file identification and recovery – ext4), we ran across a number of files (**lolitaz***) in the **/home** directory that could not be recovered. This is an obvious use case for file carving.

Since we are able to get the allocated files from the **/home** partition on **able_3**, we might want to limit our carving to unallocated blocks only. This is a common way to carve file systems – separate the allocated and the unallocated and carve those blocks only. We already learned how to extract all unallocated blocks from a file system using the TSK tool **blkls**. So we'll start by extracting the unallocated first.

Remember that the TSK tools can work directly on split images, so there is no need for us to fuse mount the image or loop mount any file systems. Running **mmls** gives us the file system offsets (if you remember, the **/home** directory was mounted on the second Linux file system at offset **104448**). We use that with our **blkls** command. You can run a quick recursive **fls** command using the **-r** option to refresh your memory on the files we are looking for. The files with the asterisk (*****) next to the inode number are deleted:

```
barry@forensicbox:carve$ mmls ../able_3/able_3.000
```

```
GUID Partition Table (EFI)
```

```
Offset Sector: 0
```

```
Units are in 512-byte sectors
```

	Slot	Start	End	Length	Description
000:	Meta	0000000000	0000000000	0000000001	Safety Table
001:	-----	0000000000	0000002047	0000002048	Unallocated
002:	Meta	0000000001	0000000001	0000000001	GPT Header
003:	Meta	0000000002	0000000033	0000000032	Partition Table
004:	000	0000002048	0000104447	0000102400	Linux filesystem
005:	001	0000104448	0000309247	0000204800	Linux filesystem
006:	-----	0000309248	0000571391	0000262144	Unallocated
007:	002	0000571392	0008388574	0007817183	Linux filesystem
008:	-----	0008388575	0008388607	0000000033	Unallocated

```
barry@forensicbox:carve$ fls -o 104448 -r ../able_3/able_3.000
d/d 11: lost+found
d/d 12: ftp
d/d 13: albert
+ d/d 14: .h
++ r/d * 15(realloc): lolit_pics.tar.gz
++ r/r * 16(realloc): lolitaz1
++ r/r * 17: lolitaz10
++ r/r * 18: lolitaz11
++ r/r * 19: lolitaz12
++ r/r 20: lolitaz13
++ r/r * 21: lolitaz2
++ r/r * 22: lolitaz3
++ r/r * 23: lolitaz4
++ r/r * 24: lolitaz5
++ r/r * 25: lolitaz6
++ r/r * 26: lolitaz7
++ r/r * 27: lolitaz8
++ r/r * 28: lolitaz9
+ d/d 15: Download
++ r/r 16: index.html
++ r/r * 17: lrkn.tar.gz
V/V 25689: $OrphanFiles
```

To obtain the unallocated blocks using `blkls`:

```
barry@forensicbox:carve$ blkls -o 104448 ../able_3/able_3.000 > able3.home.blkls

barry@forensicbox:carve$ ls -lh
total 92M
-rw-r--r-- 1 barry users 92M Aug  2 08:35 able3.home.blkls
-rwxr-xr-x 1 barry users 13K Aug  2 08:28 scalpel.conf*
```

The `blkls` command is run with the offset (`-o`) pointing to the second Linux file system that starts at sector `104448`. The output is redirected to `able3.home.blkls`. The name `home` is used to signify that this is the partition mounted as `/home` in the original `able_3` disk. Now we can see (with the `ls` command above) that we have two files in the `~/carve` directory.

`scalpel` has a number of options available to adjust the carving. There is an option to have `scalpel` carve the files on block (or cluster) aligned boundaries. This means that you would be searching for files that start at the beginning of a data block. Be careful doing that. The trade off here is that while you get fewer false positives, it also means that you miss files that may be embedded or "nested" in other files. Block aligned searching is done with the `-q <blocksize>` option. Try this option later, and compare the output. To get the block size for the target file system, you can use the `fsstat` command as we did in previous exercises.

You can carve multiple images at once with the **-i <listfile>** option, and there are other options to test data (write an audit file without carving).

In this case, we'll use an option that allows us to properly parse embedded files (**-e**). This option allows the proper pairing of headers and footers. Without the **-e** option, a header followed by another header (as with an embedded file), would result in both files sharing the same footer.

Finally, we'll use the **-o** option to redirect our carved files to a directory we are going to call **scalp_out** and the **-0** option so the output remains in a single output directory instead of categorized sub directories. Having the files in a single folder makes for easier viewing.

```
barry@forensicbox:carve$ scalpel -o scalp_out -0 -e able3.home.blkls
Scalpel version 2.0
Written by Golden G. Richard III and Lodovico Marziale.
Multi-core CPU threading model enabled.
Initializing thread group data structures.
Creating threads...
Thread creation completed.

Opening target "/home/barry/carve/able3.home.blkls"

Image file pass 1/2.
able3.home.blkls: 100.0% |*****| 91.3 MB 00:00
    ↳ ETAAAllocating work queues...
Work queues allocation complete. Building work queues...
Work queues built. Workload:
art with header "\x4a\x47\x04\x0e" and footer "\xcf\xcb" --> 0 files
art with header "\x4a\x47\x03\x0e" and footer "\xd0\xcb\x00\x00" --> 0 files
gif with header "\x47\x49\x46\x38\x37\x61" and footer "\x00\x3b" --> 0 files
gif with header "\x47\x49\x46\x38\x39\x61" and footer "\x00\x00\x3b" --> 1 files
jpg with header "\xff\xd8\xff\xe0\x00\x10" and footer "\xff\xd9" --> 6 files
jpg with header "\xff\xd8\xff\xe1" and footer "\xff\xd9" --> 0 files
png with header "\x50\x4e\x47?" and footer "\xff\xfc\xfd\xfe" --> 0 files
bmp with header "BM??\x00\x00\x00" and footer "" --> 0 files
tif with header "\x49\x49\x2a\x00" and footer "" --> 0 files
tif with header "\x4d\x4d\x00\x2a" and footer "" --> 0 files
Carving files from image.
Image file pass 2/2.
able3.home.blkls: 100.0% |*****| 91.3 MB 00:00
    ↳ ETAProcessing of image file complete. Cleaning up...
Done.
Scalpel is done, files carved = 7, elapsed = 1 secs.
```

```
barry@forensicbox:carve$ ls scalp_out/
00000000.gif 00000002.jpg 00000004.jpg 00000006.jpg
00000001.jpg 00000003.jpg 00000005.jpg audit.txt
```

The output above shows **scalpel** carving those file types in which the definitions were un-commented. Once the command completes, a directory listing shows the files (with the extension for the carved file type added) and an **audit.txt** file. The **audit.txt** file provides a log with the contents of **scalpel.conf** and the program output:

```
barry@forensicbox:carve$ less scalp_out/audit.txt
Scalpel version 2.0 audit file
Started at Fri Aug  2 08:44:13 2019
Command line:
scalpel -o scalp_out -0 -e able3.home.blkls

Output directory: scalp_out
Configuration file: /home/barry/carve/scalpel.conf

----- BEGIN COPY OF CONFIG FILE USED -----
# Scalpel configuration file

# This configuration file controls the types and sizes of files that
# are carved by Scalpel.  NOTE THAT THE FORMAT OF THIS FILE WAS
# EXTENDED in Scalpel 1.90-->!
...
----- END COPY OF CONFIG FILE USED -----

Opening target "/home/barry/carve/able3.home.blkls"

The following files were carved:
File           Start           Chop           Length           Extracted From
00000006.jpg   6586930           NO             6513             able3.home.blkls
00000005.jpg   6586368           NO             64601            able3.home.blkls
00000004.jpg   6278144           NO             15373            able3.home.blkls
00000003.jpg   6249472           NO             27990            able3.home.blkls
00000002.jpg   6129070           NO             5145             able3.home.blkls
00000001.jpg   6128640           NO             94426            able3.home.blkls
00000000.gif   6223872           NO             25279            able3.home.blkls

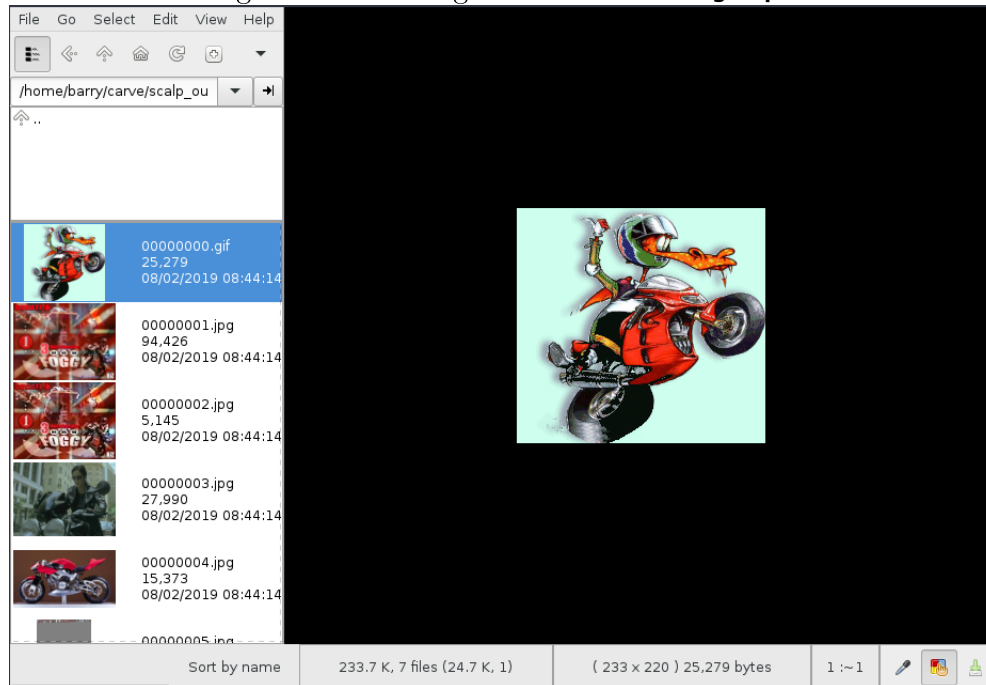
Completed at Fri Aug  2 08:44:14 2019
```

The entire **scalpel.conf** file is included in **audit.txt**. At the bottom of the output is our list of carved files with the offset the header was found at, the length of the file, and the source (what was carved). The column labeled **Chop** would refer to files that had a maximum number of bytes carved before the footer was found. You can read the **scalpel.conf** file for a more detailed description.

The files can be viewed with **display** at the command line or with a GUI viewer that can provide a thumbnail and windowed view (**sxiv** is a great CLI image viewer with thumbnail capabilities). The program **geeqie** is a simple example of a GUI viewer.

```
barry@forensicbox:carve$ cd scalp_out/
```

```
barry@forensicbox:carve$ geeqie
```

Figure 16: Viewing carved files with **geeqie**

There are other files to be found in this unallocated data. To illustrate this, let's look at the `scalpel.conf` file again and add a different header definition for a bitmap file. Open `scalpel.conf` with your text editor (**vi**)³⁰ and add the following line (shown in red) under the current `bmp` line in the `# GRAPHICS FILES` section:

```
barry@forensicbox:carve$ vi scalpel.conf
```

```
...
# BMP      (used by MSWindows, use only if you have reason to think there are
#          BMP files worth digging for. This often kicks back a lot of false
```

³⁰If you are using **vi** to edit the file, you should copy and paste the line. With the cursor on the existing line, use **yy** to copy the text (current line) and then **p** to paste on the line below. Then edit that line.

```
# positives

    bmp y   100000  BM??\x00\x00\x00
    bmp y   300000  BM??\x04\x00\x00
...

```

Here we've changed the max size to 300000 bytes, and replaced the first x00 string with x04. Save the file. Re-run **scalpel** again (write to a different output directory - **scalp_out2**), and check the output:

```
barry@forensicbox:carve$ scalpel -o scalp_out2 -0 -e able3.home.blkl
Scalpel version 2.0
Written by Golden G. Richard III and Lodovico Marziale.
Multi-core CPU threading model enabled.
Initializing thread group data structures.
Creating threads...
Thread creation completed.

Opening target "/home/barry/carve/able3.home.blkls"

Image file pass 1/2.
able3.home.blkls: 100.0% |*****| 91.3 MB 00:00
    ↳ ETAAAllocating work queues...
Work queues allocation complete. Building work queues...
Work queues built. Workload:
art with header "\x4a\x47\x04\x0e" and footer "\xcf\xc7\xcb" --> 0 files
art with header "\x4a\x47\x03\x0e" and footer "\xd0\xcb\x00\x00" --> 0 files
gif with header "\x47\x49\x46\x38\x37\x61" and footer "\x00\x3b" --> 0 files
gif with header "\x47\x49\x46\x38\x39\x61" and footer "\x00\x00\x3b" --> 1 files
jpg with header "\xff\xd8\xff\xe0\x00\x10" and footer "\xff\xd9" --> 6 files
jpg with header "\xff\xd8\xff\xe1" and footer "\xff\xd9" --> 0 files
png with header "\x50\x4e\x47?" and footer "\xff\xfc\xfd\xfe" --> 0 files
bmp with header "BM??\x00\x00\x00" and footer "" --> 0 files
bmp with header "BM??\x04\x00\x00" and footer "" -> 1 files
tif with header "\x49\x49\x2a\x00" and footer "" --> 0 files
tif with header "\x4d\x4d\x00\x2a" and footer "" --> 0 files
Carving files from image.
Image file pass 2/2.
able3.home.blkls: 100.0% |*****| 91.3 MB 00:00
    ↳ ETAProcessing of image file complete. Cleaning up...
Done.
Scalpel is done, files carved = 8, elapsed = 1 secs.
```

Looking at the highlighted output above, we can see that a total of eight files were carved this time. The bitmap definition we added clearly shows the **scalpel.conf** file can be improved on. It's also not difficult to do. Simply using **xxd** to find matching patterns in groups of files

can be enough for you to build a decent library of headers. This is particularly useful if you come across many proprietary formats.

Given that carving can be approached with a variety of algorithms, it might be a good idea to run your data through more than one tool. For this, we'll now look at **photorec**.

9.5.2 photorec

Part of the **testdisk** package, **photorec** is another carving program. It does, however, take a very different approach. **photorec** was not originally designed as a forensic utility, but rather as a data recovery tool for people who lose files from SD cards and other media. It has evolved into a very useful tool for extracting many different files from media. As part of the **testdisk** package, it is installed alongside the **testdisk** tool itself (for recovering partitions), **fidentity** (same basic idea as the **file** command, but less verbose), and **qphotorec**. **qphotorec** is a GUI front end to **photorec**.

Figure 17: **qphotorec** - GUI front end for **photorec**



We will, of course, be sticking to the command line version here (which is actually menu driven). We can compare the output received from **scalpel** with the output from **photorec**

by running the carve on the same `home.blkls` unallocated data from our `able_3` disk image. First, log in as root (**su -**) and install the `testdisk` package with **sbinstall**:

```
barry@forensicbox:carve$
```

```
su -
```

```
Password:
```

```
root@forensicbox:~# sbinstall testdisk
```

```
TestDisk is a powerful free data recovery software. It was primarily
designed to help recover lost partitions and/or make non-booting
disks bootable again when these symptoms are caused by faulty
software, certain types of viruses or human error (such as
accidentally deleting a Partition Table). Partition table recovery
using TestDisk is really easy.
```

```
PhotoRec is file data recovery software designed to recover lost files
including video, documents and archives from Hard Disks and CDROM and
lost pictures from digital camera memory.
```

```
If you want to enable the use of sudo run the script with SUDO=true
```

```
libewf is an optional dependency.
```

```
...
```

```
Proceed with testdisk? [y]
```

```
...
```

```
Package testdisk-7.0-x86_64-1_SBo.tgz installed.
```

```
Cleaning for testdisk-7.0...
```

The SlackBuild specifies that `libewf` is an optional dependency, and since we already have `libewf` installed, it will be detected and compiled in for EWF support.

Running **photorec** from the command line is simple. We'll use an option for creating a log file using `/log` (created in the current directory) and providing an output directory `/d <dirname>` (we'll use `photorec_out`). We will also point the program directly at the `able3.home.blkls` unallocated data from `able_3`. This will drop us into the **photorec** menu.

```
barry@forensicbox:~$ photorec /log /d photorec_out able3.home.blkls*)
```

The main menu appears with the `able3.home.blkls` file already selected and loaded (See Figure 18). We'll go through the menu options quickly. It's all fairly self explanatory, and additional details can be found at http://www.cgsecurity.org/wiki/PhotoRec_Step_By_Step.

Normally, the main menu would include disk partitions from internal disks and removable

Figure 18: **photorec** main menu

```
PhotoRec 7.0, Data Recovery Utility, April 2015
Christophe GRENIER <grenier@cgsecurity.org>
http://www.cgsecurity.org

PhotoRec is free software, and
comes with ABSOLUTELY NO WARRANTY.

Select a media (use Arrow keys, then press Enter):
>Disk able3.home.blkls - 95 MB / 91 MiB (R0)

>[Proceed] [Quit]

Note: Some disks won't appear unless you're root user.
Disk capacity must be correctly detected for a successful recovery.
If a disk listed above has incorrect size, check HD jumper settings, BIOS
detection, and install the latest OS patches and disk drivers.
```

media, but since we specifically called the `able3.home.blkls` file, it is loaded by default (See Figure 19). Select `[Proceed]` with the arrow keys and hit `<enter>`.

Figure 19: **photorec** running on the `able3.home.blkls` file

```
PhotoRec 7.0, Data Recovery Utility, April 2015
Christophe GRENIER <grenier@cgsecurity.org>
http://www.cgsecurity.org

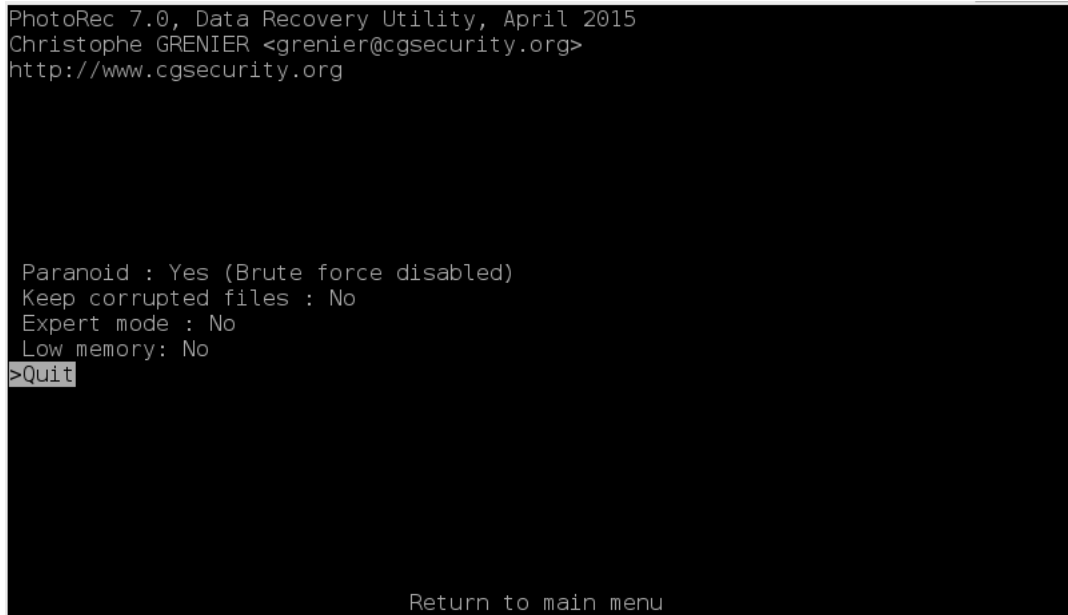
Disk able3.home.blkls - 95 MB / 91 MiB (R0)

  Partition      Start      End      Size in sectors
> P Unknown      0  0  1    11 162 45    186966

>[Search] [Options] [File Opt] [Quit]
                        Start file recovery
```

If this were a full disk image, **photorec** would display the file systems and partitions contained in the image. In this case, it is simply unallocated data and there is no partition to display. Select `[Options]` and hit `<enter>`.

The options provided are (See Figure 20):

Figure 20: Viewing **photorec** options in the running program

```
PhotoRec 7.0, Data Recovery Utility, April 2015
Christophe GRENIER <grenier@cgsecurity.org>
http://www.cgsecurity.org

Paranoid : Yes (Brute force disabled)
Keep corrupted files : No
Expert mode : No
Low memory: No
>Quit

Return to main menu
```

- **Paranoid**: Used to validate files that are carved. We'll leave it as **Yes** for now.
- **Keep corrupted files**: In normal use you might want to enable this just to be safe (collect as much data as possible). I've never found it particularly useful.
- **Expert mode**: Provides additional options for setting specific disk geometry. Unless you are working with a corrupt disk image with a mangled partition table, you can leave this at **No**.
- **Low memory**: For really large disk images where memory becomes an issue.

Obviously feel free to play with the options and explore the different menus. For this simple exercise, leaving the defaults as is will work just fine.

Return to the main menu by selecting **>Quit** and from the main menu choose **[File Opt]** and hit **<enter>**.

This will bring you to the file selection menu (See Figure 21). **photorec** will recover almost five hundred different file signatures. You can select or deselect from this menu. For now we'll leave the default file selections in place (there are a few deselected by default). Select **[Quit]** again to return to the main menu. At the main menu, select **[Search]** and hit **<enter>**.

This is where we select the file system type (See Figure 22). We'll choose **[ext2/ext3]** and hit **<enter>**, starting the search.

Once the search is complete, you will see the number of files recovered, and the output directory (**photorec_out**, which we specified on our command line). The carve is now complete.

Figure 21: **photorec** file options menu

```

PhotoRec 7.0, Data Recovery Utility, April 2015
Christophe GRENIER <grenier@cgsecurity.org>
http://www.cgsecurity.org

PhotoRec will try to locate the following files

>[X] custom Own custom signatures
[X] lcd Russian Finance 1C:Enterprise 8
[X] 3dm Rhino / openNURBS
[X] 7z 7zip archive file
[X] DB
[X] a Unix Archive/Debian package
[X] abr Adobe Brush
[X] acb Adobe Color Book
[X] accdb Access Data Base
[X] ace ACE archive
[X] ab MAC Address Book
[X] ado Adobe Duotone Options
[X] ahn Ahnenblatt
[X] aif Audio Interchange File Format
Next
Press s to disable all file families, b to save the settings
>[ Quit ]
Return to main menu

```

Figure 22: **photorec** file system selection

```

PhotoRec 7.0, Data Recovery Utility, April 2015
Christophe GRENIER <grenier@cgsecurity.org>
http://www.cgsecurity.org

P Unknown          0  0  1    11 162 45    186966

To recover lost files, PhotoRec need to know the filesystem type where the
file were stored:
[ ext2/ext3 ] ext2/ext3/ext4 filesystem
>[ Other ] FAT/NTFS/HFS+/ReiserFS/...

```

Select [Quit] in the subsequent menus and exit the program You'll be dropped back at the command prompt (See Figure 23)

Looking at a directory listing, you can see we now have a new output directory, **photorec_out.1/** along with a log file that was created with the **/log** option. Have a look at the log file, **photorec.log** with the **less** command.

```
barry@forensicbox:carve$ ls -l
```

Figure 23: **photorec** file system selection

```

PhotoRec 7.0, Data Recovery Utility, April 2015
Christophe GRENIER <grenier@cgsecurity.org>
http://www.cgsecurity.org

Disk able3.home.blkls - 95 MB / 91 MiB (R0)
  Partition      Start      End    Size in sectors
  P Unknown      0  0  1    11 162 45    186966

5 files saved in photorec_out directory.
Recovery completed.

You are welcome to donate to support further development and encouragement
http://www.cgsecurity.org/wiki/Donation

[ Quit ]

```

```

total 93516
-rw-r--r-- 1 barry users 95726592 Aug  4 12:35 able3.home.blkls
-rw-r--r-- 1 barry users    2825 Aug  4 15:07 photorec.log
drwxr-xr-x 2 barry users    4096 Aug  4 15:07 photorec_out.1/
drwxr-xr-- 2 barry users    4096 Aug  4 12:38 scalp_out/
drwxr-xr-- 2 barry users    4096 Aug  4 12:41 scalp_out2/
-rwxr-xr-x 1 barry users   12967 Aug  4 12:41 scalpel.conf*

```

```
barry@forensicbox:carve$ less photorec.log
```

```
...
```

```
Sun Aug  4 12:56:02 2019
```

```
Command line: PhotoRec /log /d photorec_out able3.home.blkls
```

```
PhotoRec 7.0, Data Recovery Utility, April 2015
```

```
Christophe GRENIER <grenier@cgsecurity.org>
```

```
http://www.cgsecurity.org
```

```
Disk able3.home.blkls - 95 MB / 91 MiB - CHS 12 255 63 (R0), sector size=512
```

```
...
```

```
blocksize=1024, offset=0
```

```
Elapsed time 0h00m00s
```

```
Pass 1 (blocksize=1024) STATUS_EXT2_ON
```

```
photorec_out.1/f0012156.gif 12156-12205
```

```
photorec_out.1/f0012206.jpg 12206-12261
```

```
photorec_out.1/f0012262.jpg 12262-12293
```

```
photorec_out.1/f0012294.bmp 12294-12863
```

```
photorec_out.1/f0012904.gz 12904-186965
```

```
Elapsed time 0h00m01s
```

```

Pass 1 +5 files
jpg: 2/4 recovered
bmp: 1/1 recovered
gif: 1/1 recovered
gz: 1/1 recovered
Total: 5 files found

```

```
12196 sectors contains unknown data, 2 invalid files found and rejected.
```

Like **scalpel**, the log output provides suitable information for inclusion in a report if needed, note that the offset locations for each carved file are given in sector offset rather than byte offset (multiply each offset given above by 512 to compare the offsets with the **scalpel** audit.txt file).

Have a look at the output of **photorec**:

```

barry@forensicbox:carve$ ls photorec_out.1/
f0012156.gif
f0012206.jpg
f0012262.jpg
f0012294.bmp
f0012904_lrkn.tar.gz
report.xml

```

The contents of the output directory show **photorec** recovered not only a few image files, but also a file called **f0012904_lrkn.tar.gz**. If you recall our **able_3** exercise, you'll remember that this was a file of some interest. **photorec** is useful for far more than just a few images. If you try and untar/extract the file, you'll find it's corrupted. Some of it, however, is still recoverable.

```

barry@forensicbox:carve$ tar tzvf photorec_out.1/f0012904_lrkn.tar.gz
drwxr-xr-x lp/lp          0 1998-10-01 18:48 lrk3/
-rwxr-xr-x lp/lp          742 1998-06-27 11:30 lrk3/1
-rw-r--r-- lp/lp          716 1996-11-02 16:38 lrk3/MCONFIG
-rw-r--r-- lp/lp        6833 1998-10-03 05:02 lrk3/Makefile
-rw-r--r-- lp/lp        6364 1996-12-27 22:01 lrk3/README
...
gzip: stdin: decompression OK, trailing garbage ignored
-rw-r--r-- lp/lp          1996 1996-11-02 16:39 lrk3/z2.c
tar: Child returned status 2
tar: Error is not recoverable: exiting now

```

There is still much information that can be gleaned from the recovery of this file. You can see the **README** is one of those files recovered. We can use this to define strings for us to search and perhaps discover where the archive was decompressed and extracted (which we

did earlier in our physical search exercise). This is one of the reasons we elect to use more than one carving utility. Differences in output can strengthen our analysis.

One question you might find yourself asking is "How do I efficiently compare carve output from two different tools to get an accurate count of files recovered?". In our very small sample produced by the exercises here, it's a fairly simple job. We just compare the image files in a graphical viewer. There are a little over a dozen total images to review. If, however, we were to carve a disk image with hundreds of unallocated image files, the comparison would be far more difficult. To address this, let's have a look at a simple program that will do the work for us.

9.5.3 Comparing and De-duplicating Carve Output

Obviously this is not a simple matter of comparing file names. The files are carved from the data blocks without any regard to directory entries or other file system information. So the tools use their own naming scheme. Interestingly **photorec** included the name of the original **lrkn.tar.gz** name of the **tar** archive in its output. This is because the name of the file is part of the file metadata (run file **f0012904_lrkn.tar.gz** and you'll see the **gzip** header contains the name).

One thing we can do is compare hashes. If hashes match, regardless of file name, then we know we have two of the same files. One simple way to do this would be to hash all the files in each directory (**photorec_out** and **scalp_out2**) and write them to a file. We could then sort this file by the hash and look for duplicates. This can be done in one command. Note that we use **f0*** and **0*** for the **md5sum** command in each directory so that we get just the carve output files and not the log/audit files from each tool.

```
barry@forensicbox:carve$ md5sum photorec_out.1/f0*scalp_out2/0*| sort
110983800a177c1746c54b15edec989a photorec_out.1/f0012156.gif
110983800a177c1746c54b15edec989a scalp_out2/00000000.gif
2d7d4def42fcbcc0c813a27505f0508b photorec_out.1/f0012904_lrkn.tar.gz
357ca99e654ca2b179e1c5a0290b1f94 photorec_out.1/f0012262.jpg
357ca99e654ca2b179e1c5a0290b1f94 scalp_out2/00000004.jpg
437a614c352b03a6a4575e9bbc2070ae photorec_out.1/f0012206.jpg
437a614c352b03a6a4575e9bbc2070ae scalp_out2/00000003.jpg
6742ca9862a16d82fdc4f6d54f808f41 scalp_out2/00000007.bmp
a0794399a278ce48bfbd3bd77cd3394d scalp_out2/00000002.jpg
aa607253fc9b0a70564228ac27ad0b13 scalp_out2/00000006.jpg
b5ca633bea09599c3fb223b4187bb544 photorec_out.1/f0012294.bmp
b6703670db3f13f23f7a3ed496a2b95c scalp_out2/00000001.jpg
f979cd849ccdd5c00fd396b600a9a283 scalp_out2/00000005.jpg
```

By sorting the output, the duplicate hashes are listed together. From the output above we can see that these two files are identical:

```
110983800a177c1746c54b15edec989a photorec_out.1/f0012156.gif
110983800a177c1746c54b15edec989a scalp_out2/00000000.gif
```

This can be re-directed to a file for later processing.

```
barry@forensicbox:carve$ md5sum photorec_out.1/f0*scalp_out2/0* | sort >
carvehash.txt
```

Well, this is fine. But it might also be nice to actually de-duplicate the files by removing one of the duplicates. Again, easy enough in our small sample here, but far more challenging and time consuming if you are dealing with hundreds or thousands of contraband images you need to sort and accurately count.

For this we can use a program called **fdupes**. **fdupes** works using both filenames and hashes to find, report, and if requested – remove duplicate files from user specified directories. It is easy to use and very effective.

```
barry@forensicbox:carve$ su -
Password:
```

```
root@forensicbox:~# sbinstall fdupes
```

```
FDUPES is a program for identifying or deleting duplicate files residing
within specified directories.
```

```
Proceed with fdupes? [y]
```

```
...
```

```
Package fdupes-1.6.1-x86_64-1_SBo.tgz installed.
```

```
Cleaning for fdupes-1.6.1...
```

We will run **fdupes** twice (always good practice). The first run will show all the duplicated files, each pair on a single line. Review the output to ensure there are no unexpected files, and then re-run the command with the **--delete** option.

```
barry@forensicbox:carve$ fdupes -R -1 photorec_out.1/ scalp_out2/
scalp_out2/00000004.jpg photorec_out.1/f0012262.jpg
scalp_out2/00000000.gif photorec_out.1/f0012156.gif
scalp_out2/00000003.jpg photorec_out.1/f0012206.jpg
```

The options we pass are **-R** for recursion. There are no sub folders in this example, but it never hurts to allow recursion. Particularly on large scale examinations where carve output can be quite massive and you might have specified categorized output for **scalpel** in particular (different file types in different directories). We also use the **-1** option to put matches on the same line. This is personal preference. Run without this option and see what you prefer.

Once the output has been previewed, re-run the command with the **--delete** option to keep only the first file in each pair (or set). If you've reviewed the output prior to deleting, then you might want to add the **-N** option for "no prompt". Use at your own discretion. Without **-N**, if you have hundreds of pairs of matching files, you'll need to confirm each deletion.

```
barry@forensicbox:~$ fdupes -R -N -delete photorec_out.1/ scalp_out2/
```

```
[+] scalp_out2/00000004.jpg
[-] photorec_out.1/f0012262.jpg

[+] scalp_out2/00000000.gif
[-] photorec_out.1/f0012156.gif

[+] scalp_out2/00000003.jpg
[-] photorec_out.1/f0012206.jpg
```

The output above indicates that the first file has been kept **[+]** and the second file deleted **[-]**. If there were more than one matching file in each set, then only the first would remain. To better control this behavior, remove the **-N** option and you can select which files to keep.

This concludes our physical carving section. We've learned how to carve files from unallocated space, view the files, sort them, and remove duplicates in an efficient manner.

9.6 Application Analysis

We've now covered several of the layers we discussed previously, including the physical and media management layers for disk information and partition layout; file system tools for gathering information on the file system statistics; and tools to work on individual files to search content and identify file types of interest. We've even done some data recovery at the physical block layer – regardless of volume and file system through carving and extensive searches. So now that we've recovered files, what do we do with them?

This is where the application layer of our analysis model comes in. For our purposes here, the term "application" can be thought of as operating system or user interactive files - that is: files that are created by applications accessed by the operating system or through user interaction (either with the operating system or external software).

In simplest terms, application analysis can be as simple as viewing the file directly for content – we've used **catdoc** and **catdov** for MS Office files, various image viewers like **geeqie**, **xv** and **display** for pictures, and simple text viewers like **less** for ASCII files. But forensic analysis is much more than simply recovering files and displaying the content. That sort of activity is really just *data recovery*. Digital forensics, however, needs to include other techniques:

- temporal analysis (*when* did it happen?)

- attribution (*who* made it happen?)
- activity mapping (*how* did it happen?)

Obviously we can glean some of this information through the analysis we've done already, using file times we see in the **istat** output or the location of files in a particular user's home directory or **Users** folder.

In order to dig a little deeper, we are going to have a look at some simple applications that will allow us to peer into the Windows Registry, Windows Event logs, and other artifacts to obtain additional forensically useful information. We'll do this using some utilities from the **libyal** project.

You can read more about **libyal** at <https://github.com/libyal/libyal/wiki>. There are a couple of important notes on these libraries we need to cover before we begin. First and foremost, make sure you understand that many of these libraries are in *alpha* or experimental status, meaning they are not fully matured and, as the above site very clearly states, are subject to break and/or change. The projects we will look at here are in alpha status. They have been tested on some simple sample files, but make sure that you test them in your own environment prior to use. These are excellent projects, and well worth keeping up with, but make sure you know what you are doing (and seeing) before using them in production. Using software that is clearly marked alpha or experimental is not recommended for production case work unless you understand and test the output for yourself. For the time being, these make for excellent tertiary cross-verification tools and vehicles for learning specific artifacts and structure.

9.6.1 Registry Parsing Exercise 1: UserAssist

Let's start our exploration of **libyal** and application analysis by looking at specific Windows registry files.

As usual, we start with the disclaimer that this section is not about learning registry forensics. It's about the tools. Of course you might gain some knowledge along the way, but that is not our purpose here. If you want to look deeper into these registry files and learn more about the art of registry forensics, then I strongly suggest you look to the excellent book³¹ written by Harlan Carvey on the subject (and browse his blog³²). You might want to have a basic understanding of registry structure before you begin this exercise, so you have some context for what's to come. And, of course there are other (faster and more comprehensive) ways to parse a registry. For example, Harlan Carvey's well known **RegRipper** will run just fine on Linux.

³¹<https://www.elsevier.com/books/windows-registry-forensics/carvey/978-0-12-803291-6>

³²<http://windowsir.blogspot.com/>

Our real purpose in this section is to show you how to do this sort of analysis at the byte level, using some common Linux tools like **xxd** and **tr**, rather than relying on more automated tools to do it for you. What we do here is not much different from what the Perl scripts in **RegRipper** do (although we simplify it somewhat here).

First, though, we need to have a registry file to work on. We'll start with the **NTUSER.DAT** file from the **AlbertE** account in our NTFS file system sample (**NTFS_Pract_2017.E01**).

We need to make sure we obtain the correct **NTUSER.DAT**. There are a couple of ways we can locate and extract the file from a disk image. You can mount the image (in our case using **ewfmount**), browse to the file and extract by copying it out of the mounted file system. This requires a few more steps than we need to do though, so we'll demonstrate it here with two simple location methods, and then extract the file with **icat**.

Since we are targeting the **AlbertE** account, and we know that a specific user's **NTUSER.DAT** file is in the **/Users/\$USERNAME/** folder, we can use **ifind** to target the specific file by name. To run **ifind**, we use **mmls** as we did previously to find the offset to the file system in our image:

```
barry@forensicbox:~$ cd NTFS_Pract_2017

barry@forensicbox:NTFS_Pract_2017$ mmls NTFS_Pract_2017.E01
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors
```

	Slot	Start	End	Length	Description
000:	Meta	0000000000	0000000000	0000000001	Primary Table (#0)
001:	-----	0000000000	0000002047	0000002048	Unallocated
002:	000:000	0000002048	0001023999	0001021952	NTFS / exFAT (0x07)

```
barry@forensicbox:NTFS_Pract_2017$ ifind -n "users/alberte/ntuser.dat" -o 2048
NTFS_Pract_2017.E01
285
```

So here we use **ifind** (find the "inode", or meta-data structure) using **-n** to find by name, at the **2048** offset we again found in our NTFS file system image by running **mmls**. The return value we get from **ifind** is **285**, the MFT entry for the **AlbertE** account's **NTUSER.DAT** file.

Alternatively, if you want to search for all the **NTUSER.DAT** files on a system, you could use **fls** with the option to recursively list all regular files (**-Fr**), grepping the output for **NTUSER.DAT**. In either case, we again find the MFT entry for **AlbertE**'s **NTUSER.DAT** is **285**:

```
barry@forensicbox:NTFS_Pract_2017$ fls -Fr -o 2048 NTFS_Pract_2017.E01 | grep
NTUSER.DAT
r/r 285-128-2: Users/AlbertE/NTUSER.DAT
r/r 286-128-2: Users/ElsaE/NTUSER.DAT
```

Once you've identified the MFT entry using one of the two methods above, you can simply extract the file with **icat**, arbitrarily naming the output (we use **NTUSER.285** here). Run the **file** command to check the resulting type:

```
barry@forensicbox:~$ icat -o 2048 NTFS_Pract_2017/NTFS_Pract_2017.E01 285 >
NTUSER.285

barry@forensicbox:~$ file NTUSER.285
NTUSER.285: MS Windows registry file, NT/2000 or above
```

Now that we have the registry file we want we can choose a specific key to search for useful information. As an example, we'll look at the **UserAssist** entries. These entries occur in the registry when a user executes a program from the desktop. **UserAssist** entries are located at **Software\Microsoft\Windows\CurrentVersion\Explorer**. For a complete explanation, I refer you again to the aforementioned book by Harlan Carvey.

So we have our registry file, **NTUSER.DAT**, and target key, **UserAssist**. We need software to access the data. For this, we install **libregf**:

```
barry@forensicbox:~$ su -
Password:

root@forensicbox:~# sbinstall libregf
...
Cleaning for libregf-20190303...

root@forensicbox:~# exit
```

You can have a look at the utilities that were installed by this package by looking at the package file in **/var/log/packages**:

```
barry@forensicbox:~$ grep usr/bin /var/log/packages/libregf-20190303-x86_64-1_SBo
usr/bin/
usr/bin/regfexport
usr/bin/regfinfo
usr/bin/regfmount
```

We can see that the package came with three executable programs placed in **/usr/bin**. We will concentrate on using **regfmount**. Much like **libewf's ewfmount** (which is also part of the **libyal** project) **regfmount** provides a fuse file system interface to a file object, in this case a registry file. The usage is very similar. First, we'll create a mount point in our current directory, followed with the registry being mounted:

```
barry@forensicbox:~$
mkdir regmount
```

```
barry@forensicbox:~$ regfmount NTUSER.285 regmount/
regfmount 20190303
```

```
barry@forensicbox:~$ cd regmount
```

```
barry@forensicbox:regmount$ ls
AppEvents/  Control\ Panel/  Environment/  Keyboard\ Layout/  Printers/  System/
Console/    EUDC/            Identities/   Network/           Software/
```

The registry file NTUSER.285 is mounted using **regfmount** on the mount point we created, **regmount** (in the current directory). When we change to the **regmount** directory, we see the contents of the registry file in the same sort of hierarchical structure as would be found in any other registry viewer. This we can now navigate and view using normal command line utilities. So let's navigate to the **UserAssist** key and view the contents.

```
barry@forensicbox:~$ cd
Software/Microsoft/Windows/CurrentVersion/Explorer/UserAssist/

barry@forensicbox:regmount/Software/Microsoft/Windows/CurrentVersion/Explorer/UserAssist$
ls
{CEBFF5CD-ACE2-4F4F-9178-9926F41749EA}/
{F4E57C4B-2036-45F0-A9AB-443BCFE33D9F}/
```

You can see once we change into that directory our prompt is quite long! I'm going to abbreviate the command prompt with ... to make the lines more readable. When we run our **ls** command, we see two cryptic looking directory (GUID) entries. Change directory into **F4E57C4B-2036-45F0-A9AB-443BCFE33D9F**³³ and the sub directory **Count/(values)**. Note that when you type the **(values)** sub directory, you will need to escape the parentheses with ****, so you will use **\(values\)**.

```
barry@forensicbox:...UserAssist$ cd \{F4E57C4B-2036-45F0-A9AB-443BCFE33D9F\}/

barry@forensicbox:...443BCFE33D9F\}$ cd Count/\(values\)/

barry@forensicbox:... (values)$
```

Now have a look at the contents of this directory.

```
barry@forensicbox:... (values)$ ls
HRZR_PGYFRFFVBA
HRZR_PGYPHNPbhag:pgbe
{0139Q44R-6NSR-49S2-8690-3QNSPNR6SS08}\\\\\\Jvaqbjf\ Snk\ naq\ Fpna.yax
```

³³This is where bash completion comes in real handy. When using the **cd** command here, type the first two characters and hit the **<tab>** key... The rest will fill in automatically. Best. Feature. Ever

```
{0139Q44R-6NSR-49S2-8690-3QNSPNR6SS08}\\\\KCF\\ Ivrjre.yax
{0139Q44R-6NSR-49S2-8690-3QNSPNR6SS08}\\\\Npprffbevrf\\\\Cnvag.yax
{0139Q44R-6NSR-49S2-8690-3QNSPNR6SS08}\\\\Npprffbevrf\\\\Erzbgr\\ Qrfxgbc\\
    ↪ Pbaarpgvba.yax
{0139Q44R-6NSR-49S2-8690-3QNSPNR6SS08}\\\\Npprffbevrf\\\\Favccvat\\ Gbby.yax
{0139Q44R-6NSR-49S2-8690-3QNSPNR6SS08}\\\\Npprffbevrf\\\\Fgvpxl\\ Abgrf.yax
{0139Q44R-6NSR-49S2-8690-3QNSPNR6SS08}\\\\Npprffbevrf\\\\Jrypbzr\\ Prague.yax
{0139Q44R-6NSR-49S2-8690-3QNSPNR6SS08}\\\\Npprffbevrf\\\\Pnyphyngbe.yax
{0139Q44R-6NSR-49S2-8690-3QNSPNR6SS08}\\\\Npprffbevrf\\\\qvfcynlfjvgpu.yax
{0139Q44R-6NSR-49S2-8690-3QNSPNR6SS08}\\\\Nqzvavfgengvir\\ Gbbyf\\\\Pbzchgre\\
    ↪ Znantrzag.yax
{9R3995N0-1S9P-4S13-0827-4802406P7174}\\\\Gnfx0ne\\\\Jvaqbjf\\ Rkcybere.yax
{9R3995N0-1S9P-4S13-0827-4802406P7174}\\\\Gnfx0ne\\\\Tbbtyr\\ Puebzr.yax
{9R3995N0-1S9P-4S13-0827-4802406P7174}\\\\Gnfx0ne\\\\Vagrearg\\ Rkcybere.yax
{9R3995N0-1S9P-4S13-0827-4802406P7174}\\\\Gnfx0ne\\\\Zbmvyyn\\ Sversbk.yax
{N77S5Q77-2R20-44P3-N6N2-N0N601054N51}\\\\Npprffbevrf\\\\Npprffvovyvgl\\\\Zntavsl.
    ↪ yax
{N77S5Q77-2R20-44P3-N6N2-N0N601054N51}\\\\Npprffbevrf\\\\Pbzznaq\\ Cebzcg.yax
```

So if you did any reading on this particular registry key, you'll find that the above entries (or "files" in our fuse mounted file system) are ROT 13 obfuscated. This means that the characters in each string above are swapped a-m or A-M for the corresponding n-z or N-Z, so an a becomes an n and a b becomes an o, and so on. We can de-obfuscate this text with the **tr** command we've used previously to replace one character with another. In this case we'll be replacing characters n-za-m with a-z, etc. Let's try this on the repeating string at the end of every line, .yax:

```
barry@forensicbox:... (values)$ echo ".yax" | tr 'n-za-mN-ZA-M' 'a-zA-Z'
.lnk
```

We can see that the .yax string at the end of each line is actually the .lnk file extension (indicating a link or shortcut file).

So what's the best way to run the above **tr** command on all the files in the /Count/(values) directory? We can go back to the short bash loop we introduced in the *Viewing Files* section of this guide:

```
barry@forensicbox:... (values)$ for file in *
    > do
    > echo $file | tr 'n-za-mN-ZA-M' 'a-zA-z'
    > done
UEME_CTLSESSION
UEME_CTLCUACount:ctor
{0139D44E-6AFE-49F2-8690-3DAFCAE6FFB8}\\\\Windows Fax and Scan.lnk
{0139D44E-6AFE-49F2-8690-3DAFCAE6FFB8}\\\\XPS Viewer.lnk
{0139D44E-6AFE-49F2-8690-3DAFCAE6FFB8}\\\\Accessories\\\\Paint.lnk
{0139D44E-6AFE-49F2-8690-3DAFCAE6FFB8}\\\\Accessories\\\\Remote Desktop Connection.lnk
```

```
{0139D44E-6AFE-49F2-8690-3DAFCAE6FFB8}\\Accessories\\Snipping Tool.lnk
{0139D44E-6AFE-49F2-8690-3DAFCAE6FFB8}\\Accessories\\Sticky Notes.lnk
{0139D44E-6AFE-49F2-8690-3DAFCAE6FFB8}\\Accessories\\Welcome Center.lnk
{0139D44E-6AFE-49F2-8690-3DAFCAE6FFB8}\\Accessories\\Calculator.lnk
{0139D44E-6AFE-49F2-8690-3DAFCAE6FFB8}\\Accessories\\displayswitch.lnk
{0139D44E-6AFE-49F2-8690-3DAFCAE6FFB8}\\Administrative Tools\\Computer Management.
    ↪ lnk
{9E3995AB-1F9C-4F13-B827-48B24B6C7174}\\TaskBar\\Windows Explorer.lnk
{9E3995AB-1F9C-4F13-B827-48B24B6C7174}\\TaskBar\\Google Chrome.lnk
{9E3995AB-1F9C-4F13-B827-48B24B6C7174}\\TaskBar\\Internet Explorer.lnk
{9E3995AB-1F9C-4F13-B827-48B24B6C7174}\\TaskBar\\Mozilla Firefox.lnk
{A77F5D77-2E2B-44C3-A6A2-ABA601054A51}\\Accessories\\Accessibility\\Magnify.lnk
{A77F5D77-2E2B-44C3-A6A2-ABA601054A51}\\Accessories\\Command Prompt.lnk
```

For review, the first line of a bash loop above means "**for** every file in the current directory (*), **do** the following **echo | tr** command", followed by the bash keyword **done** to close the loop.

You can see from the output that we've de-obfuscated the names. The de-obfuscated output matches the original output line for line. This means these are the same (third from the bottom)³⁴:

```
{9R3995N0-1S9P-4S13-0827-4802406P7174}\\\\Gnfx0ne\\\\Zbmvyyn\\ Sversbk.yax
{9E3995AB-1F9C-4F13-B827-48B24B6C7174} \\TaskBar \\Mozilla Firefox.lnk
```

That particular entry is for a link to Mozilla Firefox. The GUID value in the front of the file name represents the `FOLDERID_UserPinned` "known folder"². If we want to view the contents or "value" of the entry, we need to use the ROT-13 name on the command line. We can use **xxd** to see the raw values in hex.

```
barry@forensicbox:...(values)$
xxd \{9R3995N0-1S9P-4S13-0827-4802406P7174\}\\\\Gnfx0ne\\\\Zbmvyyn\\Sversbk.yax
00000000: 0000 0000 0400 0000 0000 0000 0400 0000  .....
00000010: 0000 80bf 0000 80bf 0000 80bf 0000 80bf  .....
00000020: 0000 80bf 0000 80bf 0000 80bf 0000 80bf  .....
00000030: 0000 80bf 0000 80bf ffff ffff c03a bc03  .....:..
00000040: f8be d201 0000 0000  .....

```

A count of the number of times this link was used can be found at offset `0x04` (highlighted in yellow). So this link was accessed 4 times, according to this entry. The date in Windows FILETIME format can be found at offset `0x3c` (highlighted in blue).

While the access count at `0x04` is easy to decipher, the Windows date value is not. I use a

³⁴The extra escape (`\`) characters in the obfuscated output is because the **ls** command escapes the spaces. The **echo** command used with the **tr** command does not.

small python script to decode the time value (the number of 100 nanosecond blocks since January 1, 1601). You can download the python script (WinTime) using **wget**:

```
barry@forensicbox:(values)$ wget http://www.linuxleo.com/Files/WinTime -O  
~/WinTime.py
```

Since we are currently in the **regmount** mount point, be sure to use the **wget -O** option to write the file to your home directory (**~/WinTime.py**). You don't want to try and download the file to the current directory (it's our fuse mounted registry mount point).

Once you have the script, you can copy the hex value and provide it as an argument to **WinTime.py**. Be sure to remove the spaces from the value (we'll use an alternative way of getting this value from **xxd** later):

```
barry@forensicbox:~$ python ~/WinTime.py c03abc03f8bed201  
Thu Apr 27 01:45:57 2017
```

If you did a full install of Slackware, Python should already be on your system. The python command points to the **WinTime.py** file we previously named with **wget -O**. The **~** indicates the file is in our home directory. This leaves us with a last execution time of April 27 at approximately 01:45. A complete forensic education regarding registry entries, interpreting dates and times, and timezone adjustment is far outside the scope of this guide, but make sure you take time settings, time zones and clock skew into account for any forensic examination where dates are meaningful. File dates and time stamps are one of the pitfalls of analysis. Read up on the subject completely before making any interpretations.

At this point it's time to unmount our fuse mounted registry file. We'll use the same **regmount** mount point in the next exercise.

```
barry@forensicbox:~$ fusermount -u regmount
```

9.6.2 Registry Parsing Exercise 2: SAM and Accounts

Let's look at another registry file, the **SAM** hive. The **SAM** hive can have a great deal of information available if there are local accounts present on the system. Again, we're not going to go through a comprehensive analysis, we're just going to have a look at a few values of one of the more important keys.

We can grab the **SAM** hive the same way we did the **NTUSER.DAT**, first searching for the proper MFT entry using **fls** and then using **icat** to extract the file:)

```
barry@forensicbox:~$ fls -Fr -o 2048 NTFS_Pract_2017/NTFS_Pract_2017.E01 | grep
SAM
r/r 178-128-2: Windows/System32/config/SAM
```

So our target MFT entry here is 178. Now we'll extract with **icat** and check the file type again with the **file** command. The file name we use on the extracted file is arbitrary. Name it however you like. Consistency is a good idea, though.

```
barry@forensicbox:~$ icat -o 2048 NTFS_Pract_2017/NTFS_Pract_2017.E01 178 >
SAM.178
```

```
barry@forensicbox:~$ file SAM.178
SAM.178: MS Windows registry file, NT/2000 or above
```

We will use the SAM mount point for this exercise as we did the previous.

```
barry@forensicbox:~$ regfmount SAM.178 regmount/
regfmount 20190303
```

Since we already pulled the NTUSER.DAT file for the AlbertE account, let's have a look at the same account in the SAM file. If we change directories down to SAM/Domains/Account/Users, we'll see the following list of potential accounts:

```
barry@forensicbox:~$ cd regmount/SAM/Domains/Account/Users/

barry@forensicbox:...Users$ ls
(values)/ 000001F4/ 000001F5/ 000003E8/ 000003E9/ Names/
```

What we see in the output above are a series of sub keys (those starting with 00000* that represent the hex value of account **Relative ID**. We can translate these with **bc**, as we would any hex value:

```
echo "ibase=16; 000001F4" | bc
```

But let's do it all at once with a **for** loop to repeat the command across all the directories (but only those that are a hex value):

```
barry@forensicbox:...Users$ for name in 00000*
> do
> echo "ibase=16; $name" | bc
> done
500
```

```
501
1000
1001
```

If you read up on Windows accounts, you'll see we have the system administrator (RID 500), the guest account (RID 501), and a pair of user accounts (1000 and 1001). There are a number of ways to associate the accounts with particular users, but we will simply navigate to the values under the 000003E8/ sub key.

```
barry@forensicbox:...Users$ cd 000003E8/(values\)/
```

```
barry@forensicbox:...(values)$ ls
F  UserPasswordHint  V
```

We have three "files" here to look at. A very quick peek at the bottom of the V file shows the username associated with this account:

```
barry@forensicbox:(values)$ xxd V
...
00000160: 0000 0001 0000 0000 0102 0000 0000 0005 .....
00000170: 2000 0000 2002 0000 0102 0000 0000 0005 ... ..
00000180: 2000 0000 2002 0000 4100 6c00 6200 6500 ... ..A.l.b.e.
00000190: 7200 7400 4500 0000 0102 0000 0700 0000 r.t.E.....
000001a0: 0300 0100 0300 0100 13c4 df6f 671a 70d2 .....og.p.
000001b0: 0c04 49e1 c16e c39a 0300 0100 0300 0100 ..I..n.....
```

The red text shows the associated account as that of AlbertE. The UserPasswordHint is fairly obvious. But let's have a look at the contents of F:

```
barry@forensicbox:~$ xxd F
00000000: 0200 0100 0000 0000 678e 5df7 f7c1 d201 .....g.].....
00000010: 0000 0000 0000 0000 20d7 bf15 76ae d201 .....v...
00000020: ffff ffff ffff ff7f 5ce9 5df2 f7c1 d201 .....\.].....
00000030: e803 0000 0102 0000 1402 0000 0000 0000 .....
00000040: 0000 0700 0100 0000 0000 4876 488a 3600 .....HvH.6.
```

Unlike the V or UserPasswordHint files, F does not display any obvious data. What you are seeing is account information for the user AlbertE, including:

- Last Login Date : offset 8
 - Password Set/Reset Date : offset 24
 - Last Failed Login : offset 40
-

...and other account information (number of logins, RID, etc.). We are going to concentrate on the dates at the offsets shown above. We've already converted similar dates using the python `WinTime.py` script. We could type each value on the command line, and run the script separately for each value. A better way, however, would be to use the command line to give us just the value we want, and pass each one to the `WinTime.py` script. We can do this with a bash `for` loop. And if you read the man page for `xxd`, you will see that we can also use different options for `xxd` to enable us to complete the date conversion without having to copy the hex value out.

Let's look at what happens if we run `xxd` with `-ps` (plain hexdump) `-s8` (seek to byte 8) `-l8` (output is 8 bytes in length). The command prompt has been truncated again for readability (F is the "file" we are viewing):

```
barry@forensicbox:(values)$ xxd -ps -s8 -l8 F
678e5df7f7c1d201
```

We find the date string extracted is exactly the format we need to pass to `WinTime.py`. In order to pass the output, we'll use command substitution. We can do this by using a subshell. By running the `xxd` command in a subshell, we can pass the value of the executed command similar to a variable using the `$`. This is done by using `$(...)` around the `xxd` command. This substitutes the output of `xxd` straight to the argument required for `WinTime.py`:

```
barry@forensicbox:~$ python ~/WinTime.py $(xxd -ps -s8 -l8 F)
Sun Apr 30 21:23:09 2017
```

This can be taken a step further. We have three separate date values to convert here. One is at offset 8 (`-s8` as we converted above). The others are at offset 24 and 40. Sounds like a perfect candidate for our now familiar bash `for` loop. We can use offsets 8, 24 and 40 as our variable, and pass those into our command substitution for `WinTime.py`. It should look something like this:

```
barry@forensicbox:(values)$ for offset in 8 24 40
> do
> python ~/WinTime.py $(xxd -ps -s$offset -l8 F)
> done
Sun Apr 30 21:23:09 2017
Thu Apr  6 01:35:34 2017
Sun Apr 30 21:23:01 2017
```

Instead of repeating the command with `-s8`, `-s24` and `-s40`, we simply create a loop with `$offset` and provide the values 8, 24, and 40 in the loop. This gives us the resulting values:

- Last Login Date : Sun Apr 30 21:23:09 2017

- Password Set/Reset Date : Thu Apr 6 01:35:34 2017
- Last Failed Login : Sun Apr 30 21:23:01 2017

Examining Windows registry files in a command line environment may not be the simplest method, but it is a great way to learn how a registry is parsed, and where information is located.

9.6.3 Application Analysis: prefetch

Understanding the caveats we provided earlier on the status of many of the `libyal` projects, be sure to browse some of them and try them out. Documentation can be sparse in places, but that's where experimentation and testing comes in. In many cases, the libraries are provided to add capabilities to other programs – the provided utilities may simply export information from an artifact to XML or text format straight to standard output. `libscca` is an example of this and is used to access Windows prefetch files.

Prefetch files can be a useful forensic artifact for any number of reasons. They can provide additional execution times for timelines, they can be used to prove program execution even when an executable has been deleted, and they can be used to correlate other artifacts created during execution. More information can be found on the Internet³⁵.

Let's have a look at a quick example, after installing `libscca`:

```
barry@forensicbox:~$ su -  
...  
root@forensicbox:~# sbinstall libscca  
libscca (libYAL Windows Prefetch File parser)  
  
libscca is a library to access the Windows Prefetch File (SCCA) format.  
  
Proceed with libscca? [y]  
Cleaning for libscca-20181227...  
  
root@forensicbox:~# exit
```

With `libscca` installed, let's look for a prefetch file to view. We'll search the NTFS image for files ending in `.pf`. You can see there are quite a few of them (output is truncated).

³⁵<http://www.forensicswiki.org/wiki/Prefetch> is a good start.

```
barry@forensicbox:~$ fls -Fr -o 2048 NTFS_Pract_2017/NTFS_Pract_2017.E01 | grep
.pf$
r/r 72-128-2:  Windows/Prefetch/58.0.3029.81_CHROME_INSTALLER-F06A66AC.pf
r/r 73-128-2:  Windows/Prefetch/AUDIODG.EXE-BDFD3029.pf
...
r/r 123-128-2: Windows/Prefetch/NMAP.EXE-69B77167.pf
...
r/r 160-128-2: Windows/Prefetch/WORDPAD.EXE-D7FD7414.pf
r/r 161-128-2: Windows/Prefetch/WUDFH0ST.EXE-AFFE87C.pf
```

Our familiar **fls** command is executed, looking for files only, recursively (**-Fr**) in the file system at offset 2048 (**-o 2048**) in our NTFS EWF files. Using **grep**, we are looking for **.pf** at the end of the line (signified by the **\\$**). The list is long, but we will look at the **NMAP.EXE** prefetch file (MFT entry 123-128-2). We can extract the file from the image with **icat**:

```
barry@forensicbox:~$ icat -o 2048 NTFS_Pract_2017/NTFS_Pract_2017.E01 123 >
nmap.pf.123
```

Let's very quickly have a look at the header of the file with **xxd**. You can immediately see why the library we just installed is called **libscca**. The prefetch header is 84 bytes long with the version at offset **0x00** and the SCCA header at offset **0x041**.

```
barry@forensicbox:~$ xxd -l 84 nmap.pf.123
00000000: 1700 0000 5343 4341 1100 0000 aaaa 0000  ....SCCA.....
00000010: 4e00 4d00 4100 5000 2e00 4500 5800 4500  N.M.A.P...E.X.E.
00000020: 0000 0200 0000 0000 d935 a382 c07b 719d  ....5...{q.
00000030: bb36 a382 0100 0000 483d 4087 1100 0000  .6.....H=@.....
00000040: 483d 4087 c029 3685 0000 0000 6771 b769  H=@...)6....gq.i
00000050: 0000 0000                                     ....
```

Some of the features we can find (be careful of byte ordering):

```
prefetch version  = 0x0017 (Version 23 - Windows 7)
SCCA header       = 0x5343 0x4341 (SCCA)
executable name   = 0x4e 0x4d 0x41 0x50 0x2e 0x45 0x58 0x45 (NMAP.EXE)
prefetch hash     = 0x69b7 0x7167 (matches the hash in the .pf filename)
```

The latest execution time can be found at offset 128 in the prefetch file (8 bytes long), and we can use **WinTime.py** again to decipher it:

```
barry@forensicbox:~$ python WinTime.py $(xxd -s 128 -l8 -ps nmap.pf.123)
Thu Apr  6 15:07:20 2017
```

Given enough information about the format, you could spend a lot of time parsing the file. There's other information stored within, including libraries and other files accessed when the executable is started. But from here we'll use **sccainfo** from **libsccto** to view the prefetch file contents, which is quite extensive.

```
barry@forensicbox:~$ sccainfo nmap.pf.123
sccainfo 20181227
```

Windows Prefetch File (PF) information:

```
Format version      : 23
Prefetch hash       : 0x69b77167
Executable filename : NMAP.EXE
Run count           : 9
Last run time       : Apr 06, 2017 15:07:20.470652700 UTC
```

Filenames:

```
Number of filenames : 53
Filename: 1          : \DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\NTDLL.DLL
Filename: 2          : \DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\KERNEL32.DLL
```

...

Volumes:

```
Number of volumes   : 1
```

Volume: 1 information:

```
Device path         : \DEVICE\HARDDISKVOLUME2
Creation time       : Apr 06, 2017 04:48:55.209910400 UTC
Serial number       : 0x5019050c
```

There are numerous utilities available to Linux users that can be found to assist in parsing files, artifacts and other data recovered from computers running operating systems other than Linux. There are, in fact, too many to list here. Sometimes it's simply a matter of finding a comparable open source project: like using **LibreOffice** to view Microsoft Office or Visio files. There are also the simple utilities that are either pre-installed or easily installed on your Linux distribution, like **catdoc** or tools like **pdfinfo** and **exiftool** for reading file metadata. There are too many to list here, but suffice to say that over the past few years application layer analysis has become much easier on Linux.

10 Integrating Linux with Your Work

This guide has covered a myriad of subjects that really just touch the surface of the command line capabilities of Linux as a forensic platform. And while it is a lengthy guide, it still only imparts a basic set of commands and utilities to allow you to learn and grow as a forensic examiner or digital investigator. The real power of Linux (as an heir of UNIX itself) is in *thinking in UNIX*. The more you use commands and get used to the output they present, the more you will learn to string them together, solving increasingly complex issues quickly and efficiently. Getting a solid grasp of commands, command history, pipes and redirection is a liberating process.

Some of the repeated exercises we've done here were designed to kick start the repetition needed to anchor the command line process into memory. It is not, however, realistic to expect that everyone will suddenly convert to Linux and forego other operating systems for forensic analysis. Linux is, at the end of the day, just another platform and tool set. I maintain that it is a useful tool set, and that there is some value in every examiner at least being familiar with it and the tools it provides. We've talked about how the command line tools we've encountered here (and there are many more) are unique when compared to common Windows tools. Much of this difference arises from the fact that they are generally designed with the UNIX approach of "do one thing and do it well". We see this in tools like **grep**, **head**, **tail**, **tr**, **sed** and utilities like **icat**, **blkls** and **catdoc**. They provide discrete output when run on their own, but as you start piping them together, we end up accomplishing multiple steps in the same command line. This becomes very powerful not only when you learn what each tool is capable of, but when you also start to think in terms of a modular command line. Learning and remembering all this, however, means more of a burden on examiner resources. And so, one of the strengths of Linux is also, in fact, one of its weaknesses when it comes to mass appeal in the forensic community.

So how do we continue to use Linux, maintain what we've learned and continue learning, while still remaining efficient? Here we will talk about how you can integrate a Linux platform into your current lab or examination processes and continue to use it, if not on a daily basis, at least enough to maintain (and continue growing) the skills we've introduced here.

We've seen tools in this guide that can be used to access file data, file system data, volume information, and block information, etc. It can be done quickly and without the need for licenses or excessive resources to load and view targeted information. Being able to accomplish this on full disk image or blocks of separated data (like the unallocated output of **blkls**, for example), and even individual files, makes Linux an excellent platform for both tool validation and the cross-verification of findings.

Validation, in this context, can be seen as comparing the output of different tools to test specific software functions (or in some cases, hardware functions) and hopefully determine that the expected output is actually produced. If your lab or organization has validation standards for forensic software and hardware, you can strengthen these by not only confirm-

ing similar functions in multiple tools, but by comparing the output of the tool being tested (a function of commercial software on Windows, for example) to an open source tool on an alternative (and also open source) operating system. Conclusive validation of functional output is far easier to ascribe to test results when those results are from entirely different systems. Linux can provide that environment where separate tools are running on an entirely different operating system kernel and environment completely, removing any potential appearance of interference.

We can also use Linux for cross-verification. In those cases where you find specific evidence with one of your standard commercial forensic tools, you can verify those results by comparing them on an alternative operating system with alternative tools. This is not the same as validation. In this case we are not testing a function, we are confirming a finding. For example, you might find a file or set of files pertinent to an investigation. The files were found in a particular volume, in a particular block (or cluster) that was associated with a particular meta-data entry (e.g. MFT). Running **mmls**, **blkstat** and **ifind**, etc. can help us verify those findings taken from a commercial tool. In cases where the data recovered may be contested or your recovery process inspected, having this cross-verification can render arguments against your procedures or tools more difficult.

As a very simple example, let's look at the SAM registry file that we worked on in the section covering application layer analysis. If we commonly use Windows as our standard forensic platform, we might extract registry files with Access Data's FTK and use tools like RegRipper to parse them. If that was the case with the SAM file we examined previously, then I might have found the following information: Parsing the file for user data, we may

Figure 24: SAM file examined with a common Windows tool

Name	SAM
File Class	Regular File
File Size	262,144
Physical Size	262,144
Start Cluster	95,487
Date Accessed	5/1/2017 13:00:42
Date Created	5/1/2017 13:00:42
Date Modified	5/1/2017 16:39:35
Encrypted	False
Compressed	False
Actual File	True
Start Sector	765,944

find that the last login for the user **AlbertE** is critical to our case and the data found might come up in testimony. Output of our primary registry analysis shows the following (output from an examination using Windows tools):

```

Username       : AlbertE [1000]
Full Name      :
User Comment    :
Account Type    : Default Admin User
Account Created : Thu Apr  6 01:35:32 2017 Z
Name           :
Password Hint   : InitialsInCapsCountToFour
Last Login Date : Sun Apr 30 21:23:09 2017 Z
Pwd Reset Date  : Thu Apr  6 01:35:34 2017 Z
Pwd Fail Date   : Sun Apr 30 21:23:01 2017 Z
Login Count     : 7

```

Because of the importance of this particular evidence to the case, we decide to cross verify the output using completely unrelated tools under Linux (we covered the steps previously). Looking at the MFT entry with TSK's **istat**, we can verify information found in our Windows software. The following **istat** command confirms the file dates and time, the size, and the location of the file.

```

barry@forensicbox:~$ istat -o 2048 NTFS_Pract_2017/NTFS_Pract_2017.E01 178

```

```

MFT Entry Header Values:

```

```

Entry: 178      Sequence: 1

```

```

...

```

```

Created:      2017-05-01 09:00:42.659179200 (EDT)
File Modified: 2017-05-01 12:39:35.889046400 (EDT)
MFT Modified:  2017-05-01 09:00:42.676485200 (EDT)
Accessed:     2017-05-01 09:00:42.676286700 (EDT)

```

```

$FILE_NAME Attribute Values:

```

```

Flags: Archive

```

```

Name: SAM

```

```

Parent MFT Entry: 69      Sequence: 1

```

```

Allocated Size: 262144      Actual Size: 262144

```

```

Created:      2017-05-01 09:00:42.659179200 (EDT)

```

```

File Modified: 2017-05-01 12:39:35.889046400 (EDT)

```

```

MFT Modified:  2017-05-01 09:00:42.676286700 (EDT)

```

```

Accessed:     2017-05-01 09:00:42.676286700 (EDT)

```

```

Attributes:

```

```

Type: $STANDARD_INFORMATION (16-0)  Name: N/A  Resident  size: 48

```

```

Type: $FILE_NAME (48-4)  Name: N/A  Resident  size: 72

```

```

Type: $SECURITY_DESCRIPTOR (80-1)  Name: N/A  Resident  size: 80

```

```

Type: $DATA (128-2)  Name: N/A  Non-Resident  size: 262144  init_size: 262144

```

```

95487 95488 95489 95490 95491 95492 95493 95494

```

```

95495 95496 95497

```

```

...

```

Note that the times are different. Obviously this is because of the application of time zones. Differences in the output are not disqualifying as cross-verification if you are able to explain why the difference occurs. This is the foundation of knowing how your software works.

Looking at the output of the Windows software registry parsing, we can verify with commands we used previously (see section 9.6.2 (SAM parsing) on page 299 for a refresher):

```
barry@forensicbox:(values)$ xxd F
00000000: 0200 0100 0000 0000 678e 5df7 f7c1 d201 .....g.].....
00000010: 0000 0000 0000 0000 20d7 bf15 76ae d201 .....v...
00000020: ffff ffff ffff ff7f 5ce9 5df2 f7c1 d201 .....\.].....
00000030: e803 0000 0102 0000 1402 0000 0000 0000 .....
00000040: 0000 0700 0100 0000 0000 4876 488a 3600 .....HvH.6.
```

Last Login Date: offset 8

```
barry@forensicbox:(values)$ python ~/WinTime.py $(xxd -ps -s8 -l8 F)
Sun Apr 30 21:23:09 2017
```

So with a simple few steps we've confirmed critical output, using different tools on a different platform, which can hopefully strengthen any testimony we may be required to give on the findings.

Cross verification can also be used to confirm the very first and most important step of any forensic process: the acquisition and proper handling of collected evidence. We can verify other tools' media hashes, collection hashes, or media identification.

If you can find a way to add Linux to your workflow, you could keep your skills current, learn additional skills, and perhaps even learn to automate some of this workflow through scripting. There are several ways you can deploy Linux in your work, including virtual machines, standalone workstations, and bootable distributions.

Virtual machines (VM) are growing in popularity, and have been for years. There are free options (like VirtualBox Qemu) that are quite robust and offer excellent compatibility and configuration options for a forensic examiner. You can run a VM on your main forensic workstation and provide it access to evidence folders and files, allowing direct interface between the tool and the target image. VMs also have a "snapshot" feature so that when work is complete, a snapshot of a clean and periodically updated operating system can be restored. Also note that VMs can be run the other way – I normally run Windows in a VM on a physical Slackware Linux workstation. The reason I do this highlights one of the drawbacks of VM usage – direct access to hardware. A VirtualBox VM, for example, will allow connections via a virtual USB controller. There are, however, times where I would want to query directly connected devices without the need of a virtual bridge. I prefer to

use Linux for that, so Windows is relegated to a VM and Slackware is given direct access to hardware. That, however, is a matter of personal preference.

The other obvious way to run Linux is to have an actual dedicated workstation. This is fine if you have one you can devote to the purpose, and it alleviates the aforementioned hardware access and interrogation issues. A full work station is particularly useful where you might want to validate or cross verify hardware identification or enumeration. Having a physical workstation requires more monetary resources and can require more configuration effort for exotic or less common hardware, but it also provides the most complete forensic access for the operating system to interact with attached hardware.

The final way you can continue using Linux is through a bootable distribution. These are always handy to keep around for times where you may need to boot a subject computer to acquire evidence or even conduct a limited examination without imaging internal media. We used this approach in our "**dd** over the wire" exercise. There are a number of good bootable distributions available suitable for forensic use. Download a couple, try them out, and see what works best for you. It may be a good idea to have several different versions for different scenarios or hardware configurations. Two bootable Linux variants that come to mind immediately are Caine and Kali Linux:

Caine: <http://www.caine-live.net/>

Kali: <https://www.kali.org/>

11 Conclusion

The examples and practical exercises presented to you here are relatively simple. There are quicker and more powerful ways of accomplishing some of what we have done in this document. The steps taken in these pages allow you to use common Linux tools and utilities that are helpful to the beginner. We've also incorporated more advanced tools and exercises to add some "real world" applicability.

Once you become comfortable with Linux, you can extend the commands to encompass many more options. Practice will allow you to get more and more comfortable with piping commands together to accomplish tasks you never thought possible with a default OS load (and on the command line to boot!). The only way to become proficient on the command line is to use it. And once you get there, you may have a hard time going back.

I hope that your time spent working with this guide was a useful investment. At the very least, I'm hoping it gave you something to do, rather than stare at Linux for the first time and wonder "what now?"

12 Linux Support

Aside from the copious web site references throughout this document, there are a number of very basic sites you can visit for more information on everything from running Linux to using specific forensic tools. Here is a sample of some of the more informative sites you will find:

12.1 Places to go for Support

Slackware. Just one of many Linux distributions.

<http://www.slackware.com>

Learn Slackware (Slackware Linux Essentials):

<https://slackbook.org/beta/>

The "unofficial" official source for online assistance is the Slackware forum at linuxquestions.org:

<http://www.linuxquestions.org/questions/slackware-14/>

Sleuth Kit Wiki:

<http://wiki.sleuthkit.org>

Sleuth Kit Forum:

<https://sleuthkit.discourse.group/>

The Linux Documentation Project (LDP):

<http://www.tldp.org>

In addition to the above list, there are a huge number of user forums, some of which are specific to Linux and computer forensics:

<http://www.forensicfocus.com>

IRC (Internet Relay Chat):

Try `##slackware` on the Freenode network (or other suitable channel for your Linux distribution of choice).

A Google search will be your very best friend in most instances.

List of Figures

1	XFCE with USB volume Win10Image inserted	41
2	Right-click context menu for disk mounting [XFCE]	42
3	XFCE Removable Media Dialog	43
4	Hardware details on a drive label	98
5	Example network acquisition diagram	137
6	Viewing an image with a mis-matched extension	189
7	An example of layers and their associated content based on Carrier's work .	215
8	The image produced by display when used directly from icat	231
9	The sector offset in able2.dd where the search hit was found.	238
10	The volume offset in able2.dd where the search hit was found.	239
11	The offset to the keyword in the volume	239
12	Identifying the data block of the keyword	240
13	Single calculation to identify the data block of the keyword	241
14	A simple example of how blkcalc is used to determine the original address of an unallocated data unit	250
15	Screenshot of LibreOffice viewing a recovered MS Office Document	264
16	Viewing carved files with geeqie	281
17	qphotorec - GUI front end for photorec	283
18	photorec main menu	285
19	photorec running on the able3.home.blkls file	285
20	Viewing photorec options in the running program	286
21	photorec file options menu	287
22	photorec file system selection	287
23	photorec file system selection	288

24	SAM file examined with a common Windows tool	307
----	--	-----

List of Command Examples

1	su session	20
2	Determine the Kernel Version	22
3	List of PCI devices with lspci	23
4	Determine the driver (module) in use	24
5	List USB devices with lsusb	25
6	List USB devices with usb-devices	25
7	Devices listed with lsblk	28
8	Output of ls SCSI	28
9	Viewing Partition Tables with fdisk	29
10	Viewing Partition Tables with gdisk	29
11	Finding USB device assignment with dmesg	31
12	Finding USB device assignment in real-time	31
13	dmesg output with NVMe	32
14	Creating a mount point as root	36
15	Collecting information to mount a volume	37
16	Mounting a partition	37
17	Change Directory to the mountpoint	37
18	umount command	38
19	Identifying the File System on an Optical Disk	38
20	Mounting Optical Media	38
21	Unmount/mnt/cdrom	39
22	Output of mount command	39
23	The /etc/fstab file	39

24	fstab mounting	40
25	Determining device volume by label	41
26	Using mount to check a desktop mounted volume:	42
27	Using the udisks2 udisksctl command	43
28	Unmounting with udisksctl	44
29	Simple ls command	45
30	Using the man command	47
31	Using the find command	47
32	pwd example	48
33	Using the file command	48
34	Using the ps command	48
35	Viewing file permissions with ls -l	49
36	Changing file permissions with chmod	50
37	Redirecting output	51
38	Redirecting output (append)	51
39	Comparison of pipes and redirection	52
40	Output of ps	52
41	Output of ps piped through grep	53
42	Using the tee command	53
43	Listing file attributes with lsattr	54
44	Changing file attributes with chattr	54
45	Try to remove an immutable file	54
46	Adding the append only attribute	55
47	Using bc	55
48	Setting the scale in bc	56

49	Non interactive bc session with a pipe	56
50	Convert Hex with bc	57
51	Bash shell arithmetic	57
52	Example lilo.conf	63
53	Viewing the boot messages with dmesg	63
54	Viewing the default runlevel	65
55	Calling the network daemons script from rc.M	69
56	OpenSSH being started from rc.inet2	69
57	Permissions of rc.sshd	70
58	Changing permissions to prevent SSH from starting at boot	70
59	Preventing service start with chmod 644	71
60	Reading rc script comments	71
61	Successful Connection with SSH	72
62	Editing /etc/hosts.deny	72
63	SSH session denied via /etc/hosts.deny	73
64	Edited /etc/hosts.allow to permit SSH on the local network	73
65	Listing empty iptables rules	75
66	Unfiltered ping (no iptables rules)	75
67	Edited /etc/rc.d/rc.firewall	75
68	Putting our firewall rules in place (iptables)	76
69	Viewing the iptables rules after starting the iptables firewall script	77
70	Ping attempt with the iptables rules in place	77
71	Selecting a package mirror	79
72	Editing the blacklist file	80
73	Compiling from source	82

74	Installing from source	83
75	Installing a package with installpkg	83
76	Installing sbotools with a SlackBuild	86
77	Viewing a SlackBuild .info file	86
78	Creating the sbotools Slackware package with a SlackBuild	87
79	Installing the sbotools package	88
80	Fetching the SlackBuilds repository	88
81	Searching for software with sbotools	89
82	Using sbofind to read a README	90
83	Setup commands for clamav	91
84	Viewing README for lshw using sbofind	91
85	Installing lshw	92
86	Listing installed SlackBuild packages using grep	92
87	Checking for software updates with sbocheck	93
88	Mounting target and checking for destination drive freespace with df -h	96
89	Running lsusb to detect a SATA USB bridge	98
90	Using lsscsi to ID an attached USB to SATA drive	99
91	Using hdparm on a subject disk	99
92	Redirecting hdparm output to a file	101
93	Using hdparm to detect an HPA	102
94	hdparm showing the existence of an HPA	102
95	Obtaining a SHA1 disk hash	104
96	Redirecting hash output to a file	104
97	Sample dd command (basic)	106
98	Comparing device hash to image hash	106

99	Splitting an image with split	107
100	Listing the split image files	107
101	Re-assembling split image files with cat and redirection	108
102	Obtaining a hash of split images with cat	108
103	Splitting images 'on the fly' with dd and split	108
104	Session example of imaging and splitting a USB thumb drive	109
105	Basic dc3dd command on an 80GB disk	112
106	The dc3dd FMT option	113
107	Split and hashed images with a logfile using dc3dd	115
108	Listing dc3dd split images	116
109	Collecting concurrent images with dc3dd	117
110	Installing libewf	118
111	Example run of ewfacquire	119
112	ewfacquire with command options	121
113	Using ewfinfo to read EWF metadata	122
114	ewfinfo on an image collected under Windows	123
115	Verifying the hash of an EWF image with ewfverify	124
116	Downloading the sample EWF files with wget	124
117	Viewing the contents of the downloaded tar archive	125
118	Extracting the E01 sample tar file	125
119	Viewing the sample E01 file metadata with ewfinfo	125
120	Verifying the sample E01 file with ewfverify	126
121	Viewing the size of the ewfexport raw image	127
122	dd with conv=noerror,sync	128
123	Running dd on a disk with errors	129

124	Interrupting ddrescue on a good disk	131
125	Viewing the interrupted map file	132
126	Viewing the size of partial ddrescue recovery	133
127	Completing the ddrescue image and viewing the map file (good disk)	133
128	First stage - Collecting good data from a bad disk with ddrescue	135
129	Second stage imaging with ddrescue (bad disk)	135
130	Using ifconfig on a collection workstation	138
131	Setting the interface with ifconfig	138
132	Checking connectivity with ping (Subject Computer to workstation)	139
133	Checking the hash of the subject computer (bootdisk)	139
134	Starting the listening nc process on the evidence collection workstation . . .	140
135	Starting the imaging process on the subject computer	140
136	Output from our over the wire dd command on the subejct computer	140
137	Check the hash of the dd network image	141
138	Using dc3dd over the network	141
139	Output of dc3dd on an network acquisition	142
140	Checking the hash of the network acquired dc3dd images	142
141	Checking the MD5 on the subect computer to compare with ewfverify . . .	144
142	Checking the hash of our image prior to compression	146
143	Simply compressing a file with gzip	147
144	Simply decompressing a file with gzip -d	147
145	Compressing and decompressing to a new file	147
146	Compressing and checking a hash without decompressing	148
147	Imaging and hashing with direct compression	149
148	Using compression on the fly over the network with gzip and dc3dd	150

149	Compressing over the network with gzip from the subject computer	150
150	Viewing the resulting log file from a dc3dd network acquisition with compression	151
151	Wiping a disk with dd	152
152	Wiping a disk with dc3dd	152
153	Confirming a zero'd drive with xxd	153
154	Using dc3dd to wipe with the hwipe option	153
155	Viewing partition information on a physical disk with fdisk	155
156	Viewing partition information on a physical disk with gdisk	155
157	Redirecting gdisk output to a file	156
158	Downloading our FAT image with wget	157
159	Running the file command on our FAT image	157
160	Using the file command on a block device	157
161	Mounting our FAT image using the loop device	158
162	Checking mount options with the mount command	158
163	Using umount to unmount the FAT file system	159
164	Viewing loop devices in the /dev directory	159
165	Using losetup to associate a file with a device	159
166	Mounting the newly associated loop device	160
167	Removing a loop association with losetup -d	160
168	Using offsets to loop mount a partition in an image	162
169	Using math expansion in the mount command to find the sector offset	162
170	Using mount and losetup to access a partition within an NTFS image	163
171	Installing multipath-tools with a manually built SlackBuild package	164
172	Downloading the kpartx practice image	165
173	Using kpartx to access partitions in an image	166

174	Listing the <code>/dev/mapper</code> nodes produced by kpartx	167
175	Using the file command on kpartx mapped partitions from an image	167
176	Mounting partitions mapped with kpartx	167
177	Removing the kpartx partition mappings	168
178	Installing afflib with sbotools	168
179	Downloading the split practice image (wget)	168
180	Viewing the contents of the split image archive	169
181	Extracting the contents of the split image file archive	169
182	Viewing the split image log for our split sample	169
183	Checking the hashes to compare with the log file (using cat and shasum) . .	170
184	Re-assembling a split image with cat	170
185	Using affuse to create a fuse mounted image of the split image	171
186	Using kpartx on our fuse mounted split images	171
187	Mounting a fuse mounted split image partition for analysis	172
188	Unmounting the partition, removing the loop mappings, and unmounting the fused split files	172
189	Verifying our hash against previous ewfverify output	174
190	Updating ClamAV with freshclam	177
191	Using clamscan on our NTFS image	177
192	Downloading the FAT file system sample with wget	179
193	Creating our case output directory	180
194	Creating an evidence mount point for our FAT image (as root)	180
195	Mounting our FAT image with the loop option	180
196	Basic ls to view a file listing	181
197	An ls command with the recursive option	182

198	Obtaining file hashes	183
199	Using find to calculate hashes of every regular file on a volume	183
200	Checking hashes with the -c option to shasum	184
201	Including modified time in a file listing	184
202	Simple file listing with paths using the file command	185
203	Simple directory listing with paths using the file command	185
204	Using the tree command to display a file listing	185
205	Using grep to search for strings in a file listing	185
206	Using find to run file on search results	186
207	Viewing the results of the find filetype command	186
208	Using grep to search for images in our filetype output	187
209	Using the strings command on an executable	188
210	Determining the file type, and then viewing a JPEG file using file and xv .	189
211	Using the grep command with a keyword list	191
212	Viewing our grep search hits with cat	192
213	Using xxd to view search hits	192
214	Using tr to translate control characters to newlines	193
215	Using wget to download the practice log files	195
216	Hashing and listing the contents of our practice logs archive	196
217	Extracting our practices logs from the archive	196
218	Viewing the logs with cat	197
219	Using tac to 'reverse' the log files	197
220	Count the number of total lines in the log files with wc	198
221	Count the number of lines in each log file along with the total using wc . . .	198
222	Using awk to print only the month and day (fields 1 and 2)	198

223	Finding unique dates in the logfile with uniq	199
224	Using grep to find a particular date	199
225	Using grep for strings at the beginning of a line	199
226	Using grep to include unknown number of spaces in the search	200
227	Using grep to find a string	200
228	Counting our grep hits with wc	200
229	Using grep and awk to collect more fields	200
230	Using tabs instead of spaces in our awk output	201
231	Creating a report file on the log analysis	201
232	Getting a sorted list of IP addresses from the logs	202
233	Downloading the carving exercise image	203
234	Viewing the carving exercise image with xxd and less	203
235	Looking for the JPEG header with xxd and grep	204
236	Calculating the decimal offset (header) using bc	204
237	Looking for the JPEG footer with xxd and grep	205
238	Calculating the decimal offset (footer) using bc	205
239	Calculating the size of the image using offsets to header and footer	205
240	Using dd to carve the JPEG with our offset and file size for skip and count .	206
241	Using xv to view the carved image	206
242	Looking at image partitions with fdisk	209
243	Using the partition information to carve partitions with dd	209
244	Checking the file system types of our partition images with file	211
245	Searching for the /etc directory in our partition images	211
246	Viewing the fstab file in our mounted partition image	212
247	Reconstructing our able_3 file system	212

248	Installing TSK with sboinstall	219
249	Viewing the installed package contents at /var/log/packages	220
250	Obtaining and checking the able2 practice image	221
251	Extracting the archive containing the able2 image	221
252	Using mmls on the able2image	221
253	Using fsstat on the able2 image	222
254	Running fls on the root directory of a file system	223
255	Running fls on the root directory of the able2 image	224
256	Using fls on a specific inode	225
257	Using fls to see a recursive listing of unallocated files	225
258	Using ffind to find files associated with an inode	227
259	Using istat to gather information about a specific inode	228
260	Listing supported TSK file systems	228
261	Redirecting the output of icat to a recovered file	229
262	Determining the type of recovered file with the file command	229
263	Listing the contents of the recovered tar archive	230
264	Listing the contents of the directory at inode 11105 using fls	230
265	Using display to show a recovered image directly from icat	231
266	img_stat and mmls on the able3 image	232
267	Running fsstat on /home (ext4)	233
268	Running fls on /home (ext4)	233
269	Running fls recursively on /home (ext4)	234
270	Running istat and icat on an ALLOCATED file in ext4	234
271	Running istat and icat on an UNALLOCATED file in ext4	235
272	Searching for a keyword in able2.dd with grep	236

273	Viewing the context of a search hit with xxd	237
274	Calculating the sector offset to a search hit with bc	237
275	Using mmls to determine which volume a sector belongs to	237
276	Calculating the byte offset to the volume containing the search hit	238
277	The volume offset of the search hit	238
278	Determine the number of bytes per block with fsstat	239
279	Calculating the block address that holds the search hit	240
280	Single calculation to determine the data block of the keyword	240
281	Determine the allocation status of the data block with blkstat	241
282	Using ifind to determine a data block's inode	241
283	Using istat to explore the inode	241
284	Using icat to recover data from the blocks in inode 10090	242
285	Saving and hashing the data recovered with icat	242
286	Using ffind to obtain the a file name to associate with our data	243
287	Using grep on split images	243
288	Calculating the file system data block for the ext4 search hit	244
289	Using blkstat to determine the status of the keyword data block	245
290	Using ifind to determine an unallocated block's inode	245
291	Using blkcat to stream the contents of a data block	245
292	Using dd to correct our recovered data's size	246
293	Obtaining the offset to our file system with mmls	247
294	Using blkls to extract the unallocated data	248
295	Using grep on our file containing unallocated blocks only	248
296	Finding our block size with fsstat	248
297	Determining the block address of the search hit in the unallocated blkls image	249

298	Calculating the block address of the unallocated unit in able2.dd	249
299	Finding and examining the inode for the unallocated search hit	250
300	Using icat to recover the unallocated data at inode 10090	251
301	Obtaining and extracting the NTFS practice image	252
302	Viewing the partition table in the NTFS image	252
303	fsstat on an NTFS file system	253
304	fls on NTFS	254
305	fls on a specific NTFS directory	254
306	Browsing directories with recursive fls	254
307	Accessing an individual NTFS attribute with icat	257
308	Using ewfmount to mount our NTFS EWF image	257
309	Using find on the loop mounted NTFS image	258
310	Using find and grep to narrow the list of files	258
311	Checking the file format and playing the MPEG video	259
312	Using fls to entries with the name jet.mpg	259
313	Using the file command on the default data stream	260
314	Using the file command on the alternate data stream	260
315	Viewing the alternate data stream	260
316	Fuse mounting the NTFS image for searching	261
317	A string search on the fuse mounted image	261
318	Using tr to make the search results readable	262
319	Finding the NTFS cluster size (block size)	262
320	Calculating the keyword offset in NTFS	263
321	Finding the MFT entry given the data block	263
322	Determining the data type for the located MFT entry	263

323	Trying to read the MS Office data with less	264
324	Redirecting the MS Office data to a file	264
325	Using icat and catdoc to view the recovered MS Office file	265
326	String search for Uranium-235	265
327	Unmounting the fuse mounted EWF image	266
328	Installing bulk_extractor	266
329	Viewing bulk_extractor 's help	267
330	Running bulk_extractor , searching for "Uranium-235"	268
331	Viewing bulk_extractor output	269
332	Viewing the find.txt feature file	269
333	Determining the MFT entry and file type of the data in our bulk_extractor search	270
334	Viewing the histogram created by the find scanner	271
335	Running bulk_extractor with a banner file and keyword file	272
336	Viewing the MSXML file with catdocx	274
337	Installing scalpel	275
338	Placing the scalpel configuration file	275
339	Viewing and editing the scalpel.conf file	276
340	Reviewing the location of the lolitaz JPG images	277
341	Extracting unallocated space for carving	278
342	Running scalpel on the able_3 images	279
343	Viewing the scalpel audit.txt file	280
344	Adding a new bitmap definition to scalpel.conf	281
345	Re-running scalpel with a new bmp definition	282
346	Installing testdisk for photorec	284

347	Running photorec	284
348	Listing the files recovered by photorec	287
349	Listing the photorec output files	289
350	Viewing the contents of the recovered tar archive from photorec	289
351	Hashing all the output files from our carving tools	290
352	Installing the fdupes utility	291
353	Running fdupes to view the duplicated files	291
354	Re-running fdupes , this time with the --delete option	292
355	Finding the MFT entry number of a registry file	294
356	Using an alternative (fls) method to find the MFT entry of NTUSER.DAT . . .	294
357	Installing libregf	295
358	Viewing the libregf package contents	295
359	Mounting the registry hive	295
360	Listing the registry keys in the NTUSER.DAT mount point	296
361	Changing directory into the target registry key	296
362	Viewing the registry key values	296
363	Using tr on a single string (.yax)	297
364	Using a bash for loop to run tr on all the 'files'	297
365	Viewing the values with xxd	298
366	De-coding the date with WinTime.py	299
367	Unmounting the fuse mounted NTUSER.DAT	299
368	Finding the MFT entry of the SAM registry hive	300
369	Extracting the SAM registry hive with icat	300
370	Using regfmount to mount the SAM hive	300
371	Viewing the accounts in the SAM file	300

372	Decoding the user accounts in the Users key with a for loop	300
373	Viewing the values for 000003E8 in the SAM file	301
374	Viewing the dates associated with the User account	301
375	Using xxd to view specific values from the registry key	302
376	Passing xxd arguments directly to WinTime.py	302
377	Using a for loop to pass the offset values to xxd	302
378	Installing libscca	303
379	Looking for prefetch files in the NTFS image	304
380	Extracting the NMAP prefetch file with icat	304
381	Viewing the prefetch file with xxd	304
382	Decoding the last execution time in the prefetch file	304
383	Viewing the prefetch file with sccainfo	305
384	Cross verification using istat	308
385	Cross verifying the date using the mounted SAM file and xxd	309
386	Decoding the date recovered from the key (F) with xxd	309