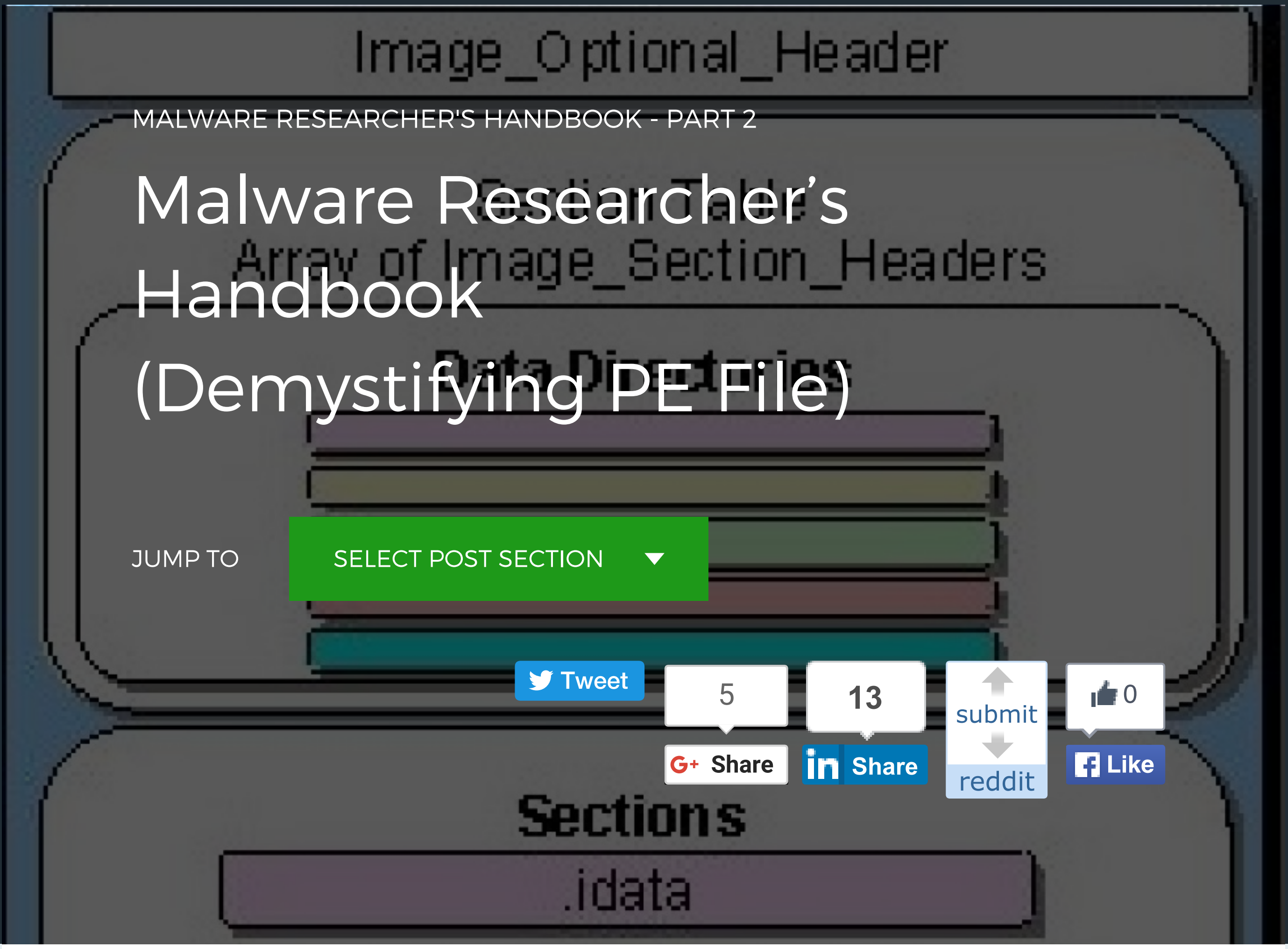


MALWARE RESEARCHER'S HANDBOOK - PART 2

Malware Researcher’s Handbook (Demystifying PE File)

JUMP TO SELECT POST SECTION

Tweet 5 Share 13 Share submit reddit 0 Like

(For the Introduction, click here:)

PE File

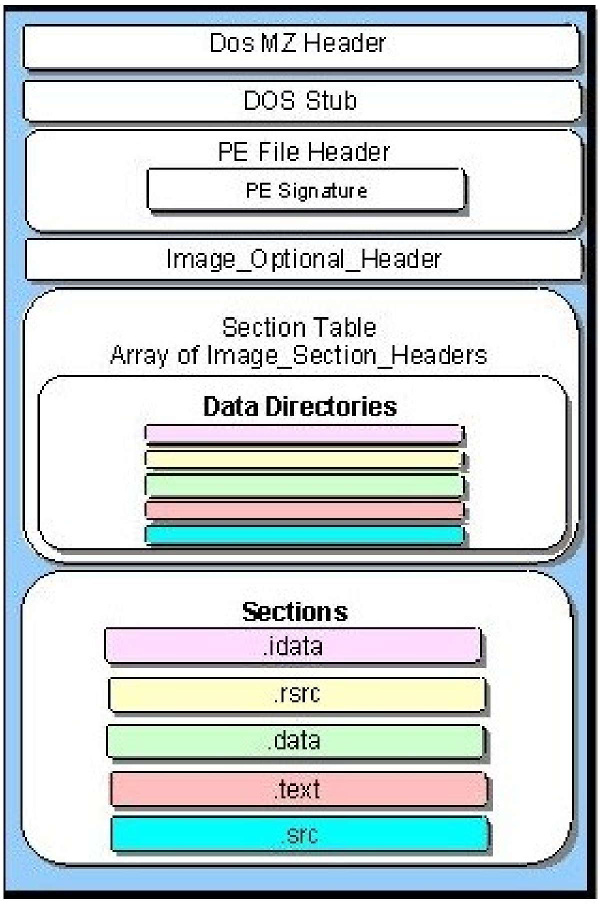
Portable executable file format is a type of format that is used in Windows (both x86 and x64).

As per Wikipedia, the portable executable (PE) format is a file format for executable, object code, DLLs, FON font files, and core dumps.

The PE file format is a data structure that contains the information necessary for the Windows OS loader to manage the wrapped executable code. Before PE file there was a format called COFF used in Windows NT systems.

Basic Structure

A PE executable basically contains two sections, which can be subdivided into several sections. One is Header and the other is Section. The diagram below explains everything



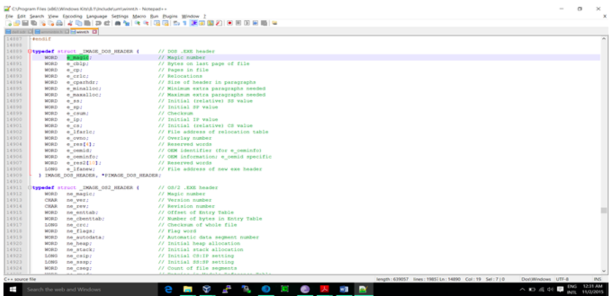
DOS Header:

DOS header starts with the first 64 bytes of every PE file. It's there because DOS can recognize it as a valid executable and can run it in the DOS stub mode. As we can investigate on the winnt.h/Windows.inc we can see below details:

Same thing can be found on the cff-explorer which is very popular malware analysis tool for PE file validation. We also have a same implementation as a picture

As we can see we have a list of structure that came under DOS header. We will not discuss everything as it is beyond our scope; we will discuss important ones that are required, such as **magic** and **ifanew** structure.

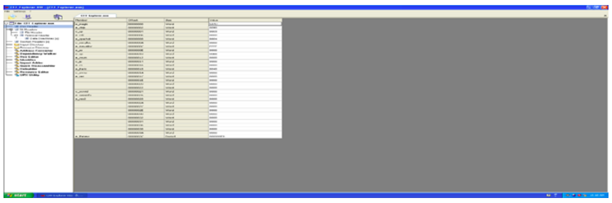
The first field, e_**magic**, is the so-called magic number. This field is used to identify an MS-DOS-compatible file type. All MS-DOS-compatible executable files set this value to 0x54AD, which represents the ASCII characters MZ. MS-DOS headers are sometimes referred to as MZ headers for this reason. It starts at offset 0 (this can be view with a hex editor). This one is developed by Mark Zbikowski (MZ)



```
#endif

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

typedef struct _IMAGE_OS2_HEADER {      // OS/2 .EXE header
    WORD   ne_magic;                    // Magic number
    CHAR   ne_ver;                      // Version number
    CHAR   ne_rev;                      // Revision number
    WORD   ne_enttab;                   // Offset of Entry Table
    WORD   ne_cbenttab;                 // Number of bytes in Entry Table
    LONG   ne_crc;                      // Checksum of whole file
    WORD   ne_flags;                    // Flag word
    WORD   ne_autodata;                 // Automatic data segment number
    WORD   ne_heap;                     // Initial heap allocation
    WORD   ne_stack;                    // Initial stack allocation
    LONG   ne_csip;                     // Initial CS:IP setting
    LONG   ne_sssp;                     // Initial SS:SP setting
    WORD   ne_cseg;                     // Count of file segments
```

C:\Program Files (x86)\Windows Kits\8.1\include\um\winnt.h - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
winnt.h
C++ source file
Search the web and Windows

We can see above that we have list of structure for Image_DOS_Header and the important header, as we already discussed

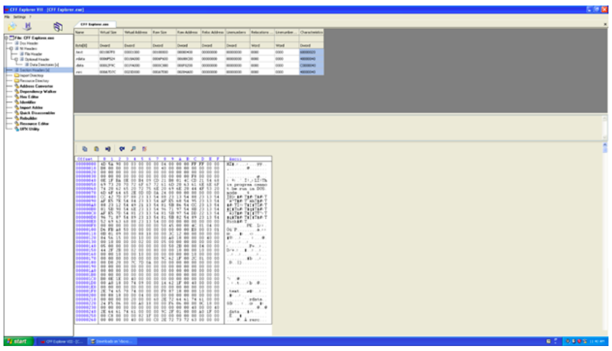


Here, using CFF, explorer we can verify the offset value of the structure and DOS MZ header and we also see that the file has the data type WORD.

ifanew is the only required element (besides the signature) of the DOS HEADER to turn the EXE into a PE. It is relative offset to the NT headers. The windows loader looks for this offset so it can skip the DOS stub and go directly to the PE header.

DOS Stub:

The DOS stub usually just prints a string, something like the message, “This program cannot be run in DOS mode.” It can be a full-blown DOS program. When building applications on Windows, the linker sends instruction to a binary called winstub.exe to the executable file. This file is kept in the address 0x3c, which is offset to the next PE header section.



The address F8000000 and the offset at the address 000000F8, where the PE starts, means the offset to the PE address and that is at the 0x00000030 address. So we are at the right address.

PE File Header

Like other executable files, a PE file has a collection of fields that defines what the rest of file looks like. The header contains info such as the location and size of code, as we discussed earlier. The first few hundred bytes of the typical PE file are taken up by the MS-DOS stub. The PE file is located by indexing the e_ifanew of the MS DOS header. The e_ifanew simply gives the offset to the file, so add the file’s memory-mapped address to determine the actual memory-mapped address.

There are some basic sub-sections defined in the header section itself; they are listed below:

Signature

Machines

NumberOfSections

SizeOfOptionalHeader

Characteristics

Signature: It only contains the signature so that it can be easily understandable by windows loader. The letters P.E. followed by two 0’s tells everything.

Machines: This is a number that identifies the type of machine on the target system, such as Intel, AMD, etc. We will target a basic structure like Intel, as shown below:

0x14d	Intel i860

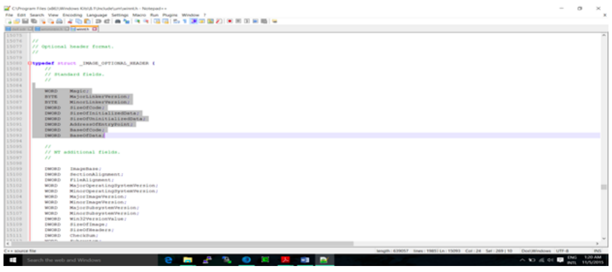
We will see above characteristics in the tool later.

NumberOfSections: This defines the size of the section table, which immediately follow the header.

SizeOfOptionalHeader: This lies between top of the optional header and the start of the section table. This is the size of the optional header that is required for an executable file. This value should be zero for an object file.

Characteristics: This is the characteristic flags that indicate an attribute of the object or image file. It has a flag called Image_File_dll, which has the value 0x2000, indicating that the image is a DLL. It has also different flags that are not required for us at this time

Image_Optional_Header: This optional header contains most of the meaningful information about the image, such as initial stack size, program entry point location, preferred base address, operating system version, section alignment information, and so forth. We can see the information in the snapshot below.



```
//
// Optional header format.
//

typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //

    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;

    //
    // NT additional fields.
    //

    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
```

We can see there are lots of headers and it is not possible to cover each and everything in detail due to space limitations, so we will discuss some of the important things that are necessary.

Standard Fields

As you see in the above picture, we have two fields that are again categorized into some headers. First we will discuss standard fields, because they are common to COFF and UNIX.

Magic: The unsigned integer that identifies the state of the image file. The most common number is 0x10b for 32-bit and 0x10b for 64-bit.

AddressOfEntrypoint: As I said, we will not discuss each header; we will discuss the important ones, and this one is very important as per the Malware Analyst’s perspective. Before getting into the details, we should know some details of PE that are required here.

RVA (relative virtual address): An RVA is nothing but the offset of some item, relative to where it is memory-mapped; or we can simply say that this is an image file and the address of the item after it is loaded into memory, with the base address of image subtracted from memory.

RVA = virtual address – base address (starting address in the memory)

The address of the entry point is the address where the PE loader will begin execution; this is the address that is relative to image base when the executable is loaded into memory.

For the program image, this is the starting address; for device drivers, this is the address of the initialization function and, for the DLL, this is optional.

Now we will start with some headers in the “Additional” section (see above image).

ImageBase: the preferred address of the image when loaded into memory. The default address is 0x00400000. An attacker can change this address depending on his requirement with an option like “-BASE:linker.”

SectionAllignment: The alignment of the section when loaded into memory. Section alignment can be no less than page size (currently 4096 bytes on the windows x86).

FileAlignment: The granularity of the alignment of the sections in the file. For example, if the value in this field is 512 (200h), each section must start at multiples of 512 bytes. If the first section is at file offset 200h and the size is 10 bytes, the next section must be located at file offset 400h: the space between file offsets 522 and 1024 is unused/undefined.

MajorSubSystemVersion: Indicates the Windows NT Win32 subsystem major version number, currently set to 3 for Windows NT version 3.10.

SizeOfImage: The size of the memory, including all of the headers. As the image is loaded into memory, it must be a multiple of SectionAllignment.

This is a combination of the MS-DOS stub, PE header, and section header rounded up to the FileAlignment. In other words, we can say that this value is the file size—the combined size of all sections of the file.

Subsystem: The subsystem is required to run the PE image. Generally, the value of subsystem is 2(0200).

NumberOfRvaAndSizes: The number of data directories in the reminder of optional header.

DataDirectories: This is another sub-section in the header section. It is nothing but the array of 16 IMAGE_DATA_DIRECTORY structures, each relating to an important data structure in the PE file, namely the Import Address Table. In the current PE file, out of 16 only 11 are used, as defined in winnt.h. Some of the directories are shown below:

```
// Directory Entries
```

```
// Export Directory
```

```
#define
```

```
IMAGE_DIRECTORY_ENTRY_EXPORT 0

// Import Directory

#define
IMAGE_DIRECTORY_ENTRY_IMPORT 1

// Resource Directory

#define
IMAGE_DIRECTORY_ENTRY_RESOURCE 2

// Exception Directory

#define
IMAGE_DIRECTORY_ENTRY_EXCEPTION 3

// Security Directory

#define
IMAGE_DIRECTORY_ENTRY_SECURITY 4

// Base Relocation Table

#define
IMAGE_DIRECTORY_ENTRY_BASERELOC 5

// Debug Directory

#define
IMAGE_DIRECTORY_ENTRY_DEBUG 6

// Description String

#define
IMAGE_DIRECTORY_ENTRY_COPYRIGHT 7

// Machine Value (MIPS GP)

#define
IMAGE_DIRECTORY_ENTRY_GLOBALPTR 8

// TLS Directory

#define IMAGE_DIRECTORY_ENTRY_TLS
9
```

```
// Load Configuration Directory

#define
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG
10
```

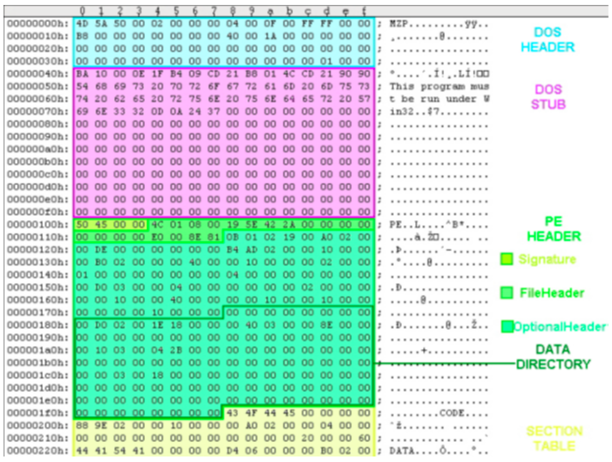
Each data directory is basically a structure defined in IMAGE_DATA_DIRECTORY.

```
typedef struct
_IMAGE_DATA_DIRECTORY {

ULONG VirtualAddress;

ULONG Size;

} IMAGE_DATA_DIRECTORY,
*PIMAGE_DATA_DIRECTORY;
```

Each data directory entry specifies the size and relative virtual address of the directory. To locate a particular directory, we have to determine the relative address from the data directory array in the optional header.

Then use the virtual address to determine which section the directory is in. Once we determine which section contains the directory, the section header for that section is then used to find the exact file offset location of the data directory.

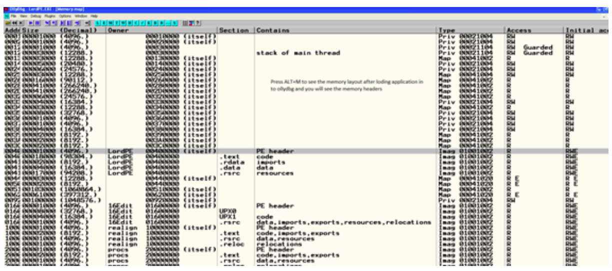
So, to get a data directory, we first need to know about sections, which are described next. An example of how to locate data directories immediately follows this discussion. We can see the various sections and headers in the following image, which is from a hex editor.



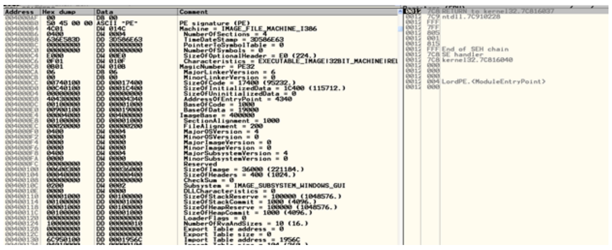
```
             0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00 ; MZP.........ÿÿ..
00000010h: B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00 ; ¸.......@.......
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 ; ................
00000040h: BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90 ; º....´.Í!¸.LÍ!
00000050h: 54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73 ; This program mus
00000060h: 74 20 62 65 20 72 75 6E 20 75 6E 64 65 72 20 57 ; t be run under W
00000070h: 69 6E 33 32 0D 0A 24 37 00 00 00 00 00 00 00 00 ; in32..$7........
00000080h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000090h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000a0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000b0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000c0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000100h: 50 45 00 00 4C 01 08 00 19 5E 42 2A 00 00 00 00 ; PE..L....^B*....
00000110h: 00 00 00 00 E0 00 8E 81 0B 01 02 19 00 A0 02 00 ; ....à.Ž..... ..
00000120h: 00 DE 00 00 00 00 00 00 B4 AD 02 00 00 10 00 00 ; .Þ......´-......
00000130h: 00 B0 02 00 00 00 40 00 00 10 00 00 00 02 00 00 ; .°....@.........
00000140h: 01 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 ; ................
00000150h: 00 D0 03 00 00 04 00 00 00 00 00 00 02 00 00 00 ; .Ð..............
00000160h: 00 00 10 00 00 40 00 00 00 00 10 00 00 10 00 00 ; .....@..........
00000170h: 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000180h: 00 D0 02 00 1E 18 00 00 00 40 03 00 00 8E 00 00 ; .Ð.......@...Ž..
00000190h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000001a0h: 00 10 03 00 04 2B 00 00 00 00 00 00 00 00 00 00 ; .....+..........
000001b0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000001c0h: 00 00 03 00 18 00 00 00 00 00 00 00 00 00 00 00 ; ................
000001d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000001e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000001f0h: 00 00 00 00 00 00 00 00 43 4F 44 45 00 00 00 00 ; ........CODE....
00000200h: 88 9E 02 00 00 10 00 00 00 A0 02 00 00 04 00 00 ; ˆž...... ......
00000210h: 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60 ; ............ ..`
00000220h: 44 41 54 41 00 00 00 00 D4 06 00 00 00 B0 02 00 ; DATA....Ô....°..
```

The first file, VirtualAddress is nothing but RVA of the table. The RVA is the address of table relative to base address of the image when the table is loaded. The second field gives size in bytes. The data directory that forms the last part of IMAGE_OPTIONAL_HEADER is listed below and we will discuss some of the important one.

To make this work more practical, we can use ollydbg or Immunity Debugger here. We will use ollydbger to see the different sections of PE file, as shown below.



After that, if we want to check the details of section of PE header using olllydbg, we have to open Appearance→PE header mode in memory layout, which is the left corner button of the ollydbg GUI. Check below and we can clearly see all the headers and sections.



```
Address   Hex dump     Data          Comment
004000AF  00           DB 00
004000B0  50 45 00 00  ASCII "PE"    PE signature (PE)
004000B4  4C01         DW 014C       Machine = IMAGE_FILE_MACHINE_I386
004000B6  0400         DW 0004        NumberOfSections = 4
004000B8  636E583D     DD 3D586E63    TimeDateStamp = 3D586E63
004000BC  00000000     DD 00000000    PointerToSymbolTable = 0
004000C0  00000000     DD 00000000    NumberOfSymbols = 0
004000C4  E000         DW 00E0        SizeOfOptionalHeader = E0 (224.)
004000C6  0F01         DW 010F        Characteristics = EXECUTABLE_IMAGE|32BIT_MACHINE|REL
004000C8  0B01         DW 010B       MagicNumber = PE32
004000CA  06           DB 06          MajorLinkerVersion = 6
004000CB  00           DB 00          MinorLinkerVersion = 0
004000CC  00740100     DD 00017400    SizeOfCode = 17400 (95232.)
004000D0  00C40100     DD 0001C400    SizeOfInitializedData = 1C400 (115712.)
004000D4  00000000     DD 00000000    SizeOfUninitializedData = 0
004000D8  40430000     DD 00004340    AddressOfEntryPoint = 4340
004000DC  00100000     DD 00001000    BaseOfCode = 1000
004000E0  00900100     DD 00019000    BaseOfData = 19000
004000E4  00004000     DD 00400000   ImageBase = 400000
004000E8  00100000     DD 00001000    SectionAlignment = 1000
004000EC  00020000     DD 00000200    FileAlignment = 200
004000F0  0400         DW 0004        MajorOSVersion = 4
004000F2  0000         DW 0000        MinorOSVersion = 0
004000F4  0000         DW 0000        MajorImageVersion = 0
004000F6  0000         DW 0000        MinorImageVersion = 0
004000F8  0400         DW 0004        MajorSubsystemVersion = 4
004000FA  0000         DW 0000        MinorSubsystemVersion = 0
004000FC  00000000     DD 00000000    Reserved
00400100  00600300     DD 00036000    SizeOfImage = 36000 (221184.)
00400104  00040000     DD 00000400    SizeOfHeaders = 400 (1024.)
00400108  00000000     DD 00000000    CheckSum = 0
0040010C  0200         DW 0002        Subsystem = IMAGE_SUBSYSTEM_WINDOWS_GUI
0040010E  0000         DW 0000        DLLCharacteristics = 0
00400110  00001000     DD 00100000    SizeOfStackReserve = 100000 (1048576.)
00400114  00100000     DD 00001000    SizeOfStackCommit = 1000 (4096.)
00400118  00001000     DD 00100000    SizeOfHeapReserve = 100000 (1048576.)
0040011C  00100000     DD 00001000    SizeOfHeapCommit = 1000 (4096.)
00400120  00000000     DD 00000000    LoaderFlags = 0
00400124  10000000     DD 00000010    NumberOfRvaAndSizes = 10 (16.)
00400128  00000000     DD 00000000    Export Table address = 0
0040012C  00000000     DD 00000000    Export Table size = 0
00400130  6C950100     DD 0001956C    Import Table address = 1956C
```

```
0012  7C8 RETURN to kernel32.7C816037
0012  7C9 ntdll.7C910228
0012  FFF
0012  7FF
0012  805
0012  001
0012  815
0012  FFF End of SEH chain
0012  7C8 SE handler
0012  7C8 kernel32.7C816040
0012  000
0012  000
0012  000
0012  004 LordPE.<ModuleEntryPoint>
0012  000
```

The data directory that forms the last part of the optional header is listed below. We will discuss more about these in section table.

Export table, import table, resource table, exception table, certificate table, base relocation table, debug, architecture, global ptr, TLS table, load config table, bound import, IAT, delay import descriptor, CLR runtime header.

The Section Table

This table immediately follows the optional header. The location of this section of the section table is

members is determined by NumberOfSections field in the file header (IMAGE_FILE_HEADER) structure. The structure is called IMAGE_SECTION_HEADER.

The number of entries in the section table is given by noofsectionfield in the file header. Each section header has at least 40 bytes of entry. We will discuss some of the important entries below.

Name1: An 8-byte null-padded UTF8

encoding string. This is can be null.

VirtualSize: The actual size of the section’s data in bytes. This may be less than the size of the section on disk.

SizeOfRawData: The size of section’s data in the file on the disk.

PointerToRawData: This is so useful because it is the offset from the file’s beginning to the section’s data.

Characteristics: This flag describes the characteristics of the section.

The PE File Section

This section contains the main content of the file, including code, data, resources and other executable files. Each section has a header and body.

An application in Windows NT typically has nine different predefined sections, such as .text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata, and .debug. Depending on the application, some of these sections are used, but not all are used.

The Executable Code: In Windows, all code segments reside in a section called .text section or CODE. We know that windows uses a page-based virtual system, which means having one large code section that is easier to manage for both the OS and application developer. This also called as entry point and thunk table, which points to IAT. We will discuss the thunk table in IAT.

.bss: This represents the uninitialized data for the application.

The .rdata represents the read-only data on the file system, such as strings and constants.

The .rsrc is a resource section, which contains resource information of a module. In many cases it shows icons and images that are part of the file’s resources.

The .edata section contains the export directory for an application or DLL. When present, this section contains information about the names and addresses of exported functions. We will discuss these in greater depth later.

The .idata section contains various information about imported functions, including the import directory and import address table. We will discuss these in greater depth later.

TLS Section

Windows supports multiple threads of execution per process; each thread has its own storage, called thread local storage (TLS). Windows has an API called the TLS API. Below is a small link that describes TLSL

https://msdn.microsoft.com/library/6yh4a9k1%28v=vs.100%29.aspx

The linker defines the .tls section in the PE file that describes the layout for TLS needed in the routines by executables and DLLs, so each time a process creates threads, a TLS is built by thread and it uses .tls as a template.

NOTE: From the malware's perspective, the array of the tls callback function started before the first instruction of the code or entry point of the exe, which does not allow a researcher to start analyzing and putting a breakpoint. This can be found by a plug-in by olly. We will discuss this in a future topic. The following link is the reference to some good material.

http://blogs.technet.com/b/mmpc/archive/2010/06/21/further-unexpected-resutls-sic.aspx

In the next installment, I will give details about later sections of a PE file, including some of the automation and cool stuff. Please comment below.

PREV: MAL..

AUTHOR
Revers3r

Revers3r is a Information Security Researcher with considerable experience in Web Application Security, Vulnerability Assessment, Penetration Testing. He is also well-versed in Reverse Engineering, Malware Analysis. He's been a contributor to international magazines like Hakin9, Pentest, and E-Forensics. In his free time, he's contributed to the Response Disclosure Program. website: www.vulnerableghost.com

When Your CEO Won't Take Security...

InfoSec Institute: Top Training Company 2015

What Is a SIEM?

CISSP Domain #1 Cheat Sheet: Security...

0 Comments InfoSec Institute Resources Login

Recommend Share Sort by Best

Start the discussion…

Be the first to comment.

ALSO ON **INFOSEC INSTITUTE RESOURCES** WHAT'S THIS?

Analyzing a DDoS Trojan

1 comment • 10 days ago

Eduard Abramovich — Hi, I am just starting with these kind of security analysis. My questions are, where did you get the …

ISIL, Terrorism and Technology: A Dangerous Mix

2 comments • 10 days ago

Pierluigi Paganini — Regarding the Snowden's case there is no political discussion about it, I just just observed …

Subscribe Add Disqus to your site Privacy DISQUS

About InfoSec

InfoSec Institute is the best source for high quality information security training. We have been training Information Security and IT Professionals since 1998 with a diverse lineup of relevant training courses. In the past 16 years, over 50,000 individuals have trusted InfoSec Institute for their professional development needs!

Connect with us

Stay up to date with InfoSec Institute and Intense School - at info@infosecinstitute.com

Like 555

Follow @infosecedu

Join our newsletter

Get the latest news, updates & offers straight to your inbox.

ENTER YO SUBSCRIBE