



INSTITUTO SUPERIOR TÉCNICO

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

FORENSICS CYBER-SECURITY

MEIC, METI

Tutorial 1

Introduction to File Forensics

2024/2025

nuno.m.santos@tecnico.ulisboa.pt

Introduction

This guide will introduce you to some basic tools used in file forensics and steganalysis. File forensics is an area of digital forensics that focuses on the analysis of file contents with the objective of uncovering traces or even evidence of malicious activities. This involves not only the analysis of visualizable data, such as images, videos or text, but also the analysis of raw, encoded or even encrypted data.

Steganalysis, a subset of file forensics, focuses specifically on identifying concealed data or files within seemingly innocuous files. By revealing these concealed elements, steganalysis plays a crucial role in identifying clandestine communication channels, uncovering hidden messages, or exposing covert operations.

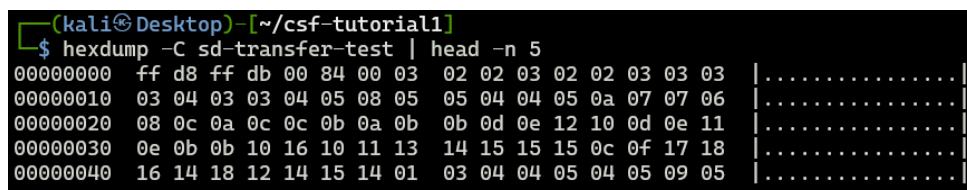
Throughout this tutorial, you will be taking the role of a forensics analyst, investigating a fictional case. The investigation you will be performing targets Eduardo Toca de Neve, suspect of leaking highly classified information from the National Salad Assembly (NSA). The authorities have successfully apprehended an SD card from Eduardo's home office and created a copy of the files found within. As the forensics analyst for this case, your job is to analyze these files in search of evidence of the suspected crime.

1 Identifying Magic Numbers

Let's begin by analyzing the file named "sd-transfer-test". One noticeable aspect is that this file lacks an extension indicating its file type. However, this means we must rely on the file's contents to determine its extension. Fortunately, most file formats make it straightforward to perform this analysis by defining unique signatures known as magic numbers. These magic numbers consist of a small sequence of bytes typically located at the beginning of a file.

To start our analysis, we need to extract the magic numbers from the file. Magic numbers are usually represented in hexadecimal format. Thus, we require a tool that allows us to view the file's contents in hexadecimal, such as the `hexdump -C` command. Optionally, since we know the magic numbers should be within the first few bytes of the file, we can filter our output to display only the initial lines of `hexdump`'s output. This can be achieved using a pipe ¹ to the `head` command:

```
$ hexdump -C {file} | head -n {number of lines}
```



```
(kali@Desktop)~[~/csf-tutorial1]
$ hexdump -C sd-transfer-test | head -n 5
00000000 ff d8 ff db 00 84 00 03 02 02 03 02 02 03 03 03 |.....|
00000010 03 04 03 03 04 05 08 05 05 04 04 05 0a 07 07 06 |.....|
00000020 08 0c 0a 0c 0c 0b 0a 0b 0b 0d 0e 12 10 0d 0e 11 |.....|
00000030 0e 0b 0b 10 16 10 11 13 14 15 15 15 0c 0f 17 18 |.....|
00000040 16 14 18 12 14 15 14 01 03 04 04 05 04 05 09 05 |.....|
```

Figure 1: Output of `hexdump` and `head` commands on `sd-transfer-test`

Once we have the magic numbers, we can search for the associated file format in signature lists such as the one found at https://en.wikipedia.org/wiki/List_of_file_signatures.

Additionally, some file formats also include a signature marking the end of the file, which, while not as useful to identify the type of a file, may be useful to detect problems in a file, as well as data that might have been hidden at the end of the file. To check the end of a file easily, you may use a similar command as the previous, but using the `tail` command:

```
$ hexdump -C {file} | tail -n {number of lines}
```

¹<https://www.geeksforgeeks.org/piping-in-unix-or-linux/>

```

(kali@Desktop)~[~/csf-tutorial1]
$ hexdump -C sd-transfer-test | tail -n 5
00028de0 ea 73 22 48 27 96 3c ed 6e 7d 29 25 ba 99 a4 51 |.s"H'.<.n})%...Q|
00028df0 e6 6d c9 ed c5 24 5f 7c d4 52 ff 00 ad 8f ea 28 |.m...$_.R....C|
00028e00 64 ec 74 76 6c 66 84 6e ea 3a f0 3a d4 db 07 a5 |d.tvlf.n.:...|
00028e10 41 a7 fd c6 fa 9a b3 5c ed ea 6e 7f ff d9 |A.....\..n...|
00028e1e

```

Figure 2: Output of hexdump and tail commands on sd-transfer-test

Lastly, while understanding this process is important, it's often unnecessary to perform it manually as there are tools available to automate this analysis. One such tool is the `file` command, included in Kali-Linux.

2 Finding Readable Strings

Next, we are going to analyze the file named “cat.jpg”. Initially, the file appears to be a simple image of a cat. However, there are many techniques that can be used to conceal information in image files without making any easily noticeable changes.

When conducting this type of open-ended analysis, it is usually recommended to start with the simplest methods. One such method involves searching for readable strings within the file's contents. By performing this search, we may quickly discover any injected information within any part of the file's contents.

In Kali-Linux, there is a command that accomplishes this: `strings`. By default, this command outputs all sequences of 4 or more ASCII characters found within a file's contents. Go ahead and try executing the `strings` command on the `cat.jpg` file.

```

(kali@Desktop)~[~/csf-tutorial1]
$ strings cat.jpg
JFIF
, #&'*)
-0-(0%()C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
+9V
g-(K
d]$E
YevD
W+!Q
%uDD
&4%F
}UU}uG
uY_Fp
fL,H
wT9w
pWac
9Qm_ms

```

Figure 3: Output of strings command on cat.jpg

As you may have noticed, the vast majority of strings shown appear random and of no interest, making manual search quite challenging. This is expected since there is a high probability that sequences of 4 or more bytes correspond to ASCII codes.

To simplify our search, we can either increase the minimum string length to search for, or filter the default output of the command for specific words. To increase the minimum string length to 8, for example, simply add `-n 8` to the `strings` command. Try it out and see if you can uncover any secrets this time.

Alternatively, if you have any leads or even just guesses about what might be hidden, you can use, for example, the `grep` command to search for specific substrings within the output of the `strings` command.

The easiest way to do this is by passing the output of one command to the other through a pipe, like this:

```
$ strings {file} | grep {substring}
```



```
(kali@Desktop)-[~/csf-tutorial1]
$ strings cat.jpg | grep 42
<42?
42\u0000
```

Figure 4: Output of strings command on cat.jpg, filtered by grep

Note: Additionally, you can use a variety of useful options for grep to facilitate your search, such as `-i` to perform a case-insensitive search, or `-B {n}` and `-A {n}` to view **n** lines before or after any line that matches the search.

3 Cracking Passwords

The next file we’re going to analyze is named “protected.zip.” Although we can view the contents of the zip file, extracting them requires us to first figure out the password used to encrypt it.

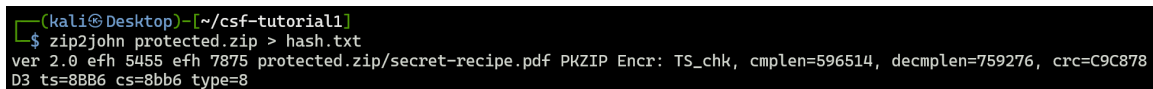
Fortunately (or unfortunately), weak and predictable passwords are quite common, making them relatively easy to guess when using the right tools. One of these tools is John the Ripper, which enables us to test a large number of passwords quickly and easily. You can install it by executing the following command:

```
$ sudo apt install john
```

To use the simplest mode of John the Ripper, all we need to provide is the hash of the password we want to discover. John functions by hashing potential passwords until one of the obtained hashes matches the supplied one.

The first step is to extract a hash from our file. Encrypted file formats commonly store the hashes of the passwords used, allowing attempted passwords to be verified before attempting to use them to decrypt the file contents. To extract the hash from our ZIP file, we only need an auxiliary tool included in the John installation, called `zip2john`.

```
$ zip2john {zip file} > {output/hash file}
```



```
(kali@Desktop)-[~/csf-tutorial1]
$ zip2john protected.zip > hash.txt
ver 2.0 efh 5455 efh 7875 protected.zip/secret-recipe.pdf PKZIP Encr: TS_chk, cmplen=596514, decmplen=759276, crc=C9C878
D3 ts=8BB6 cs=8bb6 type=8
```

Figure 5: Output of zip2john

With the hash file ready, we can now execute the simplest mode of John the Ripper, the “incremental mode.” In this mode, John will try all possible character combinations, theoretically being able to find the true password eventually, but it might take an impossibly long time to do so. You can try it out by executing the following command:

```
$ john --progress-every=1 --incremental {hash file}
```

```

(kali@Desktop)-[~/csf-tutorial1]
$ john --progress-every=1 --incremental hash.txt
Using default input encoding: UTF-8
Loaded 1 password hash (PKZIP [32/64])
Will run 16 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:00:01 0g/s 378092p/s 378092c/s 378092C/s budgail..090364
0g 0:00:00:02 0g/s 1872kp/s 1872Kc/s 1872KC/s lostrys..gubnk
0g 0:00:00:03 0g/s 3788kp/s 3788Kc/s 3788KC/s knkh26..munsh111
0g 0:00:00:04 0g/s 5314kp/s 5314Kc/s 5314KC/s 14955200..brzef
0g 0:00:00:05 0g/s 6553kp/s 6553Kc/s 6553KC/s fsosu..mocobin
0g 0:00:00:06 0g/s 7485kp/s 7485Kc/s 7485KC/s arinki3..abcdcen
0g 0:00:00:07 0g/s 8213kp/s 8213Kc/s 8213KC/s 08r..shmysin0
0g 0:00:00:08 0g/s 8789kp/s 8789Kc/s 8789KC/s jmk1..aiwdo
0g 0:00:00:09 0g/s 9353kp/s 9353Kc/s 9353KC/s lnehj..knmrq
0g 0:00:00:10 0g/s 9895kp/s 9895Kc/s 9895KC/s lyk0353..lmm5lah
0g 0:00:00:11 0g/s 10297kp/s 10297Kc/s 10297KC/s av31s3..baaumd
0g 0:00:00:12 0g/s 10657kp/s 10657Kc/s 10657KC/s efi102..efsm9j
0g 0:00:00:13 0g/s 10931kp/s 10931Kc/s 10931KC/s m1siats..m188mm6
0g 0:00:00:13 0g/s 10914kp/s 10914Kc/s 10914KC/s m1siats..m188mm6
Session aborted

```

Figure 6: Output of John in incremental mode

Note: The option `--progress-every={n}` is not necessary and serves only to print a line with some information about John's progress every **n** seconds.

Since discovering the password using this mode might take a long time, let's stop John's execution by pressing "q" or CTRL+C, so we can attempt to use a more efficient mode, the "wordlist mode." To use the "wordlist mode," we need to supply John with an additional file called a word list, essentially a text file composed of several strings to be tested as passwords, one string per line.

While you can create your own wordlist, you can also find powerful wordlists online. One such wordlist is the "rockyou" password list, originating from a security breach that leaked 32 million real passwords. To obtain this wordlist in Kali, you simply need to install the "wordlists" package and unzip the file "rockyou.txt.gz."

```

$ sudo apt install wordlists
$ sudo gunzip -d /usr/share/wordlists/rockyou.txt.gz

```

After doing this, you should see a file named "rockyou.txt" under "/usr/share/wordlists/" with a size of 134MB. You can verify this by running the following command:

```

(kali@Desktop)-[~/csf-tutorial1]
$ ls -lh /usr/share/wordlists/
total 134M
lrwxrwxrwx 1 root root 28 Apr 2 18:39 john.lst → /usr/share/john/password.lst
lrwxrwxrwx 1 root root 41 Apr 2 18:39 nmap.lst → /usr/share/nmap/nselib/data/passwords.lst
-rw-r--r-- 1 root root 134M May 12 2023 rockyou.txt

```

Figure 7: /usr/share/wordlists after unzipping rockyou

Now that we have our wordlist ready, we can execute John the Ripper in wordlist mode with the following command:

```

$ john --wordlist=/usr/share/wordlists/rockyou.txt {hash file}

```

This process may take some time depending on the complexity of the password and the size of the wordlist. You can monitor John's progress by observing the output, which will display any passwords it finds. Once John successfully cracks the password, it will display it on the screen. You can then use this password to extract the contents of the "protected.zip" file.

Remember, while using wordlists like "rockyou.txt" can be effective for cracking passwords, it's not guaranteed to work for every case. If the password is unique or not in the wordlist, you might need to

resort to more advanced techniques or custom wordlists.

4 Identifying Encodings

Finally, we will analyze the “broken.pdf”. First, we should verify if the file truly is what it claims to be – a PDF. To do this, let’s start by running the file command on it.

```
(kali@Desktop)~[~/csf-tutorial1]
$ file broken.pdf
broken.pdf: ASCII text, with very long lines (65536), with no line terminators
```

Figure 8: Output of file command on broken.pdf

According to the output of the file command, the file does not seem to be a PDF, but instead a text file. This is suspicious, so we should further investigate the type of the file. We will do this by checking if the first bytes of the file resemble PDF magic numbers.

```
(kali@Desktop)~[~/csf-tutorial1]
$ xxd broken.pdf | head -n 1
00000000: 2532 3550 4446 2532 4431 2532 4537 2530  %25PDF%2D1%2E7%0
```

Figure 9: Output of xxd command on broken.pdf

Through a simple online search, we can see that the magic numbers we expect to find for a PDF are 25 50 44 46 2d, which translate to %PDF- in ASCII. By comparing the first bytes of our file with the expected magic numbers, we see that, while they are not a direct match, they do have similarities. This might suggest that the file was originally a PDF but suffered some form of encoding, which made its magic numbers become those that we observe.

Encoding is the process of converting data from one format to another, for a variety of purposes. Table 1 provides examples of how the string ‘Hello, World!’ appears when encoded using various common encoding methods.

Encoding	Example
ASCII	Hello, World!
Base58	72k1xXWG59fYdzSNoA
Base64	SGVsbG8sIFdvcmxkIQ==
URL encoding	Hello,%20World!
Quoted-Printable	Hello,=20World!
Caesar Cipher (ROT13)	Uryyb, Jbeyq!

Table 1: Examples of common encoding types

Each encoding has specific rules for representing data, and it is these rules that may help us identify when they are being used. For example, Base64 encoded text may be easily identified through its alphabet of characters, which includes uppercase and lowercase letters, numbers, and most importantly, the characters +, / and =. Therefore, one effective way to identify an encoding is to compare the unknown string against lists of common encodings, examining which alphabet and rules best match the string in question.

Index	Binary	Char.	Index	Binary	Char.	Index	Binary	Char.	Index	Binary	Char.
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
									Padding		=

Figure 10: Base64 alphabet defined in RFC 4648

Using resources like the “Binary-to-text encoding” Wikipedia page ² as reference of several types of encoding and their corresponding alphabets and rules, discover the type of encoding that was used on the broken.pdf and decode it.

Report your findings

To conclude, after uncovering all the secrets within the acquired files, it is important to document and analyze our findings through a Digital Forensic Report. The following exercise, as well as the provided solution, serve as an example for your future assignments in the Forensics Cyber Security course, so we recommend that you review them carefully.

Exercise

Use the template that can be found in the course page to produce a report where you present all your findings and answer the three questions presented below. Justify your answers by presenting all the relevant evidence you found. Make sure to explain your hypotheses and how you have validated them.

Note: Remember, it is better to have hypotheses than false certainties.

1. Based on your analysis of the documents, can you conclude anything regarding the likely identity of the owner of this storage device? Justify your answer with relevant findings.
2. Based on the secrets you recovered, is there any indication of illicit activities involving the acquired SD card? If there's no direct evidence of any such activity, how would you interpret the data? Formulate a hypothesis regarding their purpose and justify it using the content of the recovered secrets.
3. Given your discoveries, what would be your recommendations for the subsequent course of action?

You may find an example of a report for this exercise on the course website in Course Material > Lab guides. Additionally, although not all assignments ask that you produce a timeline of events, a timeline example for this exercise is also provided on the course website.

²https://en.wikipedia.org/wiki/Binary-to-text_encoding