INSTITUTO SUPERIOR TÉCNICO

Departamento de Engenharia Informática

Forensics Cyber Security

MEIC / METI 2017-2018 – 1st Semester

# Digital Forensics Report

**Luís Aguiar 80950 & Jorge Pereira 81428 & Filipe Azevedo 82468**

## 1 Objectives of the investigation

This report aims to spell out the entire forensic investigation process in the case of Ed Winden, whose company he has been working on, suspects that he has stolen information about their revolutionary new processor. In the next sections will be explained all the steps taken since the delivery of the files recovered from a pen drive of Ed Winden himself after his trip to Lisbon. The goal is to find 4 possible hidden secrets in these files.

## 2 Artifacts for analysis

The files that have been delivered to us, apparently consist of 4 images, a text file, a zip file and a program written in python. We calculated the fingerprint using md5 to verify that the files were not changed before they were delivered to us. And the results obtained were as follows.

```
d6f38d287f9f8b3fa6c613503f5a1619  online_banking.zip
760ff6f302a6b9132983aedc9b51af1e  LOTR-Part1.txt
ccd086ec7f8d450c8d091cb6b8fe0b0c  lisbon.png
a0610a1756bc34f251716ba36b5aa197  jeronimos.png
dacadbf11a63a09fca89da083c12ac37  rossio.png
48d4eb13f4760bb18f44d425568936e0  electrico.png
ee4ce345a9519d9b3f22109828f26aac  compress.py
```

*Figure 1 - MD5 Fingerprint of all the files*

## 3 Evidence to look for

After having verified the integrity of the files, we devote our attention to the python program. We ran it, and we got the message that you can see on the right side. So we can conclude that Ed Winden had in his possession a program to hide files in images in the least significant bits (LSB image steganography). Therefore, we are hoping to find some hidden content in the images that were also delivered to us.

```
root@kali:~/Downloads# python compress.py
Ciber Securanca Forense - Instituto Superior Tecnico / Universidade Lisboa
LSB steganography tool: hide files within least significant bits of images.


Usage:
  compress.py <img_file> <payload_file> [password]

  The password is optional and must be a number.
```

*Figure 2 - Compress program running*

## 4 Examination details

With the information obtained previously from the program **compress.py**, we decided to move to the zip file to try to extract it to get more files. And with some surprise we come across a zip file protected with a password.

Since the zip file is protected, we started looking at the text file. We did some scrolling and some diagonal reading and it seemed, in fact, the first book of the *Lord of the Rings*, as the name indicated. We then accessed our arsenal of tools on Kali Linux, and used this text file to guess the password of the zip file. Since the **fcrackzip** tool needs a file with one word per line, sorted in alphabetical order, we executed the following commands to try to use the *Lord of the Rings* book as a dictionary of possible passwords.

```
root@kali:~/Downloads# grep -o -E '\w+' LOTR-Part1.txt > dictionary.txt
root@kali:~/Downloads# sort dictionary.txt | uniq  > passwords.txt
root@kali:~/Downloads# fcrackzip -v -D -p passwords.txt online_banking.zip
found file 'online_banking.docx', (size cp/uc  12936/ 22313, flags 9, chk 5e0b)
found file 'arm1_floorplan.bmp', (size cp/uc 147677/160246, flags 9, chk b0e9)
possible pw found: Frodo ()
```

*Figure 3 - Cracking the password of the zip file*

In the first step we used **grep** to get all words from the file, one per line. Next with the commands **sort** + **uniq**, we sorted the words by alphabetical order, and then removed duplicates to increase performance. Finally, we use that file with the **fcrackzip** tool, and we obtained "Frodo" as a possible password. So, we tried to unzip the file with that password, and we succeeded! Inside that zip we found two files:

- Apparently, a document created with Microsoft Word with the name "*online_banking.docx*" (with md5: b70702822417bd39a7997a0f8c73941f);
- A bmp image named "*arm1_floormap.bmp*" (with md5: f233a382d996fcc58601af79af167f2b).

We then decided to use the **file** command to verify that the files were exactly what the name indicated.

```
root@kali:~/Downloads# file arm1_floorplan.bmp online_banking.docx
arm1_floorplan.bmp:  data
online_banking.docx: Microsoft Word 2007+
```

*Figure 4 - Checking the type of the files inside the zip*

However, a strange thing happened. Although the Microsoft Word document indicated to be a Microsoft Word document (as expected), the BMP image indicated to be a file that only contained data. We then tried to open it, but we could not, as you can see below.

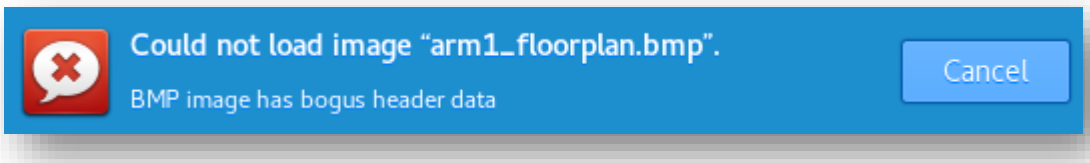


*Figure 5 - Error message given trying to open the image file*

However, the error message specified that there was an error in the header of the image and this helped us get to the next step. We opened the tool **ghex** and checked the binary of the image. We found that the file header began with ***00 00 00***, which is strange, followed by ***47 0D 0A 1A***. From here we decided to consult a website (https://goo.gl/yzGwwV) with a list of signatures of some files, and we searched for a file that had the ***47 0D 0A 01***, that followed the initial zeroes.

And, we got a match! A PNG file starts with ***89 50 4E 47 0D 0A 1A 0A***, and the ***0A***, also makes part of the header. So, we changed the ***00 00 00*** in the begging to ***89 50 4E***, using **ghex** again, and renamed the file to *"arm1_floormap.png".* We tried to open it again and we got the image showed on the side. The image appears to be a drawing of the internal components of a processor (md5: bf3ab0d53c35353b0d74eb7c2cc2e5de).

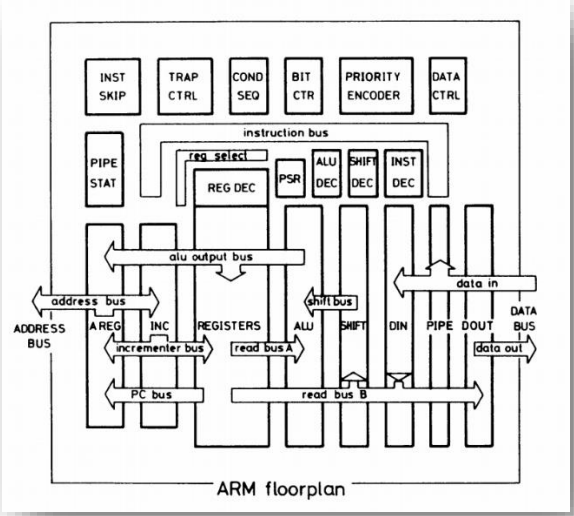


*Figure 6 - Image hidden in the zip file*

After having found this, we decided to continue the investigation. We looked at the Microsoft Word document that was also inside the zip, and it only contained one thing written "Password: 51782". As we learned in class that a doc file was nothing more than a zip file, we decided to extract it. We looked at all the files inside the extracted document and apparently everything was normal, just a bunch of xml files and a thumbnail of the document.

From here we decided to move again to the python program. Once we had run it, we decided to try opening the file to see the code. However, we were unlucky: the program was compiled. But, since it was written in python, it is relatively easy to find a decompiler, and that's what we did. After googling we found a tool called **uncompyle** (https://goo.gl/o4cFzB). We ran the compress program with it, and, eureka! We got the original program decompiled (md5: 6de42c79f6b476e8bfc8f25cac09b553).

```
root@kali:~/Downloads# uncompyle2 compress.py > compress_decompiled.py
```

*Figure 7 - Decompiling the compress program*

After analyzing the code, we concluded, as was said when we tried to run the program, that the program had as the input: an image, a file, and a password. The program hides the second parameter (the file) in the least significant bits of the image, changing its order. For example, if the file we want to hide is ***01***, the result image would be ***10***. But, before starting to hide the content itself, the size of the content that is hidden is placed in the beginning. Optionally, when a password is provided, what it does is indicate from which pixel the data starts being concealed. With this information we decided to make a program that would do exactly the opposite, and the result was the tool **obtain.py**. We then decided to run the program with the last image (*rossio.png*) that was in the files that were delivered to us, because, why not?

```
root@kali:~/Downloads# python obtain.py rossio.png > hidden_rossio.txt
```

*Figure 8 – Running the inverse of the compress program over the rossio.png image*

After this we looked over in **ghex**, and the following was obtained:

```
0000000089 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52.PNG........IHDR
0000001000 00 01 2C 00 00 01 2D 08 06 00 00 00 B2 21 5D...,...-......!]
00000020D0 00 00 00 01 73 52 47 42 00 AE CE 1C E9 00 00.....sRGB.......
0000003000 09 70 48 59 73 00 00 0B 13 00 00 0B 13 01 00..pHYs..........
000000409A 9C 18 00 00 01 59 69 54 58 74 58 4D 4C 3A 63......YiTXtXML:c
000000506F 6D 2E 61 64 6F 62 65 2E 78 6D 70 00 00 00 00om.adobe.xmp....
0000006000 3C 78 3A 78 6D 70 6D 65 74 61 20 78 6D 6C 6E.<x:xmpmeta xmln
0000007073 3A 78 3D 22 61 64 6F 62 65 3A 6E 73 3A 6D 65s:x="adobe:ns:me
0000008074 61 2F 22 20 78 3A 78 6D 70 74 6B 3D 22 58 4Dta/" x:xmptk="XM
0000009050 20 43 6F 72 65 20 35 2E 34 2E 30 22 3E 0A 20P Core 5.4.0">.
```

*Figure 9 – Information extracted from the rossio.png image with the obtain.py tool*

It looks like a PNG file, so we changed the name of the file to *hidden_rossio.png*, we open it, and, eureka! We found an image that again looks like a schematic of a processor (md5: 49f1a931bd2e74f24082cf997292cd27). From here we decided to apply the tool again over the *lisbon.png* image. However, this time we didn't succeed. Then we then tried applying the tool one more time, now over the *jeronimos.png* image. And a strange thing happened. We got a picture of what looks like the "Torre de Belém" (file "*torre_belem.png*" md5: 155460b6fc3ee0a2c9ee63a3dc28b535), Out of curiosity we decided to check if the image of the "Torre de Belém" contained something hidden, so, once again, we applied the **obtain.py** tool. We were right! Hidden in the image of "Torre de Belém" is a text file containing the following:

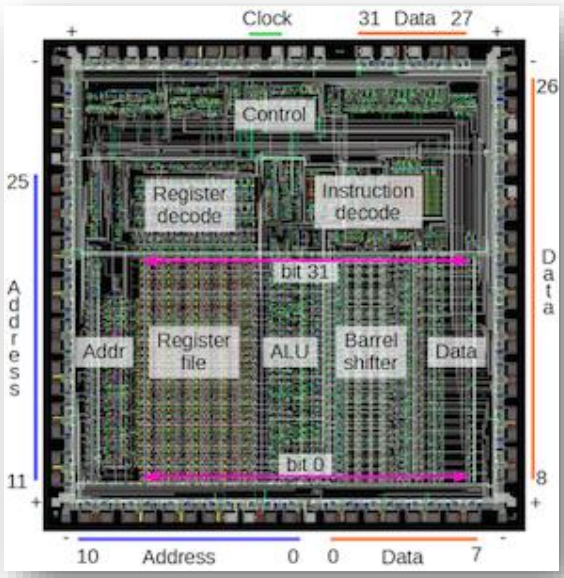


*Figure 10 - Hidden image inside the rossio image*

ARM1 chip

The ARM1 chip is built from functional blocks, each with a different purpose. Registers store data, the ALU (arithmetic-logic unit) performs simple arithmetic, instruction decoders determine how to handle each instruction, and so forth. Compared to most processors, the layout of the chip is simple, with each functional block clearly visible. (In comparison, the layout of chips such as the 6502 or Z-80 is highly hand-optimized to avoid any wasted space. In these chips, the functional blocks are squished together, making it harder to pick out the pieces.)

The diagram below shows the most important functional blocks of the ARM chip.[2] The actual processing happens in the bottom half of the chip, which implements the data path. The chip operates on 32 bits at a time so it is structured as 32 horizontal layers: bit 31 at the top, down to bit 0 at the bottom. Several data buses run horizontally to connect different sections of the chip. The large register file, with 25 registers, stands out in the image. The Program Counter (register 15) is on the left of the register file and register 0 is on the right.

Computation takes place in the ALU (arithmetic-logic unit), which is to the right of the registers. The ALU performs 16 different operations (add, add with carry, subtract, logical AND, logical OR, etc.) It takes two 32-bit inputs and produces a 32-bit output. The ALU is described in detail here.[4] To the right of the ALU is the 32-bit barrel shifter. This large component performs a binary shift or rotate operation on its input, and is described in more detail below. At the left is the address circuitry which provides an address to memory through the address pins. At the right data circuitry reads and writes data values to memory.

This text appears to be a detailed description of the operation of a processor (the file extracted from the image of the "Torre de Belém" – with the name "*hidden_belem.txt*" – will be attached to this document and it has a md5 fingerprint of: e406087f8689534997055b2eb2a64808).

We decided to apply the **obtain.py** tool on the last image (*electrico.png*) and the result was similar to the obtained result when applied to the image of Lisbon. It appeared to be nothing concealed in those images.

But we were hoping to find some more information, and it was then that we remembered that all the images created by **compress.py** had to be PNG with an RGBA color scheme. We then decided to use a tool called **pngcheck** to check the properties of all the images that were delivered to us, and as expected, both the images *rossio.png* and *jeronimos.png* followed that pattern. However, they were not the only ones. The *lisbon.png* image also followed the pattern, unlike what happened with the image *eletrico.png*, which has an RGB color scheme.

From there we decided to focus our efforts on the image of Lisbon. We ran the **obtain.py** tool again, looked at the result and apparently nothing. After this we remembered the password written on the Microsoft Word document, so we decided to use it. Then, we made another version of the **obtain.py** tool that considers the jump caused by using the password. This jump is given by the module 13 of the inserted password, multiplied by 6 (2 bits for each used color channel to hide information - RGB - in this case).

The result of this was the **obtain_lisbon.py** tool (md5: 32ca4458829617dd81d7a06b0ea9cf90). We then ran this tool on the file *lisbon.png* and the result was the text obtained below. The complete file extracted from the lisbon.png file is named *hidden_lisbon.txt* and have a md5: 68a439fadc22bf501c21e01e4ae55100.

ACCESS CODES

SERVER 1: Sr!_01llxt

SERVER 2: p_GEtKl4dA

## 5 Analysis results

After this analysis we found, in fact, business secrets of the company where Ed Winden has been working. However, we found nothing that explicitly says that it was Ed Winden himself who hid the files with the information, even though he had a tool that would allow him to do so. Also, no information was found about the servers to which the passwords hidden in the image *lisbon.png* would grant access to, which may be something to be investigated further, with access to more evidence or in contact with the company itself. In summary, the artifacts that were found in this investigation were:

| What? | Where? | MD5 |
|---|---|---|
| A drawing of the processor components. | Inside the zip file. | bf3ab0d53c35353b0d74eb7c2cc2e5de |
| A schematic of the processor components. | Hidden in the rossio.png file. | 49f1a931bd2e74f24082cf997292cd27 |
| A textual description of the processor. | Hidden inside a hidden image inside jeronimos.png file. | e406087f8689534997055b2eb2a64808 |
| Two codes to access 2 servers. | Hidden inside the lisbon.png file. | 68a439fadc22bf501c21e01e4ae55100 |

## 6 Conclusions

As was said earlier, we could not find any irrefutable evidence of Ed Winden's guilt, however, in the files that were delivered to us we found trade secrets of the company where has been working on.

25th October 2017, Instituto Superior Técnico

Luís Aguiar - 80950

Jorge Pereira - 81428

Filipe Azevedo - 82468