CHAPTER

Disk and File System Analysis

3

INFORMATION IN THIS CHAPTER

- Media Analysis Concepts
- The Sleuth Kit
- Partitioning and Disk Layouts
- Special Containers
- Hashing
- Carving
- Forensic Imaging

MEDIA ANALYSIS CONCEPTS

At its most basic, forensic analysis deals with *files* on *media*—deleted files, files in folders, files in other files, all stored on or in some container. The goal of media analysis is to identify, extract, and analyze these files and the file systems they lie upon. *Identification* includes determining which active and deleted files are available in a volume. *Extraction* is the retrieval of relevant file data and metadata. *Analysis* is the process in which we apply our intelligence to the data set and ideally come up with meaningful results.

Note that these are not necessarily discrete procedural steps. In fact, some examination processes will seem to straddle two or more of these—carving, for example, can easily be described as both identification and extraction. Nonetheless, we feel that this is a suitable model for describing *why* we as examiners are taking a particular action.

This chapter focuses primarily on the concepts behind identifying and extracting file system artifacts, and information *about* files. Deep analysis of the artifacts found in content of the files and the artifacts of interest found in specific file systems will not be covered here as this analysis makes up the bulk of Chapters 4 through 8.

While we discuss the file system analysis concepts that will be of the most use to an examiner, a full analysis of every conceivable artifact and nuance of each file system is outside the scope of this book. For greater detail on this topic, the authors highly recommend *File System Forensic Analysis* by Brian Carrier [1], the authoritative work on the subject.

File System Abstraction Model

In the aforementioned *File System Forensic Analysis*, the author puts forth a file system abstraction model to be used when describing the functions of file systems and the artifacts generated by these functions. For readers with networking backgrounds, this model is not unlike the OSI model used to describe communications systems.

As described by Carrier, the logical progression of any file system, from low level to high level, is:

- Disk

  A *disk* refers to a physical storage device—a SCSI or SATA hard drive, or a Secure Digital Card from a digital camera, for example. Analysis of items at this level is usually beyond the capabilities of most examiners—physical media analysis of conventional hard drives requires extensive specialized training and knowledge, access to a clean room, and expensive electron microscopy equipment. With the rise of flash media and Solid State Disks, however, analysis of media at this level may be in the realm of possibility for a larger pool of examiners.
- Volume

  A *volume* is created using all or part of one or more disks. A single disk may contain several volumes, or a volume may span several disks, depending on configuration. The term "partition" is often used interchangeably for a volume; Carrier makes a distinction wherein a "partition" is limited to a single physical disk, and a volume is a collection of one or more partitions. Put simply, a volume describes a number of sectors on a disk(s) in a given system. Please see Figure 3.1 for a simplified display of the delineation between a disk and volumes present on the disk.
- File System

  A *file system* is laid down on a volume and describes the layout of files and their associated metadata. Items in the file system layer include metadata specific to and solely used for the file system's operation—the Ext2 superblock is a good example.

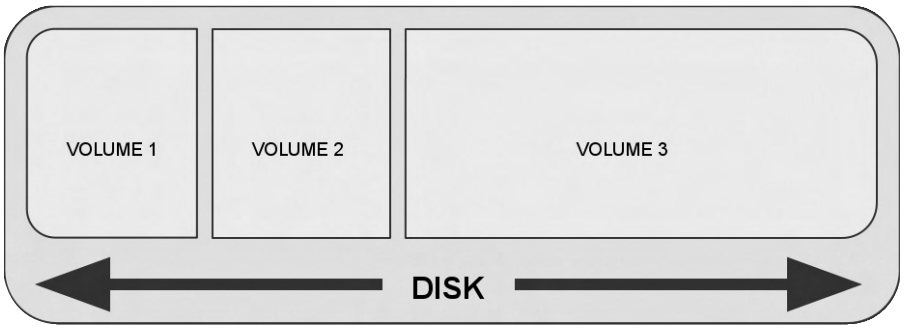

FIGURE 3.1

Disk and volumes.

- Data Unit

  A *data unit* is the smallest available freestanding unit of data storage available in a given file system. On Unix-derived file systems these are known as *blocks*. These are generally some power of 2 multiple of the physical sector size of the disk. Historically the sector size of every disk was 512 bytes—most modern file systems will use 4096 bytes (4K) or larger as the smallest addressable *data unit*. The information available at the data unit layer is simple: the content of that data unit. If that data unit is allocated to a JPEG image, the data unit will contain a portion of JPEG data. If the data unit was allocated to a text file, the data unit will contain text.
- Metadata

  Metadata refers to *data about data*. Given that the *data unit layer* holds data in a file system, the *metadata layer* then contains data about the data units. On Unix-derived file systems these metadata units are called *inodes*. The exact content of metadata units depends on the actual file system being discussed, but generally this layer will at least consist of file time stamps, file ownership information, and data units allocated to this metadata unit. We'll discuss the specific artifacts for each file system in the relevant sections later.
- File Name

  The *file name* layer is where humans operate. Unsurprisingly, this layer consists of file and folder/directory names. Once again, artifacts available in this layer vary depending on the file system. At the very least, file names have a pointer to their corresponding metadata structure.

Because this abstraction model is built with the design of Unix-derived file systems in mind, some of the separations do not map directly to the designs of file systems for other platforms. However, a good understanding of this model is imperative to truly understanding the significance of file system artifacts on *any* file system.

THE SLEUTH KIT

To process file system artifacts, we will use *The Sleuth Kit* (www.sleuthkit.org). The Sleuth Kit (TSK) is the suite of file system forensic tools originally created by Brian Carrier as an updated version of the older *Coroner's Toolkit*. The Coroner's Toolkit (TCT) was designed specifically to perform forensic analysis of compromised Unix-like systems. While being a very powerful set of early forensic tools, TCT had major shortcomings, including a lack of portability between systems and a lack of support for non Unix-like file systems. Carrier developed the Sleuth Kit to provide a highly portable, extensible, and useful open source forensics toolkit.

Installing the Sleuth Kit

The Sleuth Kit natively supports processing raw disk images (split or not), but it can also import the ability to process additional image formats from the LibEWF and

AFFLib packages installed in Chapter 2. Note that we could install precompiled Sleuth Kit packages using the Ubuntu package manager. Retrieving the source code directly and compiling ourselves minimizes the number of intermediaries involved in producing executable code. It also ensures that we have the latest version of our core tools and libraries, as package repositories may take some time to update.

Note that when executing the Sleuth Kit's configure script (`./configure`), you should see the following lines toward the end of the script's output:

```
checking afflib/afflib.h usability... yes
checking afflib/afflib.h presence... yes
checking for afflib/afflib.h... yes
checking for af_open in -lafflib... yes
checking libewf.h usability... yes
checking libewf.h presence... yes
checking for libewf.h... yes
checking for libewf_open in -lewf... yes
configure: creating ./config.status
```

This confirms that LibEWF and AFFLib are installed properly and will be used by the Sleuth Kit.

With these development libraries installed, and the Sleuth Kit configured, finishing the build and install is a simple matter executing `make` followed by `sudo make install`. This will install the suite of command-line tools that make up the Sleuth Kit.

WARNING

Got Root?

If you plan to use Sleuth Kit tools with an attached disk as the target (as opposed to an image file) remember that you will need *root* privileges. This can be accomplished either by becoming root via the "su-" command or by executing the command with root privileges using the "sudo" command, as shown in Chapter 2.

Sleuth Kit Tools

Mastering 21 separate command line utilities may seem daunting if you are not used to operating via command prompt frequently. That said, the bulk of Sleuth Kit tools are named in a logical manner, which indicates the file system layer they operate upon and the type of output you should expect from them. Since the Sleuth Kit comes from a Unix-derived pedigree, this naming is quite clear if you are familiar with the Linux command line.

The common prefixes found in the Sleuth Kit tools that indicate the file system layer of the tool are:

- "mm-": tools that operate on volumes (aka "media management")
- "fs-": tools that operate on file system structures
- "blk-": tools that operate at the data unit (or "block") layer
- "i-": tools that operate at the metadata (or "inode") layer
- "f-": tools that operate at the file name layer

There are two additional layers that don't map directly into the file system model as described:

- "j-": tools that operate against *file system journals*
- "img-": tools that operate against *image files*

Common suffixes found in Sleuth Kit tools that indicate the expected function of the tool are:

- "-stat": displays general information about the queried item—similar to the "stat" command on Unix-like systems
- "-ls": lists the contents of the queried layer, such as the "ls" command on Unix-like systems
- "-cat": dumps/extracts the content of the queried layer, such as the "cat" command on Unix-like systems

Additionally, a handful of tools provided by the Sleuth Kit don't follow this naming scheme. These are described under the "Miscellaneous Tools" section.

To demonstrate use of the Sleuth Kit, we will proceed through each layer, describing each tool present in that layer. Additionally, we will examine the use and output of the most important tools using a Linux Ext3 file system as our demonstration target.

NOTE

Sleuth Kit Disk Layer Tools

Current versions of the Sleuth Kit do not provide any tools for operating at the disk layer. Because the Sleuth Kit is a file system forensic analysis framework, this should not be surprising. That said, versions of the Sleuth Kit prior to 3.1.0 did include two tools at this layer that you may encounter in older forensic live CD distributions.

The `disk_stat` tool will show if the disk has a *Host Protected Area* (HPA) present. A HPA is one method that can be used to artificially restrict the number of sectors addressable by the operating system accessing a hard drive.

The `disk_sreset` will allow you to temporarily remove an HPA from a disk. This is a `nonpermanent` change—the HPA will return the next time the disk is powered on. Temporarily removing the HPA using `disk_sreset` enables a subsequent image capture operation to grab the entire disk, including the protected area.

Another method for restricting the displayed number of sectors is via *Device Configuration Overlay*. Both this and HPA can be detected and removed using the `hdparm` utility, which is included by default on most Linux distributions.

Other non-Sleuth Kit tools that operate at the disk layer include all of the imaging tools discussed in the *Forensic Imaging* section later in the chapter.

Volume Layer Tools

The `mmstat` command will display the type of volume system in use on the target image file or disk.

The `mmls` command parses and displays the media management structures on the image file or disk (i.e., the partition table). Note that unlike the `fdisk` command, `mmls` will clearly show nonallocated spaces before, after, or between volumes.

Here we have an example image from Digital Forensics Tool Testing archive.

```
user@forensics:~$ mmls 10-ntfs-disk.dd
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors
     Slot    Start        End          Length       Description
00:  Meta    0000000000   0000000000   0000000001   Primary Table (#0)
01:  -----   0000000000   0000000062   0000000063   Unallocated
02:  00:00   0000000063   0000096389   0000096327   NTFS (0x07)
03:  00:01   0000096390   0000192779   0000096390   NTFS (0x07)
04:  -----   0000192780   0000192783   0000000004   Unallocated
```

We can see here that the primary partition table was found in the first sector of the disk and that there are two volumes present—the first from sector 63 through sector 96389 and the second from sector 96390 through sector 192779. The `mmls` output also makes it clear that there are four "extra" sectors after the end of the last volume in addition to the standard 63 sector gap before the first volume.

Another important benefit of using mmls instead of a tool such as `fdisk` is that the offsets to individual volumes are presented as counts of 512-byte sectors. These offsets can be passed directly to higher level Sleuth Kit tools to specify a volume to analyze.

The `mmcat` streams the content of the specified volume to STDOUT (usually the console). This can be used to extract a specific volume of interest for analysis using tools that may not be able to operate on the container format or disk directly.

File System Layer Tools

The `fsstat` command displays file system information. Data of particular interest in the output of this command vary depending on the file system being examined but may include volume names, data unit sizes, and statistical information about the state of the file system. We will use output from an Ext3 file system to present the tool. Analysis of Ext3-specific information is covered in detail in Chapter 5.

```
user@forensics:~$ fsstat ubnist1.casper-rw.gen3.aff
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: Ext3
Volume Name:
Volume ID: 9935811771d9768b49417b0b3b881787
Last Written at: Tue Jan 6 10:59:33 2009
Last Checked at: Sun Dec 28 12:37:56 2008
Last Mounted at: Tue Jan 6 10:59:33 2009
Unmounted properly
Last mounted on:
Source OS: Linux
Dynamic Structure
Compat Features: Journal, Ext Attributes, Resize Inode, Dir Index
InCompat Features: Filetype, Needs Recovery,
Read Only Compat Features: Sparse Super, Has Large Files,
Journal ID: 00
Journal Inode: 8
```

As you can see from the partial tool output just given, the fsstat tool provides some basic file system information, including some information that may be of key investigative value, such as the last written and last mounted information. After this general information, the output of `fsstat` will be highly file system dependent. In the case of Ext3, statistical and layout information is provided about metadata and content structures present on the disk:

```
METADATA INFORMATION
------------------------------------
Inode Range: 1 - 38401
Root Directory: 2
Free Inodes: 36976
Orphan Inodes: 35, 20, 17, 16,
CONTENT INFORMATION
------------------------------------
Block Range: 0 - 153599
Block Size: 4096
Free Blocks: 85287
...
```

Note that this tool provides the block size used on the file system. This is important information when carving data from unallocated space.

Data Unit Layer Tools

The `blkstat` command displays information about a specific data unit. Generally, this will simply be allocation status; however, on Ext file systems, the block group to which the block is allocated is also displayed.

```
user@forensics:~$ blkstat ubnist1.casper-rw.gen3.aff 521
Fragment: 521
Allocated
Group: 0
```

The `blkls` command lists details about data units. `Blkls` can also be used to extract all unallocated space of the file system. This is useful to do prior to attempting to carve data from a file system. The following example extracts all of the unallocated space from our sample image file into a single, flat file.

```
user@forensics:~$ blkls ubnist1.casper-rw.gen3.aff > ubnist1.
  casper-rw.gen3.unalloc
user@forensics:~$ ls -lath ubnist1.casper-rw.gen3.unalloc
-rw-r----- 1 cory eng 331M Sep 2 20:36 ubnist1.casper-rw.gen3.
  unalloc
```

The `blkcat` command will stream the content of a given data unit to STDOUT. This is similar in effect to using dd to read and write a specific block. The next example uses `blkcat` to extract block 521, which we view courtesy of the `xxd` binary data viewer, which is included with the `vim` editor package on most distributions.

```
user@forensics:~$ blkcat ubnist1.casper-rw.gen3.aff 521 | xxd |
  head
0000000: 0200 0000 0c00 0102 2e00 0000 0200 0000 ........
0000010: 0c00 0202 2e2e 0000 0b00 0000 1400 0a02 ........
0000020: 6c6f 7374 2b66 6f75 6e64 0000 0c00 0000 lost+found.....
0000030: 1400 0c01 2e77 682e 2e77 682e 6175 6673 .....wh..wh.aufs
0000040: 011e 0000 1400 0c02 2e77 682e 2e77 682e .........wh..wh.
0000050: 706c 6e6b 015a 0000 1400 0c02 2e77 682e plnk.Z......wh.
0000060: 2e77 682e 2e74 6d70 021e 0000 0c00 0402 .wh..tmp.......
0000070: 726f 6673 025a 0000 0c00 0302 6574 6300 rofs.Z.....etc.
0000080: 045a 0000 1000 0502 6364 726f 6d00 0000 .Z...cdrom......
0000090: 031e 0000 0c00 0302 7661 7200 013c 0000 ......var..<..
```

The `blkcalc` command is used in conjunction with the unallocated space extracted using `blkls`. With *blkcalc*, we can map a block from `blkls` output back into the original image. This is useful when we locate a string or other item of interest in the `blkls` extract and want to locate the location of the item in our forensic image.

Metadata Layer Tools

The `istat` command displays information about a specific metadata structure. In general, any of the information listed as being contained in a metadata structure (ownership, time information, block allocations, etc.) will be displayed. As always, the exact information displayed is file system dependent. We will explore file system-specific information in subsequent chapters.

What follows is the `istat` output for inode 20 on our test Ext3 file system. Output common to other file systems includes allocation status, ownership information, size, and time stamp data. Addresses of the inode's data units will also be present but are handled in different manners by different file systems, as shown later.

```
user@forensics:~$ istat ubnist1.casper-rw.gen3.aff 20
inode: 20
Allocated
Group: 0
Generation Id: 96054594
uid / gid: 0 / 0
mode: rrw-r--r--
size: 123600
num of links: 0
Inode Times:
Accessed:         Tue Jan 6 10:59:33 2009
File Modified:    Wed Jan 7 07:59:47 2009
Inode Modified:   Wed Jan 7 07:59:47 2009
Deleted:          Wed Dec 31 16:00:17 1969
Direct Blocks:
28680 0 0 0 0 0 0 28681
0 0 0 0 0 0 0 28683
0 0 0 0 0 0 28684 0
0 0 0 0 0 0 28685
Indirect Blocks:
28682
```

The `ils` command lists the metadata structures, parsing and displaying the embedded dates, ownership information, and other relevant information. This is one of the commands that can be used to generate a *bodyfile* for timeline generation using the *mactime* command (see "Miscellaneous Tools"). Timelines are key to the investigations presented in Chapter 9.

As you can see from the argument list, the examiner can tune the `ils` output to view as much (or as little) data as necessary.

```
user@forensics:~$ ils
Missing image name
usage: ils [-emOpvV] [-aAlLzZ] [-f fstype] [-i imgtype] [-b
  dev_sector_size] [-o imgoffset] [-s seconds] image [images]
  [inum[-end]]
    -e: Display all inodes
    -m: Display output in the mactime format
    -O: Display inodes that are unallocated, but were sill open
        (UFS/ExtX only)
    -p: Display orphan inodes (unallocated with no file name)
    -s seconds: Time skew of original machine (in seconds)
    -a: Allocated inodes
    -A: Unallocated inodes
    -l: Linked inodes
    -L: Unlinked inodes
    -z: Unused inodes (ctime is 0)
    -Z: Used inodes (ctime is not 0)
    -i imgtype: The format of the image file (use '-i list' for
       supported types)
    -b dev_sector_size: The size (in bytes) of the device
       sectors
    -f fstype: File system type (use '-f list' for supported
       types)
    -o imgoffset: The offset of the file system in the image
       (in sectors)
    -v: verbose output to stderr
    -V: Display version number
```

For example, if we wanted to list all inodes that are allocated or that have been used at some point, we can do so with the `-a` and `-Z` flags:

```
user@forensics:~$ ils -aZ ubnist1.casper-rw.gen3.aff
...
st_ino|st_alloc|st_uid|st_gid|st_mtime|st_atime|st_ctime|st_
  crtime|st_mode|st_nlink|st_size
1|a|0|0|1230496676|1230496676|1230496676|0|0|0|0
2|a|0|0|1231268373|1230496676|1231268373|0|755|15|4096
7|a|0|0|1230496676|1230496676|1230496676|0|600|1|4299210752
8|a|0|0|1230496679|0|1230496679|0|600|1|16777216
11|a|0|0|1230496676|1230496676|1230496676|0|700|2|16384
12|a|0|0|1230469846|1230469846|1231311252|0|444|19|0
13|a|0|0|1230615881|1225321841|1230615881|0|755|9|4096
...
```

The `icat` command streams the data unit referenced by the specified meta data address. For example, if "file1.txt" points to inode 20, which then points to blocks 30, 31, and 32, the command "`icat` *{image_ file}* 20" would produce the same output that "`cat` file1.txt" would from the mounted file system.

The `ifind` command finds the metadata structure referenced by the provided file name *or* the metadata structure that references the provided data unit address. For example, to find the inode that owns block 28680, we can do the following:

```
user@forensics:~$ ifind -d 28680 ubnist1.casper-rw.gen3.aff
20
```

File Name Layer Tools

The `fls` command lists file names (deleted and allocated). By default it does not traverse the entire file system so you will only see the root directory of the volume being examined. This is one of the commands we can use to generate a *bodyfile* for timeline generation using the `mactime` command (see "Miscellaneous Tools"). A simple "`fls` *image*" will produce a terse directory listing of the root directory of the file system.

```
user@forensics:~$ fls ubnist1.casper-rw.gen3.aff
d/d 11:      lost+found
r/r 12:    .wh..wh.aufs
d/d 7681:    .wh..wh.plnk
d/d 23041:   .wh..wh..tmp
d/d 7682:    rofs
d/d 23042:   etc
d/d 23044:   cdrom
d/d 7683:    var
d/d 15361:   home
d/d 30721:   tmp
d/d 30722:   lib
d/d 15377:   usr
d/d 7712:    sbin
d/d 13:      root
r/r * 35(realloc):    .aufs.xino
d/d 38401:    $OrphanFiles
```

Note that the ".aufs.xino" file is listed with an asterisk—this indicates that it is deleted. The (realloc) indicates that the inode the name references has been reallocated to another file.

The `fls` man page provides more background into the various options that can be passed to the command. For interactive use, particularly important `fls` arguments include:

```
-d: Display deleted entries only
-l: Display long version (like ls -l)
-m: Display output in mactime input format with
dir/ as the actual mount point of the image
```

```
    -p: Display full path for each file
    -r: Recurse on directory entries
    -u: Display undeleted entries only
    -z: Time zone of original machine (i.e. EST5EDT or GMT)
        (only useful with -l)
    -s  seconds: Time skew of original machine (in seconds)
        (only useful with -l & -m)
```

Note that the time zone argument does not apply if you are using `-m` to create a `mactime` input file. This is only used when displaying time information to the console.

The `ffind` command finds file names that reference the provided metadata number. Using inode 20, which we located via the `ifind` command, we can discover the name associated with this inode.

```
user@forensics:~$ ffind ubnist1.casper-rw.gen3.aff 20
File name not found for inode
```

Unfortunately, no name currently points to this inode—it is *orphaned*. Just to sate our curiosity, we can check the adjacent inodes.

```
user@forensics:~$ ffind ubnist1.casper-rw.gen3.aff 19
/root/.pulse-cookie
user@forensics:~$ ffind ubnist1.casper-rw.gen3.aff 21
/root/.synaptic/lock
```

Miscellaneous Tools

The `mactime` command generates a timeline based on processing the *bodyfile* produced by `ils` and/or `fls`. To generate a timeline using the Sleuth Kit, first we need to generate the *bodyfile*. This is simply a specifically ordered pipe-delimited text file used as the input file for the mactime command.

```
user@forensics:~$ ils -em ubnist1.casper-rw.gen3.aff > ubnist1.
  bodyfile
user@forensics:~$ fls -r -m "/" ubnist1.casper-rw.gen3.aff >>
  ubnist1.bodyfile
```

This produces a text file with the metadata information of each file or inode on a single line.

```
md5|file|st_ino|st_ls|st_uid|st_gid|st_size|st_atime|st_mtime|st_
  ctime|st_crtime
0|<ubnist1.casper-rw.gen3.aff-alive-1>|1|-/----------
  |0|0|0|1230496676|1230496676|1230496676|0
0|<ubnist1.casper-rw.gen3.aff-alive-2>|2|-/drwxr-
  xr-x|0|0|4096|1230496676|1231268373|1231268373|0
0|<ubnist1.casper-rw.gen3.aff-alive-3>
  |3|-/----------|0|0|0|0|0|0|0
0|<ubnist1.casper-rw.gen3.aff-alive-4>
  |4|-/----------|0|0|0|0|0|0|0
```

```
0|<ubnist1.casper-rw.gen3.aff-alive-5>
  |5|-/----------|0|0|0|0|0|0|0
0|<ubnist1.casper-rw.gen3.aff-alive-6>
  |6|-/----------|0|0|0|0|0|0|0
0|<ubnist1.casper-rw.gen3.aff-alive-7>|7|-/rrw-------
  |0|0|4299210752|1230496676|1230496676|1230496676|0
...
0|/lost+found|11|d/drwx------
  |0|0|16384|1230496676|1230496676|1230496676|0
0|/.wh..wh.aufs|12|r/rr--r-
  -r--|0|0|0|1230469846|1230469846|1231311252|0
0|/.wh..wh.plnk|7681|d/drwx------
  |0|0|4096|1230469846|1230469897|1230469897|0
0|/.wh..wh.plnk/1162.7709|7709|r/rrw-r-
  -r--|0|0|186|1225322232|1225322232|1230469866|0
```

When generating a timeline for an actual investigation we will want to set the time zone that data originated in and possibly some additional file system-specific information. However, to generate a simple comma-separated timeline, we can issue the following command:

```
user@forensics:~$ mactime -b ubnist1.bodyfile -d > ubnist1.timeline.csv
```

Timeline analysis is quite useful when performed properly. We will discuss timeline analysis in Chapter 9.

The `sigfind` command is used to search a source file for a binary value at given offsets. Given a sequence of hexadecimal bytes, `sigfind` will search through a stream and output the offsets where matching sequences are found. `Sigfind` can be sector or block aligned, which can be of value when searching through semistructured data such as memory dumps or extracted unallocated space. This is useful for locating files based on header information while minimizing noisy false positives that may occur when simply searching through a data stream using something like the `grep` command.

Using the `sigfind` tool is quite simple.

```
-sigfind [-b bsize] [-o offset] [-t template] [-lV] [hex_
  signature] file
        -b bsize: Give block size (default 512)
        -o offset: Give offset into block where signature
           should exist (default 0)
        -l: Signature will be little endian in image
        -V: Version
        -t template: The name of a data structure template:
           dospart, ext2, ext3, fat, hfs, hfs+, ntfs, ufs1, ufs2
```

As an example, we can use sigfind to locate (at least portions of) PDF files on our test Ext3 image. PDF documents begin with the characters "%PDF." Converting these ASCII characters to their hex equivalent gives us "25 50 44 46." Using `sigfind`, we look for this at the start of every cluster boundary (which was discovered earlier using the `fsstat` tool).

```
user@forensics:~$ sigfind -b 4096 25504446 ubnist1.casper-rw.gen3.aff
Block size: 4096 Offset: 0 Signature: 25504446
Block: 722 (-)
Block: 1488 (+766)
Block: 1541 (+53)
Block: 1870 (+329)
Block: 82913 (+81043)
...
```

The output of the tool provides the offset in blocks into the image where the hit signature matched and in parentheses provides the offset from the previous match. `Sigfind` also has a number of data structure templates included, which makes identifying lost partitions or file system structures simple.

The `hfind` command is used to query hash databases in a much faster manner than grepping through flat text files.

The `sorter` command extracts and sorts files based on their file type as determined by analysis of the file's content. It can also look up hashes of extracted files and perform file extension verification.

Finally, the `srch_strings` command is simply a standalone version of the *strings* command found in the GNU *binutils* package. This tool is included to ensure that the Sleuth Kit has string extraction capability without requiring that the full *binutils* package be installed on systems where it is not normally present.

Image File Tools

We can think of the image file as a new intermediary layer that replaces the disk layer in our file system stack. Because this layer is created by an examiner, we generally don't expect to find any forensically interesting items here. However, depending on the forensic format, relevant information may be available.

The `img_stat` command will display information about the image format, including any hash information and other case-relevant metadata contained in the image. This tool is generally only useful when executed against forensic image container types. Here is the `img_stat` information from our Ext3 test image:

```
user@forensics:~$ img_stat ubnist1.casper-rw.gen3.aff
IMAGE FILE INFORMATION
--------------------------------------------
Image Type: AFF
Size in bytes: 629145600
MD5: 717f6be298748ee7d6ce3e4b9ed63459
SHA1: 61bcd153fc91e680791aa39455688eab946c4b7
Creator: afconvert
Image GID: 25817565F05DFD8CAEC5CFC6B1FAB45
Acquisition Date: 2009-01-28 20:39:30
AFFLib Version: "3.3.5"
```

The `img_cat` command will stream the content of an image file to STDOUT. This is a convenient way to convert a forensic container into a "raw" image.

Journal Tools

Many modern file systems support *journaling*. To grossly simplify, journaling file systems keep a journal of changes they are preparing to make and then they make the changes. Should the system lose power in the middle of a change, the journal is used to replay those changes to ensure file system consistency. Given this, it is possible that the journal may contain data not found anywhere else in the active file system.

The `jls` command lists items in the file system journal, and the `jcat` command streams the content of the requested journal block to STDOUT. As the information provided by these tools is highly file system specific, we will discuss the use of both of them in the relevant file system sections in the following chapters.

PARTITIONING AND DISK LAYOUTS

The two primary partitioning schemes in use today are the "Master Boot Record (MBR)" and the "GUID Partition Table (GPT)." The GPT scheme was developed as a replacement for the aging MBR scheme. The MBR partitioning method originally only allowed for four primary partitions and disks of up to 2 Terabytes, a size that is quite possible to exceed nowadays. The GPT format supports disks up to 8 Zettabytes in size and 128 primary partitions, along with many more improvements. The partition table is not likely to contain any information of relevance to most investigations. Forensic analysis of the partition table is usually limited to recovery of volumes when the partitioning structures are missing or corrupted.

Partition Identification and Recovery

Identification of deleted or otherwise missing partitions can be performed using the `sigfind` tool mentioned earlier. The tool includes a number of predefined data structure templates that will locate the tell-tale marks of a partition table or file system header. We can test this using the 10th test image from the Digital Forensic Tool Testing project (http://dftt.sourceforge.net/test10/index.html). The "dospart" template looks for the hex value "55AA" in the last two bytes of each sector, a structure common to MBR partitions.

```
user@ubuntu:~/10-ntfs-autodetect$ sigfind -t dospart 10-ntfs-
  autodetect/10-ntfs-disk.dd
Block size: 512 Offset: 510 Signature: 55AA
Block: 0 (-)
Block: 63 (+63)
Block: 96389 (+96326)
Block: 96390 (+1)
```

We can compare this with `mmls` output for the same image:

```
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors
     Slot    Start        End          Length       Description
00:  Meta    0000000000   0000000000   0000000001   Primary Table (#0)
01:  -----   0000000000   0000000062   0000000063   Unallocated
02:  00:00   0000000063   0000096389   0000096327   NTFS (0x07)
03:  00:01   0000096390   0000192779   0000096390   NTFS (0x07)
04:  -----   0000192780   0000192783   0000000004   Unallocated
```

We can see that sigfind located the 0x55AA signature in the boot sector (0), the beginning and end of the first volume (63 and 96389), and the beginning of the next volume (96390).

NOTE

Other Media Management Schemes

The Sleuth Kit is able to recognize two other volume layer layouts: Sun *slices* (used by Solaris) and BSD *disklabels* (used by BSD-based operating systems). We don't cover analysis of either platform in this book, but should you need to, you can use the Sleuth Kit on these volumes as well.

Additionally, the TestDisk tool from CGSecurity can be used to recover partitions in the case of disk corruption or intentional spoiling. TestDisk can operate on both raw and Expert Witness/E01 format files used by EnCase. An excellent tutorial on the use of TestDisk is provided at the CGSecurity site [2]. Testdisk can be installed on Ubuntu via *apt-get*. The source code and precompiled binaries for DOS, Windows, OS X, and Linux are also available from the CGSecurity site (www.cgsecurity.org).

Redundant Array of Inexpensive Disks

Redundant Array of Inexpensive Disks (RAID) is designed as a means to take multiple physical disks and address them as a single logical unit.

The most commonly used basic RAID levels are:

- **RAID 0** refers to a setup of at least two disks that are "striped" at a block level. Given two disks (0 and 1), block A will be written to disk 0, block B will be written to disk 1, and so on. This increases write speeds and does not sacrifice any storage space, but increases the fragility of data, as losing a single drive means losing half of your blocks.
- **RAID 1** is the opposite of RAID 0—blocks are mirrored across pairs of drives. This increases read speeds and reliability, but reduces the amount of available storage to half of the physical disk space.

- **RAID 5** requires at least three disks and performs striping across multiple disks in addition to creating *parity blocks*. These blocks are also striped across disks and are used to recreate data in the event a drive is lost.

Additionally, there are "nested" or "hybrid" RAID setups that combine two of these RAID levels in sequence. For example, a RAID 50 or 5+0 set would be a pair of RAID5 sets that are subsequently striped.

The Sleuth Kit has no built-in capability for dealing with RAID. The ***PyFLAG*** suite discussed in Chapter 9 includes a command line python utility called ***raid_guess.py*** that can be used to reconstruct a RAID map when given a set of disk images [3]. That said, the authors recommend using the original hardware the RAID is housed in to perform imaging whenever possible. There are many different RAID implementations in use, and recreating the logical structure after the fact can be perilous.

SPECIAL CONTAINERS

In addition to file systems in volumes on physical media, you may have to deal with file systems in other containers. One example is the Macintosh-specific DMG container discussed in the previous section. The other two major containers you are likely to encounter are *Virtual Machine Disk Images* and *Forensic Containers*.

Virtual Machine Disk Images

Virtualization applications such as VMWare, VirtualBox, Virtual PC, and QEMU allow users to run a full "virtual machine" within the host operating system. Generally, they store the file systems used by these virtual machines as virtual disk images—container files that act as a "disk" for purposes of virtualization software. If it acts like a disk for virtualization software, we should be able to get it to act as a disk for purposes of extracting artifacts. The most common virtual disk format today is *VMDK*, used by VMWare's virtualization products.

A VMWare virtual disk is defined using a *descriptor file* that defines the file(s) that makes up that particular virtual disk, as well as specifications of the "disk" being presented to the virtual machine. A disk is originally formed from the base file (or files in the case where the disk is created as a series of 2-GB split chunks). As users create snapshots of a virtual machine, files containing changes from the base image called *delta links* are created, and a new descriptor file containing information about the base and delta files is created.

The full VMDK specification is available from VMWare at http://www.vmware.com/app/vmdk/?src=vmdk.

AFFLib supports VMDK containers natively, and Sleuth Kit will import this functionality if built with AFF support. We can use any of the sleuth kit tools directly against a VMDK by specifying the "afflib" parameter to the image type argument (`-i`).

TIP

Creating VMDKs from Raw Images

In some circumstances it is useful to be able to access a raw image in a virtual machine. Two projects are available that provide just this functionality. LiveView (http://liveview.sourceforge.net/) is a graphical application targeted for use on Windows with limited Linux support that will create all the files needed to generate a VMWare-bootable virtual machine.

Raw2VMDK (http://segfault.gr/projects/lang/en/projects_id/16/secid/28/) is a command line utility that simply generates a valid VMDK file that points to your existing raw image. You can then use this VMDK in any number of ways. For example, the VMDK can be added as a secondary (read-only) disk attached to a forensics-oriented virtual machine.

NOTE

Other Virtual Disk Formats

While the most common, VMWare's VMDK is by no means the only virtual disk format in use.

VDI is the virtual disk format used by Sun's open source virtualization platform VirtualBox.
VHD is the format used by Microsoft's Virtual PC product, as well as the "built-in" virtualization capability found in Windows 7 and Server 2008.
QCOW2 is the format used currently by the open source QEMU project.

Should you need to do so, these disk formats can be converted into either VMDKs or raw images suitable for forensic processing using either the *qemu-img* utility (part of the QEMU package) or the *vboxmanage* utility from VirtualBox.

Forensic Containers

We have already spent a little time working with forensic containers, but we have not gone into detail about what exactly they are. In general, container formats geared toward forensic imaging have some functionality above and beyond what we get with a raw disk image. This can include things such as internal consistency checking, case information management, compression, and encryption. We can, of course, perform any of these tasks with a raw image as well. The difference is for a forensic container format, these functions are built into the format, reducing the administrative overhead involved with things such as ensuring that the hash and case notes for a given image are kept with that image at all times.

EWF/E01

The most commonly used forensic container format is the *Expert Witness Format* (EWF), sometimes referred to as the "E01" format after its default extension. This native format is used by Guidance Software's *EnCase* forensic suite. This "format" has changed somewhat from one release of EnCase to the next and is not an open standard. That said, the LibEWF library supports all modern variants of image files generated by EnCase in this format.

The structure of this format has been documented by its original author, Andy Rosen of ASRData, with further documentation performed by Joachim Metz during his work on the LibEWF project [4]. The EWF format supports compression, split files, and stores case metadata (including an MD5 or SHA1 hash of the acquired image) in a header data structure found in the first segment of the image file. Examiners interested in the inner workings of the EWF format should reference these documents.

AFF

The Advanced Forensics Format (AFF) is an open source format for storing disk images for forensics, as well as any relevant metadata. AFF is implemented in the LibAFF package we installed previously. The Sleuth Kit supports AFF image files through this library. AFF images can be compressed, encrypted, and digitally signed. An interesting feature of the AFF format is that metadata stored in the image file are extensible—arbitrary information relevant to the case can be stored directly in the image file in question.

AFF images can be stored in one of three methods:

- AFF—This is the default format of an AFF container; this is a single image file containing forensic data as well as case metadata.
- AFD—This format contains metadata in the image, but splits the image file into fixed-size volumes. This can be useful when transporting or archiving images via size-limited file systems or media.
- AFM—This format stores the image file as a single, solid container but stores metadata in an external file.

HASHING

One of the key activities performed at many different points throughout an examination is generation of a cryptographic hash, or *hashing.* A cryptographic hash function takes an arbitrary amount of data as input and returns a fixed-size string as output. The resulting value is a *hash* of data. Common hashing algorithms used during a forensic examination include MD5 and SHA1. MD5 produces a 128-bit hash value, while SHA1 produces a 160-bit hash value. Longer versions of SHA can be used as well; these will be referred to by the bit length of the hash value they produce (e.g., SHA256 and SHA512).

For hash functions used in forensic functions, modification of a single bit of input data will produce a radically different hash value. Given this property, it is easy to determine one of the core uses for hashing in forensic analysis: verification of the integrity of digital evidence. A hash generated from the original evidence can be compared with a hash of the bit-stream image created from this evidence—matching hashes show that these two items are the same thing. Additionally, taking an additional hash after completing examination of a forensic copy can show that the examiner did not alter source data at any time.

Other characteristics of hash functions make them valuable for additional forensic uses. Because a hash is calculated by processing the content of a file, matching hashes across various files can be used to find renamed files, or to remove "known good" files from the set of data to be examined. Alternately, the hashes of files of interest can be used to locate them irrespective of name changes or other metadata manipulations.

Many programs that implement the MD5 and SHA* algorithms are available for a variety of platforms. For simply generating a hash of a single file, the `md5sum` or `sha1sum` programs present on nearly on Linux systems are sufficient. Using these programs to generate hash lists of multiple files or multiple nested directories of files can be quite tedious. To solve this problem, Jesse Kornblum has produced the `md5deep` and `hashdeep` utilities.

`Md5deep` is a suite of hashing utilities designed to recurse through a set of input files or directories and produce hash lists for these. The output is configurable based on the examiners requirements and, despite the name, the suite includes similar tools implementing SHA* and other hashing algorithms. `Hashdeep` is a newer utility developed as a more robust hash auditing application. It can be used to generate multiple hashes (e.g., MD5 and SHA1 hashes) for files and can be used to subsequently audit the set of hashed data. After generating a base state, `hashdeep` can report on matching files, missing files, files that have been moved from one location to another, and files that did not appear in the original set. Full usage information and tutorials, source code, and binaries for Windows are available at the md5deep site [5].

As stated earlier, the fact that a change in a single input bit will change many bits in the final hash value is one of the valuable characteristics of hash functions for purposes of proving a file's content or integrity. If you instead want to prove that two files are *similar* but not identical, a standard hashing approach will not help—you will only be able to tell that two files are different, not *how* different. Jesse Kornblum's `ssdeep` was developed to provide this capability, which Jesse calls "context triggered piecewise hashes" "fuzzy hashing [6]." To simplify, fuzzy hashing breaks the input file into chunks, hashes those, and then uses this list to compare the similarity of two files. The hashing window can be tuned by the end user.

We can see the basic operation of ssdeep in the console output that follows. The author generated a paragraph of random text and then modified capitalization of the first word. The MD5 hashes are wildly different:

```
MD5 (lorem1.txt) = ea4884844ddb6cdc55aa7a95d19815a2
MD5 (lorem2.txt) = 9909552a79ed968a336ca3b9e96aca66
```

We can generate fuzzy hashes for both files by running `ssdeep` with no flags:

```
ssdeep,1.1--blocksize:hash:hash,filename
24:FPYOEMR7SlPYzvH6juMtTtqULiveqrTFIoCPddBjMxiAyejao:
  9YfQ7qYza6MdtiHrTKoCddBQxiwd,"/home/cory/ssdeep-test/lorem1.txt"
24:lPYOEMR7SlPYzvH6juMtTtqULiveqrTFIoCPddBjMxiAyejao:dYfQ
  7qYza6MdtiHrTKoCddBQxiwd,"/home/cory/ssdeep-test/lorem2.txt"
```

By inspecting both sets of fuzzy hashes visually, we can identify that they match, except for the first byte, which is where our modification occurred. Alternately, we can run ssdeep in directory mode by passing the `-d` flag, which will compare all files in a directory:

```
user@ubuntu:~/ssdeep-test$ ssdeep -d *
/home/user/ssdeep-test/lorem2.txt matches /home/user/ssdeep-
  test/lorem1.txt (99)
```

Full usage information and tutorials, source code, and binaries for Windows are available at the `ssdeep` site [7].

NOTE

Hash Collisions

Over the past few years there have been some publicized attacks against the MD5 algorithm in which researchers were able to generate two different files that generated the same MD5 hash value. All of the attacks made public thus far have been in the category of *collision attacks.* In a collision attack, a third party controls both files. This scenario is not applicable for most of the tasks we use hashing for in forensic analysis, such as verifying an image file has not been altered or verifying a file against a set of known good or bad hashes. That said, tools such as `hashdeep` can use multiple hash algorithms (in addition to nonhash data like file size) to strengthen the confidence of a hashset.

CARVING

A wise forensic examiner once said "when all else fails, we carve." Extraction of meaningful file content from otherwise unstructured streams of data is a science and an art unto itself. This discipline has been the focus of numerous presentations at the Digital Forensics Research Workshop over the years, and advancements continue to be made to this day. At its most basic, however, the process of carving involves searching a data stream for file headers and magic values, determining (or guessing) the file end point, and saving this substream out into a carved file. Carving is still an open problem and is an area of ongoing, active experimentation. Numerous experimental programs are designed to implement specific new ideas in carving, as well as more utilitarian programs geared toward operational use.

TIP

hachoir-subfile

The `hachoir-subfile` program can be used to intelligently identify files within binary streams, including unallocated space from disk images. It operates in a manner similar to the `sigfind`, but uses intelligence about file formats to provide a much stronger signal that an actual file has been located, minimizing false positives. While not a carving tool in and of itself, it can be used to positively identify files inside of a stream for subsequent manual extraction. The *hachoir* suite of programs is discussed in detail in Chapter 8.

Foremost

`Foremost` is a file carving program originally written by Jesse Kornblum and Kris Kendall at the Air Force Office of Special Investigations and later updated by Nick Mikus of the Naval Postgraduate School. It uses defined headers, footers, and knowledge of the internal structures for supported file types to aid in carving. A complete list of the file types supported natively by foremost can be found in the program's man page, but suffice it to say it includes the usual suspects: JPEG images, office documents, archive files, and more. If necessary, additional file types can be defined in a custom *foremost.conf* file. We will discuss the analysis of files and their content in Chapter 8.

Foremost can be installed easily using `apt-get` on Ubuntu or by retrieving and compiling the source (or supplied binaries) from the foremost project page at SourceForge: http://foremost.sourceforge.net/. Options that may be particularly important include:

```
-d - turn on indirect block detection (for UNIX file-systems)
-i - specify input file (default is stdin)
-a - Write all headers, perform no error detection (corrupted
     files)
-w - Only write the audit file, do not write any detected files
     to the disk
-o - set output directory (defaults to output)
-c - set configuration file to use (defaults to foremost.conf)
-q - enables quick mode. Search are performed on 512 byte
     boundaries.
```

We can perform a basic run of foremost using the Digital Forensics Research Workstop 2006 carving challenge file as input [8]. We will use the `-v` flag to increase the verbosity of the output.

```
user@ubuntu:~/dfrws $ foremost -v -i dfrws-2006-challenge.raw
Foremost version 1.5.4 by Jesse Kornblum, Kris Kendall, and Nick Mikus
Audit File
Foremost started at Sat Dec 10 21:51:55 2010
Invocation: foremost -v -i dfrws-2006-challenge.raw
Output directory: /home/user/dfrws/output
Configuration file: /usr/local/etc
Processing: dfrws-2006-challenge.raw
|------------------------------------------------------------------
File: dfrws-2006-challenge.raw
Start: Sat Jan 1 21:51:55 2011
Length: Unknown

Num    Name (bs=512)     Size        File Offset         Comment
0:     00003868.jpg       280 KB       1980416
1:     00008285.jpg       594 KB       4241920
2:     00011619.jpg       199 KB       5948928
3:     00012222.jpg         6 MB       6257664
```

```
4:    00027607.jpg       185 KB       14134784
5:    00031475.jpg       206 KB       16115200
6:    00036292.jpg       174 KB       18581504
7:    00040638.jpg       292 KB       20806656
8:    00041611.jpg         1 MB       21304832
9:    00045566.jpg       630 KB       23329792
10:   00094846.jpg       391 KB       48561152
11:   00000009.htm        17 KB       4691
12:   00004456.htm        22 KB       2281535
13:   00027496.htm       349 KB       14078061
14:   00028244.htm        50 KB       14460928
15:   00029529.htm       183 KB       15118957
16:   00032837.doc       282 KB       16812544
17:   00045964.doc        71 KB       23533568
18:   00028439.zip       157 KB       14560768
19:   00030050.zip       697 KB       15385752
20:   00045015.zip       274 KB       23047680
21:   00007982.png         6 KB       4086865      (1408 x 1800)
22:   00033012.png        69 KB       16902215     (1052 x 360)
23:   00035391.png        19 KB       18120696     (879 x 499)
24:   00035431.png        72 KB       18140936     (1140 x 540)
*|
Finish: Sat Jan 1 21:51:57 2011
25 FILES EXTRACTED

jpg:= 11
htm:= 5
ole:= 2
zip:= 3
png:= 4
```

Note that due to the intentional fragmentation of this test image, the bulk of these extracted files will not be identical to the original items. Simson Garfinkel presented research at the Digital Forensics Research workshop in 2007 that indicated that the majority of files on any give volume will be contiguous and that most fragmented files are simply split into two fragments, with a single block splitting the halves [9].

TIP

Additional Carving Utilities

`Scalpel` is a file carver forked from Foremost version 0.69 and completely rewritten with an eye toward increasing performance. The latest public release of scalpel is version 1.60, released in December 2006. The authors have presented papers referencing advanced versions of scalpel with parallelized carving support and GPU acceleration, but at the time of this publication these have not been released publicly [10].

`PhotoRec` is an advanced, cross-platform carving program distributed as part of the TestDisk program mentioned in the *Partition Identification and Recovery* section. Like TestDisk, CGSecurity provides an extensive guide that details use of the tool on their Web site [11].

The most common scenario for carving in an actual investigation is the attempted retrieval of deleted data for which metadata are no longer present or no longer linked. In these cases, extracting the unallocated space of the volume into a contiguous block using `blkls` has the potential to eliminate fragmentation caused by currently allocated blocks.

FORENSIC IMAGING

In creation of a forensic image, we are trying to capture an accurate as possible representation of source media. This is not unlike the police lines set up at a physical crime scene. These lines are put in place to minimize the amount of change that occurs in a crime scene, which in turn gives the crime scene investigators the most accurate data possible.

Imagine, then, if the crime scene investigators could create a *copy* of the actual crime scene. In the real world this is madness, but this is what we aim to do with creation of a forensic image.

A good forensic imaging process generates an exact duplicate (or a container that holds an exact duplicate) of the source media under investigation. By *exact duplicate* we mean exactly that—we aim to acquire a complete sector-for-sector, byte-for-byte copy of original media. There should be no on-disk information present on source media that do not appear in our forensic image. An ideal imaging process should not alter original media, fail to acquire any portion of original media, nor introduce any data not present on source media into the image file.

A traditional forensic analyst examining a gun used in a homicide works on the original. Why doesn't the computer forensic examiner do the same? Examiners generate forensic images for several reasons. The primary reason is to provide an *exact copy* of original media to examine. For the traditional analyst, the actual weapon is the *best evidence*. In the case of digital evidence, we can make a duplicate of source media that matches the original in every way. Working with original digital evidence can be very dangerous because the original can be altered or destroyed with relative ease. By only accessing the original media once, to generate our forensic image, we minimize our opportunities to alter the original accidentally. Another benefit of working on an image is if we make a mistake and somehow end up altering the image file in some way, we can generate a new exact duplicate from the intact original media.

Deleted Data

Another reason examiners use forensic imaging is for *completeness*. Simply examining an active file system as presented by the operating system is not sufficiently thorough for a forensic examination. Most volumes contain reams of potentially interesting data outside of the viewable, allocated files on a mounted file system. This includes several categories of "deleted data."

- ***Deleted*** files are the "most recoverable." Generally this refers to files that have been "unlinked"—the file name entry is no longer presented when a user views a directory, and the file name, metadata structure, and data units are marked as "free." However, the connections between these layers are still intact when forensic techniques are applied to the file system. Recovery consists of recording the relevant file name and metadata structures and then extracting the data units.
- ***Orphaned*** files are similar to ***deleted*** files except the link between the file name and metadata structure is no longer accurate. In this case, recovery of data (and metadata structure) is still possible but there is no direct correlation from the file name to recovered data.
- ***Unallocated*** files have had their once-allocated file name entry and associated metadata structure have become unlinked and/or reused. In this case, the only means for recovery is carving the not-yet-reused data units from the unallocated space of the volume.
- ***Overwritten*** files have had one or more of their data units reallocated to another file. Full recovery is no longer possible, but partial recovery may depend on the extent of overwriting. Files with file names and/or metadata structures intact that have had some or all data units overwritten are sometimes referred to as *Deleted/Overwritten* or *Deleted/Reallocated*.

File Slack

As mentioned previously, the minimum space that can be allocated on a volume is a single block. Assuming a 4K block size, on a standard drive with 512-byte sectors, this means the ASCII text file containing a single byte—the letter 'a'—will consume eight sectors on the disk. We provided the 'a'—where did the other 4095 bytes written to the disk come from?

The answer is, as always, it depends. Different file systems and operating systems handle this differently, but generally the process goes:

- The cluster to be used is marked as "allocated" and assigned to the file's metadata structure.
- The 'a' followed by 511 null bytes (hex 00) are placed in the first sector.

Astute readers will note that we didn't state how the next seven sectors are written to the disk. That's not an oversight—they aren't written to the disk. They retain whatever data were last stored in them during their *previous* allocation. This is what is known as *file slack* or *slack space*.

Figure 3.2 demonstrates the generation of file slack using three successive views of the same eight blocks on a disk. At first the row consists of new, empty, unallocated blocks. Then, file A is created, has eight blocks allocated to it, and those eight blocks are filled with data. File A is then "deleted" and sometime later the first five blocks are reallocated and overwritten with the content from File B. This leaves three of the blocks containing data from File A unallocated but recoverable.

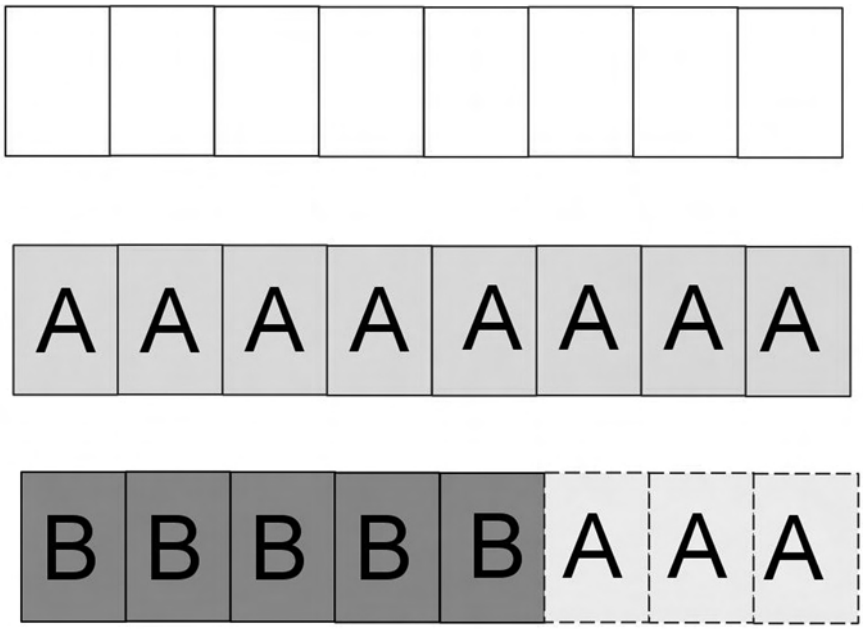

FIGURE 3.2

File slack.

NOTE

RAM Slack

While all modern operating systems pad the written sector with null bytes, this was not always the case. MS-DOS and older DOS-based versions of Microsoft Windows would pad the rest of the sector out with whatever contents of memory happened to be next to data being written. These data, between the end of allocated data and the beginning of previously allocated data, became known as *RAM slack*. Given this, RAM slack could potentially contain data that were never written to the disk, such as cryptographic keys or passphrases.

TIP

Volume or Disk?

When creating a forensic image, most of the time an examiner will use the physical disk (e.g., */dev/sda*) as input. However, in some circumstances you may be better off imaging the volume or volumes of interest (e.g., */dev/sda1*). One example is when dealing with a RAID array. Imaging physical disks requires the capability to rebuild the RAID from these disk images at a later date, which (as mentioned previously) can be difficult. Depending on the type of RAID and the utilities available to you as an examiner, this may prove to be difficult or impossible. Another example is in the case of Storage Area Network volume—with many of these systems, removing and imaging the physical drives are simply not options.

dd

The `dd` command is the most basic open source tool available to create a forensic image. Because it is nearly universally present on any Unix-like operating system and is the basis for several other forensic imaging utilities, learning its operation is valuable to any examiner. Put simply, `dd` copies data from one place to another. The user can provide various arguments and flags to modify this simple behavior, but the basic syntax of the tool is fairly clear. The excerpt from the tool help given here has the basic options you need to understand in bold.

```
user@forensics:~$ dd --help
Usage: dd [OPERAND]…
  or: dd OPTION
Copy a file, converting and formatting according to the operands.
  bs=BYTES force ibs=BYTES and obs=BYTES
  cbs=BYTES convert BYTES bytes at a time
  conv=CONVS convert the file as per the comma separated symbol list
  count=BLOCKS copy only BLOCKS input blocks
  ibs=BYTES read BYTES bytes at a time
  if=FILE read from FILE instead of stdin
  iflag=FLAGS read as per the comma separated symbol list
  obs=BYTES write BYTES bytes at a time
  of=FILE write to FILE instead of stdout
  oflag=FLAGS write as per the comma separated symbol list
  seek=BLOCKS skip BLOCKS obs-sized blocks at start of output
  skip=BLOCKS skip BLOCKS ibs-sized blocks at start of input
  status=noxfer suppress transfer statistics
```

So, to make a simple clone from one drive to another, we would invoke the tools like so:

```
dd if=/dev/sda of=/dev/sdb bs=4096
```

This takes reads from the first disk, 4096 bytes at a time, and writes the content out to the second disk, 4096 bytes at a time. If we did not provide the *block size* (`bs=`) argument, `dd` would default to reading and writing a single 512-byte sector at a time, which is quite slow.

Cloning a disk is interesting but of limited use for an examiner. For the most part, we are interested in creating a *forensic image file*—a file that contains all of the content present on the source disk. This, also, is simple to do using the same syntax.

```
user@forensics:~$ sudo dd if=/dev/sdg of=dd.img bs=32K

[sudo] password for user:
60832+0 records in
60832+0 records out
1993342976 bytes (2.0 GB) copied, 873.939 s, 2.3 MB/s
```

The key items of interest in the console output for the `dd` command are "records in" and "records out" lines. First, they match, which is good—this indicates that we did not lose any data due to drive failures, failure to write the output file fully, or any other reason. Second, the "60832+0" records indicate that exactly this many 32K blocks were both read and written. If we had imaged a drive that was not an exact multiple of 32K in size, the "+0" would instead show "+1," indicating that a partial record was read (and written).

Some of the other options of forensic interest present in the base dd command are the `conv` (convert) option. If imaging a failing or damaged hard drive, the `conv=noerror,sync` option can be used to ignore read errors, writing blocks of NULL characters in the output file for every block that was unable to be read. Additionally, in the case of a dying drive, supplying the `iflag=direct` option (use direct I/O, bypassing the kernel drive cache) and reducing the block size to 512 bytes will ensure that the amount of unrecoverable data is kept to a minimum.

WARNING

Bad Sectors

Note that using `dd` with the `conv=noerror` argument is **not** the recommended course of action when attempting to image a damaged hard drive. Given the option, we recommend using GNU `ddrescue`, a specialized version of `dd` designed to deal with retrieving data from uncooperative drives. However, in some cases your only option may be to either retrieve a partial image using `dd` or retrieve nothing at all.

dcfldd

While `dd` can and has been used to acquire forensically sound images, versions of `dd` are available that are specifically designed for forensic use. The first of these to be examined is `dcfldd`, created for the Defense Computer Forensics Laboratory by Nick Harbour. The `dcfldd` project forked from GNU `dd`, so its basic operation is quite similar. However, `dcfldd` has some interesting capabilities that aren't found in vanilla `dd`. Most of the capabilities revolve around hash creation and validation, logging of activity, and splitting the output file into fixed-size chunks. The extended `dcfldd` functions, as well as base `dd` functions, can be reviewed by passing the `--help` flag to the `dcfldd` command.

Unsurprisingly, performing the same image acquisition that was done with `dd` using `dcfldd` is quite similar. In fact, if we did not want to take advantage of the additional features of `dcfldd`, we could use the exact same arguments as before and would get the same results. In the code section following, we reimage the same device as previously, but at the same time generate a log of the md5 and sha1 hashes generated of each 512 megabyte chunk of the disk.

```
user@forensics:~$ sudo dcfldd bs=32k if=/dev/sdg of=dcfldd.img
  hashwindow=512M hash=md5,sha1 hashlog=dcfldd.hashlog
60672 blocks (1896Mb) written.
60832+0 records in
60832+0 records out
```

dc3dd

The last dd variant we will examine is `dc3dd`, a forensically oriented version created by Jesse Kornblum for the Department of Defense Cyber Crime Center. dc3dd is developed as a patch applied to GNU `dd`, rather than a fork, so `dc3dd` is able to incorporate changes made in the mainline `dd` more rapidly than `dcfldd`. `dc3dd` has all of the same extended features found in `dcfldd` and has core dd features currently absent in the latest `dcfldd` release.

We can provide the same arguments to `dc3dd` that were used previously with `dcfldd`.

```
user@forensics:~$ sudo dc3dd bs=32k if=/dev/sdg of=dc3dd.img
  hashwindow=512M hash=md5,sha1 hashlog=dc3dd.hashlog
[sudo] password for user:
warning: sector size not probed, assuming 512
dc3dd 6.12.3 started at 2010-09-03 17:34:57 -0700
command line: dc3dd bs=32k if=/dev/sdg of=dc3dd.img
  hashwindow=512M hash=md5,sha1 hashlog=dc3dd.hashlog
compiled options: DEFAULT_BLOCKSIZE=32768
sector size: 512 (assumed)
md5  0- 536870912: 07c416f8453933c80319c2d89e5533ad
sha1 0- 536870912: a222f5835ed7b7a51baaa57c5f4d4495b1ca1e79
md5 536870912- 1073741824: acac88a20c6d6b364714e6174874e4da
sha1 536870912- 1073741824:
  5b69440a15795592e9e158146e4e458ec8c5b319
md5 1073741824- 1610612736: ed9b57705e7ae681181e0f86366b85e6
sha1 1073741824- 1610612736:
  bc5369977d9a2f788d910b5b01a9a1e97432f928
md5 1610612736- 1993342976: 812c94592ec5628f749b59a1e56cd9ab
sha1 1610612736- 1993342976:
  bb789315a814159cdf2d2803a73149588b5290ee
md5 TOTAL: 58e362af9868562864461385ecf58156
sha1 TOTAL: 8eaba11cb49435df271d8bc020eb2b46d11902fe
3893248+0 sectors in
3893248+0 sectors out
1993342976 bytes (1.9 G) copied (??%), 908.424 s, 2.1 M/s
dc3dd completed at 2010-09-03 17:50:06 -0700
```

Note that `dc3dd` produces a hash log to the console as well as writing it out to the file passed in the `hashlog=` argument. Additionally, it presents the sector count rather than the block count as a summary upon completion.

TIP

Creating "Expert Witness Format" Images

It is likely that you will have to provide forensic images for use by a third party at some point in your career. Depending on the capabilities and experience of the other party, you may wish to provide them with images in the "Expert Witness Format" discussed in the *Forensic Containers* section. Note that EnCase is entirely capable of reading from "raw" images, but should you receive a specific request to provide images in this format, you can comply using open source tools.

The `ewfacquire` utility is a part of the LibEWF package and provides a robust console interface for generating EWF image files. It is invoked simply by providing the `ewfacquire` command with an input source. The program will prompt the user for information required to generate the image file.

The **guymager** application is a graphical forensic imaging utility that can be used to generate raw, AFF, and EWF image files. Note that guymager uses LibEWF for its EWF support, so functionally these two tools should be the same when generating EWF containers.

SUMMARY

This chapter discussed the core concepts of disk and file system analysis. In addition, it explored many of the fundamental concepts of forensic analysis, such as forensic imaging, dealing with forensic containers, and hashing. Through use of the Sleuth Kit, we have shown how to exploit a file system for artifacts of forensic interest. Subsequent chapters will build upon this foundation to examine and analyze higher level artifacts.

References

[1] B. Carrier, 2005. File System Forensic Analysis. Addison-Wesley, Boston, MA.
[2] TestDisk Step By Step—CGSecurity. http://www.cgsecurity.org/wiki/TestDisk_Step_By_Step.
[3] RAID Recovery—PyFLAG. http://pyflag.sourceforge.net/Documentation/articles/raid/reconstruction.html.
[4] libewf—Browse /documentation/EWF file format at SourceForge.net. http://sourceforge.net/projects/libewf/files/documentation/EWF%20file%20format/.
[5] Getting Started with Hashdeep. http://md5deep.sourceforge.net/start-hashdeep.html.
[6] J. Kornblum, Identifying almost identical files using context triggered *piecewise hashing*. Paper presented at Digital Forensics Research Workshop, 2006. Elsevier, (accessed 13.09.10).
[7] Getting Started with ssdeep. http://ssdeep.sourceforge.net/usage.html.
[8] Digital Forensics Research Workshop 2006 File Carving Challenge. http://www.dfrws.org/2006/challenge/dfrws-2006-challenge.zip.
[9] S.L. Garfinkel, Carving contiguous and fragmented files with *fast object validation*. Paper presented at Digital Forensics Research Workshop, 2007.
[10] Scalpel: A Frugal, High Performance File Carver. http://www.digitalforensicssolutions.com/Scalpel/.
[11] PhotoRec Step By Step – CGSecurity, http://www.cgsecurity.org/wiki/PhotoRec_Step_By_Step (retrieived Jan 9, 2011).