UiA University of Agder

Master

Detecting Packed PE Files

Executable file analysis for the Windows operating system.

KRISTOFFER RENSTRØM OLSEN

SUPERVISOR
Vladimir A. Oleshchuk

University of Agder, 2021
Faculty of Engineering and Science
Department of Engineering Sciences

https://t.me/learningnets

Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2). Oppgaver som er unntatt offentlighet eller taushetsbelagt/konfidensiell vil ikke bli publisert.

Vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:	Ja
Er oppgaven båndlagt (konfidensiell)?	Nei
Er oppgaven unntatt offentlighet?	Nei

https://t.me/learningnets

Acknowledgements

I want to thank my supervisor Vladimir A. Oleshchuk. Furthermore, I would like to thank the reverse engineering-focused youtube channel OALabs for being a good source of inspiration/knowledge [16].

https://t.me/learningnets

Summary

This thesis's intended reader audiences are computer nerds and reverse-engineers; since the topics in this thesis will be primarily focused on PE packers and their methods, it will also describe a program that can detect these packed PE (portable executable) binary files and describe the development of this very program. An experiment on around 225K files will be done, and its results will be shown in the later Chapter 4 Results. In the Chapter 2 Theory, the thesis will go into details around Windows PE files and their layout, so that the reader can get an understanding of what is inside the regular .exe file they have on their computers. It will also give an insight into different types of packers and tricks they employ for anti-debugging/anti-analyzation.

The results show that a packed PE image file has unique features that stand out like a sore thumb compared to benign PE files. Therefore, the thesis also touches on various types of packers and methods for detection and unpacking (dumping) the original PE file. However, the experiments are only statical analyses since the author does not want to deal with viruses on the loose.

https://t.me/learningnets

Abstract

Malware authors invent new methods regularly to hide and obfuscate their code. One of these methods is known as packing. An entire program is hidden inside another executable program; however, the hidden program is usually encrypted or obfuscated such that antivirus software cannot detect its real intent without unpacking it. This thesis will look into common PE packers and describe the development of an application used to detect packed PE binaries using static analysis. This thesis is useful for reverse engineers and antivirus developers; it will give some insight into the world of packing binaries, compression methods, and encryption methods. The thesis will also gather some statistics around packed PE binaries, using a prototype to analyze 225K viruses.

https://t.me/learningnets

Contents

Acknowledgements **ii**

Summary **iii**

Abstract **iv**

1 Introduction **1**
1.1 Background 1
1.2 Problem statement 2
1.3 Thesis structure 2
1.4 Related works 2
1.4.1 PEiD 3
1.4.2 PE-Bear 3
1.4.3 PE-detective 3
1.4.4 EMBER 3
1.4.5 PHAD 4
1.4.6 Detect it Easy (DIE) 4
1.4.7 MOV obfuscator 4
1.4.8 Capa 4

2 Theory **5**
2.1 State of the art 5
2.1.1 Different Packers 5
2.1.2 Multi-Thread Packers 5
2.1.3 UPX - Packer 6
2.1.4 Other Popular Packers 7
2.2 Processor Architechtures 7
2.2.1 The execution of a program 8
2.3 Important Windows APIs 8
2.3.1 CreateProcessInternalW 8
2.3.2 VirtualAlloc 9
2.3.3 VirtualProtect 11
2.3.4 LoadLibrary 12
2.4 PE Structure 12
2.4.1 Addresses: RVA, VA and Physical Addresses 12
2.4.2 IMAGE_DOS_HEADER 13
2.4.3 IMAGE_NT_HEADER 13
2.4.4 IMAGE_SECTION_HEADER 15
2.4.5 .text 15
2.4.6 .data 16
2.4.7 .bss 17
2.4.8 .tls 17

https://t.me/learningnets

3 **Methodolgy** 18
3.1 Assumptions and Limitations 18
3.2 Experimental design 19
3.2.1 Variable Explanation 20
3.2.2 Categories Of Viruses 20
3.3 Tools Utilized 22
3.3.1 Capstone 22
3.3.2 PeFile 22
3.3.3 scipy 22
3.3.4 PE-Bear 22
3.3.5 matplotlib 23
3.3.6 Ghidra 23
3.3.7 x64dbg And x32dbg 23
3.4 Expected Results 24

4 **Results** 26
4.1 Questions 26
4.1.1 Q1 - Detection 26
4.1.2 Q2 - Static Analysis 27
4.1.3 Q3 - Dynamic Analysis 28
4.1.4 Q4 - Different Methods 29
4.2 Experiment: Analyze Viruses 30
4.2.1 The script that gathers the data 31
4.2.2 Results for A1 - Entropy 32
4.2.3 Results for A2 - Execute rights 33
4.2.4 Results for A3 - EntryPoint 33
4.2.5 Results for A4 - DLLs 34
4.2.6 Results for A5 - Sections 35
4.2.7 Results for A6 - Section sizes 36
4.2.8 Results for A7 - String in Binary 37
4.2.9 Summary Results 38
4.3 The Prototype 39
4.3.1 Pefile 39
4.3.2 Capstone 39

5 **Discussion** 42
5.0.1 Method discussion 42
5.0.2 Result discussion 43
5.0.3 Other solutions 43
5.0.4 What to take away 44

6 **Conclusion** 45
6.0.1 The solution 45
6.0.2 Results 46
6.0.3 Difference compared to other projects 46
6.1 Future Work 46

A **Abbreviations and Glossaries** 47

Bibliography 49

https://t.me/learningnets

List of Figures

1.1 Realia Spacemaker "Shrinks your COM and EXE files" [34] 1
1.2 Pe-detective by Erik Pistelli [33] 3
1.3 The general process of a Packer [36] 4

2.1 An visual example of malicious malware "packed" in different methods [3] a) is a standard program b) Is a normal program with the virus attached to the PE file c) A compressed PE file with the decompressor inside the PE file d) Same as c but with the virus encrypted e) A program with multiple layers of encryption and compression . 6
2.2 Aegis Crypter Tool [16] 8
2.3 Detection rate of packers from black hat briefings 2006 [4] 9
2.4 Simple program that prints a string compiled with mingw-gcc . . . 10
2.5 Reflective DLL Loading detection by Windows Defender [45] 11
2.6 The two sections .text and .rdata in a bening PE file 16

4.1 The DLLs imported by Viruses 35
4.2 Figure displaying the common amount of sections in viruses. sample size of 194K viruses 36
4.3 Results of scanning the System32 folder on windows 10 37

https://t.me/learningnets

List of Listings

1	LoadLibaryA	12
2	The _DOS_HEADER from winnt.h header file in windows 10	13
3	_IMAGE_NT_HEADER for a PE file, found from winnt.h	14
4	File header found from winnt.h	14
5	Section header found from winnt.h	15
6	Optional header found from winnt.h	17
7	Example of Signature from PE-Bear [17]	22
8	Signature script used in experiments.	23
9	Example output of the custom made python script	25
10	Ghidras interpretation of assembly code seen in Figure 2.4	28
11	Bash script for analyzing all the viruses.	31
12	Entropy calculation	32
13	Str resolving function,	41

https://t.me/learningnets

List of Tables

3.1 Experiment Variables 19

4.1 Vulnerability Overview 32
4.2 Entry point / Section rights Statistics 34
4.3 DLL Imports - Viruses 35
4.4 String statistics 38
4.5 Main Results 38
4.6 The prototype's results compared with a simple signature detector . 39

https://t.me/learningnets

Chapter 1

Introduction

1.1 Background

In the computer world we live in, we use all kinds of software to fulfill our needs. Developers write the software we use, and these developers can be anyone in the entire world. Therefore a user exposes himself to a considerable risk since any program can be either malicious or benign. As such, there has been developed software to detect these malicious pieces of software. Such as antivirus software, these programs usually scan the entire computer, every single file, looking for malicious behavior; the antivirus programs also scan every new file downloaded to the computer's disk or programs that are loaded into RAM. The malicious software creators have to dodge these antivirus gazes; one of these methods this thesis will discuss is packing. The basic idea of packing can be seen in Figure 1.3

Packing of software is by no means a new phenomenon, as you can see in Figure 1.1, which is probably from around 1982 [31]. This program is not meant to be malicious, and it intends to shrink the image files on your DOS system, which saves a couple of bytes from your hard drive. Storage space was more valuable in that period as large HDD were expensive compared to today's market. Microsoft came with an answer to this software in 1985, creating their own executable packer known as EXEPACK.EXE created by the developer Reuben Borman [30]. These programs were more straightforward than today's packers; they solely compressed the data and then loaded them into memory again when executed. Packers these days use encryption, bundlers, compression, and network sources to hide their program's actual content. In addition, they can do the same packing on the same original file with a different "stub" and get a different output each time, making it nearly impossible to detect viruses based on their signatures alone.

Figure 1.1: Realia Spacemaker "Shrinks your COM and EXE files" [34]

https://t.me/learningnets

1.2 Problem statement

For my master thesis; I consider the following questions:

- *Q1 : How can one detect if a PE file is packed or contains a hidden PE file inside it?*
- *Q2 : How does one uncover the file with static analysis? Is it possible?*
- *Q3 : Is it easier to recover the original PE using dynamic analysis (i.e., Executing it)?*
- *Q4 : Which methods are best at detecting a packed binary file? What are the success rates?*

In addition to answering these questions, I will be creating a python script that will try to answer the following questions with a PE file as input.

- *A : Has the executable file been packed?*
- *B : Is the unpacking process located on the same process or in a different process/thread?*
- *C : Does the prototype recognize the packer used in the PE file?*

1.3 Thesis structure

The thesis is structured into six chapters. In Chapter 2 Theory, the thesis intends to describe state of the art and the inner details of Windows PE files. It also describes essential Windows API functionality used by PE packers. There is also a brief explanation of popular packers.

Chapter 3 Method; describes the experiments and how results will be presented. It also provides a list of all the tools used to create the prototype.

The result chapter will be dedicated to how the prototype was built, and it will present all the results found.

The discussion chapter is for discussing the validity of the experiments done and their results. It will also discuss the validity of the methodology employed for the experiments.

Finally, the conclusion will answer if the results and the thesis answered the questions asked in the problem statement. The conclusion will also hint towards future work and if this experiment was worth the time.

1.4 Related works

Most other related projects utilize header detection and other regex-like tools to extract suspicious data from a binary PE file, such as PEiD 1.4.1 and Detect it easy 1.4.6. These programs scan every single byte in the binary file; this is the most common and most straightforward way of detecting anything suspicious in the file. The scanning is quite effective and efficient; however, a crafty malware constructor may try to circumvent their triggers by laying traps such as fake sections or simply removing known headers from the file.

https://t.me/learningnets

1.4.1 PEiD

PEiD is a free tool for detecting the most common packers, cryptors, and compilers for PE files. Their Github Readme states that their program can detect over 600 different signatures in a PE file [43]. Unfortunately, the tool was written in 2008 and is therefore dated.

1.4.2 PE-Bear

PE-Bear is a graphical tool for navigating a PE file. It provides a disassembler and other general information about the critical headers present in the PE file. PE-Bear does have a method for detecting packed binaries, and that is done through signature lookup. PE-Bear's public signature files will be utilized later in this thesis. The latest update to the program was pushed May 24th, 2021, so this program is still in development, and the author claims that their program is utilized by the CIA [17].

1.4.3 PE-detective

PE-detective is an old PE packing detector. It is based on signature detection, it can scan entire directories or a single file at a time. This program was created by Erik Pistelli and is considered freeware [32].

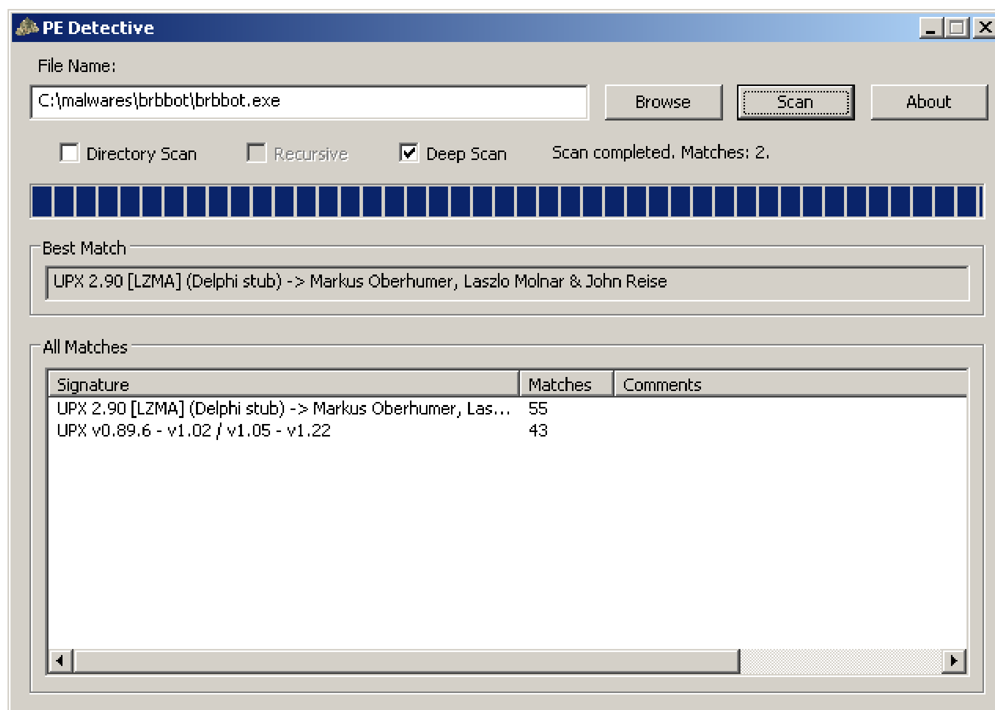

Figure 1.2: Pe-detective by Erik Pistelli [33]

1.4.4 EMBER

EMBER used 1.1 million binary files, 300K malicious, 300K benign, and 300K unlabeled for their experiments [2]. Although this dataset is large and public, the authors of the EMBER project wish that this dataset is used for machine learning malware detection programs. The viruses they have analyzed are from the large virus, malware, and URL scanner VirusTotal [46].

https://t.me/learningnets

1.4.5 PHAD

PHAD is a thesis that uses the flags for each section and devises a method to detect packed PE binaries. This Header detection method was entirely accurate, in which the rate was 93.59% with a false positive rate of 3.99% [7]. The thesis had eight distinct values calculated with the Euclidean distance and determined if the file was packed if the file's variables exceeded this number.

1.4.6 Detect it Easy (DIE)

An open-source project whose code is hosted online on Github. The program does its best to determine if the provided exe file is packed using the signature method [18].

1.4.7 MOV obfuscator

This technology is nothing other than impressive; Chris Domas, the developer, has made a program that compiles programs into only "mov" instructions. To do this, he had to define all x86_64 instructions as "mov" instructions. This creates a unique output, a bloated one for sure, but still functional. The output is about 500X the size of a program compiled with a regular compiler. However, its use-fullness is seen whenever another engineer tries to reverse engineer the program. The entire control flow of the program becomes a single "line"; there is no branching with ifs and elses, there are no jumps or calls to other functions, simply one line of mov instruction all the way through. I will not go into more details on how it is done; if it sounds interesting, be sure to visit his GitHub at [10].

1.4.8 Capa

Capa is a tool created by FireEye, which is designed to detect suspicious capabilities in PE files. It can detect Packed PE files; however, it is not its main functionality. Therefore this tool will not be used in the experiments.

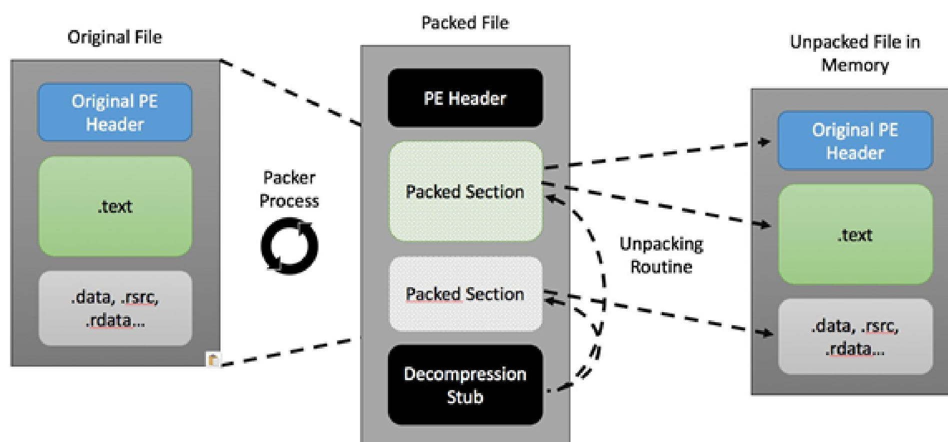

Figure 1.3: The general process of a Packer [36]

https://t.me/learningnets

Chapter 2

Theory

2.1 State of the art

The concept of packing your executable files is an old one. Vendors have been delivering this service for years; in the olden days, when storage was more expensive, users had a reason to compress their executable files. The files are reduced in size by executing compression methods on the entire executable file, thus reducing the required space to store the program on your computer. The executable (image) files in Microsoft Windows Operating systems are referred to as PE files (portable executables) and Common Object File Format (COFF); these are the .exe .dll .drv files on your Windows OS. The "portable" keyword in PE is supposed to hint that those PE files are not architecture-specific [26].

Some general terms will often occur through this thesis; make sure to look at the appendix for a short explanation of these terms A. As seen in Figure 1.3 the term "stub" is used; this word is often used regarding packers as it is a vital part of packers. The stub is equivalent to the inverse function of the packing function like you have in cryptography. Moreover, that stub has to return the original PE file in memory with no loss of data. The Figure 1.3 calls the stub "decompression stub," but it does not have to be a decompression method; it could be a decrypting stub as well.

2.1.1 Different Packers

The packers and users of packers can "pack" their binary images in theoretically infinite ways. For example, they can compress the program, encrypt that compressed data, and continue doing that one the same data; they could also salt it with data hosted on malicious servers. An example for these different methods can be seen in figure 2.1

2.1.2 Multi-Thread Packers

The unpacking routine runs whenever the user starts the program. When a multi-threaded packer is used, other threads are spawned to perform actions such as decompressing/decrypting the packed executable. The other threads also have tasks such as looking out for debuggers attaching to the primary process. They also make sure that the virus can spawn new processes that can continue delivering the payload to the target machine.

https://t.me/learningnets

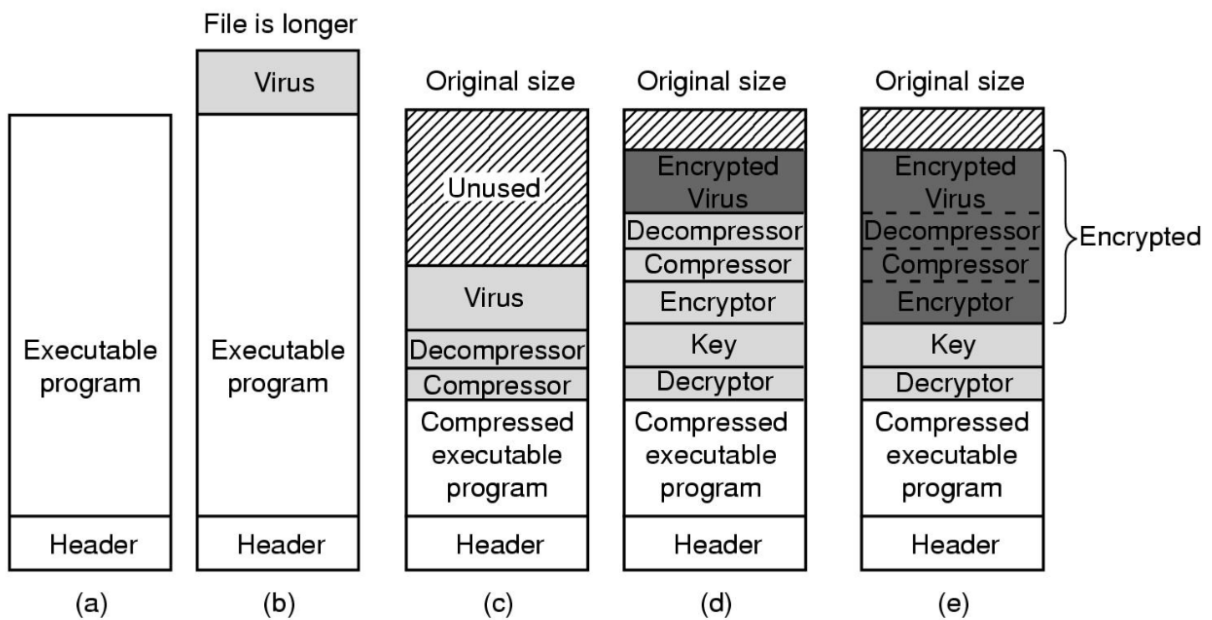

Figure 2.1: An visual example of malicious malware "packed" in different methods [3]
a) is a standard program
b) Is a normal program with the virus attached to the PE file
c) A compressed PE file with the decompressor inside the PE file
d) Same as c but with the virus encrypted
e) A program with multiple layers of encryption and compression

2.1.3 UPX - Packer

Upx is a popular open-source packer widely used; it is probably the biggest and most known one. It is developed by Markus F.X.J Oberhumer, László Molnár & John F. Reiser. Their GitHub states "UPX is portable, extendable, high-performance executable packer for several executable formats" [23]. The UPX packers are in high use by malware creators; the results from the experiments in the thesis show that 33751 of the 225420 identified PE files use the UPX packer (detected by header signature) more on this in chapter4.

The UPX developers claim that their packers can help reduce program size by around 50-70 %, which is great if space is a problem, or it could also be helpful for websites that host exe files; it helps reduce bandwidth. The compression aspect is the only one the UPX developers intend to support, however. As with encryption, they claim that it only gives "people a false feeling of security because by definition all protectors/compressors can be broken" [23]. The statement made is true to some degree; it is especially true for compression, but a key is also needed to uncover the original contents for encryption. This encryption key can be stored somewhere else, for example, on a malicious site that recognizes the fingerprint of the programmed asking for the key. However, this practice is still just obfuscation because the intent of packing is only such that the AV and Reverse Engineer will have a rougher time detecting the virus.

https://t.me/learningnets

2.1.4 Other Popular Packers

UPX is not the only leading spread packer floating around on the internet; The following packers listed under have or are still quite popular.

- Morphine - Able to encrypt the output of compressed data, and has a polymorphic engine [21].
- ASpack - ASPack is a simple packer with inbuilt security against non-professional reverse engineers [38].
- Armadillo - The first packer to introduce the anti-memory-dumping technology: Nanomites [15].
- ASProtect - Made by the same developers from ASPack, the packer promises several protections such as defense against debuggers, disassembler, unauthorized analysis, or copying. It provides encryption, compression, and integrity checks [44].
- PEBundle - Is a unique type of packer, and it provides the functionality to packer multiple executables and data files into a single file [47].
- MEW - Compresses malware using the LZMA algorithm [21].
- FSG - The abbreviation FSG stands for "Fast, small good," it is a simple packer, and it works by compressing the data sections.
- PESpin - Has protection against disassembling and patching; it compresses windows code using MASM (Microsoft Macro Assembler) [21].
- FUD Aegis Crypter - A GUI tool for creating packed malware, with plenty of features. See Figure 2.2 to see how it looks for the malware developer.

The different packers have strengths of their own, and some have complex guarding mechanisms to deter debuggers, disassemblers, emulators, virtual machines, and memory dumping. Figure 2.3 is an old data source, and the detection rates have probably improved for all the services listed.

2.2 Processor Architechtures

Standard desktop machines come with x86_64 architecture. This project is directed mainly at Windows PE files; it would typically only include x86_64 architecture processors. However, there has been a push lately towards ARM architectures as it is more energy-efficient than the standard AMD64 CPUs. Apple, for instance, has begun using ARM on its laptops. In addition, ARM has been present in phones and other appliances that require less power consumption. This thesis will only look at a few of the standard x86_64 instructions for string gathering.

https://t.me/learningnets

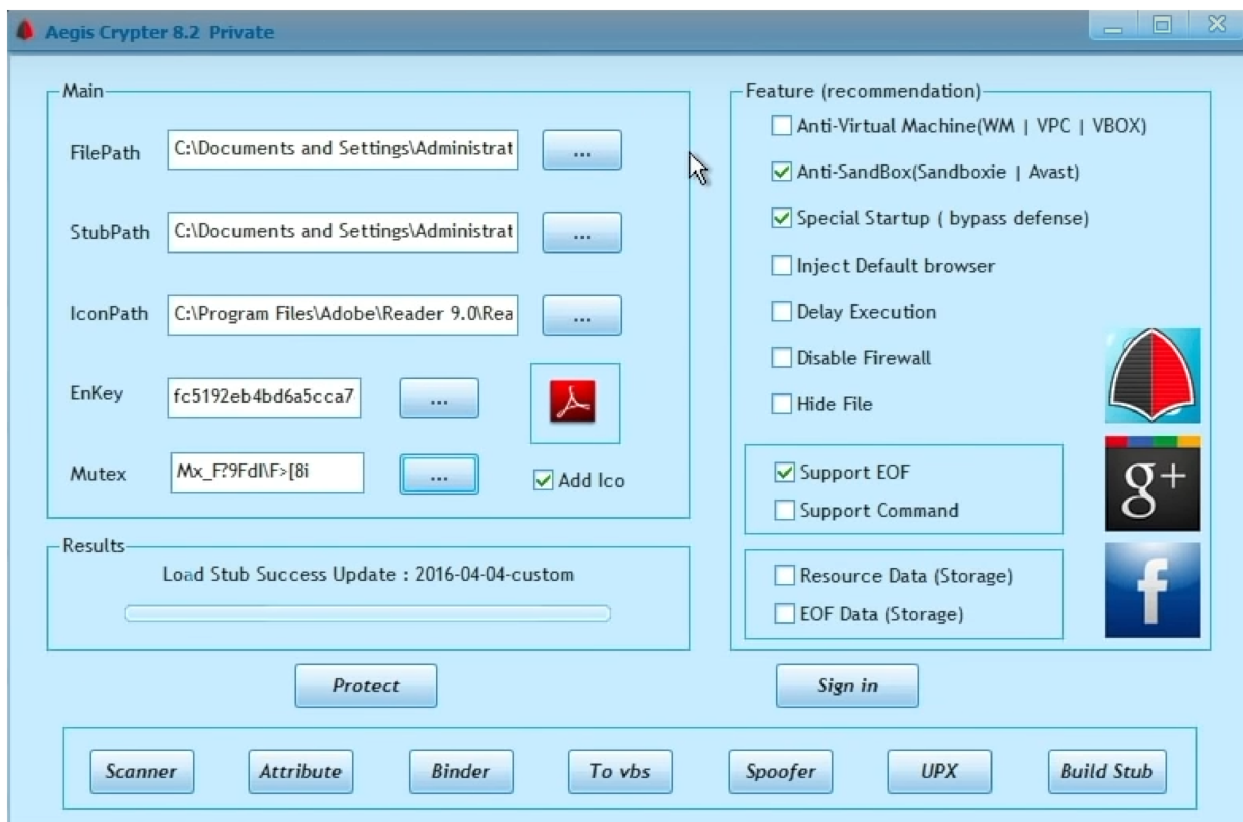

Figure 2.2: Aegis Crypter Tool [16]

2.2.1 The execution of a program

Whenever a program is started, the program counter or instruction pointer is pointed towards the program's EP (entry point). In 64 bit programs, this instruction pointer is called "RIP," whereas 32-bit programs are called "EIP." This is also a rule for registers that the CPU has, the prefixes "R" and "E" signify whether or not the program is a 32bit or 64bit program. In the case of 16bit DOS programs; they have no prefix; it is simply "IP".

The AMD64 CPU architecture has a plethora of register that is used when the program executes its instructions. as seen in figure 2.4. The "i r" command in gdb prints all registers during runtime. The ax, cx, dx, and bx are what you call general use registers; these are usually used before calling a routine (function). Additionally, before a function is called, the return point (current execution pointer) is stored in one of these variables. Nevertheless, this is also up to the compiler designer to decide which register he wants to use. The ebp and esp are referred to as stack pointers, sp is the stack pointer, while bp is the frame pointer [9].

2.3 Important Windows APIs

2.3.1 CreateProcessInternalW

CreateProcessInternalW is an API used to start a new process and thread within the calling process. This function is usually invoked after the stub has done its bidding. When invoking this function, the malware PE will be residing on its memory section of the process. The CreateProcessInternalW is invoked whenever the CreateProcessW is executed [19]. This function could also be used to make

https://t.me/learningnets

	Kaspersky	McAfee	Symantec	Microsoft
Average on the Malware Testset	83%	79%	58%	39%
ASpack	95%	97%	95%	81%
FSG	100%	100%	56%	100%
Morphine	100%	70%	100%	0%
UPX	96%	97%	92%	100%
MEW	100%	86%	53%	80%

Detection - Rate of packers

Figure 2.3: Detection rate of packers from black hat briefings 2006 [4]

multiple processes using the same process that started this one, and this is common for packers to avoid being stopped by debuggers. The malware would need to add some mutex such that it does not start an infinite amount of processes.

2.3.2 VirtualAlloc

VirtualAlloc is a vital function located in the memoryapi.h file. It reserves, commits, and changes the state of a region of pages in the virtual address space of the calling process [27]. It helps store the malware contents in memory to the calling process; you can choose what to do with the program. One old and deprecated method is to write the memory buffer contents to a file; any modern antivirus program would detect this, even windows defender would react; see figure 2.5 for an example. The windows 10 Creators Update launched in April 2017 included some new features for the Windows defender, this feature generates signals that are sent to the windows defender program whenever a program uses the functions VirtualAlloc or VirtualProtect [45]

An excellent trick to dig out the original PE malware inside is to debug the Packed PE and place a breakpoint on the return address to VirtualAlloc since the address to the new memory section will be stored in EAX/RAX at that moment. After the address is noted, you can watch a memory dump of that section and look for any suspicious data loaded into it.

Data Execution Prevention (DEP) is an essential policy in Windows, and it has been a feature available since Windows XP Service Pack 2 [40]. This feature is not unique to Windows and is called W xor X on other OSs. The memory pages are marked by a bit called the NX (No eXecute); if a memory page is marked as data, it prevents code execution on those pages. On Intel CPU's this bit is called

https://t.me/learningnets

```
(gdb) i r
eax            0x1      1
ecx            0x401940 4200768
edx            0x80     128
ebx            0x2d0000 2949120
esp            0x61ff28 0x61ff28
ebp            0x61ff28 0x61ff28
esi            0x4012e0 4199136
edi            0x4012e0 4199136
eip            0x401463 0x401463 <main+3>
eflags         0x206    [ PF IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x53     83
gs             0x2b     43
(gdb) x/10i 0x00401460
   0x401460 <main>:     push   ebp
   0x401461 <main+1>:   mov    ebp,esp
=> 0x401463 <main+3>:   and    esp,0xfffffff0
   0x401466 <main+6>:   sub    esp,0x10
   0x401469 <main+9>:   call   0x4019c0 <__main>
   0x40146e <main+14>:  mov    DWORD PTR [esp],0x405064
   0x401475 <main+21>:  call   0x403a60 <printf>
   0x40147a <main+26>:  mov    eax,0x0
   0x40147f <main+31>:  leave
   0x401480 <main+32>:  ret
(gdb) x/s 0x405064
0x405064 <__register_frame_info+4214884>:       "This is a string!"
(gdb) 
```

Figure 2.4: Simple program that prints a string compiled with mingw-gcc

https://t.me/learningnets

Figure 2.5: Reflective DLL Loading detection by Windows Defender [45]

XD (Execute Disable) and XN (eXecute Never) on ARM CPUs. WxorX prevents execution of shellcode on the stack, heap, or any data segment.

DEP has little to no performance impact when utilized on windows operating systems, and it is recommended to activate for all programs and services. You can also compile your binaries with the /NXCOMPAT link option for visual studio, indicating that your program is compatible with windows' DEP.

2.3.3 VirtualProtect

Virtual Protect is a critical function to set the appropriate privileges for sections of memory in a program. Let us say we have a program that unpacks its contents into one memory segment stretching from the address 0x04000 to 0x08000; if this content is written using the VirtualAlloc function, then the memory would be in read-only mode. The main goal is to execute it, and the hacker has to make the memory location executable and maybe even writeable. VirtualProtect changes the protection on a region of committed pages in the virtual address space of the process [28]. If the hacker succeeds, he can continue executing the new program by moving the PC (Program Counter) or start a new process with the API CreateProcessInternalW using the memory section.

https://t.me/learningnets

2.3.4 LoadLibrary

LoadLibrary is a function that loads in other DLL files and makes it possible to call their API. This function is usually called when the program includes a header file from a library; for example, #include <windows.h> would trigger the program to load KERNEL32.DLL. The LoadLibrary function is located within the Kernel32.dll [25].

To load a custom Library file in C++, it is done by invoking the function with an LPCSTR and passing it to an HMODULE variable done as such:

```
char * myDLL = "EvilDLLs\\TheEvilDLL.dll";
LPCSTR myDLLpath = (LPCSTR)myDLL;
HMODULE handle = LoadLibraryA(myDLLpath);
```

Listing 1: LoadLibaryA

The LPCSTR in listing 1 is a long pointer to a constant string. This pointer can be pointed to the string or, instead, the path in which the DLL is located. With the DLL loaded, the user can invoke any function that may reside within it; bear in mind, the dll itself may load other DLLs needed if not present already to run its functions.

2.4 PE Structure

The PE structure layout is quite similar for both 64-bit and 32-bit programs. This layout is specific to the Windows operating system and only applies to its executable files, driver, object code, and DLLs, to name a few. The following chapter will discuss the specifics for the Windows PE structure and describe an executable image file's essential parts.

First of all, we got to figure out if the file we got is a Windows PE file. To do this, we will look at the "magic bytes"; these are in the case of a windows PE file, the first two bytes, which in our PE case is "MZ" or 0x4D and 0x5A. Another hint is to search for the string "This program cannot be run in DOS mode," which should be located at the offset 0x4E and with a size of 0x26; however, as we will see later in the thesis, hundreds of ways to identify a PE executable file.

2.4.1 Addresses: RVA, VA and Physical Addresses

A program compiled into an image PE file will be placed in a structured way, with all the headers in the beginning. These headers explain where and what has to be done to execute the program. As such, there is one "address" that will be automatically generated, the psychical address. The physical address refers to the actual offset in the binary file, so for example, 0x10 would be 16 bytes in the current PE file or any other file, for that matter. However, in PE/ELF/Mach-O files, another address refers to data locations, also known as the Virtual Address. The virtual address (VA) is quite serviceable as it helps visualize which section the address is located in. For example, an address in the data section may start with a 0x4000 prefix, while an address in the text section starts with 0x1000.

https://t.me/learningnets

The virtual address's origin is from when page files were introduced. Page files isolate the process memories and process "simple" addresses instead of physical addresses on the ram stick. So when the CPU request an address at an address, it will simply send the request to the MMU (Memory Management Unit). The MMU translates the virtual address to a physical address and gets the data on the actual position on the memory stick [3].

2.4.2 IMAGE_DOS_HEADER

The DOS header is the first part of the file and is usually 0x80 bytes large (128 bytes). The first two bytes, as discussed, is just an identifier known as the magic bytes. The following bytes in this header are also exciting, specifically the four bytes from 0x3C, containing the address to the NT Header, which we will discuss later on.

```
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number
    WORD   e_cblp;                      // Bytes on last page of file
    WORD   e_cp;                        // Pages in file
    WORD   e_crlc;                      // Relocations
    WORD   e_cparhdr;                   // Size of header in paragraphs
    WORD   e_minalloc;                  // Minimum extra paragraphs needed
    WORD   e_maxalloc;                  // Maximum extra paragraphs needed
    WORD   e_ss;                        // Initial (relative) SS value
    WORD   e_sp;                        // Initial SP value
    WORD   e_csum;                      // Checksum
    WORD   e_ip;                        // Initial IP value
    WORD   e_cs;                        // Initial (relative) CS value
    WORD   e_lfarlc;                    // File address of relocation table
    WORD   e_ovno;                      // Overlay number
    WORD   e_res[4];                    // Reserved words
    WORD   e_oemid;                     // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;                   // OEM information; e_oemid specific
    WORD   e_res2[10];                  // Reserved words
    LONG   e_lfanew;                    // File address of new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Listing 2: The _DOS_HEADER from winnt.h header file in windows 10

The definition for the DOS_HEADER can be found in the winnt.h header file in windows, and it contains a short and precise description for each entry in the struct as seen in listing 2. For example, the word "WORD" is 2 bytes large, and it is an unsigned short that ranges from [0,65535]. This data type is quite common in Microsoft-related products, in comparison with ELF (Linux), in which they use the data type "uint16_t" (also 2 bytes). The LONG keyword is just the regular "long" datatype from C but capitalized.

2.4.3 IMAGE_NT_HEADER

The NT Header contains three entries, one character string that is the signature of the PE file; this is usually "PE" or 0x5045000. The IMAGE_NT_HEADER can be

https://t.me/learningnets

```c
typedef struct _IMAGE_NT_HEADERS64 {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;
```

Listing 3: _IMAGE_NT_HEADER for a PE file, found from winnt.h

seen in listing 3. Then we have the behemoth IMAGE_FILE_HEADER, which contains plenty of helpful information about the PE file; the IMAGE_FILE_HEADER can be seen in the listing 4. The Machine field describes the architecture required to run the program; for windows binaries, this field is usually 0x64 and 0x86, referring to AMD64/Intel64/x86_64 Processor architecture. Windows also supports arm for some few select boards with what they are calling "Windows 10 IoT Core".

The number of sections is a field that describes the number of sections in the PE image file. These sections contain different kinds of data. The most compelling section is the ".text" section, as it contains the instructions for the program. These instructions can be disassembled using the AMD64-IA32 manual, or you can use a pre-written disassembly program. As for the experiments this thesis will perform, the capstone disassembler will be utilized [5], more on this later in the thesis.

The number of sections in the program can hint at whether the PE file is packed. If the program contains less than the average amount of sections, it indicates that either the program was compiled with a simple assembler or linker or another unknown compiler. For example, a standard Windows program compiled with minGW GCC/G++ (The free GNU compiler for windows) typically has around 12 sections (32-bit versions). Moreover, as seen in results 4, the average number of sections in PE files located in the Windows10's System32 folder contains six to seven sections, refer to Figure 4.3. It is important to note that many of these headers can be tempered with, so the data in these structures does not necessarily reflect reality. This tampering of the headers is seen in the experiments, where the reported number of sections did not reflect the actual amount of sections.

```c
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Listing 4: File header found from winnt.h

https://t.me/learningnets

2.4.4 IMAGE_SECTION_HEADER

The Image section header describes the data each section in a Windows PE file contains. These data points will be used later in the thesis. Further details in 3.2.

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Listing 5: Section header found from winnt.h

An example of two sections can be seen in figure 2.6

2.4.5 .text

The .text section is usually not found in packed PE binaries since the packers scramble all sections and their data into their own defined sections. The main goal for a security analyst is to get access back to the original .text section, which contains all the instructions for the program.

This section may be included in a packed binary as well. However, it may just be an impostor, which sole purpose is to trick antivirus tools. Its size may vary even as an impostor; however, it is usually empty with a size of 0 or nonexistent whenever there is packing involved.

In a regular PE/ELF/Mach file, this section is the most important one that contains all instructions for the program. These instructions vary from CPU architecture but are, in essence, the same. It is up to the compiler and linker to organize the data in an executable file. The free Min-gw compiler has different procedures compared with Microsoft Visual C++ compilers. They both have a different amount of sections used. They store different data types in every PE file; from my experience, the Windows compilers are the ones who usually create the most bloat.

The .text sections flags has averagely the configuration of

- Read: True
- Write: False
- Execute: True

https://t.me/learningnets

```
⊟      140000200 2e 74 65        IMAGE_SE...
                 78 74 00
                 00 00 40 ...
   ⊞—   140000200 2e 74 65 78 74  char[8]   ".text"              Name        .text
                 00 00 00
   ⊞—   140000208 40 15 01 00     Misc                           Misc
    —   14000020c 00 10 00 00     ibo32     FUN_140001000        VirtualAddress
    —   140000210 00 16 01 00     ddw       11600h               SizeOfRawData
    —   140000214 00 04 00 00     ddw       400h                 PointerToRaw...
    —   140000218 00 00 00 00     ddw       0h                   PointerToRel...
    —   14000021c 00 00 00 00     ddw       0h                   PointerToLin...
    —   140000220 00 00           dw        0h                   NumberOfRelo...
    —   140000222 00 00           dw        0h                   NumberOfLine...
    —   140000224 20 00 00 60     SectionF...IMAGE_SCN_CNT_CODE | I... Characterist...

                          IMAGE_SECTION_HEADER_140000228          XREF[1]:     __scrt_i
⊟      140000228 2e 72 64        IMAGE_SE...
                 61 74 61
                 00 00 b6 ...
   ⊞—   140000228 2e 72 64 61 74  char[8]   ".rdata"             Name        .rdata
                 61 00 00
   ⊞—   140000230 b6 9f 00 00     Misc                           Misc
    —   140000234 00 30 01 00     ibo32     PTR_QueryPerformanceCo... VirtualAddress = 0001cab0
    —   140000238 00 a0 00 00     ddw       A000h                SizeOfRawData
    —   14000023c 00 1a 01 00     ddw       11A00h               PointerToRaw...
    —   140000240 00 00 00 00     ddw       0h                   PointerToRel...
    —   140000244 00 00 00 00     ddw       0h                   PointerToLin...
    —   140000248 00 00           dw        0h                   NumberOfRelo...
    —   14000024a 00 00           dw        0h                   NumberOfLine...
    —   14000024c 40 00 00 40     SectionF...IMAGE_SCN_CNT_INITIALI... Characterist...
```

Figure 2.6: The two sections .text and .rdata in a bening PE file

This thesis will use the notation r-x, to express the standard rights for a section.

Note that the "Write" flag is false; if this is attempted to be changed using VirtualProtect, Windows will react by sending signals to Windows Defender. Finally, the .text section should also contain the Entry point for the execution pointer. The execution pointer is the main thread's address; so to speak, it is usually referred to as the program counter. The EntryPoint can be found in the header "_IMAGE_OPTIONAL_HEADER" 6. The "optional" header is found under the NT header. The EntryPoint is a critical data point that can be used to detect packed pe files, as it should always be located inside the address space of the .text section, or at least a section with the rights r-x.

2.4.6 .data

The data section is pretty straightforward. It is simply a section in which data is stored, and it also is a section where data is written, compared with rdata (read-only). So if the program is compiled containing a char abc[20] = "Hello," it will store those bytes in a data section. However, some compilers differ in this regard, and if it is a small string, it may just store the string as an instruction such as "movabs rax,0x6f6c6c6548," which moves the string 0x6f6c6c6548, which equals "Hello." The data is stored in data segments and retrieved by doing a "lea" or equivalent instruction to get the data, which usually refers to the address to the data segment.

https://t.me/learningnets

2.4.7 .bss

The bss section contains uninitialized data; usually, either statically allocated variables are declared without any value. It is an older assembly term meaning "block started by symbol." In short, it stores uninitialized global or static variables.

2.4.8 .tls

This section enables thread-local storage, and it gives the developer the possibility to store a data object that is individual for each thread that runs in the program [26]. TLS data can be stored using the Windows API calls TlsAlloc, TlsFree, TlsSetValue, and TlsGetValue.

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    // Standard fields.
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;
    // NT additional fields.
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Listing 6: Optional header found from winnt.h

https://t.me/learningnets

Chapter 3

Methodolgy

Cybersecurity is an exciting field; it has differences from other traditional studies, as it has pretty complex and old studies regarding it. Furthermore, the field is constantly being developed, and new threats emerge that have to be solved. The methodology used for the experiments is the hypothetico-deductive research model, which is quite common for the cybersecurity field [11].

This chapter will lay the ground for an experiment. The outputs of the experiment are defined in Section 3.2.2 Experimental design. The experiment will only analyze files statically. There will be a discussion on dynamic analysis and emulation/debugging later in the Results 4.1.3.

3.1 Assumptions and Limitations

Assumptions

- Hardware is available, and that it supports virtualization (most do).
- This experiment requires a basic understanding of assembly code.
- Access to Windows 10 OS and its files
- Knowledge on how to handle viruses inside a Virtual Machine (to prevent viruses bleeding onto the host OS)
- Bash scripting knowledge.
- Assumption that the downloaded virus set is labeled correctly and truly are viruses.

Limitations

- The analysis will be performed inside a Virtual Machine with no connection to the network since I do not want to risk any spillage of viruses onto our home network. This virtualization does impact the efficiency of the experiments.
- The variations of viruses may not reflect what types of viruses are delivered to people daily.
- Time: If the program is supposed to analyze every PE downloaded, it will take a tremendous amount of time.
- RAM: Only 24GB of memory is useable, limiting the number of tasks running simultaneously.
- The virus dataset is not marked with which file is packed or not.

https://t.me/learningnets

3.2 Experimental design

Viruses are gathered from sources such as VirusTotal and VirusShare; these viruses are mostly PE files, some are other types of viruses. In addition, a giant set of viruses from the website Virus eXchange, a 61GB large folder with 271 098 viruses in it [39] was downloaded. This dataset will be the primary source of viruses for the experiment.

A script will remove any non-PE file, removing any file that lacks the two magic bytes "MZ" in the header. A simple bash script will be executed to scan every single virus file in the folder. Then, a custom-made Python script will look over the file and output the appropriate information in a JSON file. Those JSON files will be utilized for representing the results later on in Chapter 4.

For the experiments to give any valuable outputs, the variables need to be defined. These variables will be crucial to make the decision that the Binary file has been packed or not. This method is not a new one and has had plenty of studies done on it beforehand. Reports such as "Pe File Features in Detection of Packed Executables" by Dhruwajita Devi and Sukumar Nandi [8] use the variables such as section Entropy, size of uninitialized data, size of headers, and the size of raw data. These are all logical variables to take into accord for determining if a binary file is packed or not; therefore, they will be taken into accord when determining if a file is packed.

Moreover, the "PHAD" report [7] written by Yang-Seo Choi, Ik-Kyun Kim, Jin-Tae Oh, and Jae-Cheol Ryou have additional variables I will use as well, these are the following; Number of executable and writable sections, Numbers of sections with code but is not flagged as executable, if there are no executable sections in the binary if the EP is not in executable code and more. These will be taken into consideration as well. A more thorough list is detailed in Table 3.1

The Table 3.1 are all data points that are negative if they are higher, negative in the sense that the file is more likely to be packed the higher it is. All of them are a percentage of the occurrences divided by the total number of files in that set.

Variable name	Range	Description
A1	[0,1]	If the file has a section larger than 7, this datapoint is set to 1.
A2	[0,1]	If the file does not contain a ".text" section with the standard r-x rights.
A3	[0,1]	If the Entry Point is not in an execute section
A4	[0,1]	If the number of DLL's imported are less than 12
A5	[0,1]	If the number of sections is larger than 8 or less than 4
A6	[0,1]	If the size of the sections are larger than the program
A7	[0,1]	If the number of strings are less than 10.

Table 3.1: Experiment Variables

https://t.me/learningnets

3.2.1 Variable Explanation

- A1 - Entropy: The entropy describes the chaos in a section: the section data' randomness. This variable is calculated using the python function listed in Listing 12. This data point is set to one if the file contains a section with entropy higher than 7.
- A2 - Each section has its rights for be it reading, writing, or executing, and if there are multiple sections with r-x rights, it is a red flag. Since a normal PE file should only have the .text section with executing set to true. The standard rights for an executing section can be written as r-x (read and execute).
- A3 - The EP (entry point) is the address to the starting instruction for the program. It is found in the NT header. This data point is counted towards if the EP is not within a section with the standard r-x or not pointing towards any sections' address space.
- A4 - The number of imported DLLs is an intriguing data point. The number of DLLs imported may differ based on the program's functionality; the simpler the program, the fewer imported dlls. This data point is counted towards if the number of dll imports is less than 12.
- A5 - The number of sections is an input that can help determine if a file is packed. Some packers like UPX leaves only three sections when a PE file has been packed. Of course, there are no rules to this; a packer can use many sections it wants, up to a limit of 65535 (WORD-2 bytes).
- A6 - The last variable is if the sections' size is bigger than the exe file itself. If it is, the value is set to 1.
- A7 - The number of strings found in a program can hint at the program's intention. If there are no English words found in the binary, it is clear that the program wants to hide its secrets. The number of strings found within the file is the ones that intersect with an English word list file. The file with the English words is separated with a newline 0x0a and has 354 281 unique words.

3.2.2 Categories Of Viruses

The viruses are also put into different categories. These categories were labeled by the ones who provided the data source. The categories found in the dataset are the following:

- Backdoor - A backdoor is a common virus that leaves an entry point on the host machine for new sessions down the line. The Mitre ATT&CK matrix would categorize this as "Persistence."
- Constructor - Software that can create malware based on the options the hacker chooses. Like the Hacktool type of software, this may not be malicious on its own. A hacker would create the malware and then spread it through other means such as email, IRC, FTP, USB sticks, etc [42].
- DoS - An abbreviation for "Denial of service." The intention of the virus is quite obvious, as it only wants to disrupt and cause havoc on the system.

https://t.me/learningnets

- Email-Flooder - An Email flooder will grab the email application on the host computer and send spam to every known contact from the host. This type of email may contain viruses attached to the email or links to malicious websites.
- Email-Worm - The Email-Worm is a well-known type of virus that spreads itself through email, infecting the target and continues to spread itself from the target.
- Exploit - The definition of the exploit may wary, as it quite a general word. Generally, it is used as a verb, as in exploiting a weakness.
- Hacktool - The word describes itself; it is a software tool that can hack others. These types of software may not be malicious but can pose a risk for the user. As Malwarebytes labs states, "Hacktools are often downloaded from less reputable sites, which may be malware instead of the promised hacking software" [22].
- Flooder - A flooder is a virus that sends a massive amount of data to a specific target. Similar to Email-Flooders; however, the general flooder usually targets IRC clients [14].
- Hoax - This imposter "virus" is not a virus on its own; it merely states that it is a virus. For example, this type of hoax could be a simple popup message window saying, "The computer is infected pay 0.01 BTC to this address within 4 hours to solve the issue or RISK losing ALL data." This type of software could also be called "Scareware," it uses alarmist language and stresses the lack of time to solve something [41].
- IM-Flooder - A program that spams and floods up a message channel like Discord, Skype, WhatsApp [12].
- Several Trojans such as DDoS, Downloader, Ransom, Clicker, Banker are among the categories.
- Virus - A general term used to describe malware that spreads itself and infects the target.
- P2P-Worm - Malware that spreads through peer-to-peer networks.
- Rootkit - These types of malware are the worst as they plant themselves deep within the machine's configuration files and other vital parts. A famous rootkit virus that caused billions in damage is the Sony BMG Rootkit incident [29]. The primary purpose of the rootkit virus is to hide from the user, such that the user has no clue that his system is infected [6].
- Spoofer - A spoofer has the intent of spoofing another identity, usually another user on the system, such that the hacker has elevated access to the machine.

https://t.me/learningnets

3.3 Tools Utilized

3.3.1 Capstone

Capstone is a disassembler for the Python Language. It can analyze any bytes you provide it, and it will do its best to disassemble that machine code into readable assembly code. For example, the byte array 0x488b05863400000 will result with: "mov rax, qword ptr [rip + 0x3486]," which is considerably easier to read; it is also possible to create an interpreter/emulator for this type of code, and then guess what it does. This module will be critical for analyzing the PE file in question; every EXE virus file we analyze will be disassembled with capstone.

3.3.2 PeFile

It is a python module that can take a file as input or raw bytes, and it will find every header and store the interesting data in an object that can easily be accessed while analyzing the binary file. This library will be handy for my master's project as it helps me organize all related PE header data in a convenient object file that can be accessed from a variable. The other option would be to know the exact offset in the binary exe file to find the related data myself, not impossible, but it would be a huge time sink. PeFile is open source and can be found on GitHub to Ero Carrera [13].

3.3.3 scipy

Scipy will be utilized to calculate the entropy for the sections. The entropy gives an interesting data point about how chaotic the section is (i.e., how random it is). If a section has very high randomness, it is indicative of compressed or encrypted data. Furthermore, as we know, encrypted and compressed data is the product of "packing" a pe binary file/section.

3.3.4 PE-Bear

As explained in the introduction 1.4.2, PE-Bear is a disassembler, general-purpose information displayer with a solid GUI. It also has a signature list that can be downloaded. This signature will be utilized in the experiments to compare the prototype packing results versus the signature's results. The signatures come bundled in a single text file organized in a way seen in Listing 7

```
1 Armadillo 4.40 -> Silicon Realms Toolworks
2 85
3 31 2E 31 2E 34 00 00 00 C2 E0 94 BE 93 FC DE C6
4 B6 24 83 F7 D2 A4 92 77 40 27 CF EB D8 6F 50 B4
5 B5 29 24 FA 45 08 04 52 D5 1B D2 8C 8A 1E 6E FF
6 8C 5F 42 89 F1 83 B1 27 C5 69 57 FC 55 0A DD 44
7 BE 2A 02 97 6B 65 15 AA 31 E9 28 7D 49 1B DF B5
8 5D 08 A8 BA A8
```

Listing 7: Example of Signature from PE-Bear [17]

The signature seen in Listing 7 consists of three parts, firstly it is the name of the packer; second, the number of bytes to be compared with, which in this case is 85 bytes. The third part consists of the actual signature, and these are

https://t.me/learningnets

written as hex values pr byte, so the value "31" would be b"1". The signature file also contains the eventual ?? between the bytes, which means that it can be any byte from 0->255. Those "??" bytes are added as wild cards in the regex, by simply replacing them as ".." which means any two characters.

For the experiment, I wrote a small-byte comparison script, and this will later show a mistake as the detection rate is relatively low. The script that was written for scanning through the relevant virus files was relatively slow and inefficient. It simply converts the entire program into string hex values such as seen in the Listing 7. The script in question can be seen in Listing 8.

```
def findsig(data,signatures):
    sigresults = {}
    data_string = ""
    for byte in data:
        data_string+=hex(byte)[2:].upper()+" "
    data_string=data_string[:-1]
    for signame,values in signatures.items():
        bytes_sign = values["Bytes"]
        match = bool(re.findall(bytes_sign ,data_string))
        if match:
            sigresults["Name"] = signame
            return sigresults
    sigresults["Name"] = "No signatures found"
    return sigresults
```

Listing 8: Signature script used in experiments.

3.3.5 matplotlib

An open-source plotting tool for Python [20]. This tool will be used for plotting results onto graphs for visualization.

3.3.6 Ghidra

The tool Ghidra was developed by the NSA's Research Directorate for NSA's cyber-security mission and is intended for reverse engineers [1]. Ghidra allows viewing the entire binary statically, providing an easy-to-use GUI where the sections can be disassembled and interpreted as C code. This tool is not used in the experiments. However, it is used to verify that the claims made in the theory chapter are correct in practice.

3.3.7 x64dbg And x32dbg

x64dbg and x32dbg are programs for debugging 64-bit and 32-bit programs, respectively. The debugger attaches or launches the EXE file, provides all the basic debugger functionality such as GDB, and provides a nice GUI.

https://t.me/learningnets

3.4 Expected Results

Each file that is to be analyzed will give a plethora of information through its headers. Every section in the PE packed file will be in the results and exciting flags on the section; these flags describe the section's own rights to either execute, write, or read. The data will include its size, virtual address, physical address, all strings found in the section, and its entropy. The output of each file will be organized as seen in listing 9. Every file will also be put under a category, such as Backdoor, Trojan, Email-Spam, Hoax.

Every file will have an output as seen in listing 9. The output data will then be used in python scripts to present the data uncovered. For this presentation of data, matplotlib will be used, a data representation library for the Python programming language. These scripts made for presenting the data are small and easy to understand since their actions only include accessing the result JSON files and loading their data into memory [20]. Microsoft Excel will also be utilized for visualizing data.

https://t.me/learningnets

```
1        "Email-Worm.Win32.Bagz.f": {
2            "Number of sections": 4,
3            "Time spent": 2.4342153072357178,
4            "Packer detected": "UPX",
5            "Imports": "GetSystemDirectoryA,SetCurrentDirectoryA,CreateWindowExA....",
6            "Packing certainty": 90,
7            "KERNEL32.DLL": "GetModuleFileNameA,SetCurrentDirectoryA,...",
8            "SHELL32.dll": "ShellExecuteA",
9            "USER32.dll": "CreateWindowExA,SetWindowPos",
10           "Sections": {
11               ".text\u0000\u0000\u0000": {
12                   "Size": 24576,
13                   "Entropy": 7.130176741636183,
14                   "Rights": {
15                       "Read": true,
16                       "Write": false,
17                       "Execute": true
18                   },
19                   "Virtual Address": 4096,
20                   "Virtual Size": 24074,
21                   "Physical Address": 24074,
22                   "Strings": "open,name"
23               },
24               ".rdata\u0000\u0000": {
25                   "Size": 4096,
26                   "Entropy": 4.829555134574962,
27                   "Rights": {
28                       "Read": true,
29                       "Write": false,
30                       "Execute": false
31                   },
32                   "Virtual Address": 28672,
33                   "Virtual Size": 2824,
34                   "Physical Address": 2824,
35                   "Strings": "spa"
36               },
37               ".data\u0000\u0000\u0000": {
38                   "Size": 65536,
39                   "Entropy": 4.327100158011958,
40                   "Rights": {
41                       "Read": true,
42                       "Write": true,
43                       "Execute": false
44                   },
45                   "Virtual Address": 32768,
46                   "Virtual Size": 71452,
47                   "Physical Address": 71452,
48                   "Strings": "ju,is,ix,lan,ey,tu,spa,fo,hel,pl,go"
49               },
50               ".rsrc\u0000\u0000\u0000": {
51                   "Size": 4096,
52                   "Entropy": 1.869670701428837,
53                   "Rights": {
54                       "Read": true,
55                       "Write": false,
56                       "Execute": false
57                   },
58                   "Virtual Address": 106496,
59                   "Virtual Size": 928,
60                   "Physical Address": 928,
61                   "Strings": ""
62               }
63           },
64           "EntryPoint": "0x1982",
65           "Image Base": "0x400000",
66           "FileSize": 102400
67       },
```

Listing 9: Example output of the custom made python script

https://t.me/learningnets

Chapter 4

Results

4.1 Questions

Here there will be an attempt to answer the general questions asked in the problem statement in chapter 1.

4.1.1 Q1 - Detection

The first question states:

How can one detect if a PE file is packed or contains a hidden PE file inside it?

Hidden PE : Let us say there exists a hidden PE file within a PE file. Then the hidden PE file can be scattered inside the host PE file in multiple fashions. For example, it can either be scattered throughout the main PE files sections, and the malware developer can refer to the areas where the PE's bytes are hidden in the different sections to rebuild it at runtime. The rebuilding of a PE file may seem easy to do, but several protections are in place. These protections were discussed briefly in chapter 2, namely the section flags which define what can be done to the section, be it executing, reading, or writing. The trick which packer developer uses are the different API calls VirtualProtect and VirtualAlloc.

The PE contents can also be encrypted with a straightforward method like XoR every byte with a key. This key could be stored inside the PE file itself or on a network server; either way, the code has to get the key at some point on the machine locally to be able to XoR it with the data. This key could then be extracted by debugging the exe file or simply looking for keys statically (which is hard). The best method or the easiest is running the program and looking for many XoR operations. However, running the program comes with a risk; therefore, running the program on a burner (disposable) or VM (Virtual Machine) is advisable.

The encryption is usually more advanced than just XoR the bytes; for example, they could use RSA and hold onto the decryption key on a cloud server, such that the program can not be unpacked without running it. It would then request the server, and the server could confirm the packer; it could even send some general information about the running process. If there is a debugger present, the packer would most likely report that to the server, and thus the server can choose not to give the key to the process. The packer has several options to check if a debugger is attached to the process, such as using the kernel32 function IsDebuggerPresent() or the NtQueryInformationProcess() function.

https://t.me/learningnets

Detection : The reverse engineer will try to open the malware file inside a disassembler like IDA, Ghidra, or OllyDBG to discover that all the instructions look like a random mess. Most professional disassemblers also detect the packed PE because they will not find the different functions as they do not match their pre-compiled function hashes (FID). For a reverse engineer, determining if a file is packed is relatively easy, but it is more complex for a general program like an AV.

The detection part is what the experiment in this thesis mainly focuses on, and the results can be seen in Section 4.2. The methodology is described in Section 3.2; this experiment is mainly for detecting if a file is packed using statistics. This type of approach is also done in antivirus programs, as they look for known signatures and behavior in a PE file, be it the number of sections, entropy, DLLs, strings.

Detection also includes finding out which sections has the data relevant for recovering the original PE file, or as we call it, the unpacking "stub." For example, the most significant section with the highest entropy has most likely the data needed to recover it, but then you need the stub of the packer to execute it on the data to recover it. However, finding the inverse function may not be as hard as it seems, as it should be one of the sections with comprehendible code/instructions. One could also follow from the EP (entry point), interpret each instruction, and look for calls to APIs like VirtualProtect or VirtualAlloc.

4.1.2 Q2 - Static Analysis

The second question states:

How does one uncover the file with static analysis? Is it possible?

The answer here is a bit complicated, and it is both yes and no. The statical analysis does not provide a way to recover the original PE file, but it will give hints on how to do it. What is meant with "statical analysis" is that one can look at the code without actually executing it, therefore avoiding the risk of being affected. This method is what linters do when analyzing your code; they will find faults with syntax and help you fix them. However, the code we find in an image file is nonexistent because the data inside an image file is machine code.

The machine code inside the binary has to be disassembled, as done in the experiment using the python library capstone [35]. The disassembly can also be done using the manual for the target CPU architecture, such as the AMD64 programmer manual, and then write a disassembler using the AMD64 manual as a reference. However, writing a disassembler takes an awful amount of time as there are over 1000 different types of instructions to consider.

After the code is disassembled, you are left with assembly code as seen in Figure 2.4. This type of code can be interpreted as C code, which is, in return, is easier to understand. Let us take Figure 2.4 as an example; using Ghidra's [1] assembly interpreter, and we get the results as seen in Listing 10
The results from Ghidra's interpreter are pretty good, but when it comes to data types and the like, it is more of a struggle, leaving the interpretation up to the developer. The point is that it is possible to find the code for the inverse function of the packer inside the binary. Moreover, you can create a function in your

https://t.me/learningnets

```
int __cdecl _main(int _Argc,char **_Argv,char **_Env)
{
  ___main();
  _printf("This is a string!");
  return 0;
}
```

Listing 10: Ghidras interpretation of assembly code seen in Figure 2.4

favorite language with this code and then run the function on the section you believe the data is stored. In theory and indeed in practice, this will then leave you with the original PE file. The only scary thing with the method mentioned above is that you might use the wrong code and execute something else if not done correctly.

The static analysis method is also relatively slow since you have to interpret all the instructions and make your code replicating these instructions, which could take weeks. This time is quite expensive, so this should only be done if the program is vital to reverse engineer. I would recommend the Dynamic analysis approach to get the original malware binary file.

4.1.3 Q3 - Dynamic Analysis

The third question states:

Is it easier to recover the original PE using dynamic analysis (i.e., Executing it)?

By dynamic analysis, it is meant that the program is executed inside a controlled environment, for example, by a debugger such as WinDbg, x64dbg, or gdb. The debugger attaches to the process and halts its execution, leaving the execution control up to the user. Standard functionality includes dumping memory regions which help get the original PE file. However, it is crucial to choose when to dump the memory, and if the memory is dumped at the beginning of the executable, then it is obvious the decompression function has not run yet; therefore, the original PE file will not be inside the memory.

A standard method of getting the data is to use a breakpoint at the Kernel32 functions VirtualAlloc and VirtualProtect because when those functions are called, memory is allocated and given rights by the VirtualProtect function. With tools like x32dbg, one can view the memory map when those functions have returned and then look for any new suspicious memory segments with the strings "MZ" and "This program cannot be run in DOS mode." If a memory segment like that is found, one can dump that segment, and voila, the exe file is found.

There are, however, guard mechanisms against dumping the memory from a debugger, such as described in the article to Peter Ferrie [15]. "Namoites" is a type of guard mechanism; they work by putting "int 3" instructions in place of branches (if jumps). The "int 3" instruction is known as an instruction that generates a software interrupt for x86 CPUs [24]. The nanomite will then look at

https://t.me/learningnets

the unpacking method for which "int 3" is a branching instruction. The debugger does not know this and will break at each "int 3" instruction, and the control flow will then be broken.

There is a similar method to debugging, and that is simply dumping all memory while running the program. Furthermore, after the fact, search for the executable inside the memory dump. An obvious downside with this approach is that the virus gets to do its evil bidding, and disregarding the code within the virus could be detrimental.

The packed file can also be emulated, and you achieve this by emulating the instructions inside a controlled environment. Emulating it lets you look at each expression like you would with a debugger. This method is safer than just dumping the memory.

A scary part of debugging the executable itself is that some malware files execute themselves multiple times when launched. This method is called "Self-Execution," which spawns as many subprocesses as possible until it is content. The problem here is that since you launched the program in user mode, all subsequent subprocesses have the same user rights. So, you may have paused one instance whenever you debug the program, but there could be four others running such that the system gets infected. [15].

4.1.4 Q4 - Different Methods

The fourth question states:

Which methods are best at detecting a packed PE file? What are the success rates?

The manual approach to determine if a PE file is packed must be the simplest because a skilled reverse engineer will quickly see that the PE file is packed. Thus, the case of false positives will be low whenever a reverse engineer manually looks at the file in a tool like Ghidra or IDA. Nevertheless, for an antivirus program, the process of detecting a packed PE file must be automated. The AV will then employ their functions on the data to determine if it is packed. The function they use could be like the one this thesis's experiment does, which statically loads the file into a program, disassembles it, searches the PE files headers, looks for known signatures, checks for an abnormal entropy, and then estimates if it may be packed.

Another trick packer developers try to do is to make the stub (unpacking routine) as benign-looking as possible, such that it can trick antivirus checks; in addition to that, they change how the stub and general PE file is structured each time they generate a new Packed file with that malware inside, to avoid signature detection and hash databases.

So to conclude, the methods we have are the following:

- **Signature Lookup** - This is the most common for opensource and freeware tools, as those signature lists are easy to find on the internet, and they contain thousands of known signatures. The method is quite simple as one scans through the entire file and looks for any bytes matching the signature

https://t.me/learningnets

list; if there is a match, you have found the packed file. A popular tool for this is PEiD, and according to the thesis PHAD [7], it has a detection rate of 75%.

- **Dynamic Analysis** - By the use of a debugger, emulator, or a VM, one can dump the contents of the memory section the hidden PE file is temporarily stored in during execution. This approach may be the riskiest one, but it is pretty effective and quick because the reverse engineer skips the phase of actually understanding the decompression methods in the PE file.

- **Manual Detection** - By the use of a human who directly looks at the PE file in a disassembly tool like OllyDbg or Ghidra, he can determine if it is packed. This method is the best approach if there is only one file to be analyzed, but by all means, it does not hurt to put it through a detection program for extra confirmation. The success rate depends on the experience of the analyst and the cunning of the PE packer.

- **Header Detection** - A simple approach is to look through the PE file's headers and look for irregularities, such as odd section names or odd section rights. This method may easily be avoided with decoy sections with bogus code in them. The imported functions can also be found in the headers. Those imported functions can be used to gain a better understanding of the program's intent. However, those import tables are usually deleted by a malware program after those functions have been solved [15]. The success rate for this method seems to work for PHAD [7] where the detection rate is at 93.59%, but it has a higher false-positive score than signature comparison. The false-negative score is much higher for signature tools such as PEiD compared to the PHAD project.

- **Behaviour Checking** - It is also plausible to see that the program is packed if it behaves in a specific way during execution by monitoring API's frequently used or any new suspicious memory sections with data that could resemble an executable. This point could be called API hooking, which is like a debugger; you attach yourself to the process and listen for any API you may like to record. When the API is triggered, you gain access to variables such as the current registers when it was called and the stack.

4.2 Experiment: Analyze Viruses

The downloaded Viruses were around 270k; about 225k of these are PE files with the "MZ" header. The analyzing process takes its time; some of the viruses are more complex than others. Some take 0.2 seconds to analyze and run through the code, while others take 50 seconds or upwards. This limits the amount I can scan and analyze. The Virtual Machine will stay on for the entirety of my master's period, continually analyzing. This also means that any change to the code that goes through those viruses would make a huge difference, and perhaps those files that are analyzed last have a better confidence level.

https://t.me/learningnets

There is also the need to state that this experiment was not without its flaws, several hiccups regarding running out of memory, crashes, exceptions that happened during the analysis of the files, resulting in some files being incomplete and therefore lacking the critical information to give the results needed to fill the variables explained in Chapter 3. These hiccups resulted in a loss of 30 419 legitimate PE files' data, and they lacked information such as the EP(entry point) and section address space. Therefore, out of the 224 902 files analyzed, 194 482 are left to represent the results.

In addition to the 194 482 viruses, the experiments have two other data sets to use for comparison. The first dataset is the system32 files on a Windows10 machine, totaling about 587 files. Those files were selected because they are assumed to be benign. The second dataset is random exe files found around on a daily use Windows10 machine; these include software like web browsers, video games, education software, and other general daily use software. General software solutions tend not to have state of the art packers to hide their intent, yes they want to protect their IP (intellectual property), but there are other measurements in place for that, such as DRM (Digital Rights Management), DMCA (Digital Millennium Copyright Act), key certification schemes, and other in house solutions. The random exe files found on my computer and those that did not take ages to analyze amounted to 1606 files.

4.2.1 The script that gathers the data

```bash
#!/bin/bash

TIMEFILE="tids_bruk.txt"
echo "Calculating number of files..."
files=(~/Downloads/viruses-2010-05-18/*)
number_of_files=${#files[@]}
echo "Found $number_of_files"
for ((i=0; i<number_of_files;i+=24)); do
    STARTTIME=$(date +%s)

    for ((j=0;j<24;j++)); do
        timeout 5m python3 pakkeoppdag.py "${files[i+j]}" "results$j.json" &
    done

    wait
    ENDTIME=$(date +%s)
    printf "$(($ENDTIME - $STARTTIME))\n" >> $TIMEFILE
    echo "$i/$number_of_files has been scanned spent $(($ENDTIME - $STARTTIME))"
done
```

Listing 11: Bash script for analyzing all the viruses.

As seen in figure 11 there are 24 instances of the python script launched simultaneously, and this works great for my AMD 3900X 24 Thread system; however, it requires a tremendous amount of ram as well as the script loads all data in the file and tries to interpret it as instructions. The Virtual Machine has been assigned with 24 GB of ram, but this proves later to not be enough at times, causing crashes and lag. The script also runs the python scripts with a

https://t.me/learningnets

timeout command meaning that if they spend more than x amount of time, the script is killed, which sometimes is necessary since some files are humongous and require more processing than others. The script waits for all 24 processes to exit before it launches an additional 24 processes; this is achieved using the "wait" command on Linux systems.

However, the script shown in listing 11 proved to be too slow. Therefore some reorganization was done to the viruses and how they were scheduled to be scanned. The main idea is to have all 24 threads running at the same time. Nevertheless, with this script shown above, the program has to constantly read the results from the previous run, resulting in unnecessary RAM allocation and dumping; the hard disk was also pinned to 100% using the previous scanning method. As a result, the prototype's code had considerable changes at the end of the project, such that the scanning could be done quickly enough to gather necessary results before the deadline.

4.2.2 Results for A1 - Entropy

The entropy was calculated using this python3 function:

```
1  def entropy(data):
2      if len(data) == 0:
3          return 0.0
4      frequency = Counter(bytearray(data))
5      entropy = 0
6      for x in frequency.values():
7          p_x = float(x) / len(data)
8          entropy -= p_x*math.log(p_x, 2)
9
10     return entropy
```

Listing 12: Entropy calculation

The Listing 12 shows the python3 function detailing how the entropy is calculated for a section of data, this function is the same as:

$$H(X) = \sum_{i=1}^{n} P(x_i) \log_2 P(x_i)$$

File Set	Number Of Files	Avarage Entropy	Standard Deviation
Viruses	194 482	5.605	2.080
System32	587	5.010	1.258
randomexes	1606	5.039	1.404

Table 4.1: Vulnerability Overview

The table above Table 4.1 shows the results regarding entropy in the sections of the viruses. The entropy shown in the third column is the average entropy of every section in the entire result file. The viruses, for example, had 194 482 files, and all of those viruses had 587 123 sections in total; the average was done by simply summarizing the entropy and then dividing by the number of sections.

https://t.me/learningnets

The sections with the size of 0 were ignored, as there were a few of those present.

These results were not entirely expected. The entropy for the System32 and randomexe files were closer to the viruses' average entropy. The reasons can be plenty, but the most logical one is that the System32 programs are complex programs made by one of the largest software producers out there; they are most likely quite complex and contain a large amount of random data. However, the spikes of entropy were more prominent among the viruses, as seen in the standard deviation field.

The variable A1 will be in a range from 0 to 1. The variable A1 is 1 when a section with entropy higher than seven exists inside the file. As for the viruses, a section with entropy higher than seven occurs quite often 49.1% of the files contain a file with that attribute. If this number is adjusted, the more files meet the requirement, and this is logical if we look at the average entropy for each set in Table 4.1.

4.2.3 Results for A2 - Execute rights

When analyzing the benign System32 Exe files, all of them contain the ".text" section, and the rights are always r-x. So the first and most blatant hint for a file to be packed is that it lacks a ".text" section; however, some compilers have other names for the .text section, such as "/CODE." The section label can be changed as pleased using linker scripts. The data point is determined if the execute rights are set to True, False, True. Some statistics regarding the execute rights can be found in Table 4.2.

A thought for the mind is that antivirus programs can choose to enforce rules such as "there must exist a .text section" to allow the program to execute; if an AV has a check like that, the packer developer can choose to create decoy sections, or name his sections after the standard naming convention to avoid detection. For example, the variable A2 is a variable that the packer can trick since the variable A2 is fulfilled whenever there is a section with the name ".text" and has the right r-x (read, execute).

4.2.4 Results for A3 - EntryPoint

The EP (entry point) should always be pointing into a typical .text section with the execute and reading rights. The problem with this data point is that not necessarily all PE files have the .text section. If this is the case, the variable A2 will be set to 1 automatically, and A3 will depend if the EP is within a section with the rights r-x. Moreover, the results show that out of the 194482 files, only 78528 have a section with the string ".text". Therefore will most of the files have a score of 1 on this point. However, out of the 78528 sections with ".text" in them, 57209 has the EP in them. This check was achieved by checking if the address of the EP is more significant than the Virtual Address to the section and less than the size of the section plus the virtual address.

There is also the case where the EP may not be inside a .text section but is inside a section with the standard read, write and execute rights. Out of the 194482 files analyzed, the Entrypoint was found in all of them. In the results, we see that out of these 194482 files with entry points in them, 79427 is within

https://t.me/learningnets

a section with the standard rights. 79213 of the EP points to a section with all rights set to true, which is a red flag because the code should never self-modify.

File Set (Number of files)	**Number of .text sections**	**.text with standard rights**	**EP within standard rights**	**EP outside standard rights**	**EP outside sections**
Viruses 192 482	78 528	57 209	79 427	98 224	16 832
System32 587	587	587	584	0	3
randomexes 1606	1594	1590	1570	9	27

Table 4.2: Entry point / Section rights Statistics

4.2.5 Results for A4 - DLLs

There were also two other scans of other PE-related files; these files are found in the Windows 10 System32 folder. The System32 PE files are all 32bit and are a relic of the past. However, these files are fascinating to compare because we know these files are benign and developed by Microsoft developers. Finally, all the DLL files were scanned; additionally, they are also PE files; however, they are not executable. The DLLs are mainly equal to standard EXE files; they have a few differences. The main difference is that a dll is linked into the program at the time of executing, instead of building it into the program when it is compiled.

The average amount of DLLs imported in windows10/System32 is 16.85, while the average amount of DLLs imported for the virus sample is 3.73. This difference is significant, and it helps to prove that most viruses will pack their programs to hide their tricks. In the Table 4.3 below, you can see the most popular DLL imports and functions used in the virus data set.

The findings from Table 4.3 are moderately intriguing, and there is confirmation on some of the theories that the functions VirtualAlloc and LoadLibrary are essential for packers as described in Section 2.3. You can also see the use of message box and string printing functions from the user32 dll; those could intimidate the user by showing threatening messages.

https://t.me/learningnets

DLL	Number Imports	DLL	Number Imports	DLL	Number Imports
kernel32.dll	164638	user32.dll	99685	advapi32.dll	81913
Functions		**Functions**		**Functions**	
GetProc Address	108594	Message BoxA	33965	RegClose Key	46244
Load LibraryA	103544	Find WindowA	17858	RegQuery ValueExA	31176
ExitProcess	71498	Create WindowExA	17788	RegOpen KeyExA	26045
GetModule HandleA	52313	GetWindow Thread ProcessId	17388	RegSet ValueExA	22129
VirtualAlloc	47351	CharNextA	17303	RegCreate KeyExA	13969
VirtualFree	42723	wsprintfA	17260	OpenProcess Token	9447

Table 4.3: DLL Imports - Viruses

Figure 4.1: The DLLs imported by Viruses

4.2.6 Results for A5 - Sections

The number of sections found in the results wary wildly, as seen in Figure 4.2. The average amount of sections in the virus sample is 4.96, the most common amount of sections are three. The average amount of sections for the system32 exe files are 6.505. Surprisingly the difference is not as grand as expected. The average amount of sections is less for the viruses because many of them are packed, and packed PE files usually require fewer sections. Such as UPX packers only contain three sections.

However, some files contain an absurd amount of sections, for example, the file Trojan-Downloader.Win32.Banload.fnz contains 65 452 sections. This number cannot be trusted because the malware developer intentionally changed the

https://t.me/learningnets

header field NT_HEADERS.FILE_HEADER.NumberOfSections. This oversight is a flaw on the prototype's part; instead of trusting that data field, the prototype should have looked for the actual number of sections by manually counting each section. Doing the manual counting changes the result. The average of sections becomes 4.12, and the standard deviation is 2.244. Additionally, the max number of sections in the virus set becomes 36.

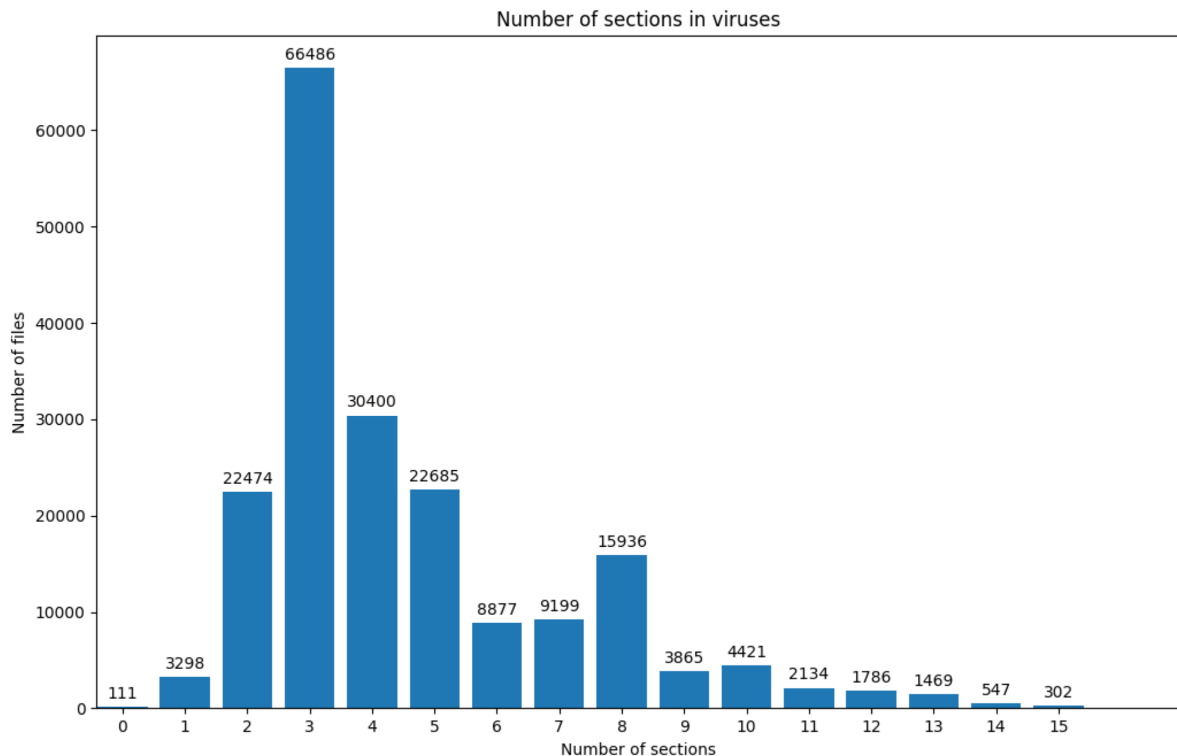

Figure 4.2: Figure displaying the common amount of sections in viruses. sample size of 194K viruses

4.2.7 Results for A6 - Section sizes

Every program has a set amount of bits it occupies on the HDD; this is also reflected in the section headers to the PE file. The variable IMAGE_SECTION_HEADER.SizeOfRawData describes the section header size on disk. There is also the datapoint VirtualSize which is how large the section will be during runtime. The VirtualSize can be either more or less than the SizeOfRawData. As stated in the explanation of the variable, see the List 3.2.1, if the sizes of each section combine differ from the size gathered from the command "wc -c file" it is set to 1.

The results show that 68 193 out of the 194 482 files have the same size as the sum of all sections' size. Meanwhile, for the System32 files, 470/587 - 80% had the same size. As for the randomexes, the results were 737/1606 - 46%.

https://t.me/learningnets

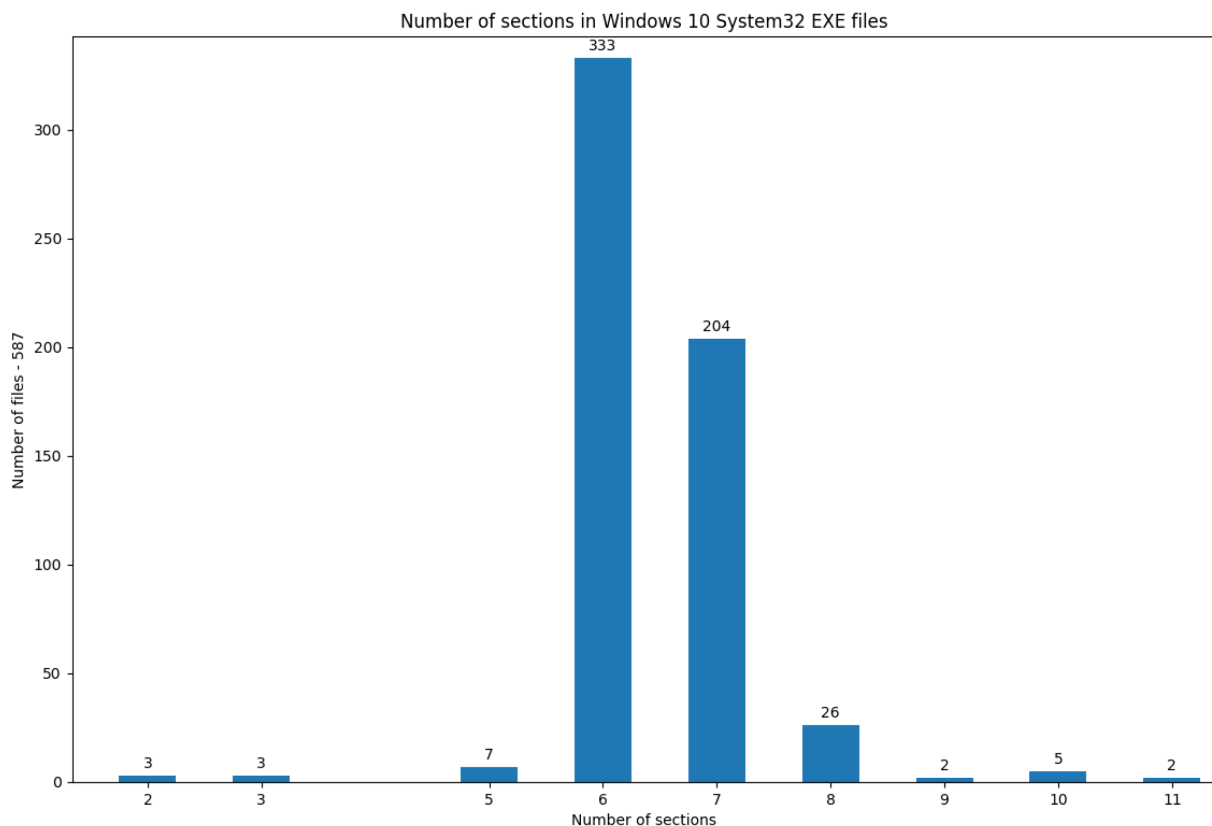

Figure 4.3: Results of scanning the System32 folder on windows 10

4.2.8 Results for A7 - String in Binary

The number of strings found inside the virus binaries varies a decent amount when looking at the results. The string searcher in the prototype is the most complex part of the code. It interprets each instruction and tries to determine if the instruction is pointing towards a string in a data segment or is storing string data on the stack. The get_str_from_instruction function is 62 lines long and has deep if lookups; it honestly is spaghetti code, but it gets the job done. The function only continues looking for a string if the instruction contains 1 of the four mnemonics ["mov", "lea", "push", "movabs"], and if it does, it will attempt to either get data if the address in the operation field is a pointer or try to interpret the operation field as a string itself. If there is an instruction like:
mov eax,0x6f6c6c6548
It would flip the number (little-endian) and then try to use the function chr() on the numbers, such that it would find the string "Hello." The byte for byte procedure may produce some errors and characters that are not wanted. In order to minimize the amount of rubbish that is translated, only characters from 0x10 to 0x7f are included.

https://t.me/learningnets

File Set (Number of files)	Number of Strings	Avarage Amount of Strings	Avarage string length	Standard Deviation String Amount	Standard Deviation String Length
Viruses 192 482	4 231 106	21.755	2.569	39.593	1.417
System32 587	17 115	29.156	4.492	57.849	2.461
randomexes 1606	134 941	84.023	4.972	221.825	2.618

Table 4.4: String statistics

The standard deviation as shown in Table 4.4 and Table 4.1 is done using the following forumla:

$$StandardDeviation = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

The results gathered from this experiment are interesting. First, it shows that a virus file is less likely to contain an English string within it, that could be because more of the viruses are packed, and therefore the strings hidden in the original malware cannot be seen.

4.2.9 Summary Results

This section is dedicated to summarising all the above results' main points into a table.

File Set	A1	A2	A3	A4	A5	A6	A7
Viruses	0.491	0.702	0.591	0.938	0.818	0.649	0.578
System32	0.030	0	0.005	0.449	0.069	0.199	0.461
Randomexes	0.083	0.009	0.022	0.661	0.263	0.541	0.363

Table 4.5: Main Results

With these results in Table 4.5 we can create a case for each file in the set and check if they are packed by checking each point individually and then compare it with the average for viruses. The average amount of offenses for the viruses is 4.735, while for the system32 files, 1.131 offenses were committed. For the random exe files on the computer, 1.890 of the variables were 1. These numbers are obtained by simply summarizing the averages in Table 4.5,
As an example, we will use the file shown in Figure 9, that file would have the array:

$$1, 0, 0, 1, 0, 1, 0$$

if this array is summarized, we get 3, the file has committed a few offenses, but these are not enough to classify this file as packed. As the results show, the viruses generally score higher on the tests, with a score of 4.735. As each file is currently being scored from 0 to 7 (only real numbers), we can label a file as packed if it commits more than four offenses. Setting the threshold to four, we get the following results seen in Table 4.6

https://t.me/learningnets

File Set	Files	Prototype	Signature	Overlap
Viruses	194 482	110 908	11 463	4277
System32	587	2	0	0
Randomexes	1606	43	63	0

Table 4.6: The prototype's results compared with a simple signature detector

The prototype has quite a high rate of labeling a file with "packed," as seen in Table 4.6. The signature column indicates how many of the 194 482 files had a signature within them, and the signature list was downloaded from PE-Bear's public list [17]. Unfortunately, the results are not expected; only 11 463 have shown a known signature. The cause of this discrepancy could be my hastily written signature lookup method 8. The script uses the inbuilt library **re** in python to compare the bytes in the PE files against the signature.

4.3 The Prototype

The prototype itself is the python3 script that gathers all the exciting data points in the binary file. It uses various tools to achieve this, and the following subsections describe the program and its logic. The main program is relatively short, with only 420 lines, but it is not the only script used to gather the data and display data. There were four other scripts dedicated to their task. The present results scripted amounted to 450 lines of code, and other small scripts such as string sanitation, consolidating JSON files, and signature scanners were created.

4.3.1 Pefile

As established in the Methodology chapter 3 the program will be written in Python3, and it will feature some powerful open-source libraries; these libraries will be vital for finishing the master in the limited six-month time period given to us.

The primary and most important library is pefile; the pefile library uses all known headers and offsets to create a comprehensive python object that is easy to navigate. Every programmer familiar with object-oriented programming will find this library helpful if they are analyzing PE-related files. All the object-related names are the same ones as you would find in the winnt header; therefore, if you want to find the number of sections in the program, you can find it at pe.NT_HEADERS.FILE_HEADER.NumberOfSections. These headers and variables are key components in the prototype program that was made. Another useful function pefile has, is that it can find import_tables at specific addresses; with this, we can determine what function the "call" instruction is calling on. For example, with the instruction "call qword ptr[rip + 0x6d9b]", we can then try to find what function is being called on looking using pefile. The lookup can be done with the function pe.get_import_table(operation address XOR image_base).

4.3.2 Capstone

As mentioned earlier in the thesis, Capstone is an open-source disassembler with excellent documentation and architectural support. The code can be found on Github to Nguyen Anh Quynh [35]. The capstone disassembler can take any

https://t.me/learningnets

data in as an argument and will return a list of instructions. These instructions include their:

- id - Instruction ID, datatype: int
- address - The Virtual address to the instruction, datatype: int
- mnemonic - The command to the instruction, for example, "lea", "mov", "push". The data type is a string.
- op_str - The operand of the instruction. The op_str contains the actual operation given in the standard Intel disassembly format, for example, "rax, qword ptr [rip + 0x3485]". The data type of this item is a string.
- Size - The size in number bytes, datatype: int
- bytes - An array of the actual bytes to the instruction.

The instructions can be used to find strings hidden in the program, strings assembled when the program runs, and put onto the stack (RAM). The number of strings can indicate obstructed strings within the program since a benign program has no intention of hiding its strings unless it is confidential. A method for determining if the number of strings hint towards a benign program is to look at the amount functions required in the program. So, for example, if there is a printf function located inside the program, but there are no strings passed to the printf, one can conclude that the string has been obstructed so that the disassembler cannot find it. Finding these function calls such as "printf" is not always straightforward though, if there has been any compression/encryption, you cannot find it; additionally, a compiler does not have to include debugger details such as symbols for the program to function.

However, there is the plausibility to utilize the functions found in the imported symbol table, and these are the functions that are resolved through a DLL file, for example, VirtualProtect from Kernel32.dll. Using this knowledge, we can find functions that require strings by looking in the official Microsoft documents for the function imported. However, this lookup after string function is not done for these experiments because of the strict time constraints. The code for the current string solving function can be seen in Listing 13. The script is not without its flaws. There are many false positives without any sanitation, resulting in lots of junk being labeled as strings. Therefore, after analyzing the PE packed file, a second script is employed, running over each string and checks if they are in the English dictionary; if not, they are thrown away. Some good strings are unfortunately lost in this process, but it gets rid of the nonsense strings.

https://t.me/learningnets

```python
def get_str_from_instruct(instruct:instruction):
    mnemonics = ["mov", "lea", "push", "movabs"]
    if instruct.mnemonic.lower() in mnemonics:
        if instruct.op_address:
            op_addr_int = int(instruct.op_address, 16)
            if op_addr_int > image_base:
                virt_addr = op_addr_int ^ image_base
            elif ("lea" in instruct.mnemonic or "mov" in instruct.mnemonic)\
                    and re.search("ip", instruct.bracket_content):
                virt_addr = instruct.address+op_addr_int+0x7
            else:
                virt_addr = op_addr_int

            if pe.get_section_by_rva(virt_addr):
                instruct.op_section = pe.get_section_by_rva(virt_addr).Name.decode("utf-8")

            if instruct.op_section and "data" in instruct.op_section:
                try:
                    stringen = pe.get_string_at_rva(virt_addr)
                    if stringen.__len__() > 0:
                        instruct.string_bin = stringen
                        instruct.decode_string()
                except:
                    pass
            else:
                if not instruct.string:
                    if re.search("ip", instruct.bracket_content):
                        try:
                            stringen = pe.get_qword_at_rva(instruct.addr\
                                +int(instruct.op_address, 16)+0x8)
                            if pe.get_string_at_rva(stringen):
                                instruct.string = pe.get_string_at_rva(stringen)
                                instruct.decode_string()
                        except:
                            pass
                    else:
                        op_addr = instruct.op_address
                        if len(op_addr) > 4:
                            instruct.op_section = "Stack string"
                            if len(op_addr) % 2 == 1:
                                op_addr = "0" + op_addr[2:]
                            else:
                                op_addr = op_addr[2:]
                            reversed_string = "".join(reversed([op_addr[i:i+2]
                                                    for i in range(0, len(op_addr), 2)]))
                            result = ""
                            for hex_code in range(0, len(reversed_string), 2):
                                tall = int(reversed_string[hex_code:hex_code+2], 16)
                                if 0x10 <= tall < 0x7f:
                                    result += chr(tall)
                            if result:
                                instruct.string = result
                            else:
                                instruct.op_section = ""
```

Listing 13: Str resolving function,

https://t.me/learningnets

Chapter 5

Discussion

5.0.1 Method discussion

The methodology of gathering information from a binary file is often used, and it has been done multiple times before. The difference this thesis makes is the variables that are taken into consideration. In addition to using the known header techniques, my code also looks through all bytes in each section and treats them as instructions; the only thing it currently finds from these are the strings they may include: stack strings or strings pointed to in other data segments. This string detection makes my solution slightly different from the plethora of other thesis doing this same experiment. It would also be possible to run the free UNIX tool "strings" to gather strings from the binary itself; however, this tool is not as flexible as the custom-made one since it only finds strings related to the entire PE files and not each specific section.

The large sample size was definitely a blunder, or rather a mistake. Although the large sample size takes days to slug through, this may vary depending on the activated functionality. The amount of RAM to hold all the related data points was significant; even with 24GB assigned to the Virtual Machine, it still used all of the memory. The ram usage could be a fault with the prototype's code. There were also attempts to run multiple programs at once; since the VM was given 24 threads to work with, it would make sense to let each thread handle one instance of the python program. However, the practice of running multiple programs at once; was too much for the old spinning HDD that had to write results and read previous results.

There are also flaws in gathering data from the files because the prototype mainly used the headers to determine several data points, such as the number of sections. Some packers try their best to remove or alter header data that can be used to detect a packed PE file [15].

https://t.me/learningnets

5.0.2 Result discussion

The results were somewhat expected, but there were interesting tidbits about how many sections a virus usually has, what kind of execution rights the sections had with EP inside them. The main gripe I have with the experiments is that they did not have exe files for the categories packed and not packed (benign). The virus files downloaded were categorized by what kind of malware they contained, but there was no information on whether they were originally packed. Therefore, comparing my detection algorithm works is more challenging because I do not have the solution to check my results. There was this idea of using a well-established tool such as PE-Bear or PEiD to make some "solution" as to if the exe file was packed, and then check my results up against theirs; however, the tools PE-Bear and PEiD is quite dated and does not have the functionality to export the results to a file. If the results from PE-Bear or PEiD were to be gathered, it would have required dumping stack memory and analyzing that to find the data stored about the results. Doing so would take an awful amount of time that I do not have.

There was confusion on my part regarding the calculation of entropy. I initially thought the base used for calculating it was 256. Having the base as 256 instead of 2 caused a bit of chaos in my results. However, this is easily fixable with a custom script only recalculating the entropy with the correct base of 2, but it did take away some of my time and caused frustration.

5.0.3 Other solutions

The Ember project [2] is a project about analyzing malware using machine learning; however, it does not focus on packers, but that does not mean they do not meet the issue since packing is a standard procedure of malware developers.

If I were to experiment again, I would try to find the 20 most popular packers and then pack a suite of standard software, such that the results could show a detection rate of packed software and the false-positive and false-negative percentage. Those three points of data would help solidify and confirm that the method devised in Chapter 3 works and is worth doing. Based on the results of the PHAD [7] one would think this method is worth doing, at least use it in tandem with other detectors.

Another thing that the experiment should incorporate more of is the use of the instructions disassembled with capstone. For example, look for standard compression code or decryption code. There are probably hundreds of other good ideas that could be done with the instructions disassembled. I could have used the instructions to validate if the section with r-x rights had logical/legit code. That could be another variable for the experiments, probably giving the prototype a better detection rate.

The functions that were found in the binaries should also be used in the experiments. There should be a variable that is one if, for example, VirtualAlloc or VirtualProtect are loaded. If there were time, this is something that should have been added.

https://t.me/learningnets

5.0.4 What to take away

The packing of PE binaries is not a new concept by any means, and it has matured quite a lot throughout the years. However, PE packers employ tons of tricks to slow down the reverse engineer, be it nanomites (decoy breakpoints), encryption, shuffling of important header data, using new API function such as IsDebuggerPresent(), hiding threads, blocking inputs, removing software breakpoints, suspending threads, guard pages to mention a few. Therefore, the field has become highly intricate since one has to know each of these tricks, and who knows how many new tricks are being employed today to trick reverse engineers?

Header detection might be good at detecting if a file is packed, but it is best to employ more methods to check if the file is packed, even though the user may suffer a slight performance hiccup. The good old saying goes as follows: "Anything that can possibly go wrong, does" [37].

https://t.me/learningnets

Chapter 6

Conclusion

6.0.1 The solution

In Chapter 1 Introduction, the thesis asked several questions; the first four questions mainly were theory/research questions. The last three were regarding the prototype. The first question A: "Is the file packed" were answered for all files in the dataset; if the answers are accurate or not, that is another topic.

As for the second question B : "Is the unpacking process located on the same process or in a different process/thread?" is a much more complex question. This question did not get answered in the experiments. To answer this question for each file, we would have to find the stub that packs out the PE file and then analyze the stub. It could also require a bit of emulation or debugging to find the proper answer for this question.

The third question C : "Does the prototype recognize the packer used in the PE file?" is somewhat answered if we link the signature script that was hastily written and the results from the main program; there are a few packers that were identified. In addition, many UPX packers were detected in the main program itself; if you count the number of files that detected the UPX packer, it will amount to 28 159 hits. Therefore, the answer to question C is mainly no.

The problem was to create an application purposed to dig information from a binary PE file, even if it was packed. The prototype saved every vital detail into result files used to determine if the binary file was packed. However, due to the complexity of these files, the success of my experiments was limited. Nevertheless, the project helped to understand the structure of PE files better and was valuable for understanding assembly and AMD64 instructions.

https://t.me/learningnets

6.0.2 Results

The virtual machine gathered the results in multiple fashions, and the methodology of getting these results changed from time to time, which consumed a considerable amount of time. Also, developing a program that was supposed to read and interpret assembly code was an inspiring learning experience. In addition, since the PE file is windows specific, you are forced to learn how the Windows OS / API calls works to a degree.

Some results were unexpected, but most of the assumptions made in the theory and method were correct. For example, the entropy is usually much higher for the virus files, meaning most of them are likely packed. The number of strings was also as expected; the viruses had fewer strings in them on average compared to the benign system32 and random exe files found on Windows10.

The results were interesting to some degree, but they lack the confidence that those results represent packed malware. It is a fact that most malware creators with an ounce of self-respect should pack their viruses, but that does not mean that the viruses used in the experiments have to be packed. We can assume they are, but we cannot be sure. However, the results shown in Table 4.5 show that there is a big difference in the file sets used, which could mean that most of the virus files are packed.

6.0.3 Difference compared to other projects

The solution was honestly a bit weak, as it only included header reading, entropy calculation, dll/imported functions, section rights, number of sections, and amount of strings. The amount of strings is at least a bit unique compared to other solutions to this problem, but it does not make it that much different. If time allowed, there would be more additions to the static detection of a packed PE file. There would also be an attempt at using an x64_86 emulator to check the behavior of the file when executed.

6.1 Future Work

The packing problem is a continually developing one, malware developers and security professionals struggle to keep their opponent in check. It is a continuous battle where new packing methods and guarding mechanisms are innovated, and solutions against them are raised. In addition, antivirus software has to overcome the tricks employed by malware writers. Thus, this field will, as stated, always be in development, evil against good.

An excellent example of the eternal struggle would be the malware attempts to detect if OllyDbg debugs the current malware. As such, the community to OllyDbg responded with creating plugins to avoid those detection methods, which in turn, the malware creators made modifications to circumvent those plugins [15].

https://t.me/learningnets

Appendix A

Abbreviations and Glossaries

Abbriviations

- PE - Portable Executable (usually an .exe file)
- DEP - Data Execution Prevention
- JIT - Just in time (Code generation or execution)
- AV - Anti Virus
- DLL - Dynamic Link Library
- BASH - Bourne Again SHell
- GNU - GNU's Not Unix
- ELF - Format of Executable and Linking Format (Linux)
- VM - Virtual Machine
- SSL - Secure Socket Layer
- HDD - Hard disk drive
- RAM - Random Access Memory
- EP - Entry point, the address to the first instruction in an image file.
- FID - Function ID (method used by Ghidra and IDA to detect known functions like printf and the like)
- MASM - Microsoft Macro Assembler
- LZMA - Lempel-Ziv-Markov chain algorithm (used for lossless data compression)
- IP - Intellectual Property
- DMCA - Digital Millennium Copyright Act
- DRM - Digital Rights Management
- TLS - Thread Local Storage
- RSA - Rivest–Shamir–Adleman (public key cryptosystem)
- CIA - Central Intelligence Agency (National defense org in US)
- COFF - Common Object File Format

https://t.me/learningnets

Glossaries

- EXE - Executable program
- Mach-O - Apple equivalent for PE and ELF files
- UPX - A open source packer and unpacker for executables
- Capstone - Open source disassembler for python
- pefile - A Tool for reading PE headers and representing them as data
- WxorX - Policy for execution of code in a process.
- Scipy - Python Tool for calculating complex math
- linter - A program that scans program code looking for syntax errors, bugs, or bad practice code. A tool for developers.
- Stub - The stub is the unpacking routine to a packed PE file.
- Signature - A signature is something that is unique for a type of file, often called "magic bytes".
- DOS - Disk operating system for older versions of Microsoft Windows
- Bundler - In PE packing terms, it means that you add multiple payload PE files inside the packed PE file.

https://t.me/learningnets

Bibliography

[1] National Security Agency. *Ghidra*. Accessed: 02-04-2021. URL: https://www.nsa.gov/resources/everyone/ghidra/.

[2] H. S. Anderson and P. Roth. "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models." In: *ArXiv e-prints* (Apr. 2018). arXiv: 1804.04637 [cs.CR].

[3] Herbert Bos Andrew S. Tanenbaum. *Modern Operating Systems*. 2014. ISBN: ISBN-10: 1-292-06142-1, ISBN-13: 978-1-292-06142-9.

[4] Tom Brosch and Maik Morgenstern. "Runtime packers: The hidden problem." In: *Black Hat USA* (2006). Accessed: 05-28-2021.

[5] Capstone. *Capstone The ultimate disassembler*. Accessed: 12-4-2020. URL: https://www.capstone-engine.org/.

[6] Ken Chiang and Levi Lloyd. "A Case Study of the Rustock Rootkit and Spam Bot." In: *HotBots* 7.10-10 (2007). Accessed: 2-5-2021, p. 7.

[7] Yang-seo Choi et al. "PE File Header Analysis-Based Packed PE File Detection Technique (PHAD)." In: *International Symposium on Computer Science and its Applications*. Accessed: 1-18-2021. 2008, pp. 28–31. DOI: 10.1109/CSA.2008.28.

[8] Dhruwajita Devi and Sukumar Nandi. "PE File Features in Detection of Packed Executables." In: *International Journal of Computer Theory and Engineering* (Jan. 2012), pp. 476–478. DOI: 10.7763/IJCTE.2012.V4.512.

[9] Advanced Micro Devices. *AMD64 ArchitectureProgrammer's Manual*. Accessed: 4-28-2021. Mar. 2021. URL: https://www.amd.com/system/files/TechDocs/24593.pdf.

[10] Chris Domas. *M/o/Vfuscator2*. Accessed: 3-1-2021. URL: https://github.com/xoreaxeaxeax/movfuscator.

[11] In: *Research Methods for Cyber Security*. Ed. by Thomas W. Edgar and David O. Manz. Syngress, 2017, p. iv. ISBN: 978-0-12-805349-2. DOI: https://doi.org/10.1016/B978-0-12-805349-2.00017-0.

[12] Kaspersky IT Encyclopedia. *IM-Flooder*. Accessed: 05-26-2021. URL: https://encyclopedia.kaspersky.com/knowledge/im-flooder/.

[13] erocarrera. *pefile*. Accessed: 12-4-2020. URL: https://github.com/erocarrera/pefile.

[14] F-Secure. *Flooder*. Accessed: 05-29-2021. URL: https://www.f-secure.com/v-descs/flooder.shtml.

[15] Peter Ferrie. *Anti-Unpacker Tricks*. Accessed: 05-30-2021. 2007. URL: https://pferrie.tripod.com/papers/unpackers.pdf.

[16] Sergei Frankoff. *How Do Packers Work - Reverse Engineering "FUD" Aegis Crypter*. Accessed: 03-05-2021. Youtube. 2018. URL: https://www.youtube.com/watch?v=uxlpRof1QWs.

[17] hasherezade. *PE-bear: What is it?* Accessed: 02-3-2021. URL: https://hshrzd.wordpress.com/pe-bear/.

https://t.me/learningnets

[18] Horsicq. *Detect It Easy*. Accessed: 04-13-2021. URL: https://github.com/horsicq/Detect-It-Easy.

[19] Fu-Hau Hsu et al. "BrowserGuard: A Behavior-Based Solution to Drive-by-Download Attacks." In: *IEEE Journal on Selected Areas in Communications* 29.7 (2011), pp. 1461–1468. DOI: 10.1109/JSAC.2011.110811.

[20] Darren Dale John Hunter and Michael Droettboom Eric Firing. *Matplotlib: Python Plotting*. Accessed: 1-23-2021. 2012. URL: https://matplotlib.org/.

[21] Matthew Jones. *Top 13 Popular Packers Used in Malware*. Accessed: 05-30-2021. URL: https://resources.infosecinstitute.com/topic/top-13-popular-packers-used-in-malware/.

[22] Malwarebytes Labs. *Hacktool*. Accessed: 05-29-2021. URL: https://blog.malwarebytes.com/detections/hacktool/.

[23] Laszlo Molnar John Reiser Markus Oberhumer. *UPX - the Ultimate Packer for eXecutables*. Accessed: 11-16-2020. URL: https://github.com/upx/upx.

[24] Microsoft. *Debugbreak*. Accessed: 05-12-2021. URL: https://docs.microsoft.com/en-us/cpp/intrinsics/debugbreak?view=msvc-160.

[25] Microsoft. *LoadLibraryA function (libloaderapi.h)*. Accessed: 3-22-2021. May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya.

[26] Microsoft. *PE Format*. Accessed: 11-15-2020. URL: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format.

[27] Microsoft. *VirtualAlloc function*. Accessed: 1-12-2021. URL: https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc.

[28] Microsoft. *VirtualProtect function*. Accessed: 1-18-2021. URL: https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect.

[29] Deirde K Mulligan and Aaron K Perzanowski. "The magnificence of the disaster: Reconstructing the Sony BMG rootkit incident." In: *Berkeley Tech. LJ* 22 (2007). Accessed: 2-4-2021, p. 1157.

[30] Michal Necasek. *EXEPACK and the A20-Gate*. Accessed: 2-17-2021. Mar. 2018. URL: http://www.os2museum.com/wp/exepack-and-the-a20-gate/.

[31] Michal Necasek. *Realia SpaceMaker*. Accessed: 2-17-2021. Apr. 2018. URL: http://www.os2museum.com/wp/realia-spacemaker/.

[32] NTCore. *PE Detective*. Accessed: 4-29-2021. URL: https://ntcore.com/?page_id=367.

[33] *PE-detective image*. Accessed: 4-29-2021. URL: https://www.aldeid.com/wiki/Explorer-Suite/PE-Detective.

[34] Ziff-Davis Publishing. *If you use DOS you need this program*. Accessed: 4-3-2021. Jan. 1983. URL: https://books.google.no/books?id=vy3cBZkjbZgC&pg=RA3-PA417&redir_esc=y#v=onepage&q&f=false.

[35] Nguyen Anh Quynh. *capstone github*. Accessed: 1-3-2021. URL: https://github.com/aquynh/capstone.

[36] Thomas Roccia. *Malware Packers Use Tricks to Avoid Analysis, Detection*. Accessed: 06-2-2021. May 2017. URL: https://www.mcafee.com/blogs/enterprise/malware-packers-use-tricks-avoid-analysis-detection/.

[37] J. Sack and 1950 Harvard Andean Expedition. *The Butcher: The Ascent of Yerupaja*. Rinehart, 1952. URL: https://books.google.no/books?id=iT4xAAAAIAAJ.

[38] Martino Sani. *ASpack manual unpacking*. Accessed: 05-31-2021. URL: http://martinosani.it/2020/02/aspack-manual-unpacking.html.

https://t.me/learningnets

[39] Jason Scott. *VX Heavens Snapshot(2010-05-18)*. Accessed: 01-03-2021. URL: https://archive.org/details/vxheavens-2010-05-18.

[40] Robert Seacord. *Secure Coding in C and C++, 2nd Edition*. Accessed: 1-18-2021. Apr. 2013. ISBN: ISBN-10: 0-321-82213-7, ISBN-13: 978-0-321-82213-0.

[41] Panda Security. *Hoax: Definition*. Accessed: 05-29-2021. URL: https://www.pandasecurity.com/en-us/security-info/hoax/.

[42] Panda Security. *Virus Encyclopedia: Constructor*. Accessed: 05-23-2021. URL: https://www.pandasecurity.com/en-us/security-info/132932/information/Constructor.

[43] Jibz xineohP Snaker Qwerton. *app-peid*. Accessed: 2-17-2021. URL: https://github.com/wolfram77web/app-peid.

[44] ASPACK software. *What is ASProtect32?* Accessed: 05-31-2021. URL: http://www.aspack.com/asprotect32.html.

[45] Microsoft Defender Security Research Team. *Detecting reflective DLL loading with Windows Defender ATP*. Accessed: 5-3-2021. Nov. 2017. URL: https://www.microsoft.com/security/blog/2017/11/13/detecting-reflective-dll-loading-with-windows-defender-atp/.

[46] VirusTotal. *virustotal*. Accessed: 3-23-2021. URL: https://www.virustotal.com/gui/.

[47] Wei Yan, Zheng Zhang, and Nirwan Ansari. “Revealing Packed Malware.” In: *IEEE Security Privacy* 6.5 (2008), pp. 65–69. DOI: 10.1109/MSP.2008.126.

https://t.me/learningnets