Introduction to The Sleuth Kit (TSK)

By Chris Marko

Rev1
September, 2005

This paper provides an introduction to The Sleuth Kit (referred to as TSK herein), from Brian Carrier, available at http://www.sleuthkit.org/. This is a free UNIX package of command line file system and media management forensic tools. It comes in source code format, and is compiled on your UNIX or Linux system of choice. Compilation is relatively simple by just typing `make`. This paper will assume you have a fundamental knowledge of Linux, and will not go into great detail about Linux installation or usage. If a Linux-based machine is not available to you, TSK is also pre-compiled and included in many forensic-related Linux distributions. Such examples include the CD-based Penguin Sleuth Kit, as well as an excellent distribution called Helix from e-fense, Inc. Furthermore, the assumption of basic knowledge of computer forensics is made, including a familiarity with different file systems, including NTFS, FAT, and EXT3.

History

TSK itself is based on designs of its predecessors The Coroner's Toolkit (TCT), as well as TCTUTILS. In order to better understand the functions of TSK, it is important to know a bit of its history. In 2000, Dan Farmer and Wietse Venema released the first version of a utility suite known as The Coroner's Toolkit (TCT). TCT was groundbreaking in that the general public finally had access to quality digital forensic tools that were not only free, but also open source. As great as TCT was, it did have its limitations. One limitation of TCT was that the file system tools only operated at the block and inode layer. Therefore, there was no concept of file and directory names during a post-mortem analysis. Another limitation was platform dependence. The analysis system had to be of the same OS type as the system being analyzed. For example, if a Solaris system was being investigated, then the analysis platform also had to be Solaris. In other words, you could not use a Linux system to perform the analysis of a Solaris system. Lastly, support for common file systems such as FAT and NTFS simply did not exist.

Brian Carrier has recognized these limitations and addressed many of them with TSK. Additionally, there is now a graphical front-end to TSK called Autopsy. However, covering Autopsy is beyond the scope of this paper, so only casual reference is made to it. Autopsy relies on the TSK tools, so a thorough understanding of this platform is important when comprehending Autopsy analysis. Furthermore, Linux has matured quite a bit since TCT first came out. Linux itself now supports the capability of recognizing more than a dozen different file system types with its own native tools.

Before jumping in and using TSK, it is also important to understand a few basic concepts. The original TSK approaches forensic analysis in a layered approach, with different tools serving a specific purpose for each layer. Each layer represents a different level of analysis of media. However, as TSK has grown, many new tools are either crossing these layers, or falling into there own unique areas. To better understand the methodology of the TSK tool names and their purpose, it is important to understand the layer concept.

The TSK layer approach starts with the "File System Layer." A disk contains one or more partitions (or slices). Each of these partitions contains a file system. Examples of

file systems include the Berkeley Fast File System (FFS), Extended 2 File System (ext2fs), File Allocation Table (FAT), and New Technologies File System (NTFS). This layer contains the values that identify how this file system is different than another file system of the same type. Management structures, such as bitmaps, are also in this category. Tools included with TSK that work in this layer start with 'fs'.[1] One such tool is the `fsstat` program. This utility displays file system details in an ASCII format, such as volume name, last mounting time, and the details about each "group" in UNIX file systems.

The next layer is the "Content Layer", or data. This is where the file and directory content is stored. Data is stored in large chunks, with names such as blocks, fragments, and clusters. In general, disk space is allocated from here when the OS needs it. The data units are a fixed size and most file systems require it to be a power of 2, such as 1024bytes or 4096bytes. The data units are called different things in different file system types, such as fragments and clusters, but they serve the same function. All tools in the TSK collection related to this layer begin with the letter 'd'.[1] The `dcat` tool can be used to display the contents of a specific unit of the file system (similar to what 'dd' can do with a few arguments). The unit size presented by `dcat` is file system dependent. The `dls` tool displays the contents of all unallocated units of a file system, resulting in a stream of bytes of deleted content. The output can be searched for deleted file content. The `dcalc` program allows one to identify the unit location in the original image of a unit in the `dls` generated image.

The third layer is known as the "Metadata Layer", or inode layer. This is where the descriptive data about files and directories are stored. This layer includes the inode structures in UNIX, MFT entries in NTFS, and directory entry structures in FAT. This layer also contains descriptive data such as dates and size as well as the addresses of the data units. The structures that the data is stored in have names such as inode and directory entry. All tools in the TSK collection related to this layer begin with an 'i'.[1] One such tool is the `ils` program, which lists some values of the metadata structures. By default, it will only list the unallocated ones. The `istat` utility displays metadata information in an ASCII format about a specific structure. One nice feature of `istat` is that it will display the destination of symbolic links. The `icat` function displays the contents of the data units allocated to the metadata structure (similar to the UNIX cat command). The `ifind` tool will identify which metadata structure has allocated a given content unit or file name.

Finally, we have the "Human Interface Layer", or file layer. The human interface layer allows one to interact with files in a manner that is more convenient than directly with the metadata layer. This is where the actual name of the file or directory is saved. In general, this is a different structure than the meta data structure. The exception to this is the FAT file system. The file names are typically saved in the data units that the parent directory allocates. The file name structure contains the name and the address of the corresponding meta data structure. All tools in this layer begin with the letter 'f'.[1]

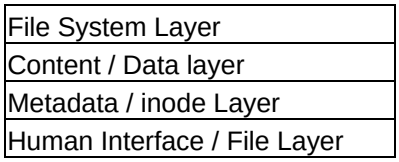

Figure 1. The Sleuth Kit Layers

The `fls` program lists file and directory names. This tool will display the names of deleted files as well. The `ffind` program will identify the name of the file that has allocated a given metadata structure. With some file systems, deleted files will be identified.

Tool Prefix	Layer / Function	Tools
disk	(Disk Tools)	disk_sreset, diskstat
img	(Image File Tools)	img_stat
mm	(Media Management Tools)	mmls
fs	File System Layer	fsstat
j	(File System Journal Tools)	jcat, jls
i	Metadata / inode Layer	ils, icat, istat, ifind
d	Content / Data Layer	dls, dcat, dstat, dcalc
f	Human Interface / File Layer	fls, ffind

Other miscellaneous tools bundled that transcend the layer methodology include `hfind`, `mactime`, and `sorter`. These will be covered in more detail below.

Disk Tools

Some ATA disks may have what is known as a Host Protected Area (HPA), which is an area of disk that is often not seen by disk imaging applications. An HPA could possibly be used to hide data from such applications, so that it is not copied during an image acquisition. The disk_stat utility included with TSK will tell you if such an area exists on a particular ATA disk. For example:

```
# disk_stat /dev/hdb
```

The `disk_sreset` utility can be used to detect and remove the HPA of an ATA disk. Running `disk_sreset` will temporarily reset the size of an ATA disk, or basically removes an HPA, if one exists. This will help alleviate the question as to whether your preferred disk imaging application will actually capture data in the HPA. After obtaining your image and the system is rebooted, resetting the disk, the HPA will return. For example, to remove the HPA on a secondary disk hdb:

```
# disk_sreset /dev/hdb
```

Image & Media Processing

The steps that follow are going to assume that you have already made a forensically sound image of the suspect media, as well as have successfully copied the image to your analysis workstation with TSK installed.

Originally, TSK itself had no way to handle an image of an entire disk, and required further steps outside of the scope of the provided tools to process an image at a partition level. However, this is where a relatively new tool called `mmls` comes in. The `mmls` utility currently supports the following file system types, which are specified with the `-t` flag:

```
- dos: DOS-based partitions (pretty much any disk in an x86 system)
- bsd: FreeBSD, OpenBSD, and NetBSD disklabels and slices
- mac: Macintosh partitions
- sun: Solaris Volume Table of Contents structures and slices
```

`mmls` can be used to retrieve information about different partitions within an image, and with the help of `dd`, can be used to extract a specific partition or partitions to a file for mounting.

Here is example `fdisk` output of a disk image:

```
# fdisk -lu images/disk.dd
      Device Boot    Start       End    Blocks   Id  System
    disk.dd1            63   2056319   1028128+   b  Win95 FAT32
    disk.dd2       2056320   8209214   3076447+  a6  OpenBSD
    disk.dd3   *   8209215  19999727   5895256+  a5  FreeBSD
```

Here is example `mmls` output of the same disk image:

```
# mmls -t dos images/disk.dd
         Slot    Start        End          Length       Description
    00:  -----   0000000000   0000000000   0000000001   Primary Table (#0)
    01:  -----   0000000001   0000000062   0000000062   Unallocated
    02:  00:00   0000000063   0002056319   0002056257   Win95 FAT32 (0x0B)
    03:  00:01   0002056320   0008209214   0006152895   OpenBSD (0xA6)
    04:  00:02   0008209215   0019999727   0011790513   FreeBSD (0xA5)
```

The first column is the `mmls` index of entries. The second column, labeled “Slot”, shows the location of the partition within the media management structures. The format of the slot field is xx:yy, where xx is the partition number, and yy is the entry within that partition table. So, 00:00 would be the first entry within the first partition table, and 01:00 would be the first entry within a second partition table (such as an extended partition). The “Start”, “End”, and “Length” columns are all displayed in sectors, and the last column, “Description”, is the type of partition found.

Now, to extract a particular partition, we can combine the use of the `dd` utility with the information provided by `mmls`. To extract the FAT partition from the previous example, note our use of the Start and Length information:

```
# dd if=disk.dd of=part1.dd bs=512 skip=63 count=2056257
```

If you are unsure of the type of image file format type, the `img_stat` tool can be used to display details about the image file. Example information includes the sector information of each split file, as well as other embedded data for other file formats. The –t flag can be used to determine the file format type.

General File System Information

The `fsstat` command can be used against a file system to return general known information, such as layout, allocation structures, and boot blocks. This information is typically retrieved from the boot sector or superblock of a file system and does not apply to any specific file or directory. Examples of information provided include the size of data units, the number of data units within the file system, and the number of metadata structures.[10]

Each file system has its own proprietary information, and because of that, `fsstat` is broken up into different sections. For example, the FAT section includes four different pieces of output. The first three are based on the file system, content, and metadata categories of the file system model used by TSK. The last section of the output contains a graphical representation of the file allocation table (FAT). While specific information about each file system type is beyond the scope of this paper, we can still take a limited look by examining the FAT output further.

The first sector of a FAT file system contains the boot sector. The boot sector contains a data structure of basic administrative information, including the layout of the file system. Following the boot sector is the first file allocation table (FAT) structure. The FAT is used to determine the next cluster in a file, as well as determine which clusters are not currently being used. In a FAT12/16 file system, the FAT immediately follows the boot sector. In a FAT32 file system, there are reserved sectors in between. A backup FAT typically follows the first FAT.

After the backup FAT (or primary if no backup FAT exists) is the start of the Data Area. This is where the directory and file contents are stored. The layout of the Data Area is different between FAT12/16 and FAT32. Under a FAT12/16 file system, the sector after the last FAT is the start of the root directory, which has a fixed size. The first cluster follows after the root directory, and is given an address of 2 (there are no clusters 0 or 1). In contrast, with FAT32, cluster 2 starts in the sector following the last FAT. The FAT32 root directory can start anywhere in the Data Area. The Data Area itself extends to the end of the file system.[13]

File and directory content are stored in what are known as clusters. A cluster is a group of consecutive sectors. The first cluster is located dozens or hundreds of sectors after the

boot sector and all FAT contents. For simplicity and consistency, TSK itself does not use cluster addresses in its output. Rather, TSK uses the sector address to examine both the data in the FAT, as well as the file's contents.

Here is an example output of using `fsstat` against a FAT32 image file:

```
# fsstat -o 63 images/disk-1.dd

FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: FAT32

OEM Name: MSDOS5.0
Volume ID: 0x6c2e5cb8

Volume Label (Boot Sector): NO NAME
Volume Label (Root Directory): FAT-VOLUME

File System Type Label: FAT32

Next Free Sector (FS Info): 173952
Free Sector Count (FS Info): 61258528

Sectors before file system: 63

File System Layout (in sectors)
Total Range: 0 - 61432496
* Reserved: 0 - 33
** Boot Sector: 0
** FS Info Sector: 1
** Backup Boot Sector: 6

* FAT 0: 34 - 15024
* FAT 1: 15025 - 30015

* Data Area: 30016 - 61432496
** Cluster Area: 30016 - 61432479
*** Root Directory: 30016 - 30047
** Non-clustered: 61432480 - 61432496

METADATA INFORMATION
--------------------------------------------
Range: 2 - 982439426
Root Directory: 2

CONTENT INFORMATION
--------------------------------------------
Sector Size: 512
Cluster Size: 16384
Total Cluster Range: 2 - 1918828

FAT CONTENTS (in sectors)
--------------------------------------------
```

```
30016-30047 (32) -> EOF
[...]
30176-30303 (128) -> EOF
30304-30335 (32) -> EOF
30336-30367 (32) -> 85984
30368-30399 (32) -> EOF
[...]
85984-86015 (32) -> 133920
[...]
133920-133951 (32) -> 146304
[...]
```

This is quite lengthy, so lets break it down a bit.

```
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: FAT32
```

This tells us that TSK does indeed think this is a FAT32 image.

```
OEM Name: MSDOS5.0
Volume ID: 0x6c2e5cb8
```

The boot sector may contain several labels, each of which would have applied when the file system was created. The above OEM Name is typically based on which OS or application formatted the file system. The volume ID should be related to the time of creation, though this may not necessarily always be accurate.[11]

```
Volume Label (Boot Sector): NO NAME
Volume Label (Root Directory): FAT-VOLUME
```

A user has the option of assigning a volume label to a file system, where it can be subsequently stored in either of two places. One is in the boot sector, and the other is in the root directory. For example, Microsoft Windows XP might only store the label in the root directory, as shown above.

```
File System Type Label: FAT32
```

The boot sector also contains a label for the file system type, though this may not necessarily be accurate. In this particular example, it is correct, though the algorithm methodology used by TSK in the first line of output above should be considered as more reliable.

```
Next Free Sector (FS Info): 173952
Free Sector Count (FS Info): 61258528
```

The two values shown above are only found in a FAT32 file system. Converted to sector values, they tell us what the next cluster that can be allocated is, as well as how many free clusters exist.

```
Sectors before file system: 63
```

This shows us how many sectors exist prior to the start of the file system. In this case, the file system starts at sector 63 of the first partition.

```
File System Layout (in sectors)
Total Range: 0 - 61432496
* Reserved: 0 - 33
** Boot Sector: 0
** FS Info Sector: 1
** Backup Boot Sector: 6
```

In our example, we see that there are 61,432,496 in the file system. Additionally, sectors 0 thru 33 are in the reserved area. The original boot sector is in sector 0, and the backup is in sector 6. The asterisks above are used to that the information falls within the parent range. So, "**" are located within a single "*".

```
* FAT 0: 34 - 15024
* FAT 1: 15025 - 30015
```

The file system has a primary, as well as a backup FAT, and the sectors for each are given above.

```
* Data Area: 30016 - 61432496
** Cluster Area: 30016 - 61432479
*** Root Directory: 30016 - 30047
** Non-clustered: 61432480 - 61432496
```

The Data Area follows the second FAT. In this case, with FAT32, the first cluster starts in the first sector 30,016 of the Data Area. Note that the size of the Data Area is not a multiple of the cluster size. In this case, there are 17 sectors at the end of the Data Area that are not allocated to a cluster, as each cluster is 32 sectors. The Root Directory can be located anywhere in the FAT32 file system, and so its location is supplied in the above example output.

```
METADATA INFORMATION
------------------------------------------
Range: 2 - 982439426
Root Directory: 2
```

The FAT file system does not assign addresses to its meta-data structures, called data entries, so TSK uses its own addressing scheme. The maximum address is related to the total number of sectors in the file system. In the example above, the range falls from 2 to 982,439,426 and the root directory has been assigned an address of 2. These are the addresses that you would use with either the `icat` or `istat` tools.

```
CONTENT INFORMATION
------------------------------------------
Sector Size: 512
Cluster Size: 16384
```

```
Total Cluster Range: 2 - 1918828
```

The above example output continued contains general content related information. It shows us that the Sector Size is 512 bytes, as well as each cluster is 16KB in size. The total cluster range is also given, even though TSK shows all addresses in sectors.

```
FAT CONTENTS (in sectors)
------------------------------------------
30016-30047 (32) -> EOF
[...]
30176-30303 (128) -> EOF
30304-30335 (32) -> EOF
30336-30367 (32) -> 85984
30368-30399 (32) -> EOF
[...]
85984-86015 (32) -> 133920
[...]
133920-133951 (32) -> 146304
[...]
```

The above is an excerpt of much more output, and acts as a graphical representation of the primary FAT structure. Each line corresponds to a "cluster run". The FAT structure contains a pointer to the next cluster in a file.

File System Content List

When starting a forensic analysis of a mounted image with TSK, one of the first things you may wish to do is obtain a full and complete list of files on the system. This can be achieved by taking advantage of the `fls` utility. When running `fls`, you will probably want to direct it to start in a specific root directory, and recursively dig down into it (-r). Further, to output the metadata details (MAC times), you can also use the –m flag.

```
# ./fls -f linux-ext3 -r -m / /dev/hda8 >/evidence/fls-output.txt
```

In the example above, a Linux ext3 file system that is mounted at /dev/hda8 is being examined, and all metadata details are being retrieved as well, starting at the root. Further, the nice thing about running `fls`, versus some other standard Linux commands (`cat`, `less`, or `find`), is that the a-time of the files is kept intact. Running `cat` or `less` will alter the A-time of a file. Running `find` against a directory tree will alter the a-time of each directory, and depending on the directives supplied to `find`, potentially each file as well.[6]

You can also use `fls` against a single, or even split image files.[12] Here is an example:

```
# fls -i split -o 63 -f ntfs images/disk1.dd.01 images/disk1.dd.02
images/disk1.dd.03
```

The optional –i flag is used to specify the image file format, if known. If it is not given, the tool will try and guess the format used. The –o flag is used to specify the offset of

where the partition starts within the image. By default, this is in sectors. If you wish to use another format, you can also use offset@blocksize. For example, to specify the partition starts at block 1000 and each block is 2,048 bytes, then you would use 1000@2048.

When specifying split image files, the names must be sorted in proper order. Wildcards may also be used. The below example is similar to the one we used previously, except that we are using a wildcard, as well as allow auto-detect to work:

```
# fls -o 63 -f ntfs images/disk1.dd.*
```

A utility that can be run against the output of `fls` to obtain a timeline of file events is `mactime`.[6] An example of using mactime against the fls output file fls-output.txt:

```
# mactime -b /fls-output.txt > /evidence/timeline-complete.txt
```

You can apply `grep` against the `mactime` output file to search for specific file types, such as hidden files that begin with a "." or locate file activity around the same time as any suspicious activity that may have occurred against the media.

If you find a specific file that should be saved for later analysis, you can use the `icat` command to save the data located at that particular inode. For example, if you want to output data from inode 312 of a ext3 file system:

```
# ./icat -f linux-ext3 /dev/hda8 312 > /evidence/inode.312
```

`fls` can be used to view all deleted file names from an image. For example, this utility can be used to retrieve deleted file information from a FAT partition.[7] To obtain a recursive list of deleted files from an image:

```
# fls -rd images/hda8.dd | less
     d/d * 232: /TEMP-823450
     r/d * 293: /TEMP-131100
```

The "d/d" follows the format of directory entry value type value and the second letter is the type according to the inode. The "d" stands for directory, and the "r" stands for file. In most instances, these should match. However, in the case of deleted files where the inode has been reallocated, it will not. The asterisk "*" shows that it is deleted. The number following is the inode number. After this is the full path of the deleted files.

Success in actually recovering deleted files will vary between file system types. For NTFS, the file content may be recovered, depending on how much disk activity has occurred since the deletion. On ext3 or Solaris UFS, deleted files are much more difficult to recover.

If you want to retrieve deleted file information or orphaned files from an NTFS partition, you can use the `ifind` utility.[8] When using `ifind` with the –p option, you can tell `ifind`

to search unallocated entries that point back to a given MFT entry address. Here is an example:

```
# ifind -f ntfs -p 31 images/img.dd
-/r * 180:     FILE1.DAT
```

Further information on finding and recovering deleted files will be covered later.

Sorting Through the File Contents

Perhaps you are presented with the task of analyzing a large system image, and have no idea as to where to start? The overall hierarchy of organization of one's personal content, along with the naming conventions for file and folders are unique to each user. What might seem logical to one individual might seem arcane or overly complex to another. This can make it difficult for an investigator to quickly identify where a user has saved information. This is particularly true if the user has taken the extra step to try and hide the data. The user may have renamed file extensions, or buried files within unsuspecting directories.

To solve this problem, the `sorter` utility becomes useful. Unlike traditional tools that may display files in the normal directory hierarchy, `sorter` lists files based on their file type. As such, all images can be grouped together or all executables can be grouped together. If the investigator is working on a case in regards to the theft of company trade secrets, he or she may start by identifying Microsoft Office documents, spreadsheets, and text files first. This tool can be used to provide helpful lists of each, regardless of where they are saved. In addition, hash databases can used to flag files that are known to be bad or ignore files that are known to be good. As such, documents that came with Windows can be excluded from the document category.[2]

A basic example of using sorter:

```
# sorter -f ntfs -d save_dir -m C:/ images/hda1.dd

     Analyzing hda1.dd
       Loading Allocated File Listing
       Processing 8213 Allocated Files and Directories
       100%

       Loading Unallocated File Listing
       Processing 158 Unallocated meta-data structures
       100%

     All files have been saved to: save_dir
```

This command will process the hda1.dd image as an NTFS file system (-f), and save all output to the save_dir directory (-d). In addition, C:/ is appended to any path as it is printed (-m). This can be useful to demonstrate the path of each file as it would have appeared on the original system.

Inside of the output directory, one will find a "sorter.sum" file. This file is a summary of how many files were found in each category, as well as how many files were ignored because of a hash database match. Each category also gets its own corresponding .txt file written to the output directory. Categories included in the default setup of `sorter` are:

- archive
- audio
- compress
- crypto
- data
- disk
- documents
- exec
- images
- system
- text
- video

File types that belong to each category can and will vary, depending on the system being used. Categories can be fully customized from their default setup, though that is beyond the scope of this paper.

Two other useful flags to be aware of are the HTML flag (-h), and the Save flag (-s). The HTML flag will change the output of each corresponding category file from flat ASCII to HTML. The Save flag will place a copy of each file in a category sub-directory. If both flags are given, thumbnails are also created of any files that fall into the images category. A thumbs-[counter].html file or files will also now be included in the images category directory, which will include 100 thumbnails per a file. This is extremely useful when one wants to quickly peruse the found images for any illegal content.

We have learned a great way to come up with an organized content list of the files within the subject image. The next step would be to reduce the number of result files to sort through by using what are known as hash databases. A hash database is a list of known files. The National Software Reference Library (NSRL) contains hashes of known good files, including operating systems and off the shelf software.[3] NSRL .ISO images are available for download at ftp://ftp.nist.gov/pub/itl/div897/nsrl. Any file that has a matching hash in a database used by sorter will be ignored and not added to any category file. Instead, they will be documented in an "exclude.txt" file for future reference. Before a known good file is simply ignored and documented in the "exclude.txt" file, the extension of the file will be checked and, if a mismatch is found, added to a file called "mismatch_exclude.txt".

Any custom hash databases, as well as the NSRL database, must be indexed prior to usage. The tool to index a database is the `hfind` utility. `hfind` will be covered in more detail later when we cover Hash Databases.

To take advantage of leveraging a hash database in our use of `sorter`, one can use the NSRL flag (-n), or the custom hash database flag (-x). Another flag is the known bad file flag (-a), which allows you to use a hash database of known bad files. Any file hash match with the known bad files list will be added to an "alert.txt" file.

Here is an example of using all of these hash flags:

```
# sorter -f ntfs -d save_dir -h -s -n /usr/local/nsrl/NSRLFile.txt -x
/usr/local/forensics/hash/win2k.txt -a
/usr/local/forensics/hash/porn.txt -m "C:/" images/hda1.dd
```

The above example will process the hda1.dd image as an NTFS file system (-f), and as we learned previously, a C:/ is appended to any path outputted with the –m flag. All data is saved in the save_dir folder thanks to the –d flag, and not only is output in HTML format (-h), but a copy of each file is saved (-s) as well. Files in the NSRL (-n), as well as files in the custom hash database (-x) are ignored in order to help reduce the results. However, any file found in the known bad file database (-x) will be flagged and listed in the "alert.txt" file.

A final flag to be aware of with `sorter` is the –l flag. This tells sorter to write nothing to disk, and instead output all findings to STDOUT. This can be used to direct output to another machine on the network during an initial incident response. In the example below, a machine is listening at 172.16.5.81 on port 8888:

```
# sorter -f linux-ext2 -l /dev/hda1 | nc -w 10 172.16.5.81 8888
```

Hash Databases

Now, lets take a step back and learn more about the utility that creates the hash databases `sorter` can use, `hfind`.[5] `hfind` basically allows the user to look up hashes for NIST NSRL, Hashkeeper, as well as MD5sum hash databases. Prior to any lookups being performed, the database must be indexed by running `hfind` with the index flag (-i), followed by the index type. The type may be nsrl-md5 (NSRL database indexed with MD5 values), nsrl-sha1 (NSRL database indexed with SHA-1 values), md5sum (a database created by the md5sum tool), or hk (HashKeeper database indexed with MD5 values).

For example, to index an MD5 NSRL database, the following command may be used:

```
# hfind -i nsrl-md5 /usr/local/nsrl/NSRLFile.txt
```

Once the database has been indexed, you can look up any value. An example of looking a value up:

```
# hfind /usr/local/nsrl/NSRLFile.txt 917c4a96fc6bd589fe522765391077e8
    917c4a96fc6bd589fe522765391077e8        Hash Not Found
```

Note that you do not need to specify any database types when performing your lookup. You may also try and pipe a value in:

```
# md5sum unknown.dat | hfind /usr/local/nsrl/NSRLFile.txt
    F99AC9C11F6D745310F53A25D46BE551        MTRUSHMO.WMF
```

Another command line flag that can be used is to provide a file list of hashes (-f). Here is an example of a text file called hashes.txt that has one hash per a line, with a total of two hashes:

```
# hfind -f hashes.txt /usr/local/nsrl/NSRLFile.txt
    917c4a96fc6bd589fe522765391077e8        Hash Not Found
    F99AC9C11F6D745310F53A25D46BE551        MTRUSHMO.WMF
```

Searching for Binary Data

A useful tool that can be used to search a file for a binary value is the `sigfind` utility. This utility allows one to search for a supplied data structure, file header, or other type of low-level value.[9]

`sigfind` will read the supplied input file, in block size chunks, and look for the specified signature at the given offset. If a match is found, `sigfind` will print the sector that the match was found, as well as the distance between it and the previous hit. The distance can be useful in situations where the looking for data structures that may be stored in a pattern (i.e. every 1024 bytes).

In this example, `sigfind` is used to search for 0x55a, which both FAT and NTFS use as the last two bytes in their relative 512-byte boot sectors.

```
# sigfind -o 510 -b 512 55AA images/fat32.dd
Block size: 512  Offset: 510  Signature: 55AA
Block: 0 (-)
Block: 1 (+1)
Block: 2 (+1)
Block: 6 (+4)
Block: 7 (+1)
Block: 8 (+1)
Block: 12 (+4)
[...]
```

In the above example, we see that a match is found in sectors 0, 1, 2, 6, 7, and 8. The sectors shown have the value in bytes 510 and 511 in each 512-byte sector.

Various templates exist for the differences between different file systems. Such included templates include DOS, EXT2/3 superblocks, FAT boot sector, NTFS boot sector, and more. Simply use `sigfind` with the `-t` flag to pass along a template.

File System Journal Tools

Some file systems, such as NTFS and ext3, have what is known as a journal (ext3 is actually ext2 with journaling capabilities added). A journal is responsible for recording metadata (and sometimes content) updates that are made against the file system. Tools can take advantage of this information and potentially recover recently deleted data.

One utility that can be used to list the contents of a file system journal is the `jls` utility. `jls` lists the records and entries in a file system journal. If an inode is given, then it will look at the supplied inode for a journal. If not, it will use the default location. The output from jls will list both the journal block number, as well as the description. For example:

```
# jls -f linux-ext3 images/img.dd
```

Note the use of the –f flag, which specifies the file system type. The optional –o flag could be used to specify the sector offset where the file system starts in the image. Non-512 byte sectors can be specified in the offset@blocksize format, such as 32@2048.

The `jcat` utility can then be used to show the contents of a block in the file system journal. The inode address of the journal may be provided. If no address is given, the default location will be used. The block address given is a journal block address, and not a file system block. For example:

```
# jcat -f linux-ext3 images/img.dd 34 | xxd
```

The –f flag specifies the file system type. In the example above, 34 is the inode. As with `jls`, you can also optionally use the –o flag.

Metadata & inodes

When encountering the situation where further information about a particular inode is desired, look no further than the `istat` utility. Here is an example:

```
# istat images/wd0e.dd 493
     inode: 493
     Not Allocated
     uid / gid: 1000 / 1000
     mode: rw-------
     size: 92
     num of links: 1
     Modified: 08.10.2001 17:09:49 (GMT+0)
     Accessed: 08.10.2001 17:09:58 (GMT+0)
     Changed: 08.10.2001 17:09:49 (GMT+0)
     Direct Blocks:
     59382
```

From the output, we can see the inode number of 493. We can also see that this inode is currently not allocated, which means the file has been deleted. Further, we can see the UID and GID associated with the inode, as well as the rwx security settings. MAC time

information is also available. The direct block is the actual fragment associated with the inode. If you wish to see what the original file associated with this inode was, we can take advantage of the `ffind` utility as seen in this example:

```
# ffind -a images/wd0e.dd 493
     * /dev/.123456
```

The leading asterisk "*" tells us that the file has been deleted. However, at one point, the full path and file of "/dev/.123456" was allocated inode 493, which, when combined with the previous information from `istat`, was allocated fragment 59382.

If you wish to confirm the allocated fragment, you can use the `ifind` utility with the –a flag, telling it to find all occurrences:

```
# ifind -a images/wd0e.dd 59382
     493
```

Remember, `ifind` will search for the meta-data structure that has allocated a given disk unit, or a given filename. An example of searching for a filename:

```
# ifind -f linux-ext2 -n "/etc/" images/wd0e.dd
```

If you would like to search an NTFS partition for unallocated MFT entries for a supplied parent inode, you can use the –p flag. Here is an example:

```
# ifind -f ntfs -p 5 -l -z EST5EDT images/ntfs-wd0e.dd
```

The –p searches the unallocated MFT entries for a parent inode of 5. Additionally, the –l flag says to provide details, and the –z flag will automatically set the time zone to correct times.

Once you find a particular inode of interest, you may wish to copy this file data. The `icat` utility can be used in such a situation. For example, if you want to output data from inode 493 from the image:

```
# icat –f linux-ext3 images/wd0e.dd 493 > /evidence/inode.493
```

Using the `ils` utility, information associated with unallocated metadata can be obtained. By default, `ils` lists only the inodes of deleted files. When a file is deleted, the time associated with the file is typically updated. In the case of FAT partitions, the time zone must be provided. Under many conditions it will not be possible to link the original name to the metadata, though it will still provide some idea as to when activity occurred. An example of using `ils` on an OpenBSD file system:

```
# ils -f openbsd -m images/root.dd >> data/body
```

Content & Data Layer

A disk image may include unallocated disk space. The `dls` utility can be used to extract unallocated disk units from such an image. For example:

```
# dls images/wd0e.dd > /evidence/wd0e.dls
```

After using your preferred UNIX utilities to search the extracted image (note that this extracted image is not a real file system), you can use the `dcalc` utility to return the address in the original image of whatever address you give it from your `dls` generated image. For example, lets use the UNIX strings utility to extract text from our dls image, and then search for the term “abcdefg”.

```
# strings -t d evidence/wd0e.dls > /evidence/wd0e.dls.str
# grep "abcdefg" evidence/wd0e.dls.str | less
     10389739: abcdefg
```

The above tells us at the string is located at byte 10389739. Next, lets determine the fragment size of the original file system image this was extracted from with the `fsstat` utility.

```
# fsstat images/wd0e.dd
     <...>
     CONTENT-DATA INFORMATION
     ------------------------------------------
     Fragment Range: 0 - 266079
     Block Size: 8192
     Fragment Size: 1024
```

This shows us that the fragment size is 1024 bytes. 10389739/1024=10146 (roughly). This means that the string “abcdefg” is located in fragment 10146 of the `dls` extracted image. To view the full fragment from the `dls` image, we can use `dd` against the original image to extract this spot.

```
# dd if=images/wd0e.dd bs=1024 skip=10146 count=1 | less
```

Next, lets use the `dcalc` utility to convert the address.

```
# dcalc -u 10146 images/wd0e.dd
     59382
```

Here, we learn that the string “abcdefg” is located at fragment 59382 of the original image. To view the statistics of a the data unit, including allocation status and group number, one can use the `dstat` utility.

```
# dstat images/wd0e.dd 59382
Sector: 59382
Allocated (Meta)
```

To view the contents of this particular fragment, the `dcat` utility can be used.

```
# dcat images/wd0e.dd 59382 | less
```

Note that this may return a bunch of binary data that appears as gibberish, or readable text depending on whatever data is written at the address indicated.

Conclusion

We have learned a bit about the basic tools that make up TSK in this paper. TSK certainly is not the complete end-all solution to performing any forensic analysis. No tool is. It does, however, provide you with a wealth of potential capabilities that may or may not necessarily be found in other packages. Furthermore, the ability to use these utilities from the command line directly against a file provide a mechanism of "raw access and feedback" against data not necessarily found with other applications. If this raw access is to daunting or complicated, the web browser-based Autopsy program may be used to achieve many of these same results. As a screwdriver and hammer has its place in every person's toolbox, TSK certainly has a place in every computer forensic investigators' arsenal of low-level system utilities.

Bibliography

1. Carrier, Brian. (2003, February). *A High Level Overview of Autopsy and TASK* in The Sleuth Kit Informer. Retrieved from http://www.sleuthkit.org/informer/sleuthkit-informer-1.html
2. Carrier, Brian. (2003, April). *Sorting Out the Sorter* in The Sleuth Kit Informer. Retrieved from http://www.sleuthkit.org/informer/sleuthkit-informer-3.html
3. http://www.nsrl.nist.gov
4. http://www.blackhat.com/presentations/bh-usa-03/bh-us-03-willis-c/bh-us-03-willis.pdf
5. Carrier, Brian. (2003, August). *Finding Hashes with hfind* in The Sleuth Kit Informer. Retrieved from http://www.sleuthkit.org/informer/sleuthkit-informer-7.txt
6. Carrier, Brian. (2003, November). *UNIX Incident Verification with The Sleuth Kit* in The Sleuth Kit Informer. Retrieved from http://www.sleuthkit.org/informer/sleuthkit-informer-10.txt
7. Carrier, Brian. (2004, May). *TSK FAT File Recovery* in The Sleuth Kit Informer. Retrieved from http://www.sleuthkit.org/informer/sleuthkit-informer-14.txt
8. Carrier, Brian. (2004, September). *NTFS Orphan Files* in The Sleuth Kit Informer. Retrieved from http://www.sleuthkit.org/informer/sleuthkit-informer-16.txt
9. Carrier, Brian. (2004, November). *Detecting Host Protected Areas (HPA) in Linux* in The Sleuth Kit Informer. Retrieved from http://www.sleuthkit.org/informer/sleuthkit-informer-17.txt
10. Carrier, Brian. (2005, January). *Description of the FAT fsstat Output* in The Sleuth Kit Informer. Retrieved from http://www.sleuthkit.org/informer/sleuthkit-informer-18.txt
11. Craig Wilson. (2005, June). *Volume Serial Numbers & Format Verification Date/Time*. Digital Detective White Paper. Retrieved from http://www.digital-detective.co.uk/documents/Volume%20Serial%20Numbers.pdf
12. Carrier, Brian. (2005, March). *New Image File Support* in The Sleuth Kit Informer. Retrieved from http://www.sleuthkit.org/informer/sleuthkit-informer-19.txt
13. *FAT: General Overview of On-Disk Format* in Microsoft Extensible Firmware Initiative Hardware White Paper. Retrieved from http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx.