Digital Forensics

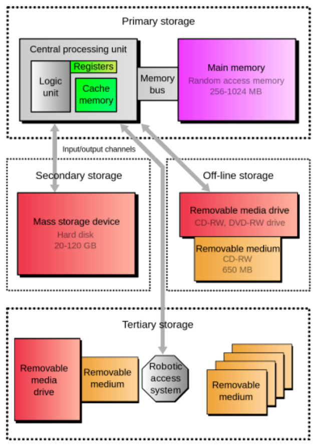
Federico Conti

2024/25

Contents

File Systems

In a computer, data storage is organized in a hierarchical manner.



At the top, we have fast and directly accessible storage, such as registers and cache, while at the bottom, we have slower but larger storage, like hard drives or SSDs.

We focus on secondary (external) memory storage:

- not directly accessible by CPU –> data must be transferred to main memory (RAM) before it can be processed
- data transferred in blocks –> rather than individual bytes
- significantly slower
- non-volatile –> retains data even when power is turned off

Floppies/HDs/CDs/DVDs/BDs/SD-cards/SSDs/pendrives. . . are all block devices; following File System Forensic Analysis's terminology.

A volume is a collection of addressable blocks

- these blocks do not need to be contiguous on a physical device
- a volume can be assembled by merging smaller volumes

A partition is a contiguous part of a volume

- partitioning is optional: some removable storage does not use it

By definition, both disks and partitions are volumes. In this part of the course we deal with block-device (forensics) images, like the one acquired from actual devices.

Users don't interact directly with storage blocks—instead, they work with files and directories.

The file system creates this illusion; i.e., it handles the mapping between files/directories and a collection of blocks (usually, clusters of sectors). Consists of on-disk data structures to organize both data and metadata. There exist various file-system formats (e.g., FAT, NTFS, . . .)

(high-level) formatting a volume means to initialize those structures.

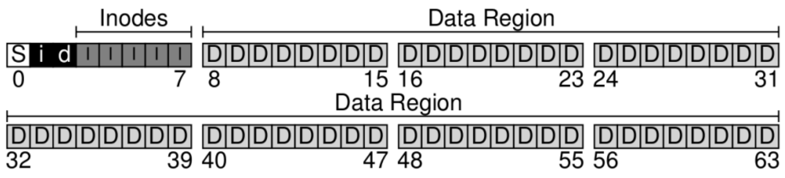
VSFS (Very Simple File-System)

In Unix-like file systems (e.g., EXT4, see `man mkfs.ext4`), each file (or other filesystem object) has an associated inode that stores its metadata. However, inode does not store file names, which are instead kept in directory structures.

Every file system object has an inode, including:

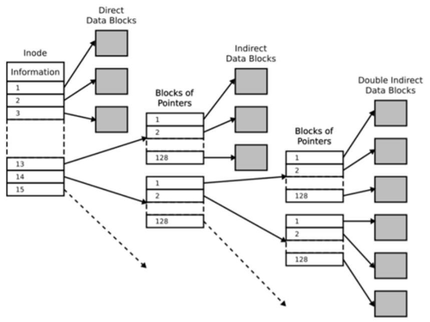
- regular file
- directory
- symbolic link
- FIFO
- socket
- character device
- block device

Formatting means preparing: the superblock, i-node/data bitmaps, i-node table, data region.



An inode contains:

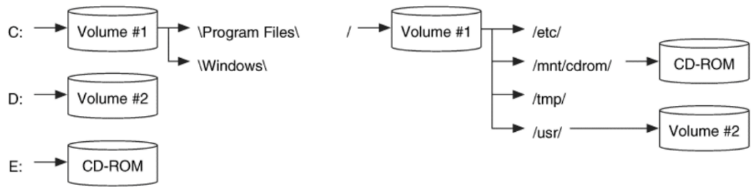
- file type
- UID/GID/permission-bits
- time information
- size in bytes
- number of hard links (AKA names)
- pointers to data blocks



To use the end-user view, a file system, stored on a **block device**, must be mounted (or "parsed")

Most modern operating systems automatically mount external storage devices when they are connected.

- In Unix/Linux there is a single root directory (/), and additional volumes are mounted within this hierarchy
- in Windows each volume (storage device/partition) is assigned a drive letter (C:, D:, E:)



Block devices can be seen as "files" themselves

- Linux special files, typically under /dev
  - various "aliases" in /dev/disk –> /by-id; /by-uuid; /bypath
  - `lsblk` lists information about available block devices

Viceversa, (image) files can be seen as block devices.

1. `losetup` command allows an image file to be treated as a virtual block device (loop device).
   - `--list`
   - `--find [--show] [--partscan] image`
   - `--detach[-all]`
2. Then, we can `mount` them.
   - Instead of manually setting up a loop device, mount can automatically create one:
   - `offset=<byte_offset>` starting point within an image file (use `fdisk -l image_file.img`)
   - `mount [-o loop] image` instead of manually setting up a loop device
   - `ro` read-only

3. Umount and check umount /dev/sda1 fsck /dev/sda1

Example

```
xz -dk two-partitions.dd.xz

# FIRST METHOD

losetup -r -o $((1*512)) /dev/loop0 two-partitions.dd  # First partition
losetup -r -o $((1026*512)) /dev/loop1 two-partitions.dd  # Second partition

# losetup -r -o $((1*512)) --find --show /tmp/two-partitions.dd
# losetup -r -o $((1026*512)) --find --show /tmp/two-partitions.dd

fdisk -l /dev/loop1 # check

mount -o ro /dev/loop0 /mnt/two-partition
mount -o ro /dev/loop1 /mnt/two-partition

# SECOND METHOD

losetup -r --find --show --partscan two-partitions.dd

mount -o ro /dev/loop0p1 /mnt/part1
mount -o ro /dev/loop0p2 /mnt/part2

umount /dev/loop0p1
umount /dev/loop0p1
```

The Sleuth Kit (TSK)

The Sleuth Kit (TSK) is a forensic toolkit that provides different layers of analysis for digital investigations. Each layer focuses on specific aspects of a digital storage system, allowing forensic examiners to extract and interpret data at various levels.

- `img_` for images
- `mm` (media-management) for volumes
- `fs` for file-system structures
- `j` for file-system journals
- `blk` for blocks/data-units
- `i` for inodes, the file metadata
- `f` for file names

Typically followed by:

- `stat` for general information
- `ls` for listing the content
- `cat` for dumping/extracting the content

Example

```
img_stat two-partitions.dd
img_cat two-partitions.dd
```

```
img_stat canon-sd-card.e01
```

When analyzing file systems, we categorize data into essential and non-essential based on their reliability and importance.

- Essential Data = Trustworthy & required for file retrieval.
  - If name or location were incorrect, then the content could not be read
- Non-Essential Data = Can be misleading & needs verification.
  - the last-access time or the data of a deleted file could be correct but we don't know

The Volume (or Media Management) layer in The Sleuth Kit (TSK) focuses on analyzing and managing disk partitions. This layer is crucial for identifying partition structures, extracting partitions, and verifying file system integrity.

- `mmstat image` displays the type of partition scheme
- `mmls image` displays the partition layout of a volume
- `mmcat image part_num` outputs the contents of a partition

Example

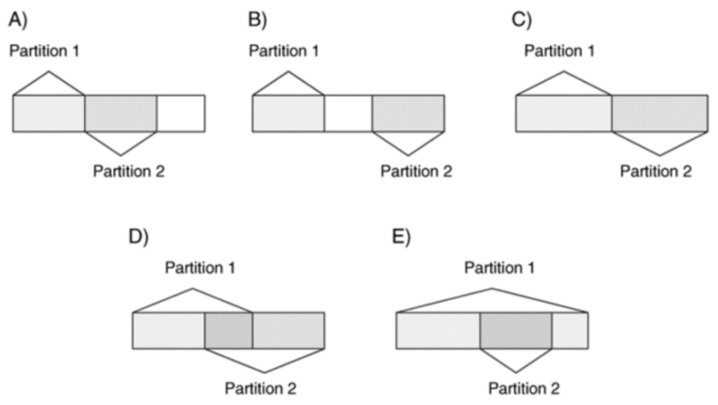
```
mmstat canon-sd-card.e01

mmls canon-sd-card.e01

##OUT##
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
001:  -------   0000000000   0000000050   0000000051   Unallocated
002:  000:000   0000000051   0000060799   0000060749   DOS FAT16 (0x04)
###

mmcat canon-sd-card.e01 2 > canon-sd-card.img
dd if=canon-sd-card.e01 of=canon-sd-card.dd bs=512 skip=51

sha256sum  canon-sd-card.img  canon-sd-card.dd # ARE DIFFERENT !!!!!!!!!
```

DOS (or MBR) partition tables

The concept of MBR was introduced in 1983 with PC DOS 2.0.

It contain:

- machine code for the boot loader, which usually loads and executes the active-partition Volume Boot Record
- a 32-bit unique identifier for the disk, located at offset 440 (0x1B8).
- information on how the disk is partitioned –> four 16-byte entries (each at offset 446 (0x1BE)), allowing up to four primary partitions.
- last two bytes of the MBR contain the signature bytes: 0x55 0xAA.



1. Valid Configurations (A, B, and C):
    - These configurations ensure that partitions are either adjacent or properly aligned without overlap.
    - Partitions are defined in a way that does not create ambiguity in data storage.
2. Invalid Configurations (D and E):
    - D and E depict overlapping partitions, which is problematic.
    - Overlapping partitions may cause data corruption, boot issues, or system conflicts because two partitions would claim the same disk space.

CHS (Cylinder-Head-Sector) is the early method for addressing physical blocks on a disk.

It used a 3-byte structure:

- 10 bits for Cylinders (tracks stacked vertically)
- 8 bits for Heads (read/write heads on a disk platter)
- 6 bits for Sectors (sections of a track)

Replaced by Logical Block Addressing in ’90s.

- To convert you need to know the number of heads per cylinder, and sectors per track, as reported by the disk drive
- Yet, many tools still aligned partitions to cylinder boundaries

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	Master Boot Code															
….																
1A0																
1B0									Disk Signature						Boot ind[1]	Start head
1C0	start Sect[2]	start Cyl[2]	Sys ID[3]	End Head	End sect[2]	End Cyl[2]	Relative Sectors				Total Sectors					
1D0																
1E0																
1F0															55	AA

1. Boot indicator 0x00 = non-boot, 0x80 = bootable
2. Starting sector & starting cylinder are allocated bits, not bytes (0x1C0-0x1C1) same goes for end head and end sector

BIT	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Value	Starting sector							Starting Cylinder								

3. Common partition values.

0x01	FAT12 <32MB
0x04	FAT16 <32MB
0x05	MS Extended partition using CHS
0x06	FAT16B
0x07	NTFS, HPFS, exFAT
0x0B	FAT32 CHS
0x0C	FAT32 LBA
0x0E	FAT16 LBA
0x0F	MS Extended partition LBA
0x42	Windows Dynamic volume
0x82	Linux swap
0x83	Linux

0x84	**Windows hibernation partition**
0x85	Linux extended
0x8E	Linux LVM
0xA5	FreeBSD slice
0xA6	OpenBSD slice
0xAB	Mac OS X boot
0xAF	HFS, HFS+
0xEE	MS GPT
0xEF	Intel EFI
0xFB	VMware VMFS
0xFC	VMware swap

A Master Boot Record (MBR) is typically 512 bytes and laid out like this:

```
Offset (hex) | Size | Description
---------------------------------------------
0x000        | 446  | Bootstrap code area
0x1B8        | 4    | Disk signature (sometimes called "unique MBR signature")
0x1BC        | 2    | Usually 0x0000 or may be used for copy-protection, etc.
0x1BE        | 16   | Partition entry #1
0x1CE        | 16   | Partition entry #2
0x1DE        | 16   | Partition entry #3
0x1EE        | 16   | Partition entry #4
0x1FE        | 2    | MBR signature (0x55AA)
```

Each 16-byte partition entry has the structure:

```
Byte | Description
----------------------------------------
0    | Boot indicator (0x80 = bootable; 0x00 = non-bootable)
1-3  | Starting CHS (Head-Sector-Cylinder) - often unused in modern disks
4    | Partition type (ID)
5-7  | Ending CHS (Head-Sector-Cylinder)
8-11 | Relative sectors (start in LBA)
12-15| Total sectors in this partition
```

Example

Use ImHex, writing proper patterns, to extract disk and partition information from mbr{1,2,3}.dd. Then, answer the following questions:

1. What are the three disk signatures?
2. Is there any MBR with inconsistent partitioning?
3. Are there MBRs without bootable partitions?

4. What is the largest FAT (id=4) partition?
5. Are CHS information always present?

(fdisk -l mbr2.dd)

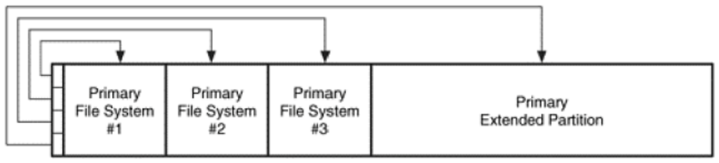
```
tar -tJf MBR123_and_GPT.tar.xz
tar -xJvf MBR123_and_GPT.tar.xz mbr1.dd

xxd -s 0x1B8 -l 4 mbr1.dd
```

Pattern editor

```
#include <std/mem.pat>
struct PartitionEntry {
  u8  bootIndicator;
  u8  startCHS[3];
  u8  partitionType;
  u8 endCHS[3];
  u32 relativeSectors;
  u32 totalSectors;
};

struct MBR {
  u8 bootCode[0x1B8];        // 446 bytes
  u32 diskSignature;         // offset 0x1B8
  u16 reserved;              // offset 0x1BC (often 0x0000)
  PartitionEntry partitions[4];  // 4 partition entries, each 16 bytes
  u16 signature;             // offset 0x1FE, should be 0x55AA
};

 MBR seg[while(!std::mem::eof())] @ 0x00;
```

Extended partitions

MBR has only 4 slots for primary partitions.



To work around this limitation, one slot can be used for the primary extended partition, a partition containing other partitions.

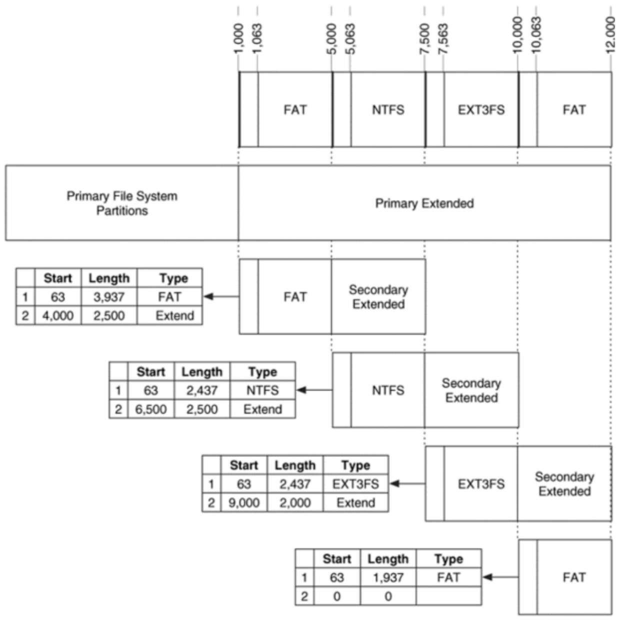
Beware of logical-partition addressing, which uses the distance from the beginning of a partition (vs the physical-addressing, from the beginning of the whole disk).

Inside the primary extended partition we find secondary extended partitions, containing

- a partition table t (with the same 512-byte structure)
- a secondary file-system partition p (logical partition), which contains a FS or other data

The partition table (t) describes:

1. Location of p (logical partition) relative to t.
2. Next secondary extended partition (if any), w.r.t. the primary extended partition



Example

ext-partitions.dd (SHA256: b075ed83211…) contains three partition tables: one primary, two extended. Analyze them with ImHex, and compare the result w.r.t. fdisk/mmls Source: (https://dftt.sourceforge.net/test1/index.html)

```
 mmls ext-partitions.dd

      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
001:  -------   0000000000   0000000062   0000000063   Unallocated
002:  000:000   0000000063   0000052415   0000052353   DOS FAT16 (0x04)
003:  000:001   0000052416   0000104831   0000052416   DOS FAT16 (0x04)
004:  000:002   0000104832   0000157247   0000052416   DOS FAT16 (0x04)
005:  Meta      0000157248   0000312479   0000155232   DOS Extended (0x05) #15724*512 = address
006:  Meta      0000157248   0000157248   0000000001   Extended Table (#1)
007:  -------   0000157248   0000157310   0000000063   Unallocated
008:  001:000   0000157311   0000209663   0000052353   DOS FAT16 (0x04)
009:  -------   0000209664   0000209726   0000000063   Unallocated
010:  001:001   0000209727   0000262079   0000052353   DOS FAT16 (0x04)
011:  Meta      0000262080   0000312479   0000050400   DOS Extended (0x05)
012:  Meta      0000262080   0000262080   0000000001   Extended Table (#2)
013:  -------   0000262080   0000262142   0000000063   Unallocated
014:  002:000   0000262143   0000312479   0000050337   DOS FAT16 (0x06)
```

GPT - GUID PArtition Tables

A Universally/Globally Unique IDentifier (UUID/GUID) is a 128-bit label.

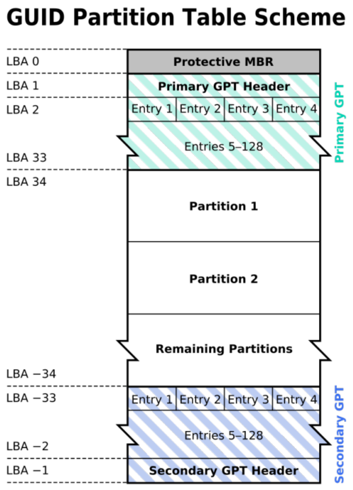
- Uniqueness: Properly generated UUIDs are statistically unique, meaning the probability of duplication is extremely low.
- Standard Format: UUIDs are typically written in a 32-character hexadecimal format divided into five groups: 8-4-4-4-12, separated by hyphens.

```
uuidgen
```

```
bdeec955-b1b8-44a2-8034-15507d431aca
```

The GPT format, used by the Extensible Firmware Interface (EFI), which replaced BIOS, is the current standard on PCs; it

- starts with a protective MBR
- supports up to 128 partitions
- uses 64-bit LBA addresses
- keeps "mirrored" backup copies of
- important data structures



Example

Use ImHex, writing proper patterns, to extract disk and partition information from gpt.dd. Then, answer the following questions:

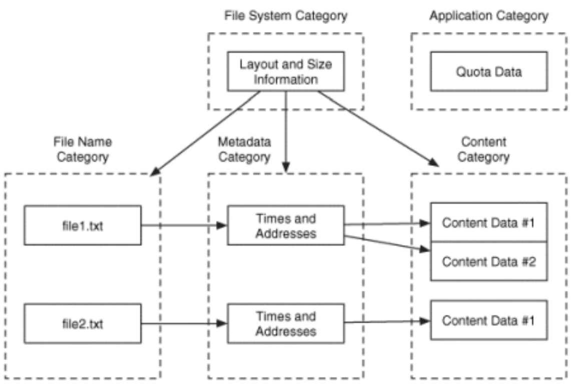
1. What is the disk GUID?
2. How many partitions are there?
3. What are the partition names?
4. Can you find the partition type GUIDs in the previous table?

```
gdisk -l gpt.dd
mmls -t gpt gpt.dd
```

File System Analysis

A reference model for a file system based on different categories of data that are involved in file storage and management.

- File System Category: layout and size information about the entire file system, such as: file system parameters (e.g., block size, total size) the structure or mapping of data storage
- Content The actual data, stored in clusters/blocks/data-units
- MetaData: Data that describes files: size, creation date
- File Name Data that assign names to files
- Applliccation: Data not needed for reading/writing a file; e.g., user quota statistics or a FS journal



To get the general details of a file-system - `fsstat [-o sect_offs] image`

Example

1. Find the OEM Name and Volume Label (Boot Sector) in canon-sd-card.e01

```
mmls canon-sd-card.e01

##OUT##
      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
001:  -------   0000000000   0000000050   0000000051   Unallocated
002:  000:000   0000000051   0000060799   0000060749   DOS FAT16 (0x04)
###

fsstat -o 51 canon-sd-card.e01
```

2. Check whether the partition types are correctly set inside two-partitions.dd

```
mmls two-partitions.dd

##OUT##
      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
001:  -------   0000000000   0000000000   0000000001   Unallocated
002:  000:000   0000000001   0000001025   0000001025   DOS FAT16 (0x06) # wrong
003:  000:001   0000001026   0000002047   0000001022   DOS FAT12 (0x01)
###

fsstat -o 1 two-partitions.dd # FAT12
fsstat -o 1026 two-partitions.dd # FAT12
```

Example

inside the image file two-partitions.dd

1. look for the strings

- “didattica”
- “wDeek”"
- "tool

- “secret”

```
strings two-partitions.dd | grep -E "didattica|wDeek|tool|secret" # secret,wDeek,didattica
or
strings two-partitions.dd | ag "didattica|wDeek|tool|secret"
or
xxd -g1 two-partitions.dd| grep -C 3 ecre
```

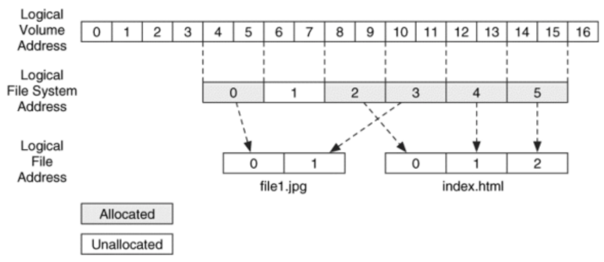
2. (ro) mount its partitions, and look for the same strings inside the contained files

```
losetup -r --find --show --partscan two-partitions.dd

mount -o ro /dev/loop0p1 /mnt/part1
mount -o ro /dev/loop0p2 /mnt/part2

grep -rE "didattica|wDeek|tool|secret" /mnt/part1 # null
grep -rE "didattica|wDeek|tool|secret" /mnt/part2 # tool
```

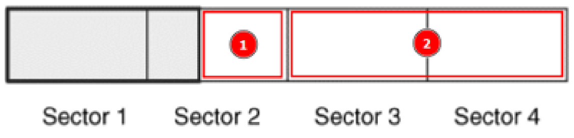
Do some string appear only in one of the two searches? Can you guess why?

Each sector can have multiple addresses, relative to the start of the…

- storage media: physical address
- volume: (logical) volume address
- FS [data area]: (logical) FS address AKA (logical) cluster number
- file: (logical) file address AKA virtual cluster numbers



When writing a 612-byte file in a file system with 2K clusters (where each sector is 512 bytes), the way data is allocated creates slack space—unused but allocated storage that may contain remnants of previous data.



When investigating deleted files, forensic analysts use two major approaches:

1. Metadata-based

   If the file is deleted but metadata still exists, we can recover:

   - File size, timestamps, and allocated sectors/clusters.
   - Orphaned files (files with no full path reference)

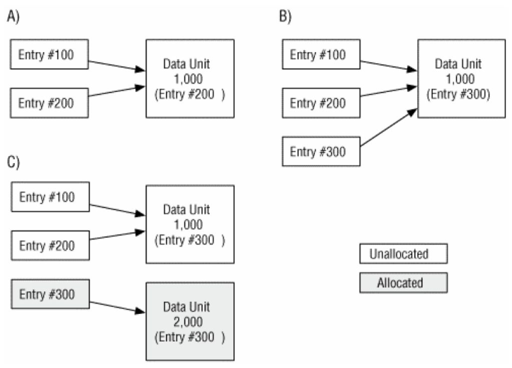
2. Application-Based

Used when metadata is unavailable:

```
- Typically from un-allocated space
- Does not need any FS information
```

Example

Where do data-units come from?


A)
Entry #100
Entry #200
Data Unit 1,000 (Entry #200)
B)
Entry #100
Entry #200
Entry #300
Data Unit 1,000 (Entry #300)
C)
Entry #100
Entry #200
Data Unit 1,000 (Entry #300)
Entry #300
Data Unit 2,000 (Entry #300)
Unallocated
Allocated

A.

- Entry#100 initially points to Data Unit 1,000.
- Entry #200 is created after #100 is deleted and reuses the same data unit.

This means that even after deletion, old data might still be recoverable unless overwritten.

C.

- Entry #300 is now assigned a completely new Data Unit (2,000).
- Entry #100 and #200 had used Data Unit 1,000, but it is now unallocated.

The original content in Data Unit 1,000 might still be present but no longer linked to any active file. (Carving).

TSK metadata commands

The Sleuth Kit (TSK) provides powerful commands to analyze file system metadata, particularly focusing on inodes, which store key file attributes.

1. `ils [-o sect_offs] image` - list inode information
   - `-r` → Lists only removed (deleted) files
   - `-a` → Lists only allocated (active) files
   - `-m` → Displays inode details in a format compatible with mactime (used for timeline analysis)
2. `istat [-o sect_offs] image inum`- dumps detailed metadata of a specific file or inode

!!! TSK uses the inode abstraction even for file systems that do not natively have them.

- Some file systems (e.g., FAT32) do not have inodes, but TSK emulates them to allow a consistent analysis approach

3. `ifind [-n filename] [-d data-unit] [-o offset] image` - viceversa, to find the inode corresponding to a data-unit or file name
   - `strings -t d disk-image.dd | grep "password"`- gives you an offset (e.g., 123456) where the data appears.
   - `ifind -d $((123456/4096)) disk-image.dd`- $((n/block-size)) returns the inode number
4. `ffind [-o sect_offs] image inum` - lists the names using the inode (useful when names are not inside the "inode")
5. `icat [-o sect_offs] image inum` - extracts and displays the contents of a file based on its inode number.
   - `-s` → Includes slack space (unused space in the last cluster of a file)
   - `-r` → Attempts to recover deleted files

   Note: deleted content may be present in unallocated data (without metadata pointing to it). To check/dump blocks:

    - `blkstat image block` - displays metadata about a specific block (e.g., allocation status, timestamps, etc.)
    - `blkcat image block [how-many-blocks]` - outputs the raw content of a specific block
    - `blkls` - lists or outputs blocks too

6. `fls [-o sect_offs] image [inum]` - list files inside the directory corresponding to the inode number
7. `ffind [-o sect_offs] image inum` - lists the names using the inode (useful when names are not inside the "inode")

Example

Let's find out why some of the following strings appear in one search and not the other

```
mmls two-partitions.dd

##OUT##
      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
001:  -------   0000000000   0000000000   0000000001   Unallocated
002:  000:000   0000000001   0000001025   0000001025   DOS FAT16 (0x06)
003:  000:001   0000001026   0000002047   0000001022   DOS FAT12 (0x01)
###


strings -t d two-partitions.dd | grep -E "didattica|wDeek|tool|secret"

##OUT##
  20514 and I have a secret message ;)
 547396 wDeek
 547436 /home/gio/didattica/file-systems/vol_fs_analysis/examples/pp/test
###
```

- End of first partition = (1025*512) = 524800 –> "secret" is in the firts partition.
- Start of second partition = (1025*512) = 525312 –> "wDeek" is int the second partition.
- End of second partition = (1025*512) = 1048064 —> "didattica" is in the seconf

1. Find "secret"
    - Sector number of "secret" = (20514/512) = 40 **at begining of the disk, but partition start at sector 1**
    - Offset = (40-1) = 39

```
ifind -d 39 -o 1 two-partitions.dd # get 4 (a indo of block 39 f partition start 1)

istat -o 1 two-partitions.dd 4

##OUT##
Directory Entry: 4
Allocated
File Attributes: File, Archive
Size: 34
Name: HELLO.TXT

Directory Entry Times:
Written:        2023-03-16 08:57:32 (EDT)
```

```
Accessed:       2023-03-16 00:00:00 (EDT)
Created:        2023-03-16 08:57:32 (EDT)

Sectors:
39 0 0 0 # use only one sector

###

icat -s -o 1 two-partitions.dd 4

##OUT##
Hi there! ...
###

# We note that the size of ls hello is (34B) < oh the size dd rows (64)

dd if=two-partitions.dd bs=512 count=1 skip=40 | hexdump -C
ls -l hello.txt
```

2. Find "wDeek" and "didattica"

   - Sector number of "secret" = (547396/512) = 1069 **at begining of the disk, but partition start at sector 1026**
   - Offset = (1069-1026) = 43

```
ifind -d 43 -o 1026 two-partitions.dd # get 6 (a indo of block 43 of partition start 1026)

istat -o 1026 two-partitions.dd 6

##OUT##

Directory Entry: 6
Not Allocated # DELETED --> not mounted by OS
File Attributes: File, Archive
Size: 4096
Name: _EST~1.SWP

Directory Entry Times:
Written:        2023-03-16 09:04:10 (EDT)
Accessed:       2023-03-16 00:00:00 (EDT)
Created:        2023-03-16 09:04:10 (EDT)

Sectors:
43 44 45 46 47 48 49 50

###

icat -o 1026 two-partitions.dd 6 | strings

##OUT##
b0VIM 8.2
root
wDeek
```

```
/home/gio/didattica/file-systems/vol_fs_analysis/examples/pp/test
3210
#"!

###
```

3. Find “tool”

```
fls -rp two-partitions.dd -o 1026

##OUT##
r/r 4:  wikipedia.txt
r/r * 6:        .test.swp
r/r * 8:        test
v/v 16083:      $MBR
v/v 16084:      $FAT1
v/v 16085:      $FAT2
V/V 16086:      $OrphanFiles
###


istat -o 1026 two-partitions.dd 4

##OUT##
Directory Entry: 4
Allocated
File Attributes: File, Archive
Size: 3934
Name: WIKIPE~1.TXT

Directory Entry Times:
Written:        2023-03-16 09:04:24 (EDT)
Accessed:       2023-03-16 00:00:00 (EDT)
Created:        2023-03-16 09:04:24 (EDT)

Sectors:
39 40 41 42 55 56 57 58 # we see that the cluster is not consecutive and string "tool"
                        # is fragmented in "to...ol"
###

icat -o 1026 two-partitions.dd 4 | strings | hexdump -C | grep -C 6 tool
```

Carving

Is a method of recovering files without relying on metadata (like file names, paths, or inodes). It works by identifying file signatures (headers & footers) and extracting the data between them.(E.g., 0xFF 0xD8 and 0xFF 0xD9 for JPEG files).

Example

Inside eighties.dd (cc121c3a037f904a4fa5ef51263df9fdb800d89af7330df22615802b81821f9d) there is a FAT file system with some deleted content. In particular, there were files with the following SHA256 hashes:

- 4410aaee5ae15917c064f80a073ec75260482b7035fad58c85f1063d0b795733

- 1b756ad00ad842c3356c093583e2e4fab2540e15ca88750606f45f7efd1f4d26
- 592f47dfcbeda344fc394987b6e02a65a35d4d849d35d2fc821e5be1889c645d
- 8a461036c70736eb4ca83e9062318c8293be2baad1c475c41c1945221559048e
- 0d176b77f6b81468eb7ba367d35bdcbd8fdfc63445c2cc83c5e27c5e0b4c1a14

Can you recover and identify them?