PalArch's Journal of Archaeology of Egypt / Egyptology

UNDERSTANDING OF PE HEADERS AND ANALYZING PE FILE

Maxim Demidenko[1*], *Alzhan Kaipiyev*[2], *Maryam Var Naseri*[3], *Yogeswaran A/L Nathan*[4], *Nor Afifah Binti Sabri*[5]

[1*]Student at Asia Pacific University.

[2]Student at Asia Pacific University.

[3]Lecturer at Asia Pacific University.

[4]Lecturer at Asia Pacific University.

[5]Lecturer at Asia Pacific University.

[1*]tp039365@mail.apu.edu.my,[2]tp039400@mail.apu.edu.my,[3]maryam.var@staffemail.apu.edu.my,[4]yogeswaran.nathan@apu.edu.my,[5]afifah@staffemail.apu.edu.my

Maxim Demidenko, Alzhan Kaipiyev, Maryam Var Naseri, Yogeswaran A/L Nathan, Nor Afifah Binti Sabri. Understanding of PE Headers and Analyzing PE File-- Palarch's Journal of Archaeology of Egypt/Egyptology 17(4), 8611-8620. ISSN 1567-214x.

keywords: Portable Executable, Reverse Engineering, Cyber Security.

ABSTRACT:
The purpose of the paper is to explain the workflow of the PE files in detailed format. This includes understanding of all data structures inside the file and useful information that is possible to get. After the explanation the simple Python script is provided as the way to get this data structures.

INTRODUCTION

The topic is part of the Malware static analysis technique. All data structures are containing official information from Microsoft documentation explained by writer. Also, all links to Microsoft official documentation are provided in reference. The Python scripting is done using "pefile" library and it was developed by Ero Carrera. This module allows to extract the important part of the information from the file that could help in investigation. For this module Python should be upgraded to Python 3.

MATERIALS AND METHODS

The first thing before understanding what is Portable Executable file (PE). Reader should understand basics principle how digital storage and processing unit are working inside the systems.

- **BYTE** – unit of storage of information. A set of bits processed by a computer at once. In modern computing systems, a byte consists of eight bits and, accordingly,takes one of 256 different values.
- **WORD and DWORD** – machine-dependent and platform-dependent value, measured in bits or bytes equal to the capacity of the processor registers. The recording of information in memory, as well as its extraction from memory is made by addresses. This property of memory is called addressability.
- **RVA**-Relative Virtual Address the address of the file after its loaded inside the memory. It differs from the position the file in hard disk.
- **RAW**- designation for an indefinite volume file system. The RAWFS component exists and is built into the kernel itself, but the only purpose of this component is to respond to requests from applications about the size of the volume and the name of the file system.

With the advent of the Windows NT 3.1 operating system, Microsoft switched to the PE format. All later versions of Windows, including Windows 95/98 / ME, support this format. The format retained limited support for the existing (MZ) to bridge the gap between DOS-based systems and NT systems. The first 2 bytes of the PE file contain the signature 0x4D 0x5A - "MZ" (as a successor of the MZ format). Next - the double word at offset 0x3C contains the address of the PE header. The latter begins with the signature 0x50 0x45 - "PE".

2.1.PE Headers

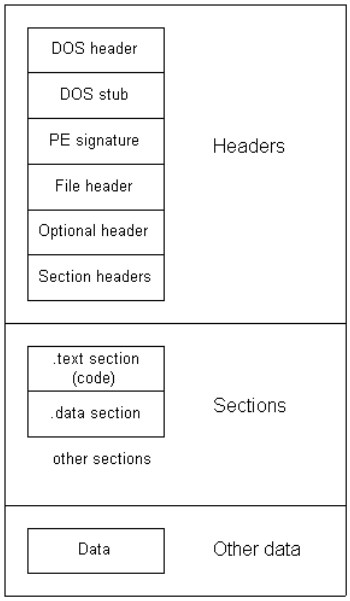

Figure 1 PE file global structure

PE format is the format of executable files of all 32-bit and 64-bit Windows systems. Currently there are two PE file formats: PE32 and PE32 +. PE32 format for x86 systems, and PE32 + for x64. The described structures can be observed in the "*WINNT.h*" header file that comes with the Windows SDK.(Microsoft, 2019)

Also, PE is a modified version of the COFF file format for Unix. PE / COFF is an alternative term for developing Windows.

2.1.1. DOS-Header (IMAGE_DOS_HEADER)

```
1. typedef struct_IMAGE_DOS_HEADER{
2. chare_magic[2]={'M','Z'};
3. WORDlastsize;
4. WORDnblocks;
5. WORDnreloc;
6. WORDhdrsize;
7. WORDminalloc;
8. WORDmaxalloc;
9. WORD*ss;// 2 byte value
10. WORD*sp;// 2 byte value
11. WORDchecksum;
12. WORD*ip;// 2 byte value
13. WORD*cs;// 2 byte value
14. WORDrelocpos;
15. WORDnoverlay;
16. WORDreserved1[4];
17. WORDoem_id;
18. WORDoem_info;
19. WORDreserved2[10];
20. DWORDe_lfanew;
21. }
```

Data Structure 1 DOS-Header.

The Image_DOS_Header or simply DOS Header is the fit structure appearing in each executable. The most important fields in this structure are “e_magic” and “e_lfnew”. The magic number of the executable file should be filled with “MZ” value. MZ is the unique identifier of the executable files and stands for Mark Zbikowski (the creator of the DOS). All other fields are not that useful in file analysis because they are only helpers in programs execution.

After first 0x64 Bytes of information file continues with the DOS-stub. Usually this section does not contain anything but “This program cannot be run in DOS mode.”, or “This program must be run under win32” which are default for some compilers. This section is fully responsible for executable behavior if it would be launched on DOS and made for backwards compatibility. Due to the progress of computing there is almost impossible to find the program that will have unique execution statements for DOS.

2.1.2. PE-Header (IMAGE_NT_HEADER)

Image_NT_Header or PE header is the next structure after the DOS header and it takes next 0x18 bytes of memory. This header contains of 0x4 bytes allocated for signature and two large structs that would be analyzed next. Those structs are: IMAGE_FILE_HEADER and IMAGE_OPTIONAL_HEADER. (Microsoft, 2018).

```
1. typedefstruct _IMAGE_NT_HEADERS{
2. DWORDSignature;
3. IMAGE_FILE_HEADERFileHeader;
4. IMAGE_OPTIONAL_HEADEROptionalHeader;
```

```
5. } IMAGE_NT_HEADERS,*PIMAGE_NT_HEADERS;
```

Data Structure 2 PE Header

2.1.3. File-Header or COFF Header (IMAGE_FILE_HEADER)

This header contains only the information about the file characteristics. Shortly, "Machine" is 2 byte identifier to represent the system architecture, "NumberOfSections" is a number of sections in PE file that is limited to 96, "TimeDateStamp" represents the creation date of the file, "PointerToSymbolTable" the shift of RAW to characters table based on "SizeOfOptionalHeader" (Both are used rarely and usually filled with 0), "Characteristics" represents some file characteristics (using predefined constants). (Microsoft, 2018)

```
1. typedef structIMAGE_FILE_HEADER{
2. WORDMachine;
3. WORDNumberOfSections;
4. DWORDTimeDateStamp;
5. DWORDPointerToSymbolTable;
6. DWORDNumberOfSymbols;
7. WORDSizeOfOptionalHeader;
8. WORDCharacteristics;
9. } IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Data Structure 3 File-Header.

2.1.4. OptionalHeader (IMAGE_OPTIONAL_HEADER)

Despite the name, Image_Optional_Header or Optional Header is not optional. This header is mandatory for proper execution of the PE file. This header has two main format types: PE32 and PE32+ (IMAGE_OPTIONAL_HEADER32 and IMAGE_OPTIONAL_HEADER64 accordingly). The type is stores in the Magic number in first 0x16bit of the struct. (Microsoft, 2018)

```
1. typedefstruct _IMAGE_OPTIONAL_HEADER {
2. WORDMagic;
3. BYTEMajorLinkerVersion;
4. BYTEMinorLinkerVersion;
5. DWORDSizeOfCode;
6. DWORDSizeOfInitializedData;
7. DWORDSizeOfUninitializedData;
8. DWORDAddressOfEntryPoint;
9. DWORDBaseOfCode;
10. DWORDBaseOfData;
11. DWORDImageBase;
12. DWORDSectionAlignment;
13. DWORDFileAlignment;
14. WORDMajorOperatingSystemVersion;
15. WORDMinorOperatingSystemVersion;
16. WORDMajorImageVersion;
17. WORDMinorImageVersion;
18. WORDMajorSubsystemVersion;
19. WORDMinorSubsystemVersion;
20. DWORDWin32VersionValue;
```

```
21. DWORDSizeOfImage;
22. DWORDSizeOfHeaders;
23. DWORDCheckSum;
24. WORDSubsystem;
25. WORDDllCharacteristics;
26. DWORDSizeOfStackReserve;
27. DWORDSizeOfStackCommit;
28. DWORDSizeOfHeapReserve;
29. DWORDSizeOfHeapCommit;
30. DWORDLoaderFlags;
31. DWORDNumberOfRvaAndSizes;
32. IMAGE_DATA_DIRECTORYDataDirectory[IMAGE_NUMBEROF_DIRECTO
   RY_ENTRIES];
33. } IMAGE_OPTIONAL_HEADER,*PIMAGE_OPTIONAL_HEADER;
```

Data Structure 4 Optional Header

Some Explanations:

- **MajorLinkerVersion** - The major version number of the linker.
- **MinorLinkerVersion**- The minor version number of the linker.
- **SizeOfCode** - The size of the code section, in bytes, or the sum of all such sections if there are multiple code sections.
- **SizeOfInitializedData** -The size of the initialized data section, in bytes, or the sum of all such sections if there are multiple initialized data sections.
- **SizeOfUninitializedData** -The size of the uninitialized data section, in bytes, or the sum of all such sections if there are multiple uninitialized data sections. (Microsoft, 2018)

The struct Image Data Directory is saving predefined constants for VA and RVA. Constants are defined with numbers from 0 to 14 and have the format of "IMAGE_DIRECTORY_ENTRY_~~TYPE~~". And have thow predefined types: EXPORT; IMPORT; RESOURCE;EXCEPTION; SECURITY; BASERELOC;DEBUG; COPYRIGHT; ARCHITECTURE; GLOBALPTR; TLS; LOAD_CONFIG; BOUND_IMPOT; IAT; DELAY_IMPORT; COM_DESCRIPTOR.

```
1. typedefstruct _IMAGE_DATA_DIRECTORY {
2. DWORDVirtualAddress;
3. DWORDSize;
4. } IMAGE_DATA_DIRECTORY,*PIMAGE_DATA_DIRECTORY;
```

Data Structure 5 Image Data Directory

2.1.5. Section-header (IMAGE_SECTION_HEADER)

Data Directory is followed by Data Section. Table of section are divided by some king of island which has their "NumberOfSections". Each section has their own rules, rights and instructions. Their size 0x28 bytes. Number of Sections are located in file Header. Name of the section has length in 8 symbols". Virtual Size DWORD is a size of the virtual section. SizeOfRawData the size of the section in file. VirtualAddress is RVA of the section. Characteristics is an instruction for section which will load it in file.

Section with resources should followed by .rsrc, but other section can be anything. Section is an environment that loading itself in virtual memory and working inside this memory. The virtual address is created in ImageBase. (Microsoft, 2019)

```
1. typedefstruct _IMAGE_SECTION_HEADER {
2. BYTEName[IMAGE_SIZEOF_SHORT_NAME];
3. DWORDPhysicalAddress;
4. DWORDVirtualSize;
5. DWORDVirtualAddress;
6. DWORDSizeOfRawData;
7. DWORDPointerToRawData;
8. DWORDPointerToRelocations;
9. DWORDPointerToLinenumbers;
10. WORDNumberOfRelocations;
11. WORDNumberOfLinenumbers;
12. DWORDCharacteristics;
13. } IMAGE_SECTION_HEADER,*PIMAGE_SECTION_HEADER;
```

Data Structure 6 Section Header.

2.1.6. Export table (Mostly used in .dll)

In the very first element of the DataDirectory array, the RVA is stored on the export table, which is actually represented by the IMAGE_EXPORT_DIRECTORY structure. This table is characteristic of dynamic library files (.dll). The main task of the table is to link the exported functions with their RVA. (Pietrek, 2010)

```
1. typedefstruct _IMAGE_EXPORT_DIRECTORY {
2. DWORDCharacteristics;
3. DWORDTimeDateStamp;
4. WORDMajorVersion;
5. WORDMinorVersion;
6. DWORDName;
7. DWORDBase;
8. DWORDNumberOfFunctions;
9. DWORDNumberOfNames;
10. DWORDAddressOfFunctions;
11. DWORDAddressOfNames;
12. DWORDAddressOfNameOrdinals;
13. }IMAGE_EXPORT_DIRECTORY,*PIMAGE_EXPORT_DIRECTORY;
```

Data Structure 7 Export Directory

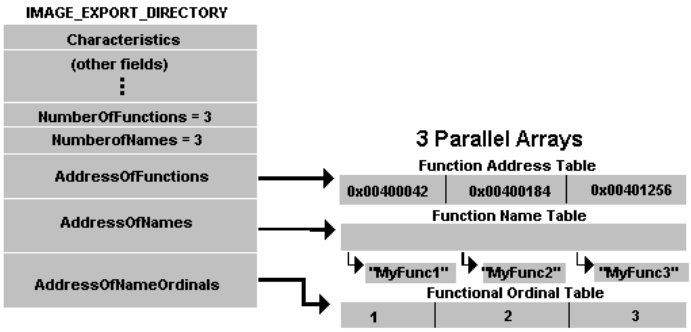

Figure 2 Image Export Directory

This structure contains three pointers to three different tables. This is a table of names(AddressOfNames),ordinals(AddressOfNamesOrdinals), addresses(AddressOfFunctions).

The Name field stores the RVA of the dynamic library name. The ordinal is like an intermediary, between the table of names and the table of addresses and is an array of indices (the size of the index is 2 bytes).

2.1.7. Imports (IMAGE_IMPORT_DESCRIPTOR)

The Import Descriptor is the following table after the table of exports. To calculate its address, it is needed to find DataDirectory struct. After getting the RVA of "IMAGE_DIRECTORY_ENTRY_IMPORT" it is needed to get RAW value and follow the path shown on "Figure 3 Import descriptor path"

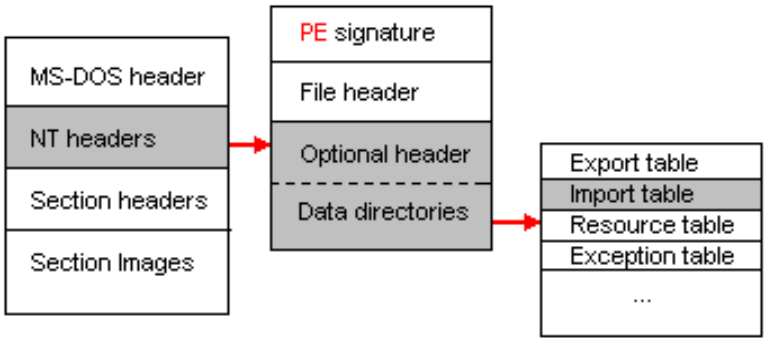

Figure 3 Import descriptor path

```
1. typedefstruct _IMAGE_IMPORT_DESCRIPTOR {
2. DWORDCharacteristics;
3. DWORDOriginalFirstThunk;
4. DWORDTimeDateStamp;
5. DWORDForwarderChain;
6. DWORDName;
7. DWORDFirstThunk;
8. }IMAGE_IMPORT_DESCRIPTOR,*PIMAGE_IMPORT_DESCRIPTOR;
```

Data Structure 8 Import Descriptor

RESULTS AND IMPLEMENTATION

Firstly, pre-install the module into system to make everything work. Open and run CMD with administrative privilege and type the following commands inside it. (Carrera, 2019)

- $git clone https://github.com/erocarrera/pefile.git
- $ cd pefile
- $ pip install -r requirements.txt
- $ python setup.py install

After all this procedure are done. Then open the Python in your IDE or CMD. Then type “import pefile”. Load the file inside your code and use function pefile.PE(file path).

```
import pefile

path = "C:/Users/thego/Desktop/4.exe"
pe = pefile.PE(path)

print("e_magic value: %s" % hex(pe.DOS_HEADER.e_magic))
print("Signature value: %s" % hex(pe.NT_HEADERS.Signature))
```

Figure 4 Python Code Example Snippet 1

This following screenshot showing the small snippet that allow to find the e_magic value (the address of the MZ string) and Signature value is a 4-byte signature that identifies the file as a PE format image file.

```
e_magic value: 0x5a4d
Signature value: 0x4550

Process finished with exit code 0
```

Figure 5 Python Code Example Output 1.

This script will expand more information about Data Directory with the Virtual address and size of the directory.

```
path = "C:/Users/thego/Desktop/4.exe"
pe = pefile.PE(path)

for data_dir in pe.OPTIONAL_HEADER.DATA_DIRECTORY:
    print(data_dir)
```

Figure 6 Python Code Example Snippet 2

The output will show all IMAGE_OPTIONAL_HEADER. This directory has information about address and size that has found in file.

```
[IMAGE_DIRECTORY_ENTRY_EXPORT]
0x168       0x0    VirtualAddress:                0x0
0x16C       0x4    Size:                          0x0
[IMAGE_DIRECTORY_ENTRY_IMPORT]
0x170       0x0    VirtualAddress:                0x4B54
0x174       0x4    Size:                          0xC8
[IMAGE_DIRECTORY_ENTRY_RESOURCE]
0x178       0x0    VirtualAddress:                0x7000
0x17C       0x4    Size:                          0x1E0
```

Figure 7 Python Code Example Output 2

This script search for DLL inside the file using DIRECTORY_ENTRY_IMPORT. Address and size pairs for special tables that are found in the image file and are used by the operating system (for example, the import table and the export table). (Microsoft, 2019).

```
for entry in pe.DIRECTORY_ENTRY_IMPORT:
    dll_name = entry.dll.decode('utf-8')
    if dll_name == entry.dll.decode('utf-8'):
        print(entry.dll.decode('utf-8'))
        for func in entry.imports:
            print("\t%s at 0x%08x" % (func.name.decode('utf-8'), func.address))
```

Figure 8 Python Code Example Snippet 3

This output shows all DLL and Functions used inside it. That could help in future debugging or Dynamic Analysis. The address of the entry points relative to the image base when the executable file is loaded into memory. For program images, this is the starting address.

```
    GetModuleHandleW at 0x00404028
    UnhandledExceptionFilter at 0x0040402c
USER32.dll
    GetWindowTextW at 0x00404058
    GetForegroundWindow at 0x0040405c
MSVCP140.dll
    ?sputn@?$basic_streambuf@DU?$char_traits@D@std@@@std@@QAE_JPBD_J@Z at 0x00404034
    ?sputc@?$basic_streambuf@DU?$char_traits@D@std@@@std@@QAEHD@Z at 0x00404038
    ?flush@?$basic_ostream@DU?$char_traits@D@std@@@std@@QAEAAV12@XZ at 0x0040403c
    ?setstate@?$basic_ios@DU?$char_traits@D@std@@@std@@QAEXH_N@Z at 0x00404040
    ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A at 0x00404044
    ?uncaught_exception@std@@YA_NXZ at 0x00404048
    ?_Xlength_error@std@@YAXPBD@Z at 0x0040404c
    ?_Osfx@?$basic_ostream@DU?$char_traits@D@std@@@std@@QAEXXZ at 0x00404050
VCRUNTIME140.dll
    __std_exception_copy at 0x00404064
    memset at 0x00404068
```

Figure 9 Python Code Example Output 3.

CONCLUSION

In conclusion should be said that Malware static analysis technique one of the important parts of fighting the malware. Malware is causing a critical threat to the world. This topic was created to teach a basic technique of analyzing the malware using the simple tools and Python scripting that allow Malware analysis to automate the work experience.

ACKNOWLEDGMENT

The authors would like to thank their teacher and mentor Maryam Var Naseri.

REFERENCES

Carrera, E. (2019, May 20). *PEFILE git hub*. Git Hub: https://github.com/erocarrera/pefile

Microsoft. (2018, 05 12). *IMAGE_FILE_HEADER structure*. Microsoft: https://docs.microsoft.com/en-us/windows/desktop/api/winnt/ns-winnt-_image_file_header

Microsoft. (2018, 05 12). *IMAGE_NT_HEADERS32 structure*. Microsoft: https://docs.microsoft.com/en-us/windows/desktop/api/winnt/ns-winnt-_image_nt_headers

Microsoft. (2018, 05 12). *IMAGE_OPTIONAL_HEADER32 structure*. Microsoft: https://docs.microsoft.com/en-us/windows/desktop/api/winnt/ns-winnt-_image_optional_header

Microsoft. (2019, 12 05). *IMAGE_SECTION_HEADER structure*. Microsoft: https://docs.microsoft.com/en-us/windows/desktop/api/winnt/ns-winnt-_image_section_header

Microsoft. (2019, 03 18). *PE Format*. Microsoft: https://docs.microsoft.com/en-us/windows/desktop/Debug/pe-format

Pietrek, M. (2010, 06 30). *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format*. Microsoft: https://docs.microsoft.com/en-us/previous-versions/ms809762(v=msdn.10)#IMAGE_EXPORT_DIRECTORY