Chapter 6

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) [4]

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. [3]) A DLL is loaded into the context of another process. EXE files have the file ending *.exe*. They usually don't export any functions. 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 6.1)

Both, EXE and DLL files, are considered as *image files* by the PE/COFF specification, because they contain executable code. In contrast to image files are object files (Common Object File Format or COFF), which don't contain executable code. The Common Object File Format is not an issue in the thesis.

citation

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*.

6.1 General Structure and PE Headers

Figure 6.1 illustrates the structure of a PE file. An executable 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.

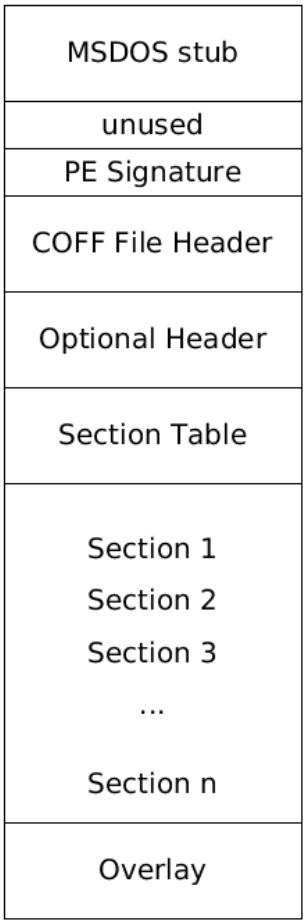

Figure 6.1: Structure of a PE file

Signatures are used to determine if a file is of a certain file format. 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 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.

MZ, PE00

Right after the signature follow the COFF File Header, the Optional Header and the Section Table. The COFF File Header 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.

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 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 or the resource table.

references

The Section Table 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 Headers described above are located at a fixed file offset, the rest of the PE contains data defined by pointers in the PE Headers. Data that was appended to the file, but is not part of the PEformat is called *overlay*. Overlay is not mapped into memory. The overlay is used by some applications as an easy way to store arbitrary data.

6.2 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 [4]. 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. Some of these special sections are described right after.

sections that are recognized by portex

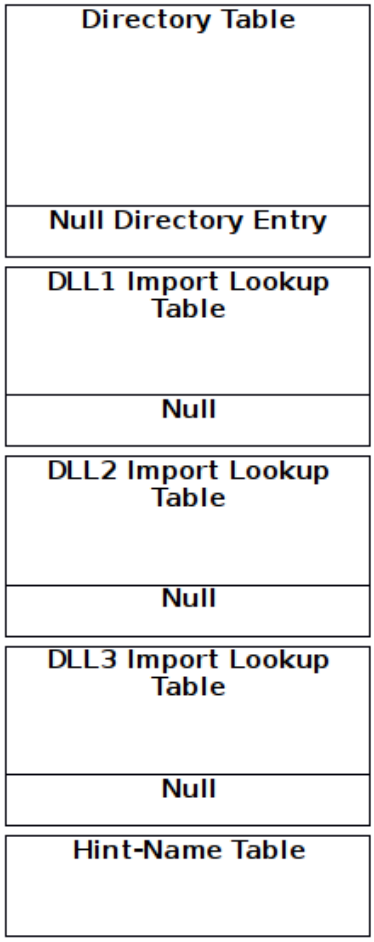

Figure 6.2: Typical Import Section Layout by [4, p. 61]

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 6.2

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 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.

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 6.1 shows example contents for a DLL with two exported symbols: `DLL2Print` and `DLL2ReturnJ`.

Listing 6.1: 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)
```

```
(0x7bcf,DLL2ReturnJ)

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

1, 2

Export Entries Resolved
-----------------------

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

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 [4, 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 6.3 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 [4, 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 6.2

Listing 6.2: Example for a Debug Section entry, output by *PortEx*

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

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
```

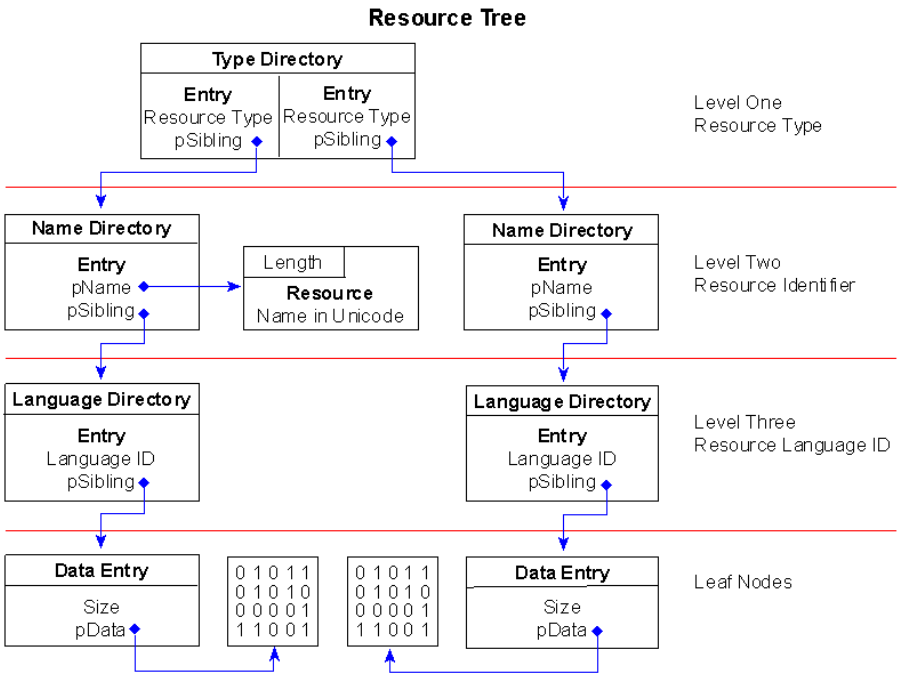

Figure 6.3: Resource tree structure by [1]

Bibliography

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

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

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

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

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