INSTITUTO SUPERIOR TÉCNICO

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

# FORENSICS CYBER-SECURITY

MEIC, METI

# Tutorial III

## Introduction to File System Forensics

2023/2024

nuno.m.santos@tecnico.ulisboa.pt

## Introduction

This guide aims to provide a hands-on introduction to file system forensics and to introduce you to the usage of well-known file system analysis tools such as The Sleuth Kit (TSK), as well as Scalpel and Foremost, two file carving tools which may help to collect evidence in the case file metadata is not available. The forensic images required for these exercises can be downloaded from the course website.

Before we begin, make sure to prepare the environment for forensic analysis. Please refer to the Kali Linux forensic testbed that you have set up in Tutorial I. We also suggest you to review the slides of theory classes 9 and 10 which provide you the technical background that will allow you to understand the steps presented herein.

## 1 Deleted File Identification and Recovery (Ext2)

The goal of this exercise is to train you to the identification and recovery of deleted files from a forensic disk image containing an Ext2 file system. In this particular case, the file can be recovered based on leftover unallocated metadatada that can be retrieved using The Sleuth Kit (TSK). TSK is a collection of command line tools and a C library that allows you to analyze disk images and recover files from them. We will start by getting familiar with a couple of file system analysis tools provided by TSK, like `fsstat`, `fls`, or `mmls`, running them against a disk image.

Before starting your analysis, download and extract the `able2` disk image into some folder in the Kali forensic environment by executing the following commands:

```
$ wget https://turbina.gsd.inesc-id.pt/csf2324/t3/able2.tar.gz
$ tar -xvzf able2.tar.gz
```

### 1.1 List the partition table

The first tool we are going to use, `mmls`, provides access to the partition table within an image, and gives the partition offsets in sector units. We may run the following command:

```
$ mmls able2.dd
```

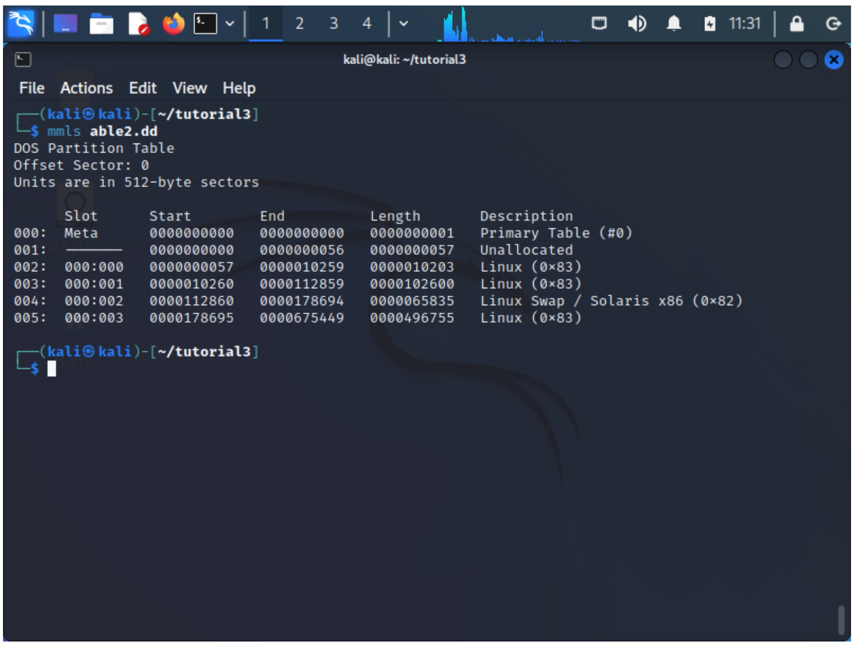


**Figure 1:** Output of `mmls able2.dd`

In this analysis, suppose that the information we are looking for is located on the root file system of our image. The root ( / ) file system is located on the second partition. Looking at our `mmls` output, we can see that that partition starts at sector 10260 (numbered 03 in the `mmls` output, or slot 000:001).

### 1.2 Gather file system information

Next, we need to learn what's inside our target partition. The `fsstat` command provides specific information about the file system in a volume. We can run TSK's `fsstat` command with `-o 10260` to gather file system information at that offset: (Figure 2)

```
$ fsstat -o 10260 able2.dd
```

This specifies that we want information residing on the partition that starts at sector offset 10260.

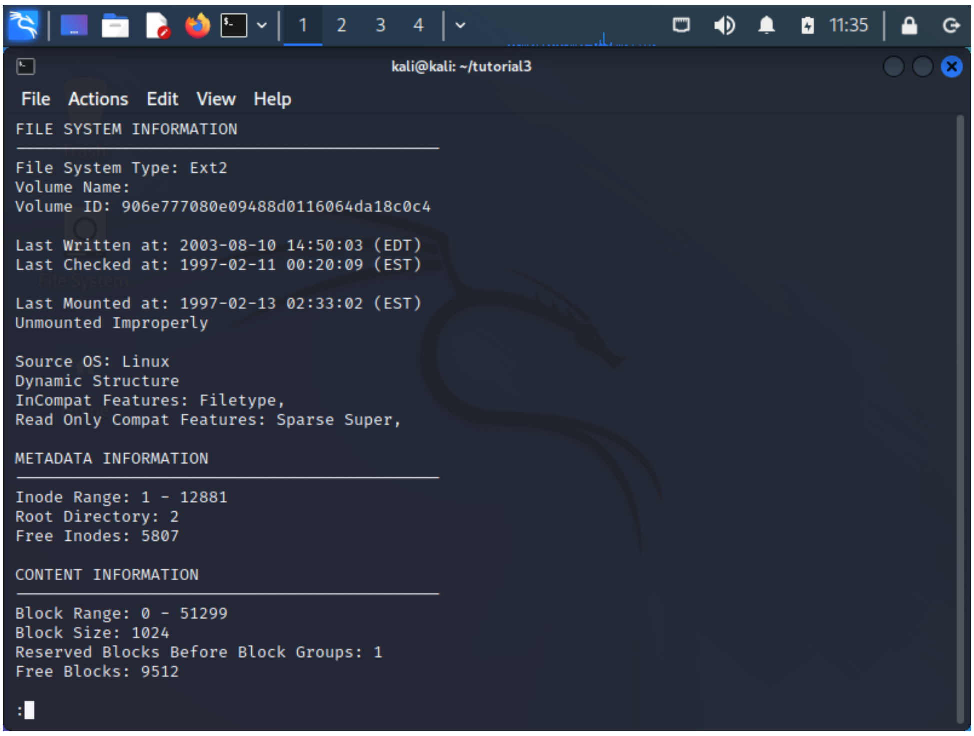


**Figure 2:** Output of `fsstat -o 10260 able2.dd | less`

### 1.3 List file names and directories

We can get more information using the `fls` command, which lists the file names and directories contained in a file system, or in a given directory. If you run the `fls` command with only the `fls` option to specify the file system, then by default it will run on the file system's root directory (see Figure 3). This is inode 2 on an EXT file system and MFT entry 5 on an NTFS file system. In other words, on an EXT file system, any of the below commands will yield the same output (Fig 3):

```
$ fls -o 10260 able2.dd
```

<u>or</u>

```
$ fls -o 10260 able2.dd 2
```

There are several points we want to take note of before we continue. Let's take a few lines of output and describe what the tool is telling us. Have a look at the last three lines from the above `fls` command.

```
...
r/r 1042:    .bash_history
```

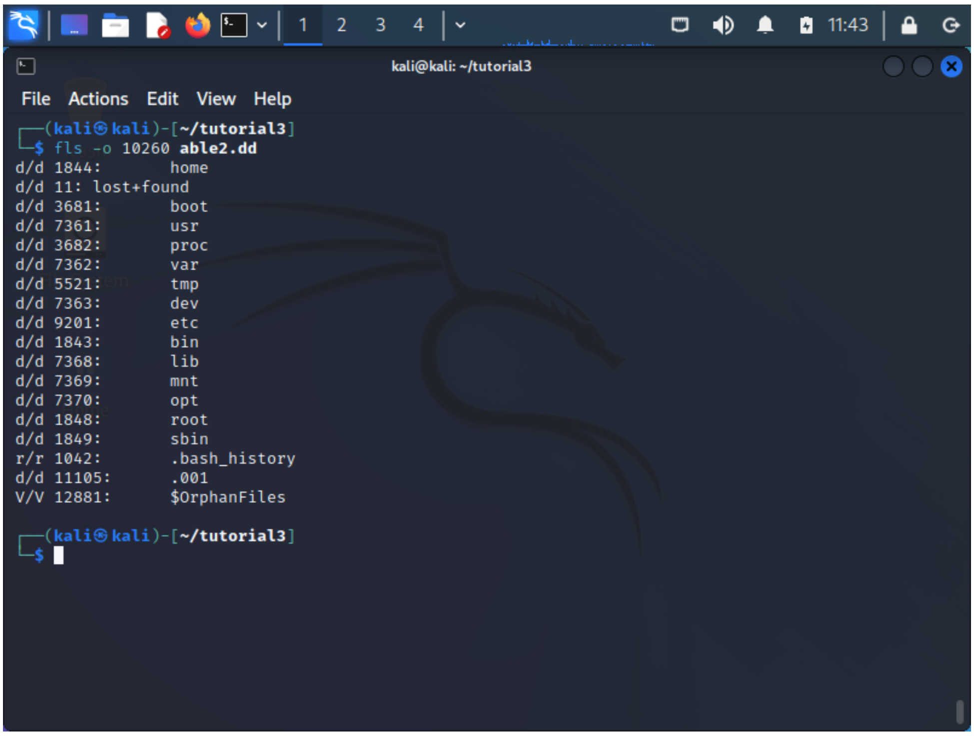


**Figure 3:** Output of `fls -o 10260 able2.dd`

```
d/d 11105:   .001
d7d 12881    $OrphanFiles
```

Each line of output starts with two characters separated by a slash. This field indicates the file type as described by the file's directory entry, and the file's metadata (in this case, the inode because we are looking at an EXT file system). For example, the first file listed in the snippet above, `.bash_history`, is identified as a regular file in both the file's directory and inode entry. This is noted by the `r/r` designation. Conversely, the following two entries (`.001` and `$OrphanFiles`) are identified as directories. The next field is the metadata entry number (inode, MFT entry, etc.) followed by the filename. In the case of the file `.bash_history` the inode is listed as 1042.

Note that the last line of the output, `$OrphanFiles` is a virtual folder created by TSK and assigned a virtual inode. This folder contains virtual file entries that represent unallocated metadata entries where there are no corresponding file names. These are commonly referred to as "orphan files", which can be accessed by specifying the metadata address, but not through any file name path.

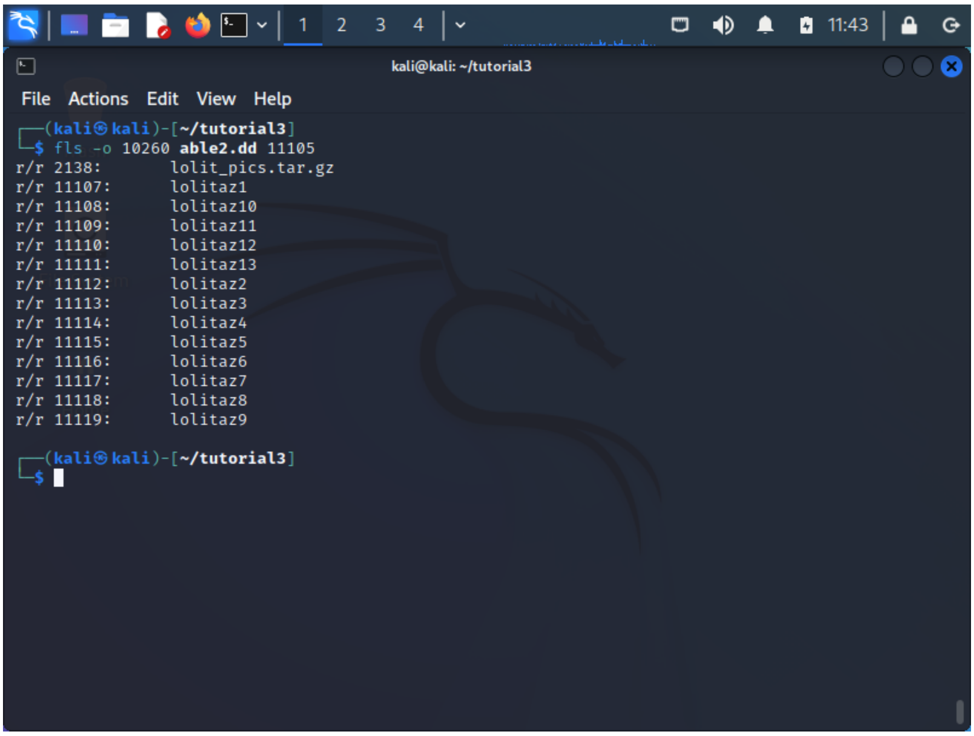


**Figure 4:** Output of `fls -o 10260 able2.dd 11105`

In Figure 4, we continue to run `fls` on directory entries to dig deeper into the file system structure (we could also have used `-r` for a recursive listing). By passing the metadata entry number of a directory, we can view its contents. For example, have a look at the `.001` directory in the listing above. This is an unusual directory and would cause some suspicion. It is hidden (starts with a "."), and no such directory is common in the root of the file system. To see the contents of the `.001` directory, we would pass its inode to `fls` as it can be observed in Figure 4.

## 1.4 Uncover deleted files

Mostly important for this exercise, `fls` can also be useful for uncovering deleted files. For example, if we wanted to exclusively check deleted entries that are listed as files (rather than directories), and we want the listing to be recursive, we could run the command:

```
$ fls -o 10260 able2.dd -Frd able2.dd
```

This command runs against the partition in `able2.dd` starting at sector offset 10260 (-o 10260), showing only file entries (-F), descending into directories recursively (-r), and displaying deleted entries (-d). The output of the above command can be observed in Figure 5.

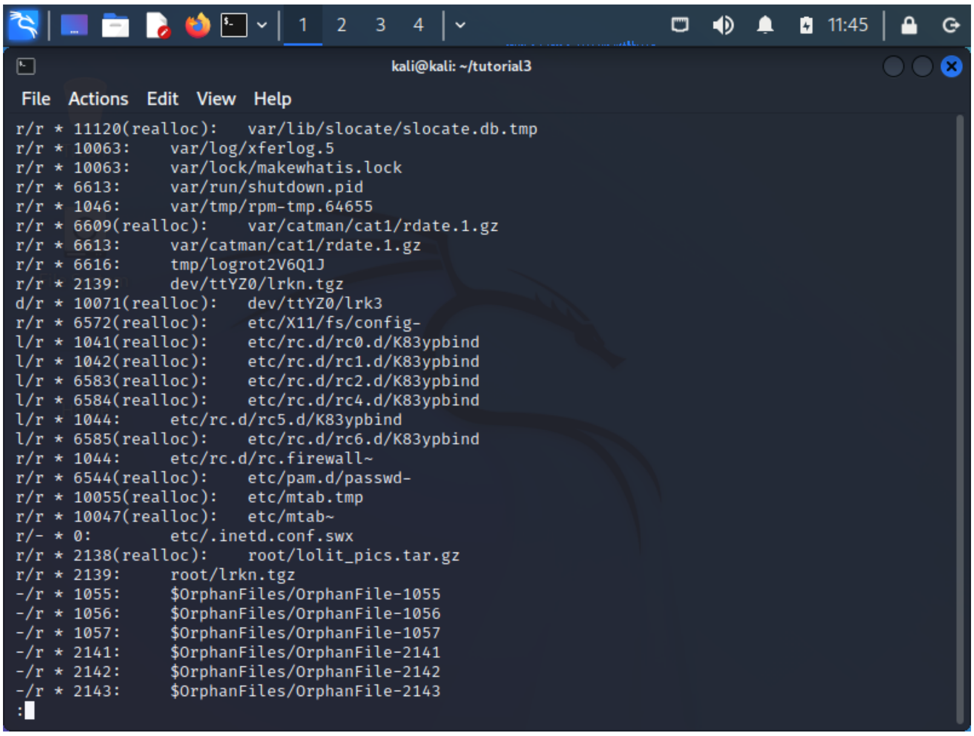


**Figure 5:** Output of `fls -o 10260 able2.dd -Frd able2.dd | less`

Notice that all of the files listed have an asterisk (*) before the inode. This indicates the file is deleted, which we expect in the above output since we specified the `-d` option to fls. We are then presented with the metadata entry number (inode, MFT entry, etc.) followed by the filename.

Have a look at the line of output for inode number 2138 (`root/lolit_pics.tar.gz`). The inode is followed by `realloc`. Keep in mind that `fls` describes the file name layer. The `realloc` means that the file name listed is marked as unallocated, even though the metadata entry (2138) is marked as allocated. Thus, the inode from our deleted file may have been "reallocated" to a new file.

In the case of inode 2138, it looks as though the `realloc` was caused by the file being moved to the directory `.001` (see the `fls` listing of `.001` on the previous page – inode 11105). This causes it to be deleted from it's current directory entry (`root/lolit_pics.tar.gz`) and a new file name created (`.001/lolit_pics.tar.gz`). The inode and the data blocks that it points to remain unchanged and in "allocated status", but it has been "reallocated" to the new name.

## 1.5 Find all file names associated with a particular inode

Let's continue our analysis by taking advantage of metadata (inode) layer tools included in TSK. In a Linux EXT type file system, an inode has a unique number and is assigned to a file. The number corresponds to the inode table, allocated when a partition is formatted. The inode contains all the metadata available for a file, including the modified/accessed/changed times and a list of all the data blocks allocated to that file. If you look at the output of our last `fls` command, you will see a deleted file called `lrkn.tgz` located in the `/root` directory:

```
...
r/r * 2139:   root/lrkn.tgz
...
```

The inode displayed by `fls` for this file is 2139. This inode also points to another deleted file in `/dev` (same file, different location). We can find all the file names associated with a particular metadata entry by using the command: (Figure 6)

```
$ ffind -o 10260 -a able2.dd 2139
```

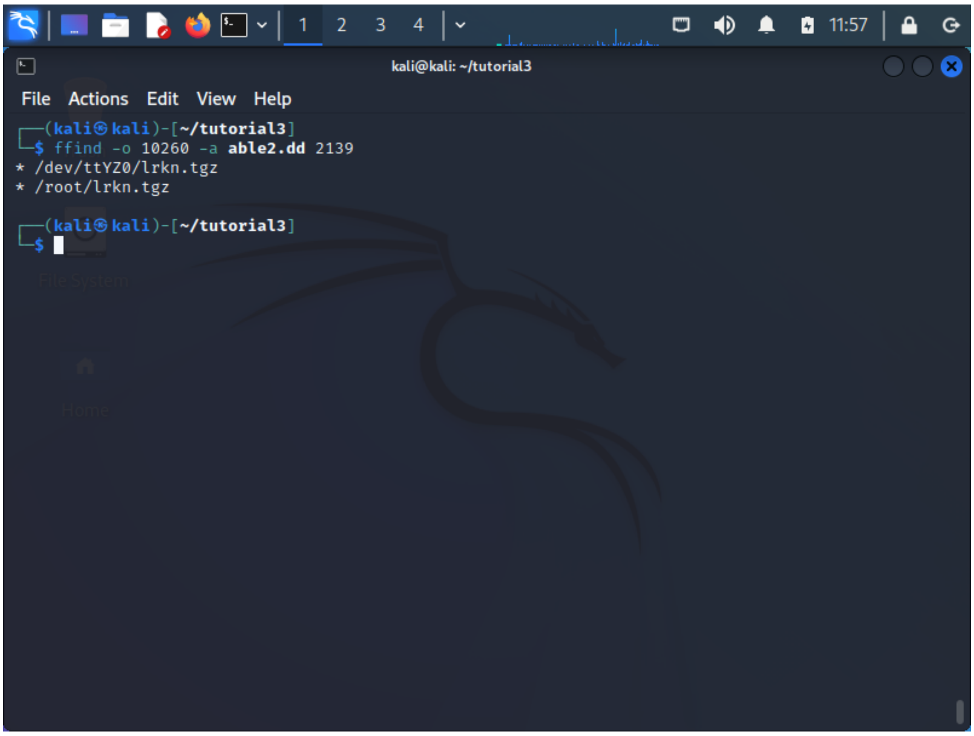


**Figure 6:** Output of `ffind -o 10260 -a able2.dd 2139`

Here we see that there are two file names associated with inode 2139, and both are deleted, as noted again by the asterisk (the `-a` ensures that we get all the inode associations).

## 1.6 Gather information about an inode

Continuing on, we are going to use `istat`. Remember that `fsstat` took a file system as an argument and reported statistics about that file system. `istat` does the same thing; only it works on a specified inode or metadata entry. In NTFS, this would be an MFT entry, for example.

To gather information about inode 2139 (Figure 7), we may execute:

```
$ istat -o 10260 able2.dd 2139
```

This command reads the inode statistics on the file system located in the `able2.dd` image in the partition at sector offset 10260 (`-o` 10260), from inode 2139 found in our `fls` command. There is a large amount of output here, showing all the inode information and the file system blocks ("Direct Blocks") that contain all of the file's data.

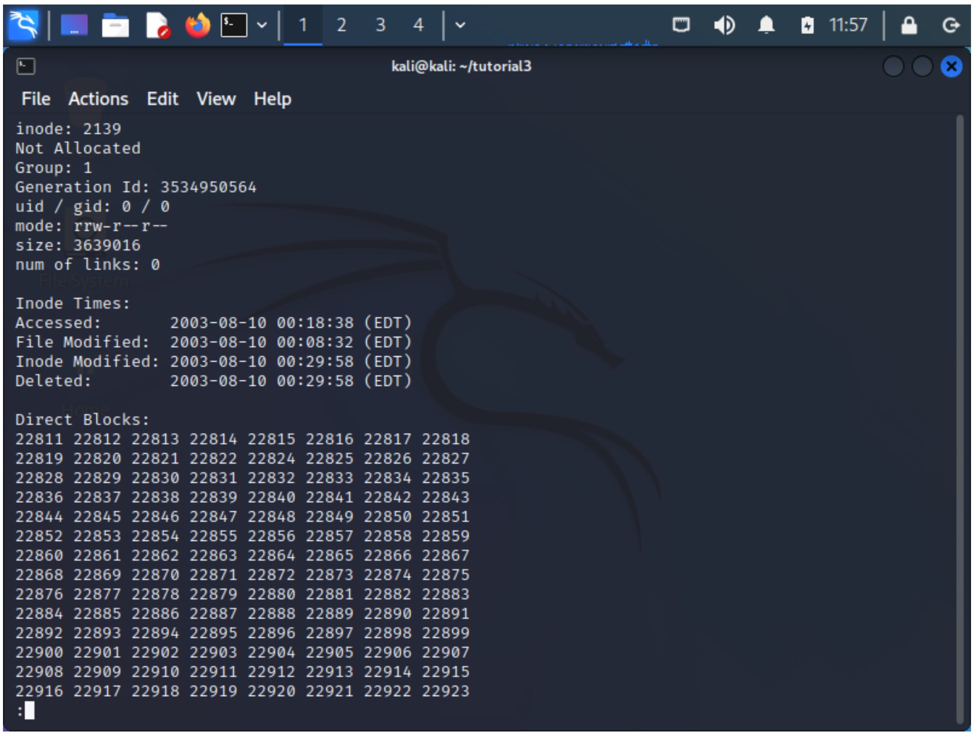


**Figure 7:** Output of `istat -o 10260 able2.dd 2139 | less`

Keep in mind that the Sleuth Kit supports a number of different file systems. istat (along with many of the Sleuth Kit commands) will work on more than just an EXT file system. The descriptive output will change to match the file system istat is being used on. You can see the supported file systems by running:

```
$ istat -f list
```

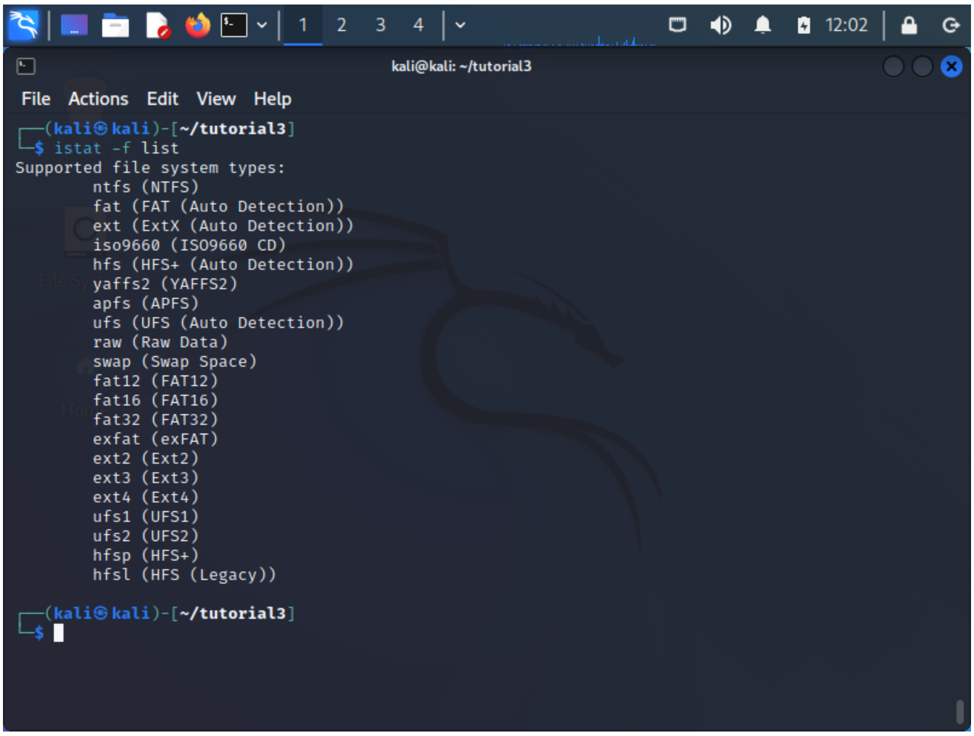


**Figure 8:** Output of `istat -f list`

### 1.7 Extract and inspect the deleted file based on its former inode

We now have the name of a deleted file of interest (from `fls`) and the inode information, including where the data is stored (from `istat`). Next we are going to use the `icat` command from TSK to grab the actual data contained in the data blocks referenced from the inode. `icat` also takes the inode as an argument and reads the content of the data blocks that are assigned to that inode, sending it to standard output. Remember, this is a deleted file that we are recovering here. We are going to send the contents of the data blocks assigned to inode 2139 to a file for closer examination:

```
$ icat -o 10260 able2.dd 2139 > lrkn.tgz.2139
```

This command runs the `icat` tool on the file system in our `able2.dd` image at sector offset 10260 (`-o` 10260) and streams the contents of the data blocks associated with inode 2139 to the file `lrkn.tgz.2139`. Now that we have what we hope is a recovered file, we can inspect the recovered data with the `file` command (Figure 9).

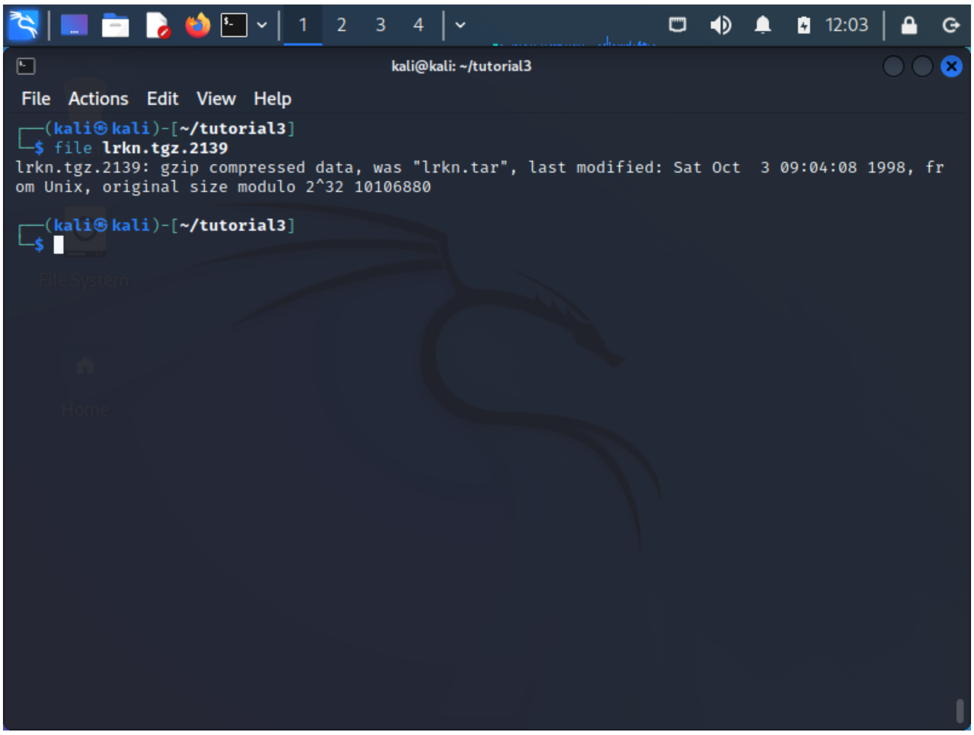


**Figure 9:** Output of `file lrkn.tgz.2139`

Have a look at the contents of the recovered archive by executing:

```
$ tar -tzvf lrkn.tgz.2139
```

Recall that the `t` option to the tar command lists the contents of the archive (Figure 10).

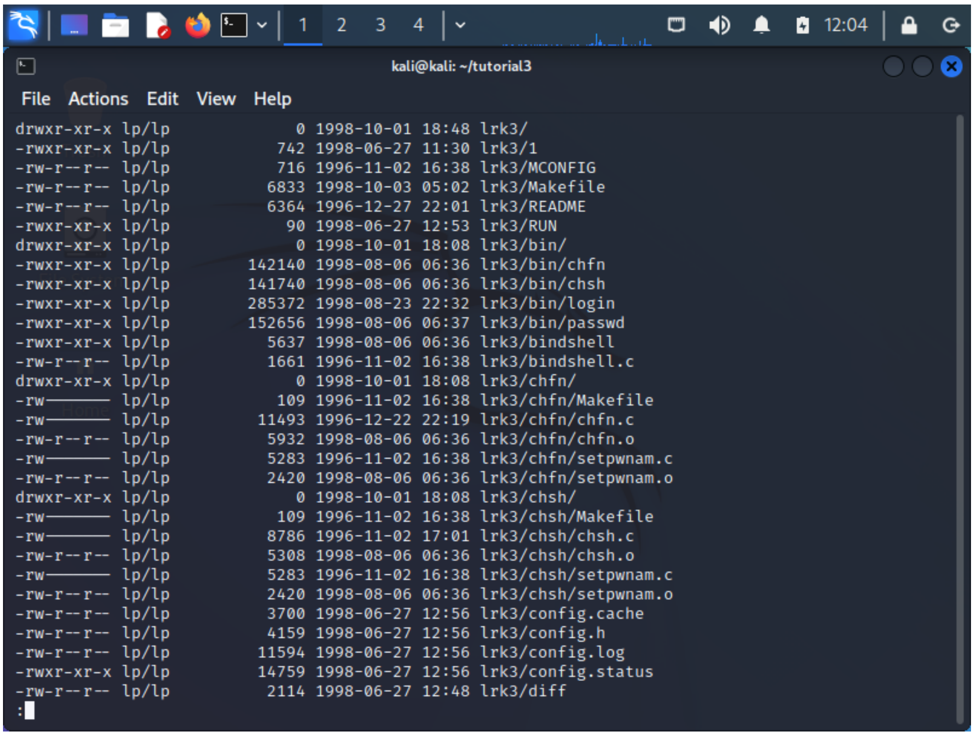


**Figure 10:** Output of `tar -tzvf lrkn.tgz.2139 | less`

We can notice that there is a `README` file included in the archive. Rather than extracting the entire contents of the archive, we will first extract and inspect the `README` file using the command:

```
$ tar -xzvOf lrkn.tgz.2139 lrk3/README > lrkn.2139.README
```

In this tar command we specify that we want the output sent to stdout (`-O` option) so we can redirect the extracted file (`lrk3/README`) to a new file called `lrkn.2139.README`.

If you read the file (for example with `cat lrkn.2139.README | less`), you will find out that we have uncovered a "rootkit", full of programs used to hide a hacker's activity.

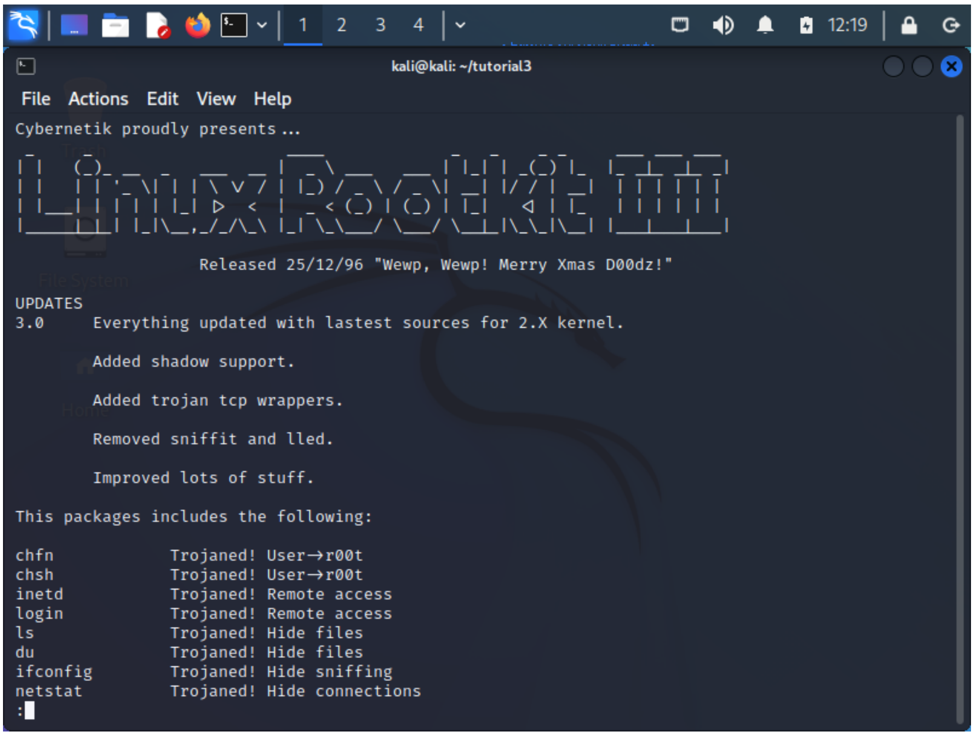


**Figure 11:** Output of `cat lrkn.2139.README | less`

### 1.8 More on icat

Let's now look at a different type of files recovered by `icat`. Recall our previous directory listing of the `.001` directory at inode 11105 (`fls -o 10260 able2.dd 11105`), depicted in Figure 12.

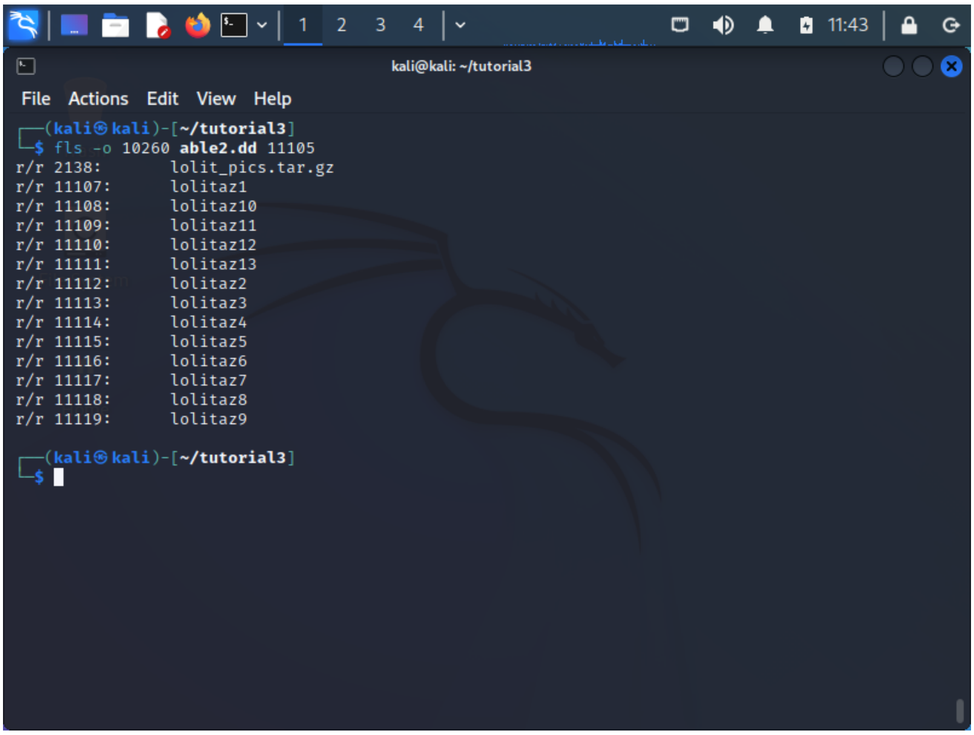


**Figure 12:** Output of `fls -o 10260 able2.dd 11105`

We can determine the contents of the (allocated) file with inode 11108 by using `icat` to stream the inode's data blocks through a pipe to the command `file`:

```
$ icat -o 10260 able2.dd 11108 | file -
/dev/stdin: GIF image data, version 89a, 233 x 220
```

The output shows that we are dealing with a picture file. You may do the same thing with the `display` command to show its contents:

```
$ icat -o 10260 able2.dd 11108 | display
```

**Note:** You might have to install ImageMagick, through the following command:

```
$ sudo apt install imagemagick
```

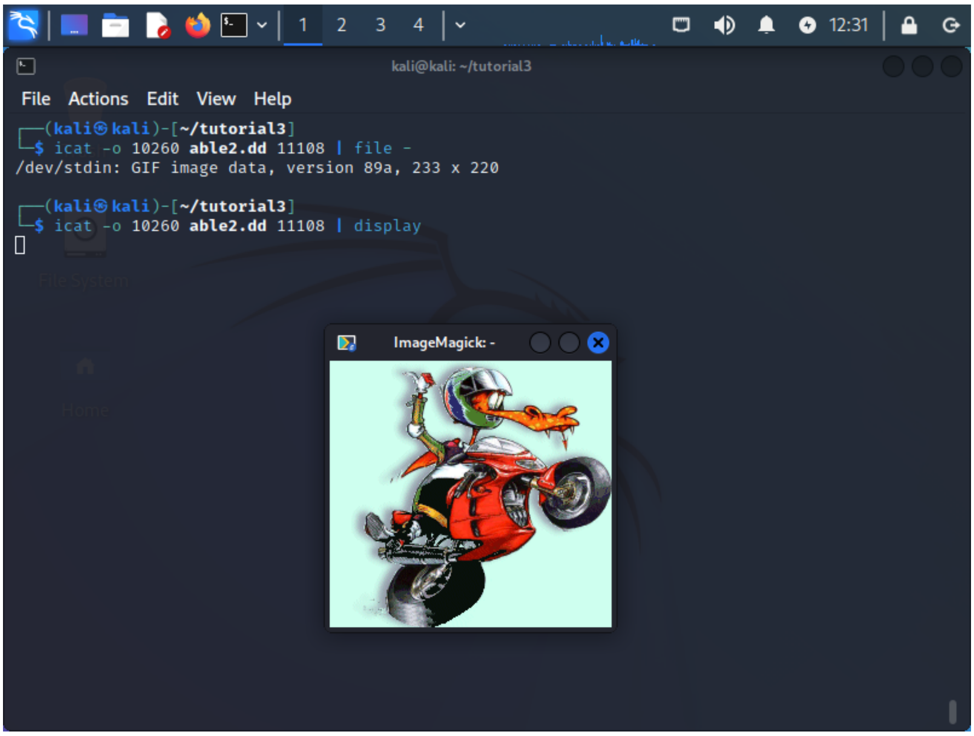


**Figure 13:** Output of `icat -o 10260 able2.dd 11108 | display`

## 2 Deleted File Identification and Recovery (Ext4)

In this exercise, you will be exposed to the limitations of the techniques introduced above when analyzing file systems where the metadata's block pointers are cleared upon deletion of a file; such is the case of Ext3 and Ext4 file systems. To demonstrate this feature, we will use the `able_3.dd` image as our target for analysis: an Ext4 file system. Before we begin, download the Ext4 disk image as follows:

```
$ wget https://turbina.gsd.inesc-id.pt/csf2324/t3/able_3.tar.gz
$ tar -xvzf able_3.tar.gz
```

This disk image is divided on a set of four split images. However, since TSK supports the inspection of individual split images, we may resort to TSK tools directly and do not need to fuse all splits of the image (something that can be accomplished by using `affuse`, for instance). To inspect the first split, start by running:

```
$ mmls able_3/able_3.000
```

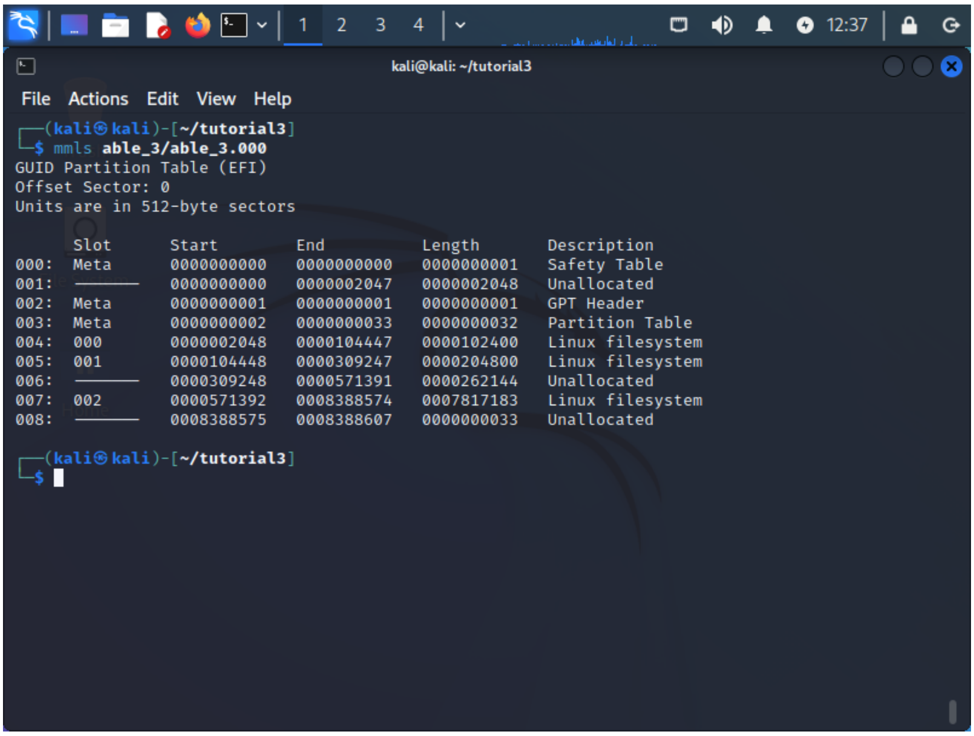


**Figure 14:** Output of `mmls able_3/able_3.000`

We are particularly interested in examining the `/home` directory, which is on the partition at offset 104448. To check which files are present in the `/home` directory, you can run:

```
$ fls -o 104448 -r able_3/able_3.000
```

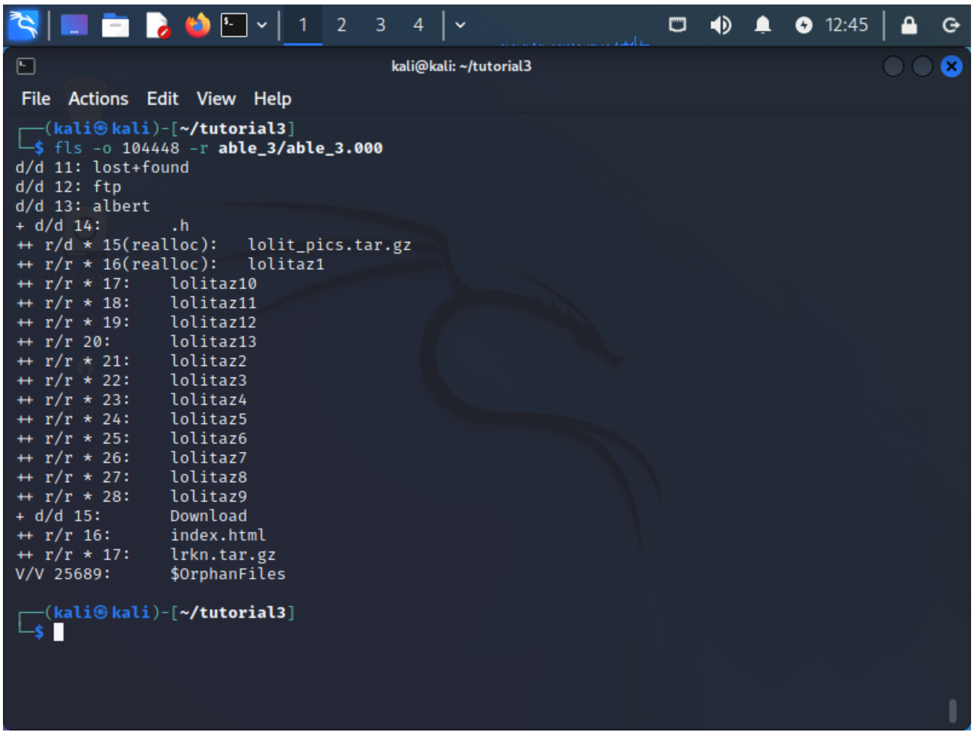


**Figure 15:** Output of `fls -o 104448 -r able_3/able_3.000`

You can see some familiar files in this output. We see a number of `lolitaz` files we saw on `able2`, and we also see the `lrkn.tar.gz` file from which we recovered the `README` from. For this exercise, we will be interested in the `lolitaz` files.

There is a single allocated file in that directory called `lolitaz13`. You can compare the output of `istat` and a follow-up `icat` command between the allocated file `lolitaz13` (inode 20), and one of the deleted files - we'll use `lolitaz2` (inode 21).

Figure 16 shows the output of `istat -o 104448 able_3.000 20`, where we observe that the inode is allocated and that the data it points to can be found in the direct blocks listed in the bottom.

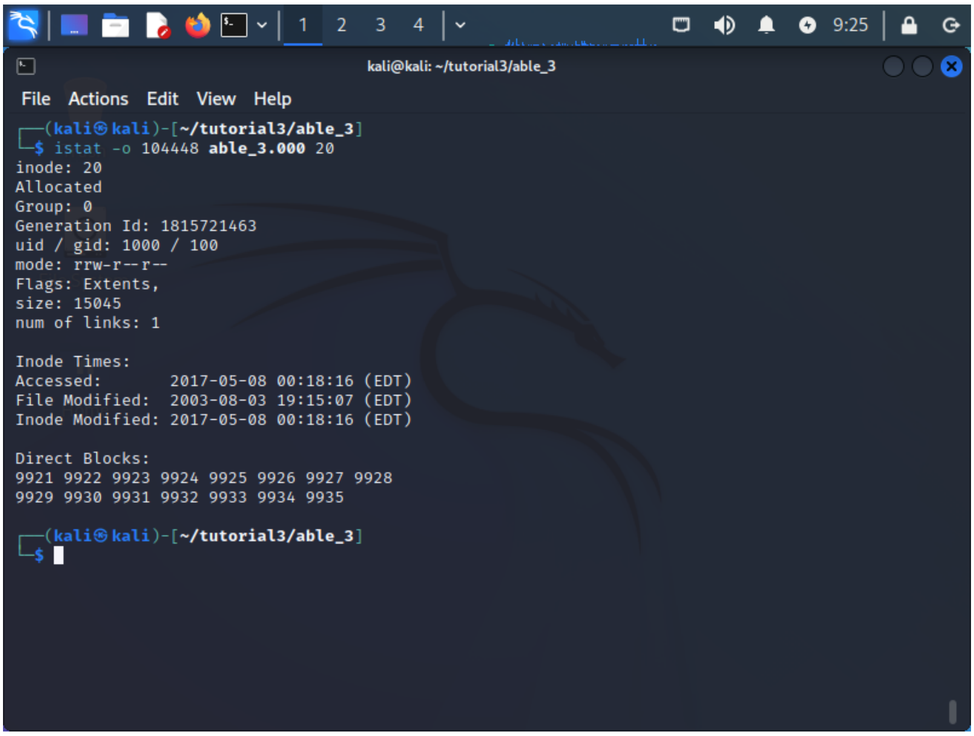


**Figure 16:** Output of `istat -o 104448 able_3.000 20`

We can now inspect the initial content pertaining to this inode by running:

```
$ icat -o 104448 able_3.000 20 | xxd | head -n 5
```

The output of this command can be found in Figure 17 and shows the expected signature of a `jpg` image.

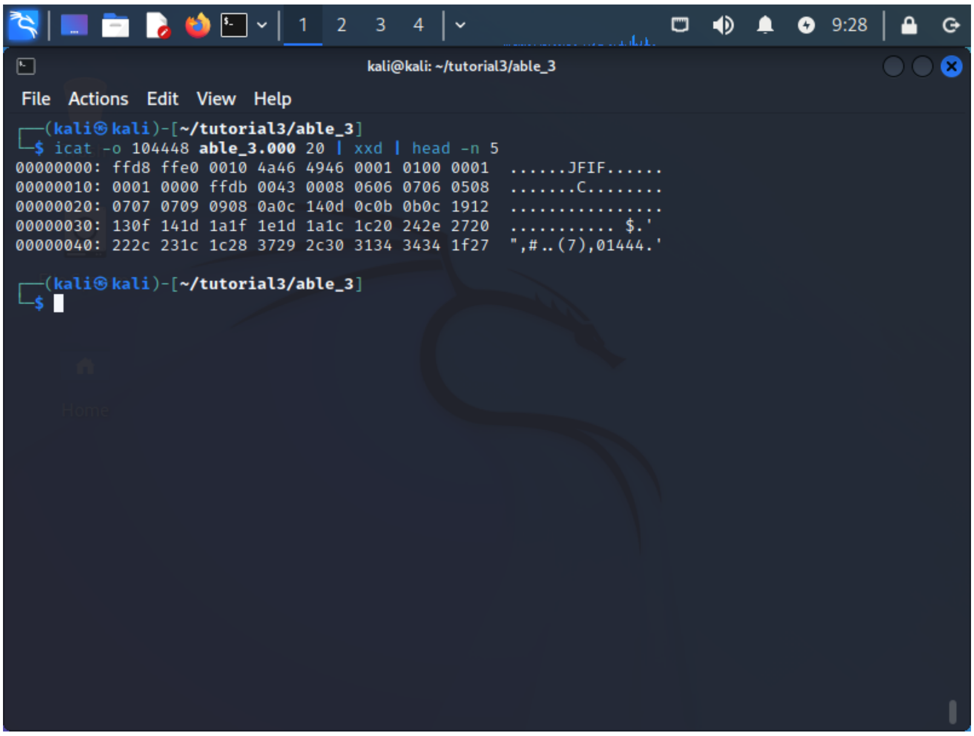


**Figure 17:** Output of `icat -o 104448 able_3.000 20 | xxd | head -n 5`

In contrast, we observe that inode 21 points to an unallocated file (Figure 18). As such, executing `icat -o 104448 able_3.000 21 | xxd | head -n 5` yields no output. On an Ext4 file system, when an inode is unallocated the entry for the Direct Blocks is cleared. There is no longer a pointer to the data, so commands like `icat` will not work. Remember that `icat` uses the information found in the inode to recover the file. In this case, there is none.

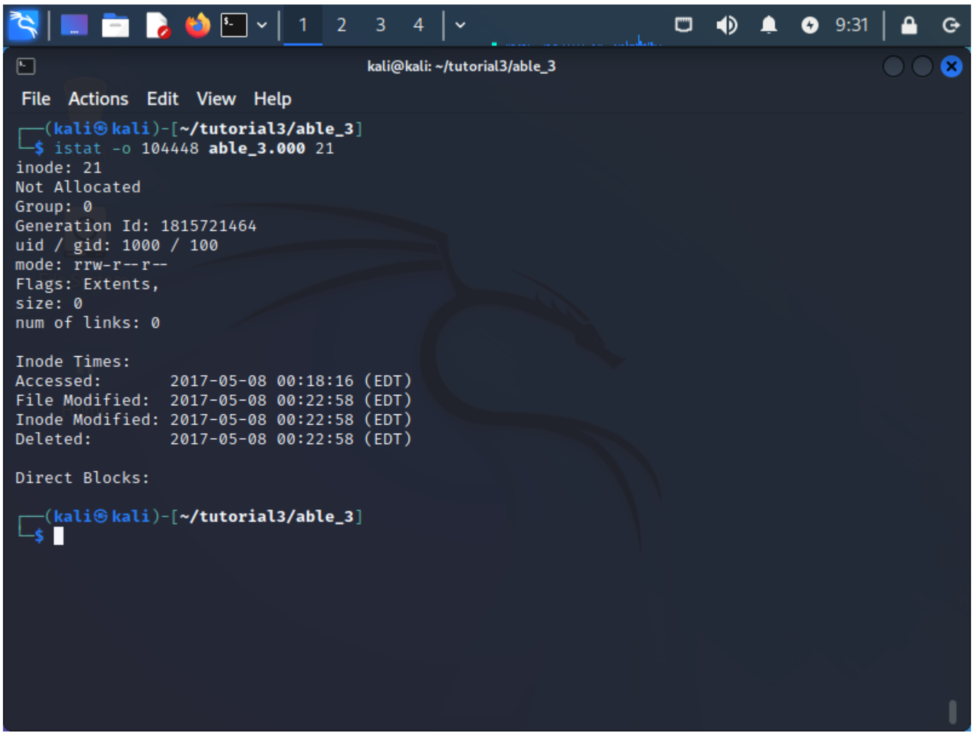


**Figure 18:** Output of `istat -o 104448 able_3.000 21`

Albeit we are unable to recover the data of unallocated files through the above method, we can still apply a number of techniques to attempt the recovery of deleted files. The next section introduces the file carving technique which we will use to recover multiple `lolitaz` files.

# 3 File Carving

File carving is a method for recovering files and fragments of files when directory entries are corrupt or missing. In a nutshell, file carving is a process used to extract structured data out of raw data present in a storage device, without the assistance of metadata provided by the file system that originality created the file. We will be using two well known carving tools, Scalpel (Section 3.1) and Foremost (Section 3.2), so as to retrieve files based on specific characteristics present in the structured data.

## 3.1 Scalpel

Scalpel is a filesystem-independent carver that reads a database of header and footer definitions and extracts matching files or data fragments from a set of image files or raw device files.

Before running Scalpel, you must define which file types are to be carved by the tool. Scalpel's configuration file (`/etc/scalpel/scalpel.conf`) starts out completely commented out (Figure 19). We will need to uncomment some file definitions in order to have Scalpel work. **Those of you using a different distribution to the one supplied may have to check online for the location of your scalpel configuration, or install it if it is not installed by default.**

First, you should copy `/etc/scalpel/scalpel.conf` to your working directory and edit it, so we can later instruct Scalpel to perform an analysis according to this configuration file. Alternatively, you may also directly edit `/etc/scalpel/scalpel.conf`.

```
$ cp /etc/scalpel/scalpel.conf .
$ open scalpel.conf
```

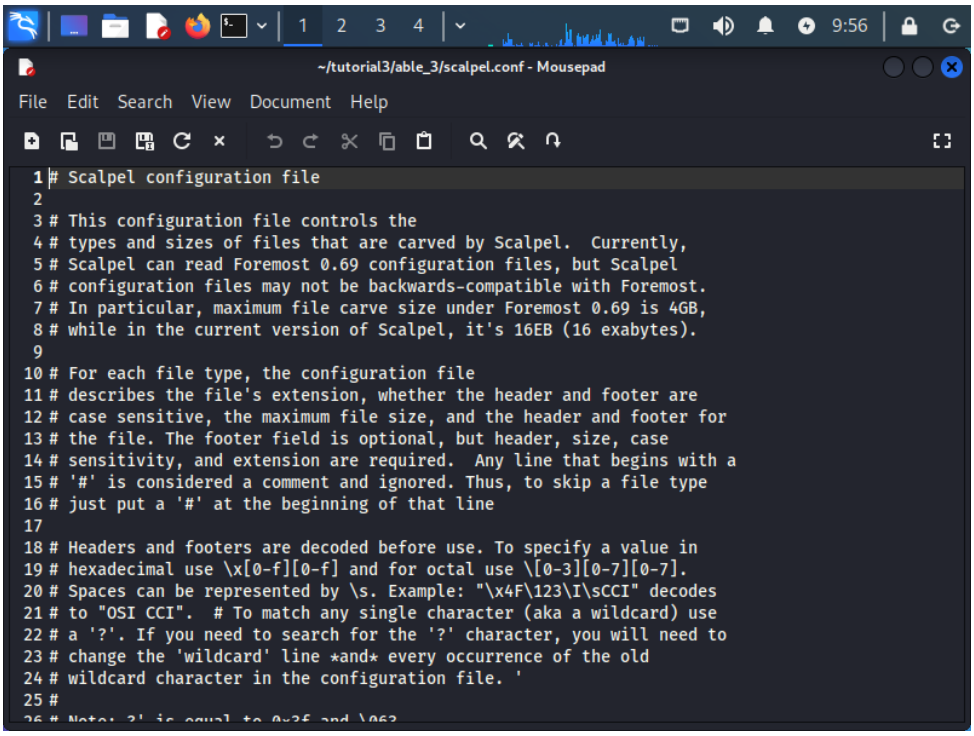


**Figure 19:** Scalpel config file

For the purpose of our exercise, scroll down to where the `#GRAPHICS FILES` section starts and uncomment every line that describes a file in that section. When we run Scalpel these uncommented lines will be used to search for patterns. That section should look like the one represented in Figure 20 when you are done.

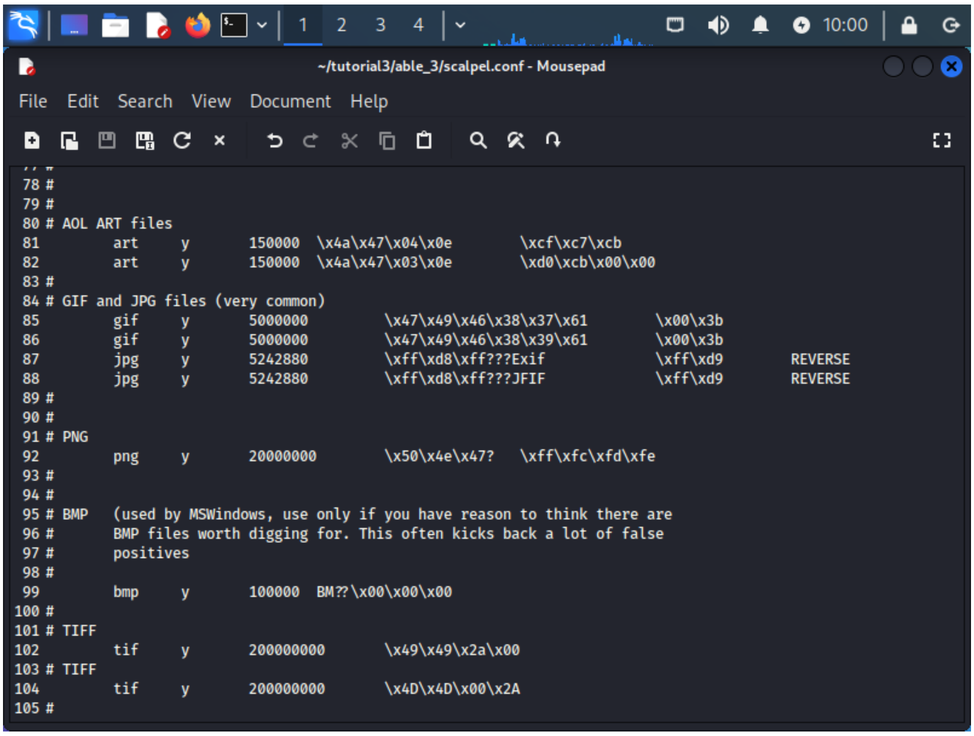

**Figure 20:** Scalpel config file changes

In this exercise, we aim to recover the `lolitaz` files found in Section 2. Since we are able to retrieve allocated files with TSK tools, we shall focus our carving process on unallocated data only. We can extract unalocated blocks by using the TSK tool `blkls`.

```
$ blkls -o 104448 able_3.000 > home.blkls
```

The `blkls` command is run with the offset (`-o`) pointing to the second Linux file system that starts at sector 104448. The output is redirected to `home.blkls`. The name “home” helps us to remember that this is the partition mounted as `/home`.

Scalpel has a number of options available to adjust the carving (be sure to check the manual!). There is an option to have Scalpel carve the files on block (or cluster) aligned boundaries. This means that you would be searching for files that start at the beginning of a data block. This should be done with caution. While you may get fewer false positives, it also means that you will miss files that may be embedded or “nested” in other files. Block aligned searching is done by providing the `-q <blocksize>` option. Try this option later, and compare the output. To get the block size for the target file system, you can use the `fsstat` command as we did in previous exercises.

In this case, we’ll use an option that allows us to specify which configuration file we want to use (`-c`). Finally, we’ll use the `-o` option to redirect our carved files to a directory we are going to call `scalp_out` and the `-O` option so that the output remains in a single output directory instead of categorized sub directories. Having the files in a single folder makes it easier to inspect recovered evidence. With everything put together, you may run:

```
$ scalpel -c scalpel.conf -o scalpel_out -O home.blkls
```

The output of the above command (Figure 21) shows scalpel carving those file types in which the definitions were uncommented.

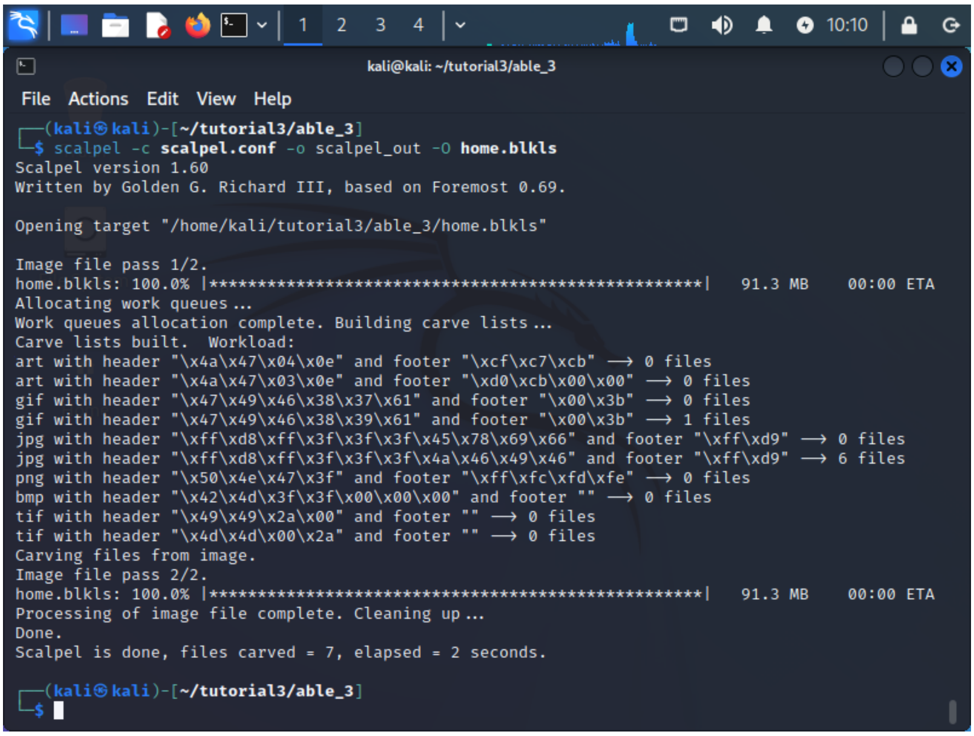


**Figure 21:** Scalpel usage

Once the command completes, a directory listing shows the carved files, and an `audit.txt` file providing a log with the contents of `scalpel.conf` and the program output (Figure 22). At the bottom of the output is our list of carved files with the offset the header was found at, the length of the file, and the source (what was carved). The column labeled Chop would refer to files that had a maximum number of bytes carved before the footer was found.

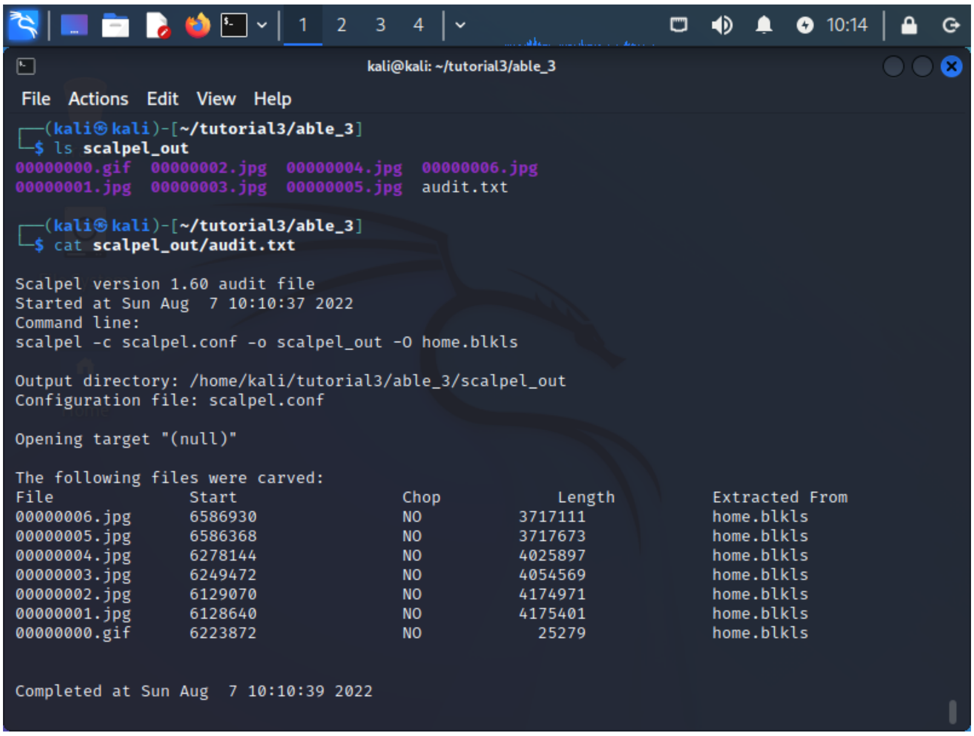


**Figure 22:** Scalpel output files

However, there are other files to be found in this unallocated data. To illustrate this, let's look at the `scalpel.conf` file again and add a different header definition for a bitmap file. Open `scalpel.conf` with your text editor and add the following line (highlighted in Figure 23) under the current `bmp` line in the `#GRAPHICS FILES` section:

```
        bmp        y         300000   BM??\x04\x00\x00
```

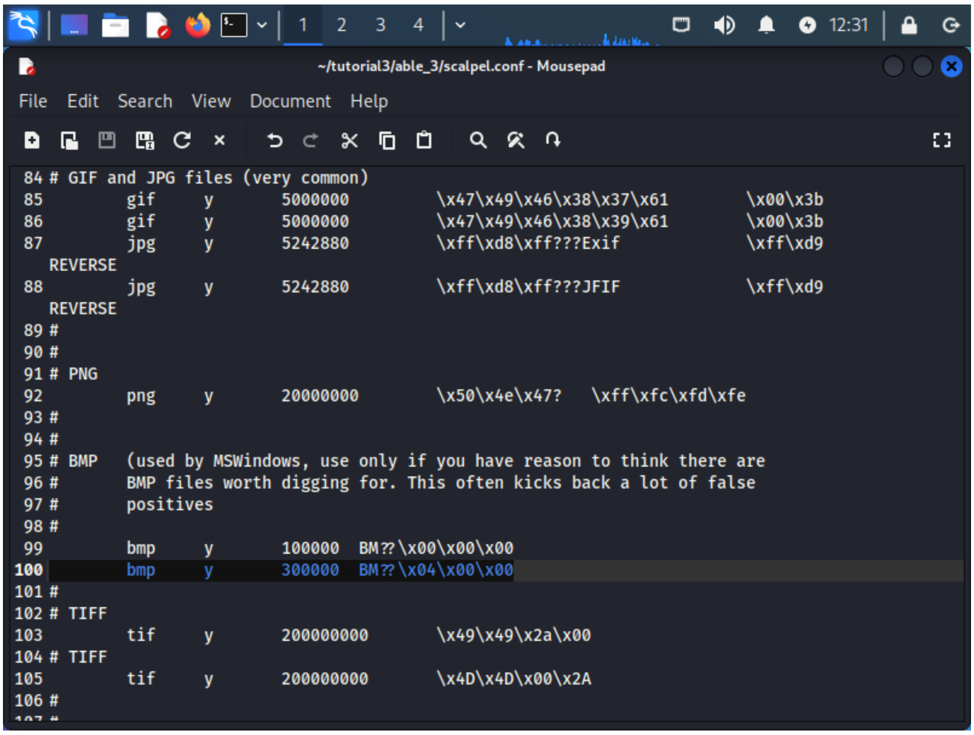


**Figure 23:** Scalpel config file changes

Here we changed the max size to 300000 bytes, and replaced the first `x00` string with `x04`. Save the file, re-run scalpel (write to a different output directory – scalpel_out2), and check the output (Figure 24).

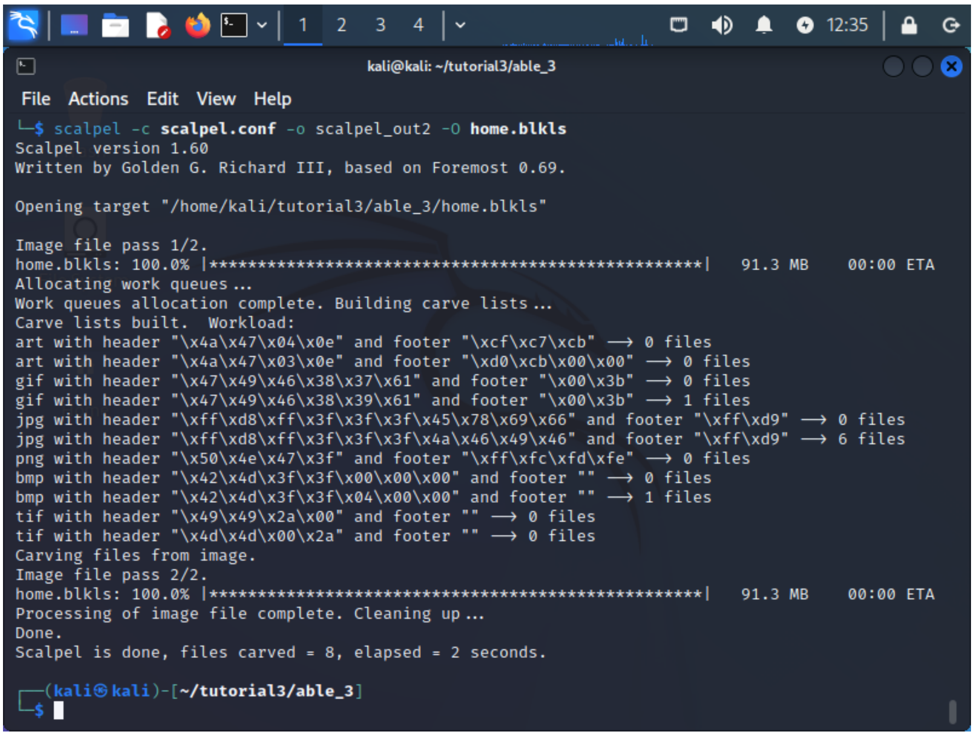


**Figure 24:** New scalpel output

Looking at the output above, we can see that a total of eight files were carved this time. The bitmap definition we added shows the `scalpel.conf` file can be easily improved on. Simply using `xxd` to find matching patterns in groups of files can be enough for you to build a decent library of headers, particularly if you come across many proprietary formats.

## 3.2 Foremost

File carving can be approached with a variety of tools, such as `foremost`, which also retrieves files by examining its internal structure. In this exercise, refer to `foremost` manual[1] and online walkthroughs to carve out the files from `home.blks`. Does `foremost` yield the same results as Scalpel?

[1] `https://www.systutorials.com/docs/linux/man/8-foremost/`