

Digital Forensics

Federico Conti

2024/25

Contents

console.dd	3
Initial Setup	3
Analysis Process	3
Data Recovery	6
Advanced File System Analysis	6
 corrupted.dd	 8
Initial Setup	8
Analysis Process	9
Fix FAT Table	10
zxgio	10
Unlocated Space	11
 strange.dd	 13
Partition Scheme Identification	13
File System Type Identification	14
File System extraction	18
Failed attempts	19

console.dd

The purpose of this analysis is to investigate a provided disk image (`console.dd`), which appears to be an unpartitioned FAT filesystem containing only a single JPEG file. There is suspicion that the volume was recently reformatted to conceal prior data. The goal is to reconstruct the original partition scheme and recover as much of the original content as possible.

Initial Setup

```
diff console.dd.sha256 <(sha256sum console.dd)
```

```
file console.dd
```

```
# Output:
```

```
console.dd: DOS/MBR boot sector, code offset 0x3c+2, OEM-ID "mkfs.fat", sectors/cluster 4, root entries
```

```
fdisk -l console.dd
```

```
# Output:
```

```
Disk console.dd: 4 MiB, 4194304 bytes, 8192 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x00000000
```

```
sudo mount -oro console.dd /mnt/console1
```

```
strings -e S xbox.jpg | hexdump
```

```
# Output:
```

```
00000000 ff d8 ff e0 0a 4a 46 49 46 0a 32 22 33 2a 37 25 |.....JFIF.2"3*7%|
```

```
# SHA xbox.jpg
```

```
88f83bef7713f5e38aa59da9b71ec53081fe373593758879a4ce06a658cceed0 xbox.jpg
```

- The hash verification confirms that the image file `console.dd` is intact and unaltered.
- The image is detected as a FAT12 filesystem.
- No active partitions were listed, reinforcing that the current filesystem is unpartitioned.
- Upon mounting, only a single JPEG file (`xbox.jpg`) was present.
- The JPEG magic number (FFD8 FFE0) confirms the file type.

At this stage, the disk image appears as an unpartitioned FAT12 filesystem with a single valid JPEG file. However, further analysis is required to detect remnants of any previous partitioning.

Analysis Process

```
mmstat console.dd
```

```
# Output:
```

```
gpt
```

```
mm ls console.dd
```

```
# Output:
```

```
GUID Partition Table (EFI)
Offset Sector: 0
Units are in 512-byte sectors
```

	Slot	Start	End	Length	Description
000:	-----	0000000000	0000002047	0000002048	Unallocated
001:	002	0000002048	0000008158	0000006111	Linux filesystem
002:	Meta	0000008159	0000008190	0000000032	Partition Table
003:	-----	0000008159	0000008191	0000000033	Unallocated
004:	Meta	0000008191	0000008191	0000000001	GPT Header

img_stat console.dd

Output:

IMAGE FILE INFORMATION

Image Type: raw

Size in bytes: 4194304

Sector size: 512

Surprisingly, mmstat detected traces of a GUID Partition Table (GPT), which is inconsistent with a simple FAT12 format. This indicates remnants of a previous partition scheme.

Findings

- A previous GPT structure is partially detectable.
- There's evidence of a Linux filesystem starting at sector 2048.
- The presence of unallocated spaces and partition table metadata confirms that the disk was likely reformatted over an existing partitioned structure.

Using ImHex

Analysing the disk image with ImHex revealed further information on the GPT structure:

- It appears that the disk was quickly reformatted to Logical Block Addressing (LBA) 0 with a FAT12 filesystem. This action overwritten the primary GPT header, rendering it partially corrupted and unable to identify the original partition entries.
- Despite this, remnants of the GPT structure are still detectable, suggesting that the disk previously contained a

more complex partition scheme.

Given:

1. The disk image size is 4,194,304 bytes.
2. The sector size is 512 bytes.

We confirm there are exactly 8192 sectors. According to the GPT standard, the backup GPT header resides at the last sector (LBA 8191).

```
#include <std/mem.pat>

struct PartitionEntry {
    u8  bootIndicator;
    u8  startCHS[3];
    u8  partitionType;
    u8  endCHS[3];
    u32 relativeSectors;
    u32 totalSectors;
};

struct GPTHeader {
    char signature[8];           // "EFI PART"
    u32 revision;                // typically 0x00010000
    u32 headerSize;              // usually 92 (0x5C)
    u32 headerCRC32;             // CRC32 of the header
    u32 reserved;                // must be zero
    u64 currentLBA;              // LBA of this header
    u64 backupLBA;               // LBA of the backup GPT header
    u64 firstUsableLBA;
    u64 lastUsableLBA;
    u8  diskGUID[16];            // 128-bit GUID for the disk
    u64 partitionEntryLBA;       // LBA where partition entries start
    u32 numberOfPartitionEntries; // number of partition entries
    u32 sizeOfPartitionEntry;    // size of each partition entry (often 128)
    u32 partitionEntryArrayCRC32; // CRC32 of the partition entries
    u8  reserved2[420];          // 512 - 92 = 420 (fills one 512-byte sector)
};

struct GPTPartitionEntry {
    u8  partitionTypeGUID[16];
    u8  uniquePartitionGUID[16];
    u64 firstLBA;
    u64 lastLBA;
    u64 flags;
    u16 partitionName[36]; // 72 bytes (UTF-16)
};

//SECONDARY

// LBA = fdisk -l console.dd
GPTHeader Secondary_GPTHeader @ (8191 * 512);
GPTPartitionEntry Secondary_gptPartitions[128] @ (Secondary_GPTHeader.partitionEntryLBA * 512);
```

Result of GPT Analysis

- The secondary GPT header was successfully located at LBA 8191.
- The partition entries were parsed, revealing that: > Entry 2 defines a partition of type Linux File System. > This partition starts at sector 2048, consistent with previous findings from mmls.

The screenshot shows a hex editor on the left and a GPT partition table on the right. The hex editor displays raw data from address 003FBE20 to 003FBF70. The ASCII column shows the text "rG.y=i.G", "#I x!3.", "L i n u", "x . f i l e s . y", "s t e m .", and "s t e m .". The GPT partition table on the right lists the following partitions:

Name	Start	End	Size	T	Value	Comment
Secondary_gptPartiti	0x003FBE00	0x003FF0FF	16384	byt	GPT [...]	
▶ [0]	0x003FBE00	0x003FBE7F	128	bytes	str { ... }	
▶ [1]	0x003FBE80	0x003FBEFF	128	bytes	str { ... }	
▼ [2]	0x003FBF00	0x003FBF7F	128	bytes	str { ... }	
▶ partitionTypeGUID	0x003FBF00	0x003FBF0F	16	bytes	u8 [...]	
▶ uniquePartitionGU	0x003FBF10	0x003FBF1F	16	bytes	u8 [...]	
firstLBA	0x003FBF20	0x003FBF27	8	bytes	u64 2048	
lastLBA	0x003FBF28	0x003FBF2F	8	bytes	u64 8158	
flags	0x003FBF30	0x003FBF37	8	bytes	u64 0	
▶ partitionName	0x003FBF38	0x003FBF7F	72	bytes	u16 [...]	
▶ [3]	0x003FBF80	0x003FBFFF	128	bytes	str { ... }	
▶ [4]	0x003FC000	0x003FC07F	128	bytes	str { ... }	
▶ [5]	0x003FC080	0x003FC0FF	128	bytes	str { ... }	
▶ [6]	0x003FC100	0x003FC17F	128	bytes	str { ... }	
▶ [7]	0x003FC180	0x003FC1FF	128	bytes	str { ... }	
▶ [8]	0x003FC200	0x003FC27F	128	bytes	str { ... }	
▶ [9]	0x003FC280	0x003FC2FF	128	bytes	str { ... }	

Data Recovery

```
fsstat -o 2048 console.dd
```

Output:

```
File System Type: NTFS
Version: Windows XP
Cluster Size: 4096
Total Sector Range: 0 - 6109
```

The partition was identified as NTFS, confirming that the original system was likely Windows-based.

```
dd if=console.dd of=ntfs.dd bs=512 skip=2048 count=6110
```

```
mount -oro ntfs.dd /mnt/console1
```

Output:

```
file ps5.jpg
```

```
ps5.jpg: JPEG image data, JFIF standard 1.01, resolution (DPI), density 72x72, segment length 16, b
```

SHA ps5.jpg

```
200ff11c196aeeaecd7a4021aa40a47459a252b5776ecdd3104f2e5537eb75a2 ps5.jpg
```

Upon inspection of the mounted partition, a file named ps5.jpg was discovered. The file ps5.jpg is a valid JPEG image, confirming successful recovery of at least part of the original data stored prior to the reformatting attempt.

Advanced File System Analysis

To ensure a thorough investigation, advanced forensic tools were employed to enumerate files, recover orphaned data, and extract detailed metadata from the NTFS Master File Table (MFT).

```
fls -rp console.dd -o 2048
```

#

```
r/r 4-128-1: $AttrDef
```

```
...
```

```
r/r 0-128-1: $MFT
```

```
...
```

```
r/r 64-128-2: ps5. # uniquely identifies the file or directory within the Master File Table.
```

```
V/V 65: $OrphanFiles
```

```
-/r * 16: $OrphanFiles/OrphanFile-16
```

```
...
```

Presence of multiple orphaned files under \$OrphanFiles, suggesting incomplete deletions or filesystem inconsistencies prior to the reformat.

To recover any residual files not linked within the filesystem, foremost was executed:

```
foremost -i console.dd -o recover/
```

Result:

- The carving process did not detect any deleted files.
- All files recovered by foremost were consistent with those already identified through filesystem analysis (ps5.jpg and xbox.jpg).
- No additional user files, fragments, or hidden data were found beyond the active filesystem entries

```
88f83bef7713f5e38aa59da9b71ec53081fe373593758879a4ce06a658cceed0 00000049.jpg
```

```
200ff11c196aeeaecd7a4021aa40a47459a252b5776ecdd3104f2e5537eb75a2 00006128.jpg
```

For detailed file metadata, the NTFS Master File Table (MFT) was extracted and analyzed using specialized tools.

```
icat -r -o 2048 console.dd 0 > MFT.bin
```

```
MFTECmd.exe -f MFT.bin --csv .\ --csvf console_mft.csv
```

- The file ps5.jpg was confirmed as an active file (InUse=True) with a creation and modification date of April 11, 2023.
- No Alternate Data Streams (ADS) or special attributes were detected. (parsing with TimelineExplorer)

corrupted.dd

This section details the forensic analysis conducted on the disk image `corrupted.dd`. The objective was to investigate the file system structure, identify signs of corruption, recover inaccessible data, and locate specific string patterns within the image.

Initial Setup

```
file corrupted.dd
```

```
# Output:
```

```
DOS/MBR boot sector, code offset 0x3c+2, OEM-ID "mkfs.fat", Bytes/sector 2048, FATs 3, root entries
```

```
fsstat corrupted.dd
```

```
# Output:
```

```
...
File System Type: FAT12
OEM Name: mkfs.fat
Volume ID: 0xc8269037
Volume Label (Boot Sector): BILL
Volume Label (Root Directory): BILL
File System Type Label: FAT12
...
File System Layout (in sectors)
Total Range: 0 - 719
* Reserved: 0 - 0
** Boot Sector: 0
* FAT 0: 1 - 1
* FAT 1: 2 - 2
* FAT 2: 3 - 3
* Data Area: 4 - 719
** Root Directory: 4 - 11
** Cluster Area: 12 - 719
```

```
METADATA INFORMATION
```

```
-----
Range: 2 - 45831
Root Directory: 2
```

```
CONTENT INFORMATION
```

```
-----
Sector Size: 2048
Cluster Size: 2048
```


Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
00000000:	E8	3C	90	6D	6B	66	73	2E	66	61	74	00	08	01	01	00	<.mkfs.fat....
00000010:	03	00	02	D0	02	F8	01	00	10	00	02	00	00	00	00	002.....This
00000020:	00	00	00	00	80	00	29	37	90	26	C8	42	49	4C	4C	20)7.&BILL
00000030:	20	20	20	20	20	20	46	41	54	31	32	20	20	20	0E	1FFAT12..
00000040:	BE	5B	7C	AC	22	C0	74	08	56	B4	0E	BB	07	00	CD	10	.[."t.V.....
00000050:	5E	EB	F0	32	E4	CD	16	CD	19	EB	FE	54	68	69	73	20	^..2.....This
00000060:	69	73	20	6E	6F	74	20	61	20	62	6F	6F	74	61	62	6C	is not a bootabl
00000070:	65	20	64	69	73	68	2E	20	20	50	6C	65	61	73	65	20	e disk. Please
00000080:	69	6E	73	65	72	74	20	61	20	62	6F	6F	74	61	62	6C	insert a bootabl
00000090:	65	20	66	6C	6F	70	70	79	20	61	6E	64	00	0A	70	72	e floppy and..pr
000000A0:	65	73	73	20	61	6E	79	20	68	65	79	20	74	6F	20	74	ess any key to t
000000B0:	72	79	20	61	67	61	69	6E	20	2E	2E	2E	20	00	0A	00	ry again
000000C0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000D0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000E0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000F0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000100:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000110:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000120:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000130:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000140:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000150:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000160:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

```

2 struct bootSector { // VBR
3     u8 jumpInstruction[3];
4     char oemID[8];
5     u16 bytesPerSector;
6     u8 sectorsPerCluster;
7     u16 reservedSectors;
8     u8 numberOfFATs; //3
9     u16 rootEntries;
10    u16 totalSectors16;
11    u8 mediaDescriptor;
12    u16 sectorsPerFAT16;
13    u16 sectorsPerTrack;
14    u16 numberOfHeads;
15    u32 hiddenSectors;
16    u32 totalSectors32;
17    u8 driveNumber;
18    u8 reserved1;
19    u8 bootSignature;
20    u32 volumeSerialNumber;
21    char volumeLabel[11];
22    char fileType[8];
23    u8 bootCode[0x1C0];
24    u16 signature;
25 };

```

A sector 0 directly contains a FAT12 Boot Sector, so there is no MBR/GPT table listing partitions, and it is not a bootable image.

- **Volume Label:** BILL
- **Sector Size:** 2048
- **Cluster Size:** 2048
- **Num FATs:** 3

Analysis Process

Using The Sleuth Kit (TSK):

```

fls -r -p corrupted.dd | grep '\.TXT'
# Output:
r/r 45: HOMEWORK.TXT
r/r 32: NETWORKS.TXT
r/r * 36:      _EADME.TXT

```

HOMEWORK.TXT (pseudo-inode 45)

Status: Allocated
 Size: 6 bytes
 Sector: 619
 Readability:
 Can read it both by using `icat` and by mounting the image.

NETWORKS.TXT (pseudo-inode 32)

Status: Allocated
 Size: 17,465 bytes
 Starting Sector: 345
 Readability:
 Cannot read it by mounting the image, but you can read it using `icat`.
 Explanation: This indicates that the mounted file system has issues following the cluster chain, likely due to corruption in the FAT.

EADME.TXT (pseudo-inode 36)

Status: Deleted
 Size: 60,646 bytes

Sectors: 457 to 486

Readability:

Since this file is deleted, it is expected not to appear in the mounted file system. However, you can recover it using `icat` if the data has not been overwritten.

```
istat corrupted.dd 32
```

Output:

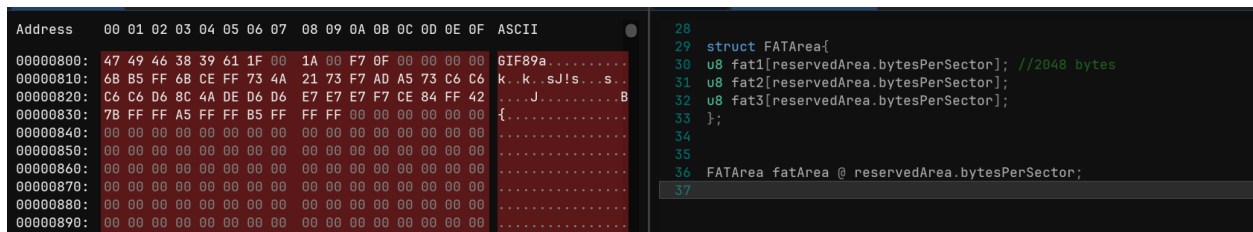
```
Directory Entry: 32 #pseudo inode by TSK
Allocated
File Attributes: File, Archive
Size: 17465
Name: NETWORKS.TXT
...
Sectors: 345
```

Fix FAT Table

Rebuild the cluster chain in the FAT for corrupted files, particularly for `NETWORKS.TXT`.

Analyzing the first FAT it was discovered that `FAT0` was overwritten with non-FAT data, likely a fragment of a GIF file.

```
xxd -s $((2048)) -l 2048 corrupted.dd | less
```



```
Address 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ASCII
00000800: 47 49 46 38 39 61 1F 00 1A 00 F7 0F 00 00 00 00 6IF89a.....
00000810: 68 B5 FF 6B CE FF 73 4A 21 73 F7 AD A5 73 C6 C6 k..k..sJ!s...s..
00000820: C6 C6 D6 8C 4A DE D6 D6 E7 E7 E7 F7 CE 84 FF 42 ....J.....B
00000830: 7B FF FF A5 FF FF B5 FF FF FF 00 00 00 00 00 {.....
00000840: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000850: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000860: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000870: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000880: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000890: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

28
29 struct FATArea{
30     u8 fat1[reservedArea.bytesPerSector]; //2048 bytes
31     u8 fat2[reservedArea.bytesPerSector];
32     u8 fat3[reservedArea.bytesPerSector];
33 };
34
35
36 FATArea fatArea @ reservedArea.bytesPerSector;
37
```

A known good copy of `FAT2` was used to restore `FAT0`:

```
cp corrupted.dd corrupted_fixed.dd
```

```
dd if=corrupted_fixed.dd of=corrupted_fixed.dd bs=2048 skip=3 seek=1 count=1 conv=notrunc
```

After fixing FAT:

- The cluster chain has been rebuilt.
- Both `.TXT` files can now be mounted correctly, allowing the recovery of the hash for `NETWORKS.TXT`.

```
sha256sum *.TXT
```

```
9b4a458763b06fefc65ba3d36dd0e1f8b5292e137e3db5dea9b1de67dc361311 HOMEWORK.TXT
```

```
e9207be4a1dde2c2f3efa3aeb9942858b6aaa65e82a9d69a8e6a71357eb2d03c NETWORKS.TXT
```

zxgio

Inside the file `corrupted.dd`, there are some occurrences of the string `zxgio` (without quotes). Below is the analysis:

```
strings -t d corrupted.dd | grep -E zxgio
```

```
512 zxgio
```

```
2832 zxgio
```

```
724025 zxgio
```

```
1267712 zxgio
```

From fsstat on intact image:

Offset (byte)	Sector	File System Area
512	0	Boot Sector
2832	1	FAT 0 (corrupted)
724025	353	Cluster Area (slack space)
1267712	619	Cluster Area (inside HOMEWORK.TXT 619-619 (1) -> EOF)

1. Verify sector 353:

- Sector 353 contained the string in slack space (confirmed via dd and xxd).
- Offset 1,267,712 confirmed within HOMEWORK.TXT.
- No occurrence found within actual data of NETWORKS.TXT.

```
istat corrupted_fixed.dd 32
# Output:
Directory Entry: 32
Size: 17465
Name: NETWORKS.TXT
Sectors: 345 346 347 348 349 350 351 352 353
```

```
icat corrupted_fixed.dd 32 | grep zxgio
# Output:
NULL
```

2. Slack Space Inspection:

- It can be confirmed that the string is contained in the slack space of cluster 353

```
dd if=corrupted_fixed.dd bs=2048 skip=353 count=1 of=sector353.bin
xxd sector353.bin | less
# Output:
tware....zxgio..
```

Unlocated Space

The image corrupted.dd has a size of 721 sectors, while the FAT12 file system only uses sectors 0-719. Sector 720, being outside the file system, was extracted and analyzed.

```
dd if=corrupted.dd bs=2048 skip=720 count=1 of=unused_sector720.bin
```

```
file unused_sector720.bin
```

```
# Output:
ASCII text
```

```
echo "ascii text" | base64 -d > hidden_file
```

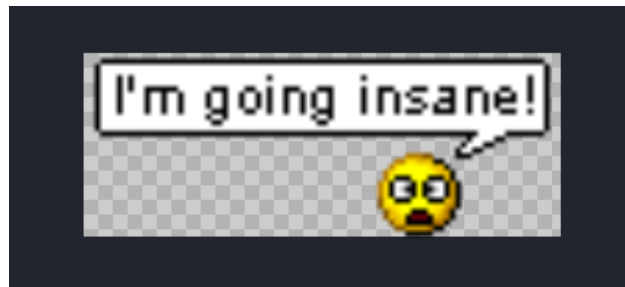
```
file hidden_file
```

```
# Output:
GIF image data, version 89a, 86 x 33
```

Results:

- The sector contains a long ASCII string without line terminators.
- Analysis revealed it to be Base64 encoding.

- Decoding the string produced a file recognized as a GIF image.



strange.dd

This section analyzes the disk image `strange.dd`, which exhibits an unusual dual file system configuration. The image uses a GPT partition scheme with a single partition labeled as `Microsoft basic data`. Within this partition, both FAT32 and Ext3 file systems coexist, creating an ambiguous setup that challenges traditional forensic tools. The analysis explores the partition scheme, identifies the file systems, and extracts their contents.

Partition Scheme Identification

- The disk image `strange.dd` uses a GPT partition scheme.
- It includes a Protective MBR and GPT header.
- Only one partition entry is defined in Primary GPT Entries, labeled as `Microsoft basic data`.

```
fdisk -l strange.dd
```

#Output

```
Disk strange.dd: 8 GiB, 8589934592 bytes, 16777216 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: B5D4692F-0DB0-4356-B0E5-5A70BACB2347
```

```
mmls strange.dd
```

#Output

Slot	Start	End	Length	Description
000: Meta	0000000000	0000000000	0000000001	Safety Table
001: -----	0000000000	0000002047	0000002048	Unallocated
002: Meta	0000000001	0000000001	0000000001	GPT Header
003: Meta	0000000002	0000000033	0000000032	Partition Table
004: 000	0000002048	0016777182	0016775135	Microsoft basic data
005: -----	0016777183	0016777215	0000000033	Unallocated

The screenshot shows a forensic analysis tool interface. The left pane displays a hex dump of the disk image with ASCII characters on the right. A red box highlights the GPT header area starting at address 0000000002. The right pane shows a tree view of the disk structure, with 'Primary_GPTHeader' highlighted in a red box.

Figure 1: Primary GPT Header

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
000003E0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000003F0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000400:	A2	A0	D0	EB	E5	B9	33	44	87	C0	68	B6	B7	26	99	C73D..h.&.
00000410:	91	85	A4	D0	C1	B8	A5	4E	91	A1	30	95	A5	A7	67	50N...gP
00000420:	00	08	00	00	00	00	00	00	DE	FF	FF	00	00	00	00	00M.i.c.r.
00000430:	00	00	00	00	00	00	00	00	40	00	69	00	63	00	72	00o.s.o.f.t..b.a.
00000440:	6F	00	73	00	6F	00	66	00	74	00	20	00	62	00	61	00	s.i.c..d.a.t.a.
00000450:	73	00	69	00	63	00	20	00	64	00	61	00	74	00	61	00	
00000460:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000470:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000480:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000490:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000004A0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000004B0:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

Figure 2: entry[0] in Primary GPT Entries

File System Type Identification

Extract only the partition part to better analyze it

```
mmcat strange.dd 4 > microsoftdata.dd
```

The disktype command reveals something unusual about this partition: Both file systems appear to coexist within the same partition space of ~8 GiB.

```
fsstat microsoftdata.dd
```

#Output

Multiple file system types detected (EXT2/3/4 or FAT)

```
disktype microsoftdata.dd
```

#Output

```
--- microsoftdata.dd
```

Regular file, size 7.999 GiB (8588868608 bytes)

FAT32 file system (hints score 3 of 5)

Unusual sector size 2048 bytes

Volume size 7.931 GiB (8515354624 bytes, 519736 clusters of 16 KiB)

Volume name "FAT32LABEL"

Ext4 file system

Volume name "ext3label"

UUID 66A53AB6-90CE-4122-8B31-8102D0845496 (DCE, v4)

Last mounted at "/dir/dev1"

Volume size 7.999 GiB (8588865536 bytes, 2096891 blocks of 4 KiB)

Potential Scenarios: - Possible file system-in-file system nesting (e.g., FAT embedded within an EXT partition). - Hidden Volumes: There could be a FAT file system hidden at a specific offset within the EXT3 partition.

ImHex analysis

The analysis confirms the following:

1. Recognizes the FAT32 boot sector at the start of the partition.
 - FAT32 characteristics:
 - Sector size: 2048 bytes.
 - Contains only 1 FAT table instead of the standard 2.
 - No backup boot sector is present.
 - Volume label: "FAT32LABEL".
2. An Ext3 SuperBlocks file system resides in the 1024 offset:

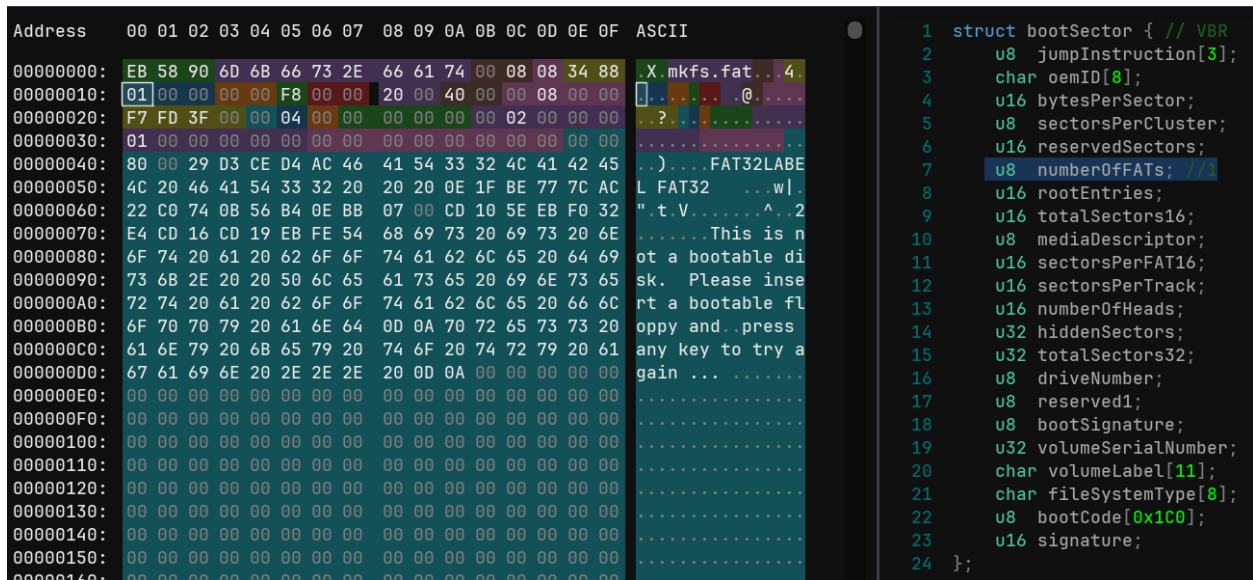


Figure 3: FAT boot sector

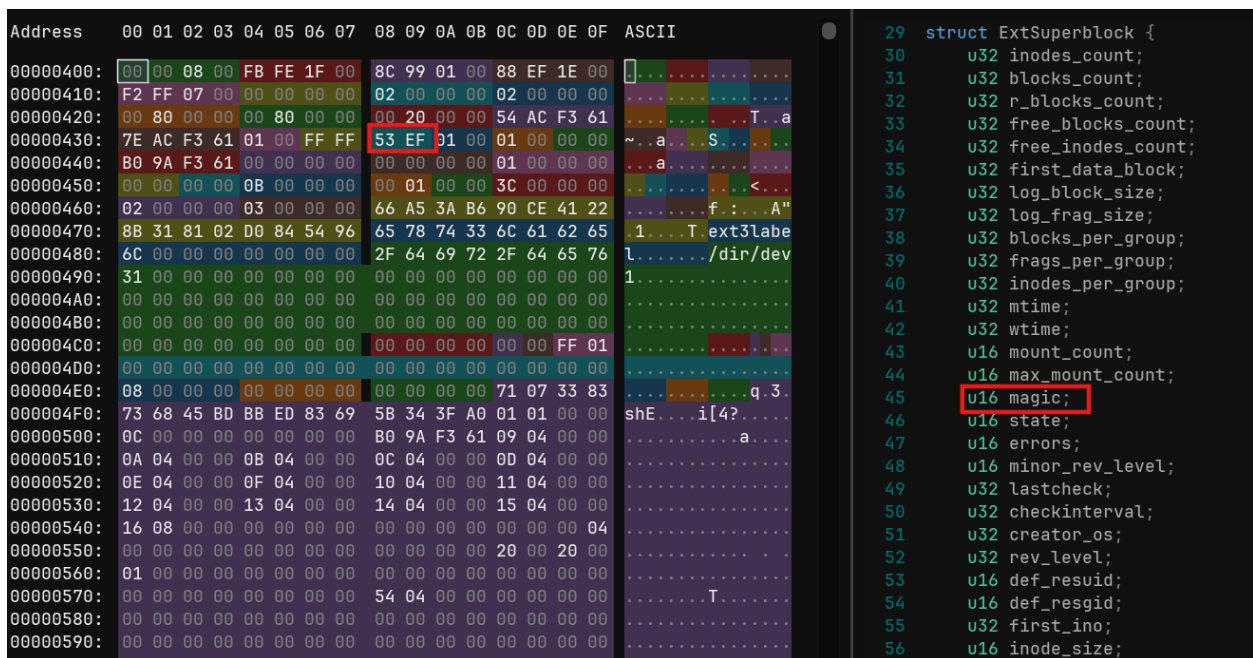


Figure 4: Ext3 First SuperBlock

This dual file system setup demonstrates forensic techniques (TSK), making it challenging to identify and analyze the true contents of the disk image.

After some attempts to get some hints from strings, here's probably the more informative combination of parameters:

```
strings --radix=d microsoftdata.dd | grep -i -E 'ext3|hint|jpg'
```

#Output

```
1144 ext3label
```

```

4206644 ext3_nashorn_1.jpg
4206672 ext3_nashorn_2.jpg
4206700 ext3_nashorn_3.jpg
4268084 ext3_nashorn_1.jpg
4268112 ext3_nashorn_2.jpg
4273272 ext3label
4321332 ext3_nashorn_1.jpg
4321360 ext3_nashorn_2.jpg
4321388 ext3_nashorn_3.jpg
73506912 FAT32_~1JPG
73507008 FAT32_~2JPG
73507104 FAT32_~3JPG
134217848 ext3label
402653304 ext3label
671088760 ext3label
939524216 ext3label
1207959672 ext3label
3355443320 ext3label
3623878776 ext3label
6576668792 ext3label
6576670208 There is a hint here

```

Analysing the HINT

```
xxd -s 6576670208 -l 512 microsoftdata.dd
```

This is the suggested paper: **4.2. Example B: Ext3 and FAT32**

An **Ambiguous File System Partition** is a deliberately crafted partition where show:

- it is possible to create ambiguous file system partitions by integrating a guest file system into the structures of a host file system: integrating a fully functional FAT32 into Ext3.
- Traditional forensic tools may detect one, both, or even get confused.

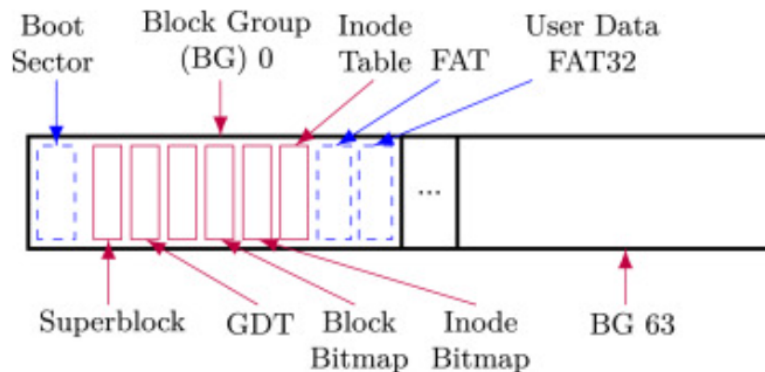


Figure 5: overview of the construction

“the superblock of Ext3 has a fixed offset of 1024 bytes, which provides enough space for another data structure to be placed before. Therefore, the Ext3 file system serves as the host file system for the combination with FAT32.”

This technique demonstrates the challenges in identifying and analyzing the true contents of a disk image, requiring advanced forensic methods to uncover hidden data.

Map Both File Systems Precisely

1. FAT

```
fsstat -f fat microsoftdata.dd
```

#Output

```
FILE SYSTEM INFORMATION
```

```
-----  
File System Type: FAT32
```

```
OEM Name: mkfs.fat  
Volume ID: 0xacd4ced3  
Volume Label (Boot Sector): FAT32LABEL  
Volume Label (Root Directory): FAT32LABEL  
File System Type Label: FAT32  
Next Free Sector (FS Info): 4295003172  
Free Sector Count (FS Info): 0
```

```
Sectors before file system: 2048
```

```
File System Layout (in sectors)  
Total Range: 0 - 4193782  
* Reserved: 0 - 34867  
** Boot Sector: 0  
** FS Info Sector: 1  
** Backup Boot Sector: 0  
* FAT 0: 34868 - 35891 # FAT0  
* Data Area: 35892 - 4193782  
** Cluster Area: 35892 - 4193779  
*** Root Directory: 35892 - 35899  
** Non-clustered: 4193780 - 4193782
```

```
METADATA INFORMATION
```

```
-----  
Range: 2 - 266105029  
Root Directory: 2
```

```
CONTENT INFORMATION
```

```
-----  
Sector Size: 2048  
Cluster Size: 16384  
Total Cluster Range: 2 - 519737
```

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

```
-----  
35892-35899 (8) -> EOF  
35900-36059 (160) -> EOF  
36060-36227 (168) -> EOF  
36228-36379 (152) -> EOF
```

- size of the FS is $2048 * 4193782 = 8588865536$ bytes.

2. Ext3

```
fsstat -f ext3 microsoftdata.dd
```

#Output

```
....
CONTENT INFORMATION
-----
Block Range: 0 - 2096890
Block Size: 4096
Free Blocks: 2027400

BLOCK GROUP INFORMATION
-----
Number of Block Groups: 64
Inodes per group: 8192
Blocks per group: 32768
....
Group: 0:
Inode Range: 1 - 8192
Block Range: 0 - 32767
Layout:
    Super Block: 0 - 0
    Group Descriptor Table: 1 - 1
    Data bitmap: 513 - 513
    Inode bitmap: 514 - 514
    Inode Table: 515 - 1026
    Data Blocks: 1027 - 32767
Free Inodes: 8181 (99%)
Free Blocks: 0 (0%) # !!!
Total Directories: 2
....
```

- the FS size is $4096 * 2096890 = 8588861440$ bytes.

To protect FAT32 data from being overwritten by the Ext3 file system, the group descriptor table, the superblock (and their copies) and the respective block bitmaps had to be manipulated. Vice versa, clusters occupied by the Ext3 file system had to be marked as bad in the FAT.

- Ext3 avoids overwriting Fat32 by marking 0 free blocks in the first group.
- all groups have almost all free blocks except for the first one, where there are no free blocks.

Summary

Offset (Hex)	Offset (Dec)	Content
0x00000000	0	FAT32 Boot Sector
0x00000400	1024	Ext3 Superblock
0x00008834	34868	FAT
0x00008C34	35892	FAT32 Data Area Start
0x003FFDF5	4193781	FAT32 Data Area End

File System extraction

```
sudo mount -o loop,ro -t ext3 microsoftdata.dd /mnt/strange/ext3
```

#Output

```
sha256sum ext3_nashorn_*
8b79029a06610f29ba1c16e4cd4cf498e196e3a7f67a53efebb32f720f3d472d ext3_nashorn_1.jpg
0cb84374324e13606bb22b4164323bb487f9088e4a2cc700673180256174e294 ext3_nashorn_2.jpg
193067cecbd63195bfab2f3f702cc44ff3c6e6fa8de5335a405fbeb9955c3512 ext3_nashorn_3.jpg
```

```
sudo mount -o loop,ro -t vfat microsoftdata.dd /mnt/strange/fat32
```

#Output

```
sha256sum fat32_nashorn_*
8b79029a06610f29ba1c16e4cd4cf498e196e3a7f67a53efebb32f720f3d472d fat32_nashorn_1.jpg
0cb84374324e13606bb22b4164323bb487f9088e4a2cc700673180256174e294 fat32_nashorn_2.jpg
193067cecbd63195bfab2f3f702cc44ff3c6e6fa8de5335a405fbeb9955c3512 fat32_nashorn_3.jpg
```

Failed attempts

Try to mount the image by jumping directly to the Ext3 superblock FAIL

```
mount -o ro,loop,offset=1024 microsoftdata.dd /mnt/strange
```

#Output

```
mount: /mnt/strange: wrong fs type, bad option, bad superblock on /dev/loop0, m
missing codepage or helper program, or other error.
```

```
    dmesg(1) may have more information after failed mount system call.
```

Let's go look for the backup superblock and mount with respect offset also fail:

- To protect FAT32 data from being overwritten by the Ext3 file system, the group descriptor table, the superblock (and their backup) and the respective block bitmaps are manipulated,

```
dumpe2fs microsoftdata.dd | grep -i superblock
```

#Output

```
Primary superblock at 0, Group descriptors at 1-1
Backup superblock at 32768, Group descriptors at 32769-32769
...
...
...
```

Zeroing out the first 512 bytes allowed successful mounting of the Ext3 file system; however, the FAT32 file system specifications were lost in the process.

```
cp microsoftdata.dd microsoftdata_clean.dd
```

```
dd if=/dev/zero of=microsoftdata_clean.dd bs=512 count=1 conv=notrunc
```

```
sudo mount -o ro,loop microsoftdata_clean.dd /mnt/strange
```