Library for Static Analysis of PE Malware

by Katja Hahn

Master Thesis

HTWK Leipzig

Fakultät Informatik, Mathematik und Naturwissenschaften

First Assessor:	Prof. Dr. rer. nat. habil. Michael Frank (HTWK Leipzig)
Second Assessor:	Max Mustermann

Leipzig, September 2014

Contents

List of Figures

List of Tables

List of Acronyms

DLL	Dynamic-Link Library
EXE	Executable File
IAT	Import Address Table
NZ	New Executable
PE	Portable Executable
PE/COFF specification	*Microsoft Portable Executable and Common Object File Format Specification*
PE32+	Portable Executable with a 64-bit address space
PE32	Portable Executable with a 32-bit address space
RVA	Relative Virtual Address
VA	Virtual Address
VM	Virtual Machine

Chapter 1

Introduction

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, At accusam aliquyam diam diam dolore dolores duo eirmod eos erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd magna no rebum. sanctus sea sed takimata ut vero voluptua. est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat.

Consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus.

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo

Chapter 2

Malware

2.1 Malware Taxonomy

2.1.1 Behavioural Malware Types

Usually malware analysts make guesses about the malware's behaviour and shape their further analysis to confirm (or refute) these guesses. This approach helps to speed up the analysis. [4, p. 3] Hereafter is an overview to the different types of malware depending on its behaviour.

Definition 1 (Downloader) *A* downloader *is a piece of software that downloads other malicious programs. (cf. [4, p. 3])*

Definition 2 (Rootkit) *A* rootkit *is a software that has the purpose of hiding the presence of other malicious programs or activities. (cf. [4, p. 4])*

A rootkit may conceal login activities, log files and processes. Rootkits are often coupled with backdoor functionality (see definition 3).

Definition 3 (Backdoor) *A* backdoor *allows access to the system by circumventing the usual access protection mechanisms. (cf. [4, p. 3])*

The backdoor is used by the attacker or other malicious programs to get access to the system later on.

Definition 4 (Launcher) *A* launcher *is a software that executes other malicious programs. (cf. [4, p. 4])*

A launcher mostly uses unusual techniques for running the malicious program in the hopes of providing stealth.

Definition 5 (Spam-sending malware) Spam-sending malware *uses the victim's machine to send spam. (cf. [4, p. 4])*

Attackers use this kind of malware to sell their spam-sending services.

Definition 6 (Information stealer) *An* information stealer *is a malicous program that reads confidential data from the victim's computer and sends it to the attacker. (cf. [4, p. 4])*

Examples for information stealers are: keyloggers, sniffers, password hash grabbers [4, p. 3] and also some kinds of deceptive malware. The latter makes the user input confidential data by convincing the user that it provides an advantage. An example for a deceptive information stealer is a program that claims to add more money to the user's Paypal account; actually it sends the Paypal credentials the user puts into the program to the attacker's e-mail server.

Definition 7 (Botnet) *A* botnet *is a collection computer programs on different machines that recieve and execute instructions from a single server.*

While some botnets are used legally, malicious botnets are installed without consent of the computer's owners and may be used to perform distributed denial of service attacks or for spam-sending (see definition 5).

Definition 8 (Scareware) Scareware *tries to trick a user into buying something by frightening him. (cf. [4, p. 4])*

A typical scareware example is a program that looks like an antivirus scanner and shows the user fake warnings about malicious code found on the system. It tells the user to buy a certain software in order to remove the malicious code.

Definition 9 (Virus) *A* virus *recursively replicates itself by infecting or replacing other programs or modifying references to these programs to point to the virus code instead. A virus possibly mutates itself with new generations. (cf. [9, p. 27, 36])*

A typical virus is executed if the user executes an infected host file.

Definition 10 (Worm) "Worms *are network viruses, primarily replicating on networks." [9, p. 36]*

Typically worms don't need a host file and execute themselves without the need of user interaction. [9, p. 36] But there are exceptions from that: e. g. worms that spread by mailing themselves need user interaction. A worm is a subclass of a virus by definition 10.

2.1.2 Mass Malware and Targeted Malware

Malware is not only classified by behaviour, but also by the attacker's goals. If the malware was designed to infect as many machines as possible, it is a *mass malware.* A *targeted malware* on the other hand was written to infect a certain machine, organization or company.

2.2 Malware Analysis

Definition 11 "Malware analysis *is the art of dissecting malware to understand how it works, how to identify it, and how to defeat or eliminate it." [4, p. xxviii]*

Static Analysis

Definition 12 Static analysis *is the examination of a program without running it. [4, p. 2]*

Static analysis includes e. g. viewing the file format information, finding strings or patterns of byte sequences, disassembling the program and subsequent examination of the intructions.

Dynamic Analysis

Definition 13 Dynamic analysis *is the examination of a program while running it. [4, p. 2]*

Dynamic analysis includes e. g. observing the program's behaviour in a Virtual Machine (VM) or a dedicated testing machine or examining the program in a debugger.

2.3 Malware Detection by Antivirus Software

2.4 Malware Hiding Techniques

Chapter 3

Portable Executable Format

The Portable Executable (PE) is a file format for image files used by Microsoft products for 32- and 64-bit system architectures. It is the successor of the New Executable (NZ) file format for 16-bit systems. The PE format is described in the *Microsoft Portable Executable and Common Object File Format Specification* (PE/COFF specification) [6]

PE file types, which are relevant for this thesis, are Dynamic-Link Library (DLL) and EXE files. DLL files export functions or data other programs can use. They can have various file endings, including *.sys*, *.dll*, *ocx*, *.cpl* and *.drv*. (cf. [5]) A DLL is loaded into the context of another process. EXE files have the file ending *.exe*. They usually don't export any symbols. The system creates a new process upon launching the EXE. The system recognizes the file type by a certain flag in the PE headers. (see page 8)

other file types? FON?

Both, EXE and DLL files, are considered as *image files* by the PE/COFF specification, because they have been processed by a linker and are used as input for the loader. In contrast to image files are *object files* (Common Object File Format or COFF), which are used as input for a linker (cf. [6, p. 8]). The Common Object File Format is not an issue in the thesis.

PortEx extracts the information from the PE format to assist in analysing malware. Therefore knowledge about the PE format is neccessary to understand the inner workings of the library *PortEx*.

3.1 General Concepts

add section with terms (VA, RVA, linker, section)? Or: explain how PE is loaded and explain terms there?

This section explains some frequent terms that are also used by the PE/COFF specification and necessary to understand the descriptions of the PE format.

Definition 14 (RVA) Relative Virtual Addresses (RVA) *are used while the image file is loaded in memory. They are relative to the base address of the image file, which is the address of the first byte where the image is loaded in memory (cf. [6, p. 19]).*

Definition 15 (VA) *A* Virtual Address (VA) *is the same as a Relative Virtual Address (RVA) with the base address added (cf. [6, p. 19]).*

Definition 16 (section) *A defined unit of data or code within an image file is called a* section *(cf. [6, p. 19]). Sections are defined by their section header in the Section Table.*

Definition 17 (entry point) *The* entry point *is a Relative Virtual Address (RVA) to the starting address for EXE files or to the initialization function for device drivers. (cf. [6, p. 18])*

define def: prefix for autoref

3.2 General Structure

Figure 3.1 illustrates the structure of a PE file. It consists of the so called PE Header, followed by the Section Table and the sections. The overlay is optional data appended to the file. As different resources use the term *PE Header* with variable meanings , the following defintion by the PE/COFF specification will be used in the following:

or not use it commonly

avoid konjunktivb

Definition 18 (PE Header) *The* PE Header *"consists of a MS-DOS stub, the PE signature, the COFF file header, and an optional header." [6, p. 11]*

The different parts of the PE are explained hereafter.

A PE file always starts with the MS-DOS Stub. This is an application which is able to run in MS-DOS. The standard MS-DOS Stub prints the message "This program cannot be run in DOS mode" and closes right after.

The operating system determines the file format by looking for specific signatures. The file format signature is usually at the very beginning of the file. Since the PE starts with the MS-DOS Stub, which has a file format signature itself, the PE signature is placed after. The offset to the PE signature is defined in location 0x3c of the MS-DOS Stub, thus enables Windows to properly execute the PE file. Right after the PE signature follows the PE Header.

MZ, PE00

The first part of the PE Header is the COFF File Header. It contains information about the type of the target machine, the number of sections, a time date stamp that indicates when the file was created, the size of the Optional Header and flags that indicate file characteristics including a flag that indicates whether the file is a DLL.

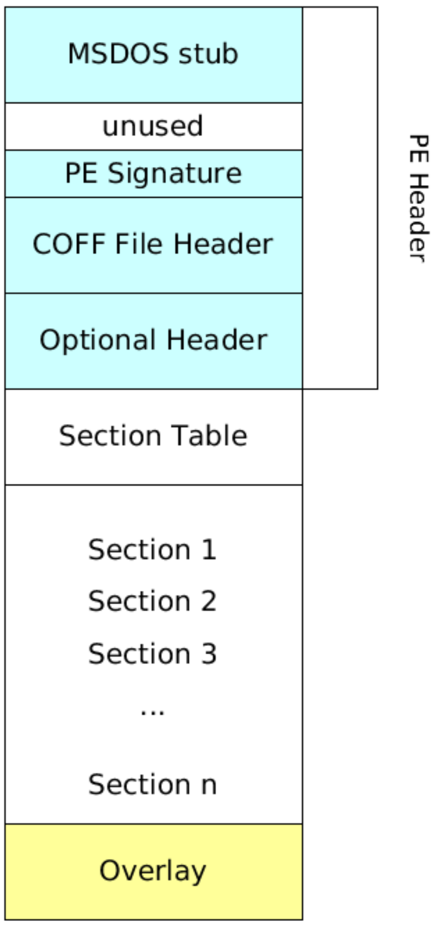


Figure 3.1: Structure of a PE file

The Optional Header follows right after the COFF File Header at a certain offset from the beginning of the PE Header. Despite its name the Optional Header is mandatory for image files. Only object files don't need it. The Optional Header has three parts: Standard Fields, Windows Specific Fields and a Data Directory Table.

The Standard Fields of the Optional Header contain information necessary for loading and running the file. They determine for example, whether the image file allows a 64-bit address space (PE32+) or is limited to a 32-bit address space (PE32). They also declare i. a. the size of initialized and uninitialized data, the size of the code, the linker versions and the entry point (see PE Header 17) of the image file.

The Windows Specific fields provide additional information for the Windows loader and linker like the operating systems the image file can run on, alignment values, dll characteristics and the number of data directories in the Data Directory Table.

A Data Directory Table entry consists of address and size for a table or string that the system uses. Examples are the import table, the export table and the resource table. (see [6, pp. 24])

The Section Table is placed right after the Optional Header. It consists of the section headers for the sections that make up the rest of the PE file. A section header describes i. a. characteristics, size, name and location of a section.

While the PE Header and Section Table described above are located at a fixed file offset, the rest of the PE contains data defined by pointers in the PE Header or the Section Table. The sections contain arbitrary data, only some sections have a special meaning and are explained in *Special Sections* below. An EXE file has at least one section containing executable code.

Data that was appended to the file, but is not part of the PE format is called *overlay*. Overlay is not mapped into memory. The overlay is used by some applications as a way to store arbitrary data without having to deal with the PE format.

3.3 Special Sections

Sections may contain arbitrary information, which is only relevant to the application using them; but some sections have a special meaning. Their format is described in the PE/COFF specification [6]. These sections are recognized by entries in the Data Directory Table of the Optional Header or certain flags in the Section Table. They have typical section names which are also used in the PE/COFF specification to refer to the sections. These names are not mandatory, but a convention. That's why they can not be relied on while trying to find certain sections in a PE. Not only malware writers misuse the section names to obscure their purpose, but also legimate compilers and packers violate the convention. Some of these special sections are described right after.

example!

sections that are recognized by portex

Export Section

The *.edata Section* or *Export Section* is generally found in DLLs. The section begins with the Export Directory Table, which contains general information and addresses to resolve imports from this section. The Export Directory Table points to an array of addresses called Export Address Table. Each address either points to code or data within the current image file, or is a forwarder address which points to a symbol in another DLL.

Other image files have two ways to import symbols from the current image file: They either use an index into the Export Address Table (the index is also called *ordinal*) or they use a public name of the symbol. Ordinals are defined in the Ordinal Table; public names are defined in the Export Name Table.

Entries of the Ordinal Table correspond to the Export Name Pointer Table entries by their position. Every entry is an ordinal that represents an index in the Export Address Table.

The Export Name Pointer Table is an array of addresses which point to names of the Export Name Table. These names are null-terminated ASCII strings. They are the public names that other image files can use to import the symbols.

Listing 3.1 shows example contents for a DLL with two exported symbols: `DLL2Print` and `DLL2ReturnJ`. It also illustrates in lines 34-40 how the information from the different tables is combined.

Listing 3.1: Example for Export Section contents, output by *PortEx*

```
 1  Export Directory Table
 2  ......................
 3
 4  Minor Version: 0 (0x0)
 5  Address Table Entries: 2 (0x2)
 6  Ordinal Base: 1 (0x1)
 7  Name Pointer RVA: 31664 (0x7bb0)
 8  Export Flags: 0 (0x0)
 9  Ordinal Table RVA: 31672 (0x7bb8)
10  Number of Name Pointers: 2 (0x2)
11  Major Version: 0 (0x0)
12  Time/Date Stamp: 1317493556 (0x4e875b34)
13  Name RVA: 31676 (0x7bbc)
14  Export Address Table RVA: 31656 (0x7ba8)
15
16  Export Address Table
17  ....................
18
19  0x1030, 0x1050
20
21  Name Pointer Table
22  ..................
23
24  RVA    ->  Name
25  ****************
26  (0x7bc5,DLL2Print)
```

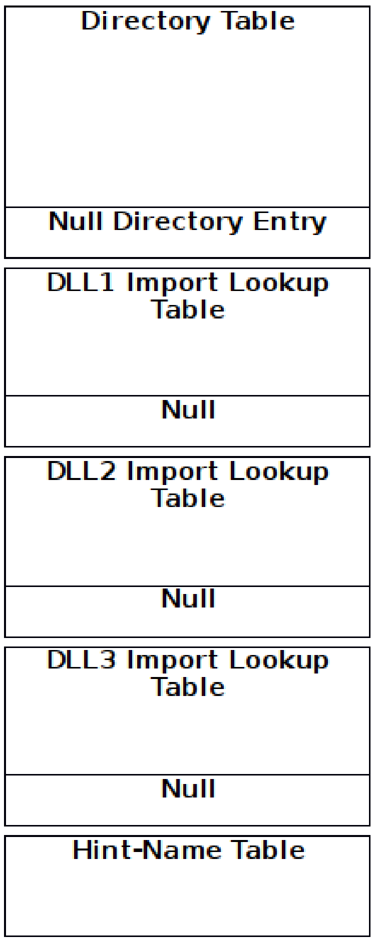


Figure 3.2: Typical Import Section Layout by [6, p. 61]

```
(0x7bcf,DLL2ReturnJ)

Ordinal Table
..............

1, 2

Export Entries Summary
----------------------

Name, Ordinal, RVA
...................
DLL2Print, 1, 0x1030
DLL2ReturnJ, 2, 0x1050
```

Import Section

Every image file that imports symbols has an *Import Section*, also called *.idata Section*. The Import Section contains the Import Directory Table, several Import Lookup Tables, the Hint-Name Table and the Import Address Table (IAT). A typical layout of the Import Section is in Figure 3.2

Every Import Directory Table entry points to an Import Lookup Table. Each Import Lookup Table describes the imported symbols of a single DLL.

The Hint-Name table entries have a hint and an ASCII name of the import. Each hint is an index to the Export Name Pointer Table (see section 3.3) of the DLL the image is importing from. Hints speed up the lookup of imports.

Null entries mark the end of the Import Directory Table and the Import Lookup Table.

The IAT is identical to the Import Directory Table except while the image is bound. In the latter case the IAT entries are overwritten with memory addresses of the imported symbols.

Resource Section

Resources of a PE can be i. a. icons, text, windows, copyright information. They are saved as an entry in the *Resource Section*, which also has the name *.rsrc Section*. The Resource Section is build up as a tree with the actual resource addresses as leaves. While 2^{31} tree levels can be used according to the PE/COFF specification [6, p. 100], Windows only uses three levels with the first level node being the type, the second being the name and the third being the language information. Figure 3.3 illustrates the structure of a resource tree.

Debug Section and Debug Directory

Whereas most sections can be at an arbitrary location in the file, the *Debug Section* (aka *.debug Section*) must be placed at the very end of the image file. The reason is that the loader doesn't map this section into memory. Image files contain per default of the linker only a Debug Directory (as pointed to by the Data Directory Table), but not a Debug Section (see [6, p. 78]). Thus the Debug Directory is either located in the Debug Section if it exists, in any other section of the PE or not in any section at all.

Every Debug Directory entry defines i. a. size, location and the type (format) of a debug information block. An example is in Listing 3.2

Listing 3.2: Example for a Debug Section entry, output by *PortEx*

```
Debug Section
-------------
```

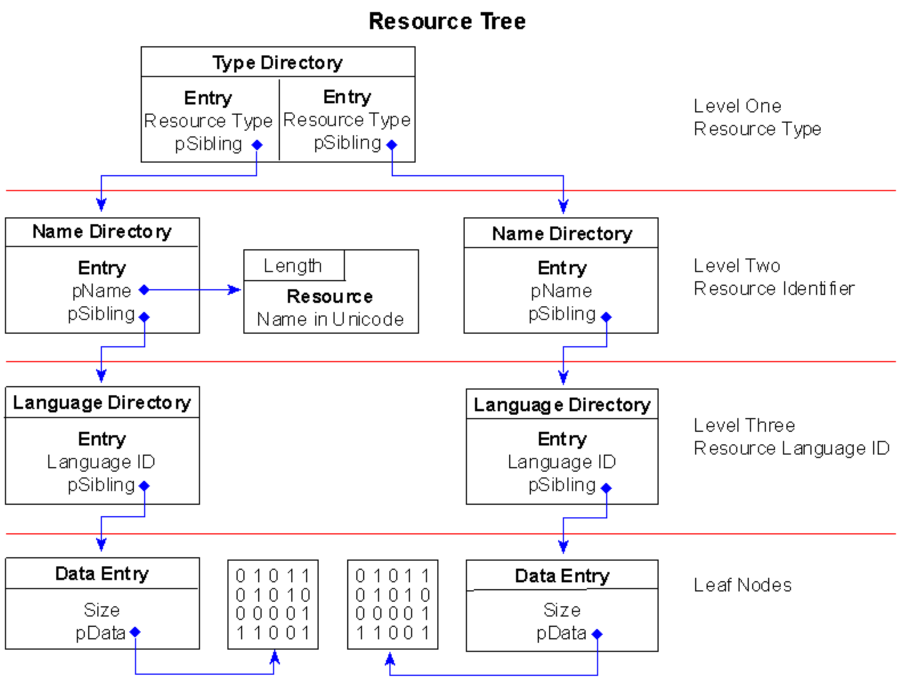


Figure 3.3: Resource tree structure by [3]

```

Address of Raw Data: 828652 (0xca4ec)
Minor Version: 0 (0x0)
Size of Data: 35 (0x23)
Characteristics: 0 (0x0)
Major Version: 0 (0x0)
Pointer to Raw Data: 764140 (0xba8ec)
Type: Visual C++ debug information
Time date stamp: Thu Oct 14 16:33:20 CEST 2010
```

3.4 Loading Process

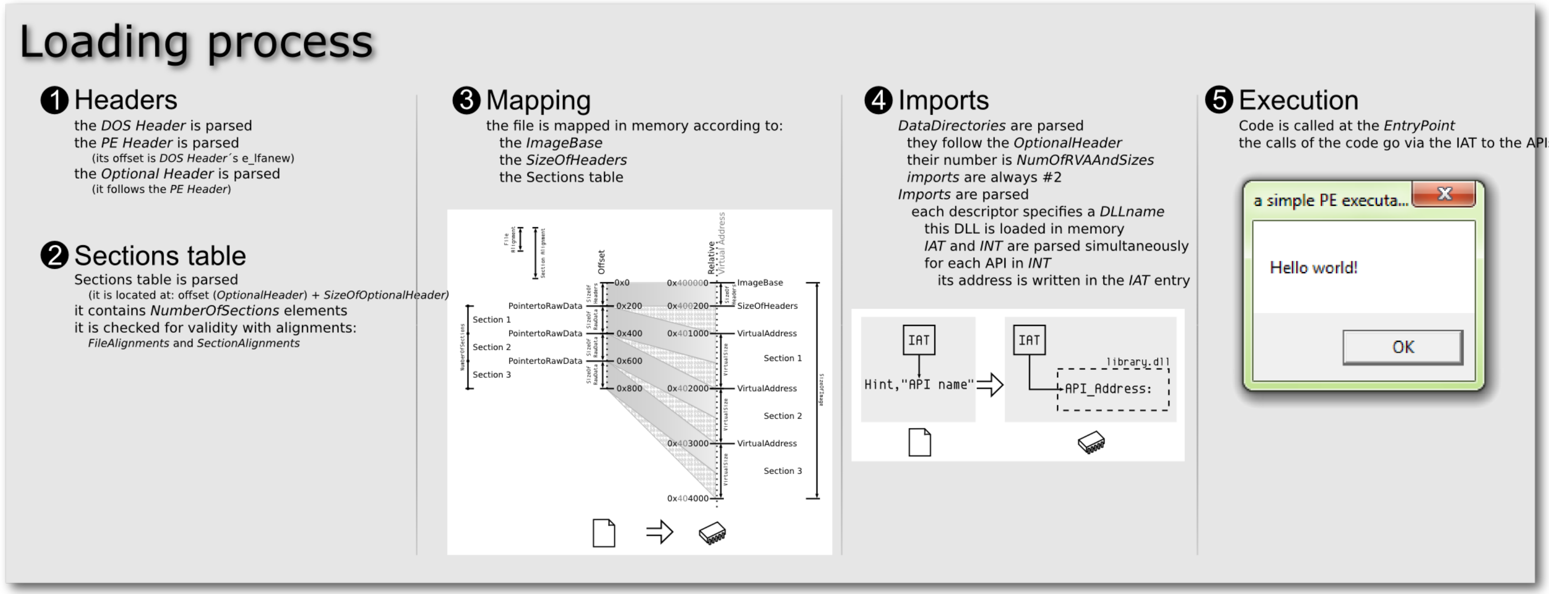
test [1] why?

add reference to figure

3.5 PE Malformations

Definition 19 (Malformation) *A* malformation *is data or layout of a PE file that violates the PE/COFF specification.*

Malformations are either accidental results of PE modifications or done on purpose. A malformation doesn't necessarily prevent the Windows loader from



Loading process

❶ Headers

the *DOS Header* is parsed
the *PE Header* is parsed
(its offset is *DOS Header´s* e_lfanew)
the *Optional Header* is parsed
(it follows the *PE Header*)

❷ Sections table

Sections table is parsed
(it is located at: offset (*OptionalHeader*) + *SizeOfOptionalHeader*)
it contains *NumberOfSections* elements
it is checked for validity with alignments:
FileAlignments and *SectionAlignments*

❸ Mapping

the file is mapped in memory according to:
the *ImageBase*
the *SizeOfHeaders*
the Sections table

❹ Imports

DataDirectories are parsed
they follow the *OptionalHeader*
their number is *NumOfRVAAndSizes*
imports are always #2
Imports are parsed
each descriptor specifies a *DLLname*
this DLL is loaded in memory
IAT and *INT* are parsed simultaneously
for each API in *INT*
its address is written in the *IAT* entry

❺ Execution

Code is called at the *EntryPoint*
the calls of the code go via the IAT to the APIs

Figure 3.4: Loading Process of a PE by [1]. Note that the term PE Header is used synonymously to COFF File Header in this graphic

running the file. The Windows loader doesn't work in full compliance with the PE/COFF specification to maintain backward compatibility with obsolete compilers and files. Malware writers utilize the loader's behaviour to create normally working PE files that can not be parsed by most tools used for malware analysis. Some malformations, like setting reserved fields, are also done to hide information in a PE; for example marking a host file as infected to prevent a virus from infecting it twice.

Accidental malformations occur if the malware writer doesn't know the PE/COFF specification enough to perform modifications in compliance with it. An example is a virus that spreads by adding a new section to the host file, copying the own code into it and changing the entry point to the beginning of the new section. The changes done to the host file can lead to subsequent malformations without impairing the Windows loader while running the file.

Sheehan et al states that 68 % of all image files have malformations. [7, slide 7] Because *PortEx* specializes in PE malware, one goal of *PortEx* is to parse malformated PEs correctly and to recognize malformations.

The following malformation examples are modifications done on purpose to deceive malware analysis tools.

3.5.1 Simple Malformations

A malformation is *simple* if it concerns a single field or data table in the PE (cf. [10, slide 7]). The malformation examples described hereafter are in ascending order of their complexity.

remove?

Too Many Sections

According to the PE/COFF specification the number of sections is limited to 96 [6, p. 12]. While Windows XP refuses to execute a PE with more sections, Windows Vista, 7 and 8 run it nevertheless. The number of sections is defined in the COFF File Header in a 16-bit value. If the maximum number of 0xffff (65535) sections is set, tools might fail to allocate enough memory to load the file and crash subsequently. An example is IDA v5.3.

PE Header in Overlay

pespecreferences, usual vs change

The PE Header usually follows after the MS-DOS Stub

As explained in section 3.2 the address to the beginning of the PE signature and the following PE Header is located in file offset 0x3c within the MS-DOS Stub. This address is a 32-bit value and can be changed to point to the overlay of the file as illustrated in Figure 3.5. (cf. [10, slide 13])

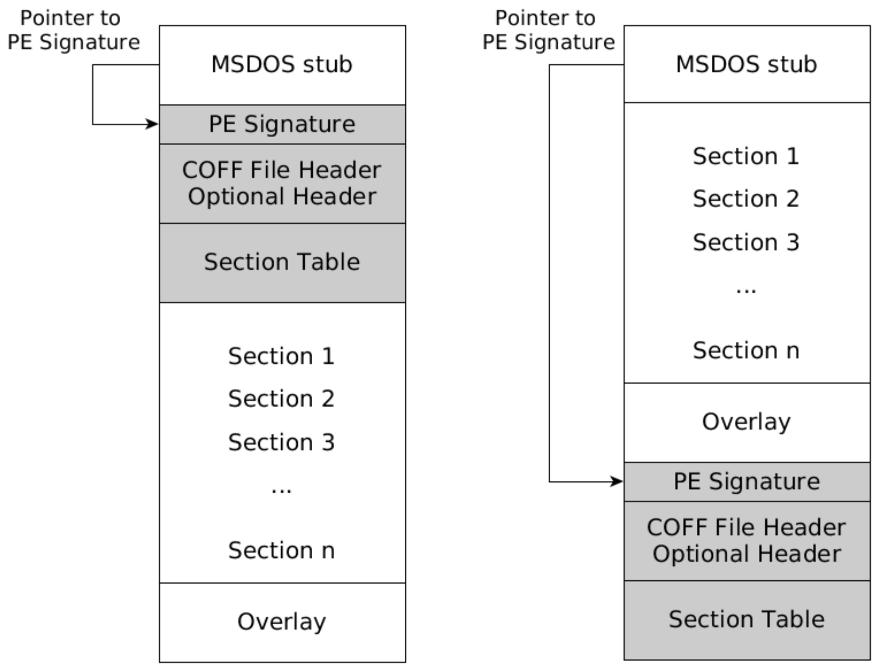


Figure 3.5: PE Header in Overlay (cf. [10, slide 13])

The overlay is never mapped into memory, so the PE Header won't be present in memory as well. The Windows loader reads it from disk and the file will be executed nevertheless. Tools that read the PE Header from memory can break because of that malformation.

Section Table in Overlay

The idea from the previous malformation is extended by moving only the Section Table to the overlay. The Optional Header has a variable size. The offset from the beginning of the Optional Header and its size determine the beginning of the Section Table. The size is defined in the field `SizeOfOptionalHeaders` the COFF File Header. It is a 16-bit value, so the maximum value for the size is 65535. If the end of the file is smaller than the offset of the Optional Header plus its size, the Section Table can be moved to the very end of the file. Figure 3.6 illustrates the malformation.

As a result of this modification the Section Table will not be mapped to memory. A tool that parses the memory content will not be able to find the Section Table. Pericin demonstrates this in his talk at the BlackHat Conference with the debugger OllyDbg. [10, min. 14:45]

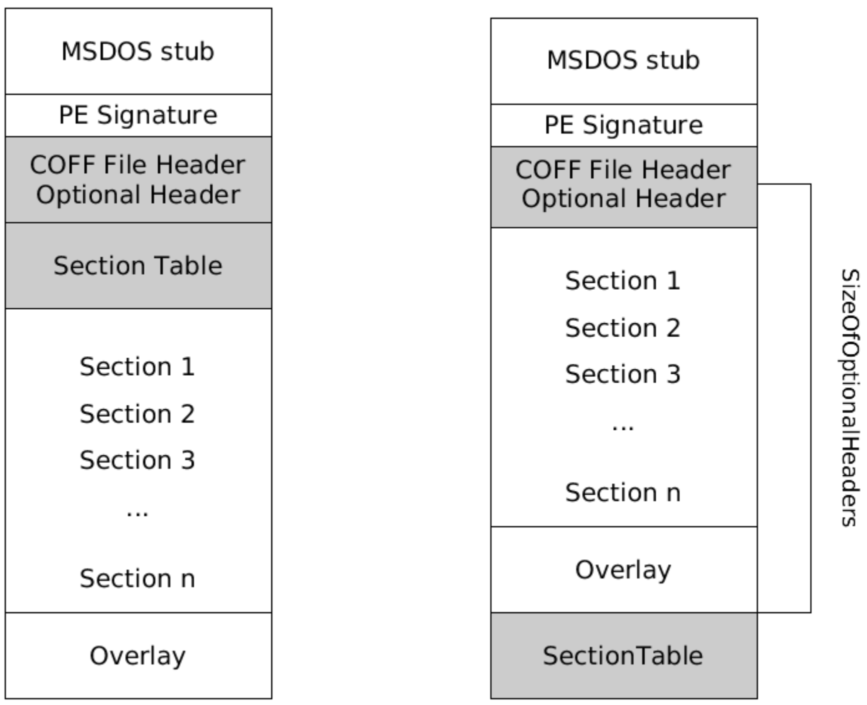


Figure 3.6: Section Table in Overlay (cf. [10, slide 14])

Dual PE Header

The `SizeOfHeaders` field in the Optional Header defines the "combined size of an MS-DOS Stub, PE header, and section headers" [6, p. 20]. But it also determines the Virtual Address (VA) of the first section implicitly. [10, slide 15]

If the `SizeOfHeaders` field is modified to a smaller value, the modification will result in loading only part of the PE Header to memory. The Windows loader still processes most of the PE Header on disk regardless of the `SizeOfHeaders` field. So setting the VA of the first section in the Section Table to the same value of the `SizeOfHeaders` field will make the section part of the PE Header in memory whereas the PE Header read on disk is another one. Figure 3.7 illustrates the malformation.

The Data Directory Table is parsed from memory by the operating system. That means writing a different continuation of the PE Header to the first section will result into e. g. different imports or exports used by the operating system than the ones parsed on disk by reverse engineering tools.

Writeable PE Header

The PE Header usually has only `execute` and `read` attributes enabled. There are two possibilities to make it writeable.

The first possibilitiy is setting the `FileAlignment` and the `SectionAlignment` to 0x200 (512) or smaller. Both values are part of the Optional Header. 0x200 is the minimum value according to the PE/COFF specification [6, p. 19]. If the value in the PE Header is smaller, the Windows loader rounds it up to 0x200. These changes force the Windows loader to provide full access to the PE Header. (cf. [2, p. 6])

The second possibility is placing the PE Header in a section as explained in *Dual PE Header* and enabling the `write` attribute in the section header.

This malformation is necessary for some of the complex malformations following after.

3.5.2 Complex Malformations

A malformation is complex if it concerns multiple fields or data tables in the PE (cf. [10, slide 7]).

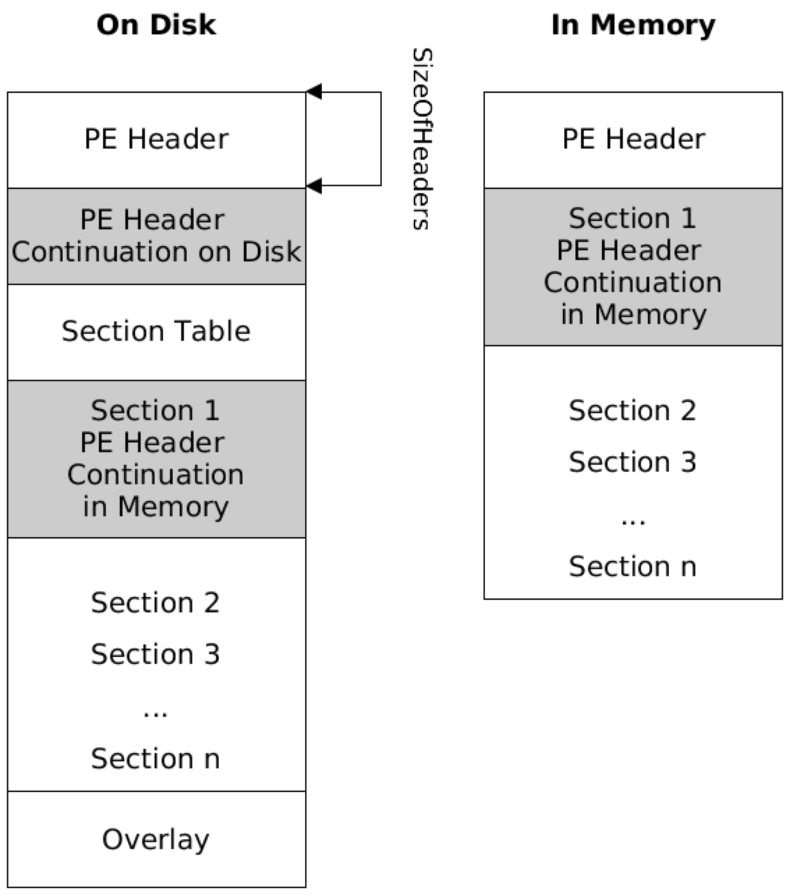


Figure 3.7: Dual PE Header

No Sections

Despite having no sections a PE is able to execute code by placing instructions within the PE Header. This is demonstrated by Sotirov in his *Tiny PE* project. [8]

To make this work, the file has to be modified so that the physical offsets are the same as their virtual counterparts. The PE Header is made writable as explained in *Writeable PE Header* and the number of sections is set to 0. The `SizeOfHeaders` field in the Optional Header needs to be large to enough to make Windows load the whole file into memory. That way the PE can execute code inside its PE Header.

A PE without sections may be partly unreadable by reverse engineering tools as a result of incorrect address conversions; or might be able to break tools that expect the file to have at least one section. An example is Hiew v8.03 which doesn't recognize a sectionless PE as PE file.

Chapter 4

Static Analysis Library

Chapter 5

Evaluation

Bibliography

[1] Corcami: *A Windows Executable Walkthrough.* https://code.google.com/p/corkami/wiki/PE101?show=content.

[2] Corporation, ReversingLabs: *Undocumented PECOFF.* BlackHat USA 2011, Las Vegas, 2011.

[3] Kath, Randy: *The Portable Executable File Format from Top to Bottom.* http://www.csn.ul.ie/~caolan/publink/winresdump/winresdump/doc/pefile2.html, 2013.

[4] Michael Sikorski, Andrew Honig: *Practical Malware Analysis.* No Starch Press, Inc., 2012.

[5] Microsoft Cooperation: *What is a DLL?* https://support.microsoft.com/kb/815065/EN-US, December 2007.

[6] Microsoft Cooperation: *Microsoft PE and COFF specification.* http://msdn.microsoft.com/library/windows/hardware/gg463125, 2013.

[7] Sheehan, Casey, Nick Hnatiw, Tom Robinson and Nick Suan: *Pimp My PE: Parsing Malicious and Malformed Executables.* Virus Bulletin Conference 2007. Sunbelt Software, 2007.

[8] Sotirov, Alexander: *Tiny PE.* http://www.phreedom.org/research/tinype/.

[9] Szor, Peter: *The Art of Computer Virus Research and Defense.* Addison Wesley Professional, February 2005.

[10] Vuksan, Mario and Tomislav Pericin: *Constant Insecurity: Things You Didn't Know About (PECOFF) Portable Executable Format.* BlackHat USA 2011, Las Vegas. ReversingLabs, 2011.

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig, ohne Hilfe Dritter verfasst habe, dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und Zitate kenntlich gemacht habe. Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden.

Leipzig, den ____________________ ____________________

Unterschrift