

Dissecting PE Headers

Introduction

In the Windows Operating System, you might have often seen files with extension .exe. The .exe extension here stands for executable. As the name suggests, an executable file contains code that can be executed. Therefore, anything that needs to be run on a Windows Operating System is executed using an executable file, also called a Portable Executable file (PE file), as it can be run on any Windows system. A PE file is a Common Object File Format (COFF) data structure. The COFF consists of Windows PE files, DLLs, shared objects in Linux, and ELF files. For this room, we will only be covering the Windows PE files.

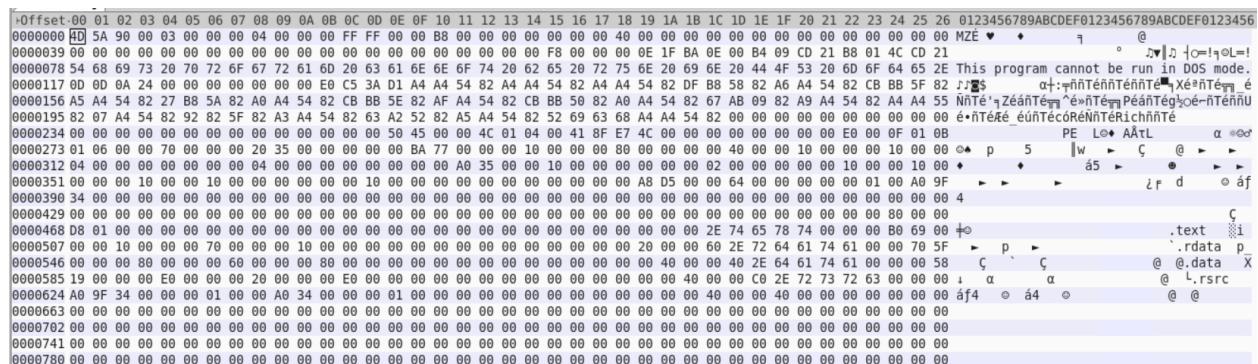
Given this background knowledge, it becomes essential to understand the PE file to learn malware analysis. This room will introduce us to the following concepts:

- Understanding the different headers in a PE file
- Learning how to read PE headers
- Identify packed executables
- Use the information from PE headers to analyze malware

Before going through this room, it is recommended that you complete the [Intro to Malware Analysis](#) room.

Overview of PE Headers

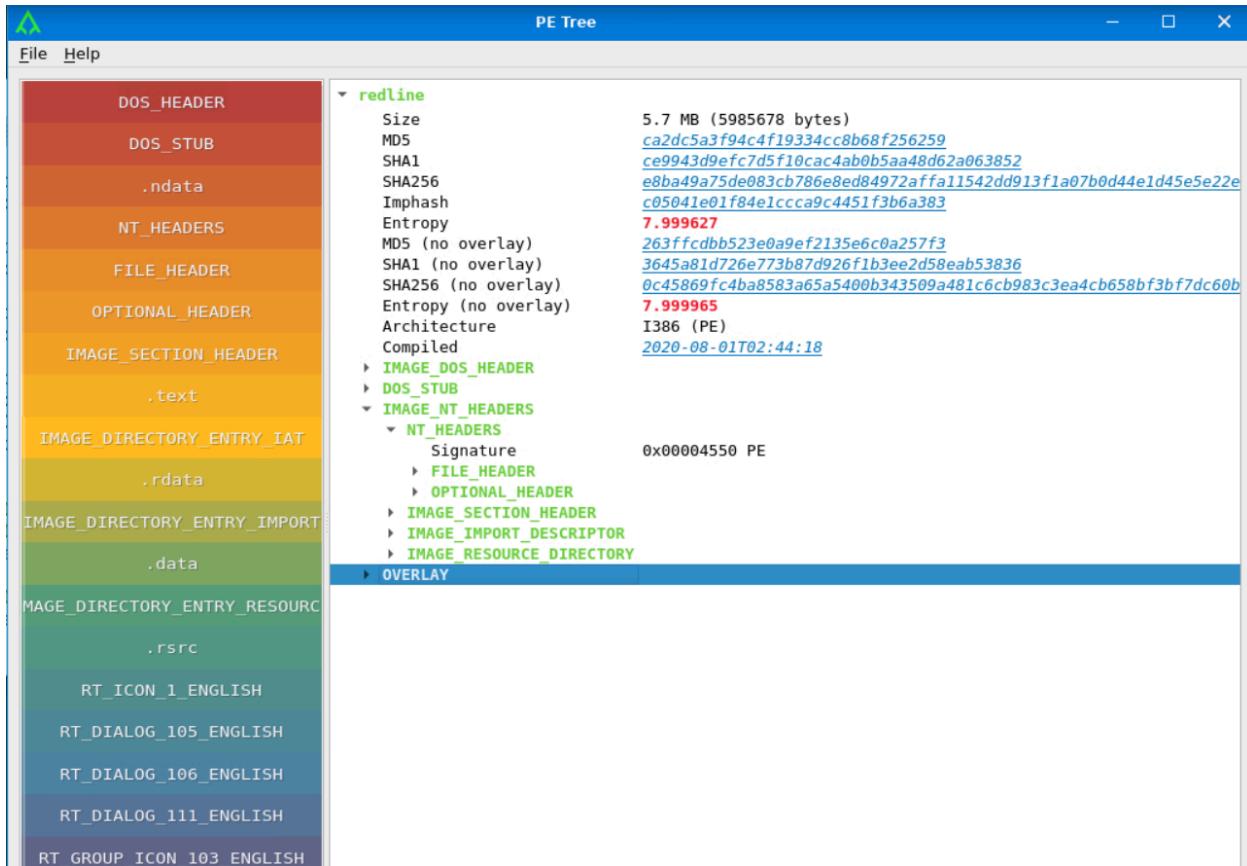
On disk, a PE executable looks the same as any other form of digital data, i.e., a combination of bits. If we open a PE file in a Hex editor, we will see a random bunch of Hex characters. This bunch of Hex characters are the instructions a Windows OS needs to execute this binary file.



In the upcoming tasks, we will try to make sense of the Hex numbers we see in the above screenshot and describe them as Windows understands them. We will also

explore how to leverage this information for malware analysis. We will use the wxHexEditor utility, present in the next task's attached VM to perform this task.

As we view the file in a Hex editor, we observe that manually interpreting all this data might become too tedious. Therefore, we will use a tool pe-tree in the attached VM to help us analyze the PE header. This is what we see when we open a PE file using pe-tree.



In the right pane here, we see some tree-structure dropdown menus. The left pane is just shortcuts to the dropdown menus of the right pane. Some of the important headers that we will discuss in this room are:

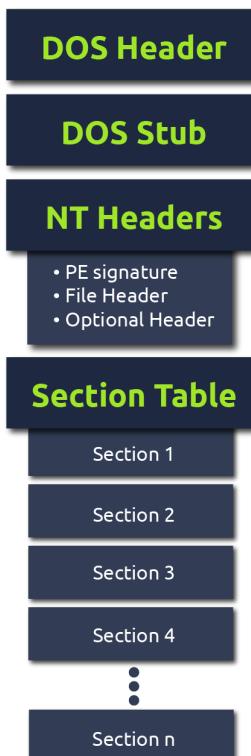
- IMAGE_DOS_HEADER
- IMAGE_NT_HEADERS
- FILE_HEADER
- OPTIONAL_HEADER
- IMAGE_SECTION_HEADER
- IMAGE_IMPORT_DESCRIPTOR

All of these headers are of the data type STRUCT. A struct is a user-defined data type that combines several different types of data elements in a single variable. Since it is user-defined, we need to see the documentation to understand the type for each

STRUCT variable. The documentation for each header can be found on [MSDN](#), where you can find the data types of the different fields inside these headers.

Please remember that while we use the tools mentioned earlier, various other tools perform similar tasks. However, the goal is not to learn about the tools but instead the PE format so that we can perform the same analysis using any other tools we come across, providing the same functionality.

Now, let's move to the following tasks and learn about each of these parts of a PE file.



Answer the questions below:

What data type are the PE headers?

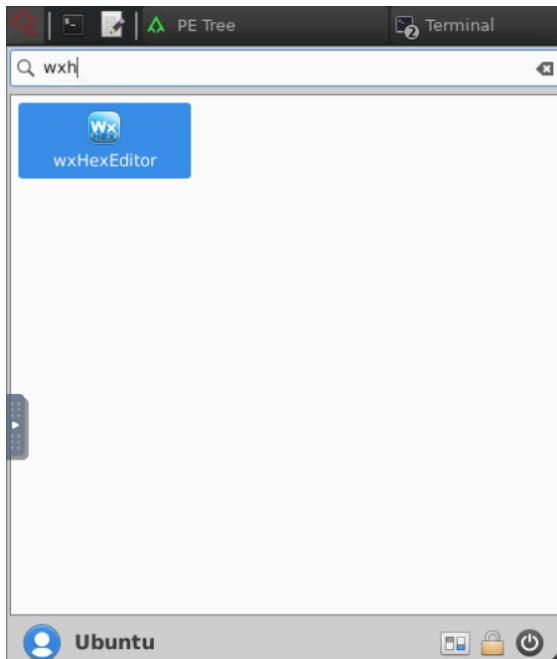
Answer: **STRUCT**

IMAGE_DOS_HEADER and DOS_STUB

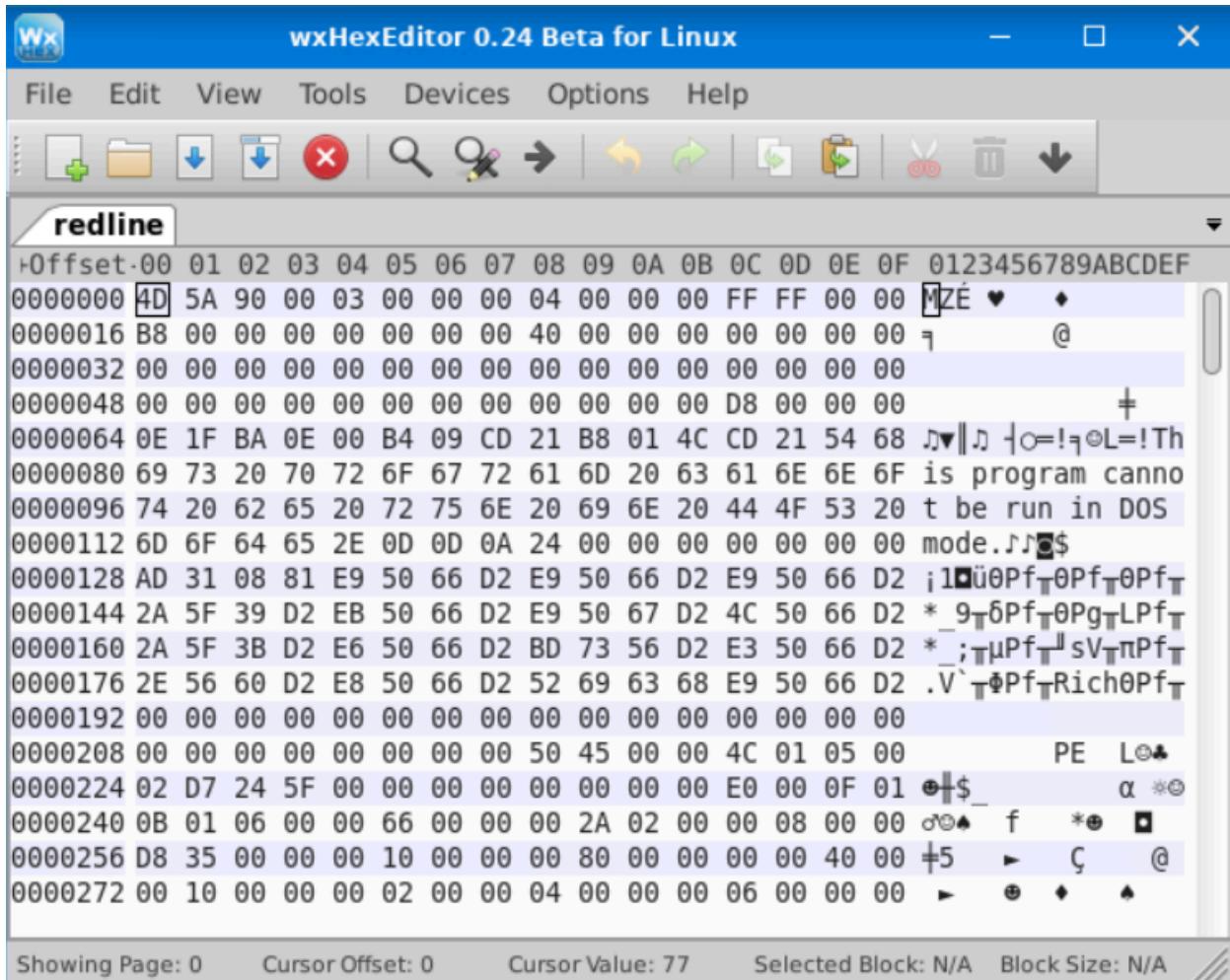
For this task, we will need to use the attached VM. For this purpose, press the 'Start Machine' button on the top-right corner of this task to start the attached machine. The machine will start in a split-screen view. In case the machine is not visible, use the blue Show Split View button at the top-right of the page. Alternatively, you can log in to the machine using the following credentials:

- Username: ubuntu
- Password: 123456

Once the machine starts, we can find that there is a directory on the Desktop of the machine named Samples. In this directory, we will find a few PE files. Let's open the file named redline in a Hex editor to see what it looks like. We can use the wxHexEditor utility in the attached VM to open the required file. To open the wxHexEditor, press the menu on the top left corner of the VM and search for wxHexEditor, as shown in the following screenshot.



This is what the redline PE will look like in the Hex Editor.



Since it seems a little too complex to comprehend, let's use the help of the pe-tree utility to see what the PE header looks like. If we run the following command in the terminal in the attached VM, it will open the redline PE file in the pe-tree utility.

pe-tree Desktop/Samples/redline

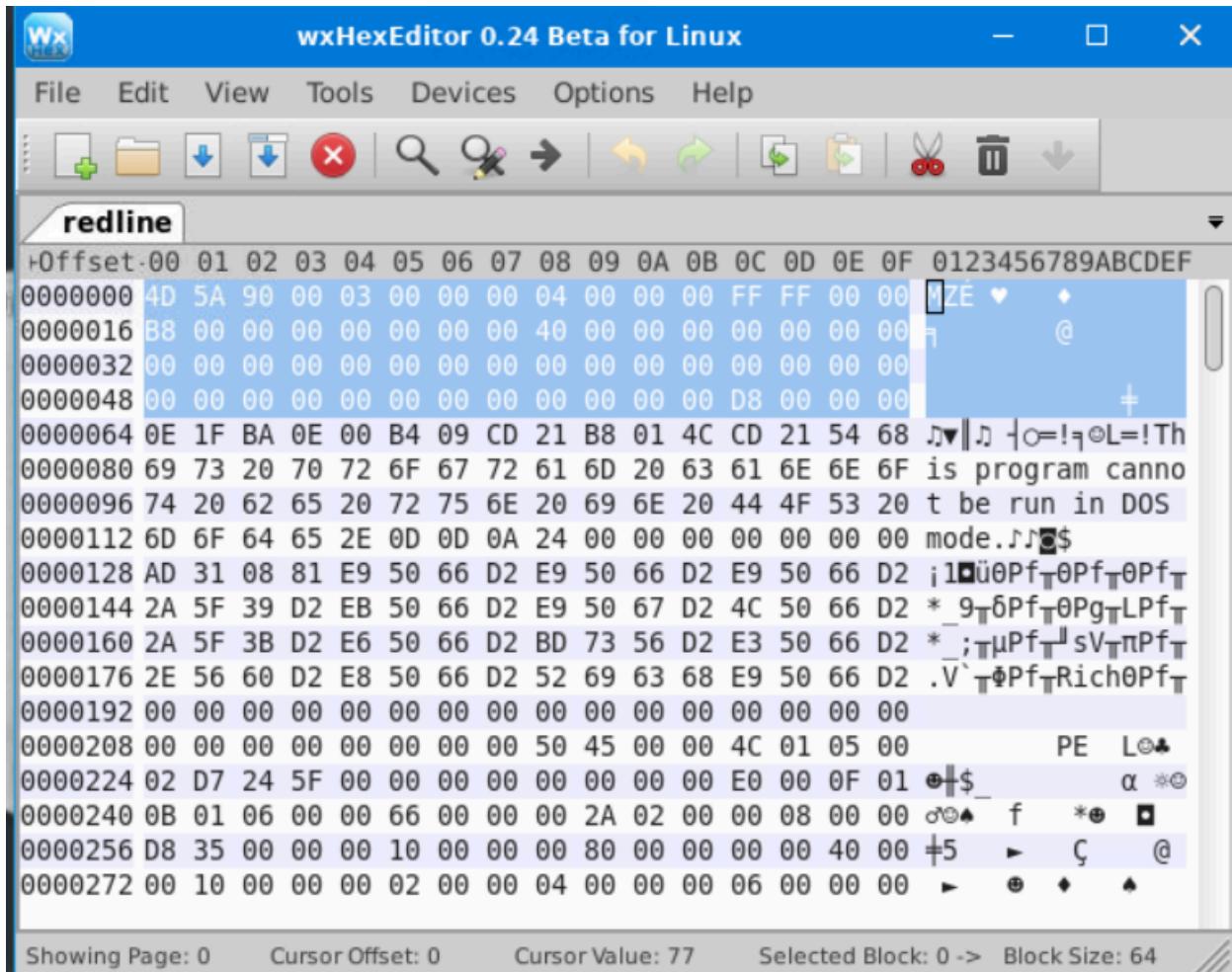
Please note that the pe-tree utility will take roughly 8 minutes to open. In the meanwhile, you can continue reading the text and come back once it has opened. This is what the pe-tree output will look like when we open the redline utility.



The above screenshot shows some basic information about the PE file in the right-hand pane. We see the size, hashes, Entropy, architecture, and compiled date of the PE file. This information is not extracted directly from the header; instead, it is calculated or extracted from different parts of the header, as we will see later. The header starts below this information, with the heading IMAGE_DOS_HEADER. Let's dive into that and learn what information that contains.

The IMAGE_DOS_HEADER

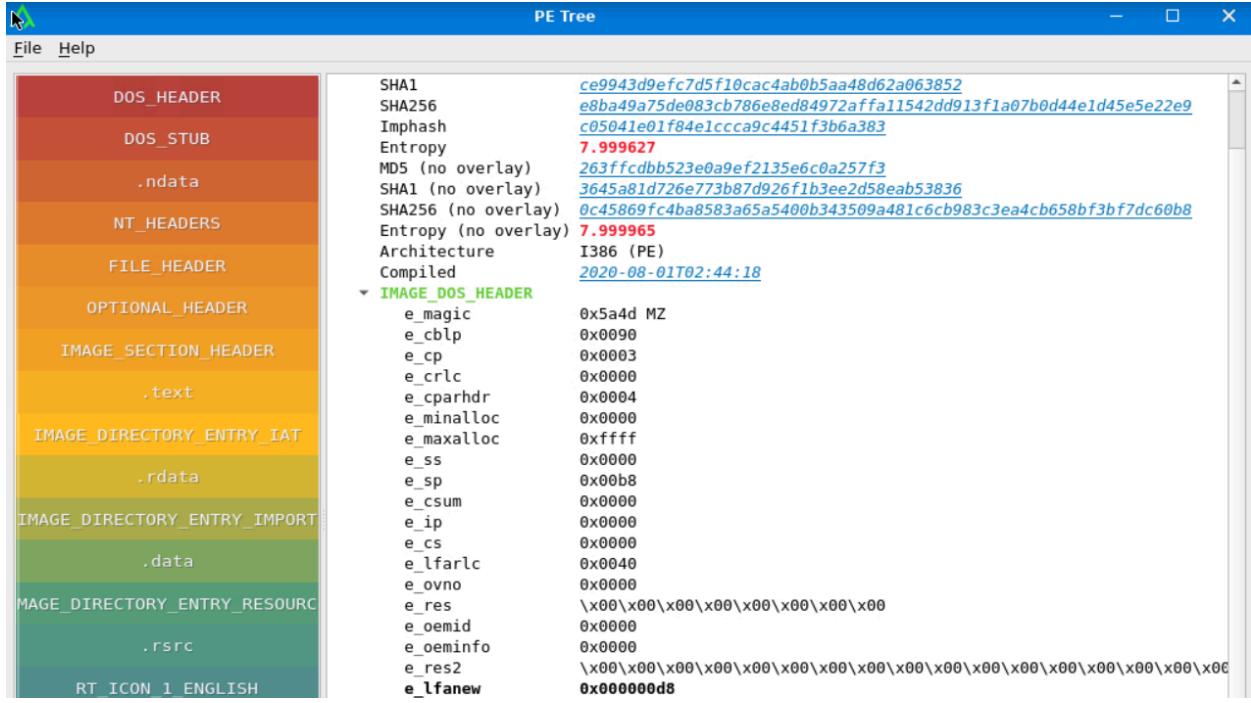
The IMAGE_DOS_HEADER consists of the first 64 bytes of the PE file. We will analyze some of the valuable information found in the IMAGE_DOS_HEADER below. The below screenshot has the IMAGE_DOS_HEADER highlighted in the Hex Editor.



In the screenshot above from the Hex Editor, notice the first two bytes that say 4D 5A. They translate to the MZ characters in ASCII, as shown in the right pane of the Hex Editor. So what do these characters mean?

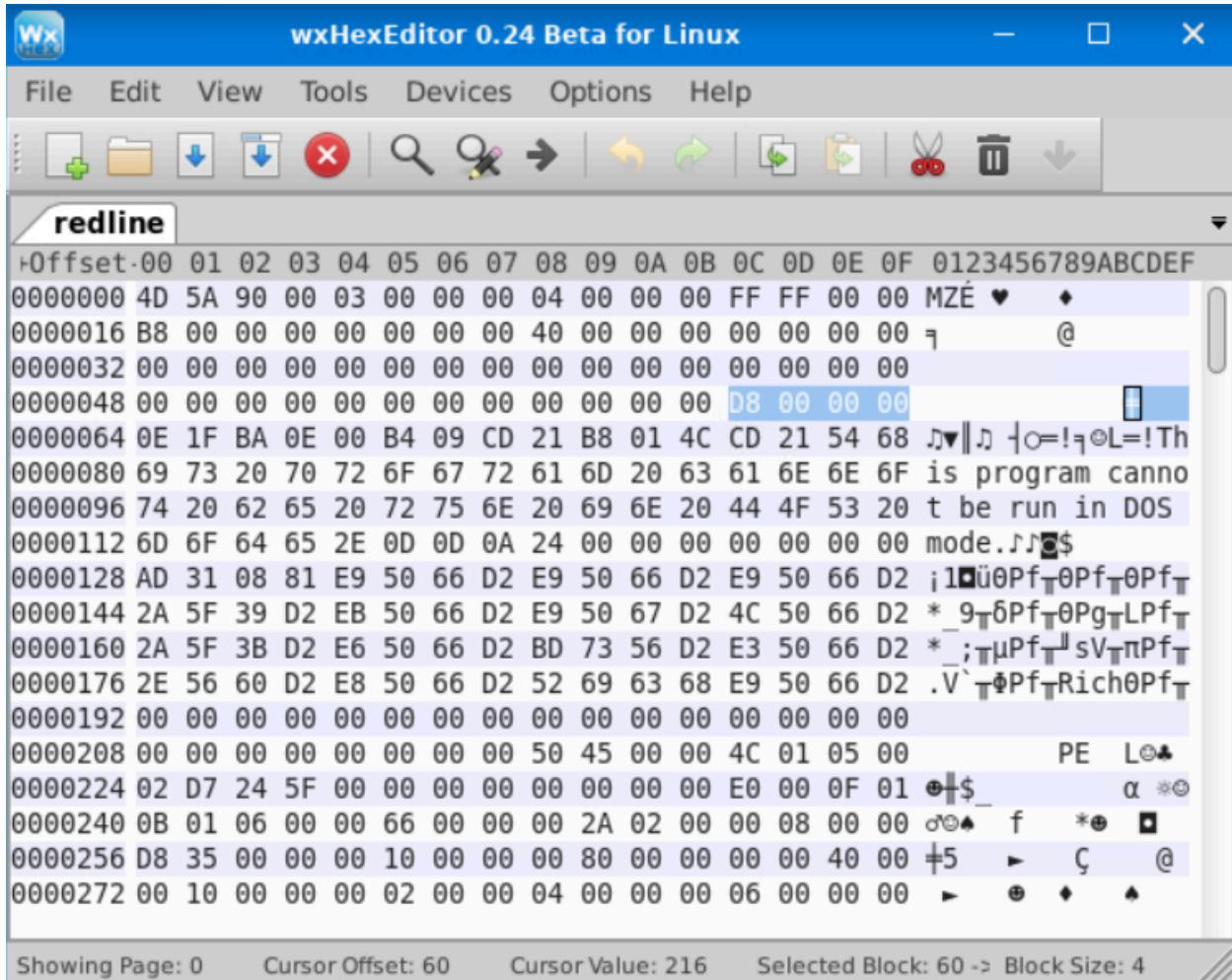
The MZ characters denote the initials of [Mark Zbikowski](#), one of the Microsoft architects who created the MS-DOS file format. The MZ characters are an identifier of the Portable Executable format. When these two bytes are present at the start of a file, the Windows OS considers it a Portable Executable format file.

This is what it will look like when we expand the IMAGE_DOS_HEADER dropdown menu:



Notice the first entry in the IMAGE_DOS_HEADER dropdown menu. It says e_magic and has a value of 0x5a4d MZ. This is the same as what we saw in the Hex Editor above, but the byte order is reversed due to [endianness](#). The Intel x86 architecture uses a little-endian format, while ARM uses a big-endian format.

The last value in the IMAGE_DOS_HEADER is called e_lfanew. In the above screenshot, it has a value of 0x000000d8. This denotes the address from where the IMAGE_NT_HEADERS start. Therefore, in this PE file, the IMAGE_NT_HEADERS start from the address 0x000000d8. We can see this value highlighted in the Hex editor below.



We have to remember that the byte order is switched when we compare those reported by the pe-tree utility and those we see in the Hex editor due to endianness.

The IMAGE_DOS_HEADER is generally not of much use apart from these fields, especially during malware reverse engineering. The only reason it's there is backward compatibility between MS-DOS and Windows.

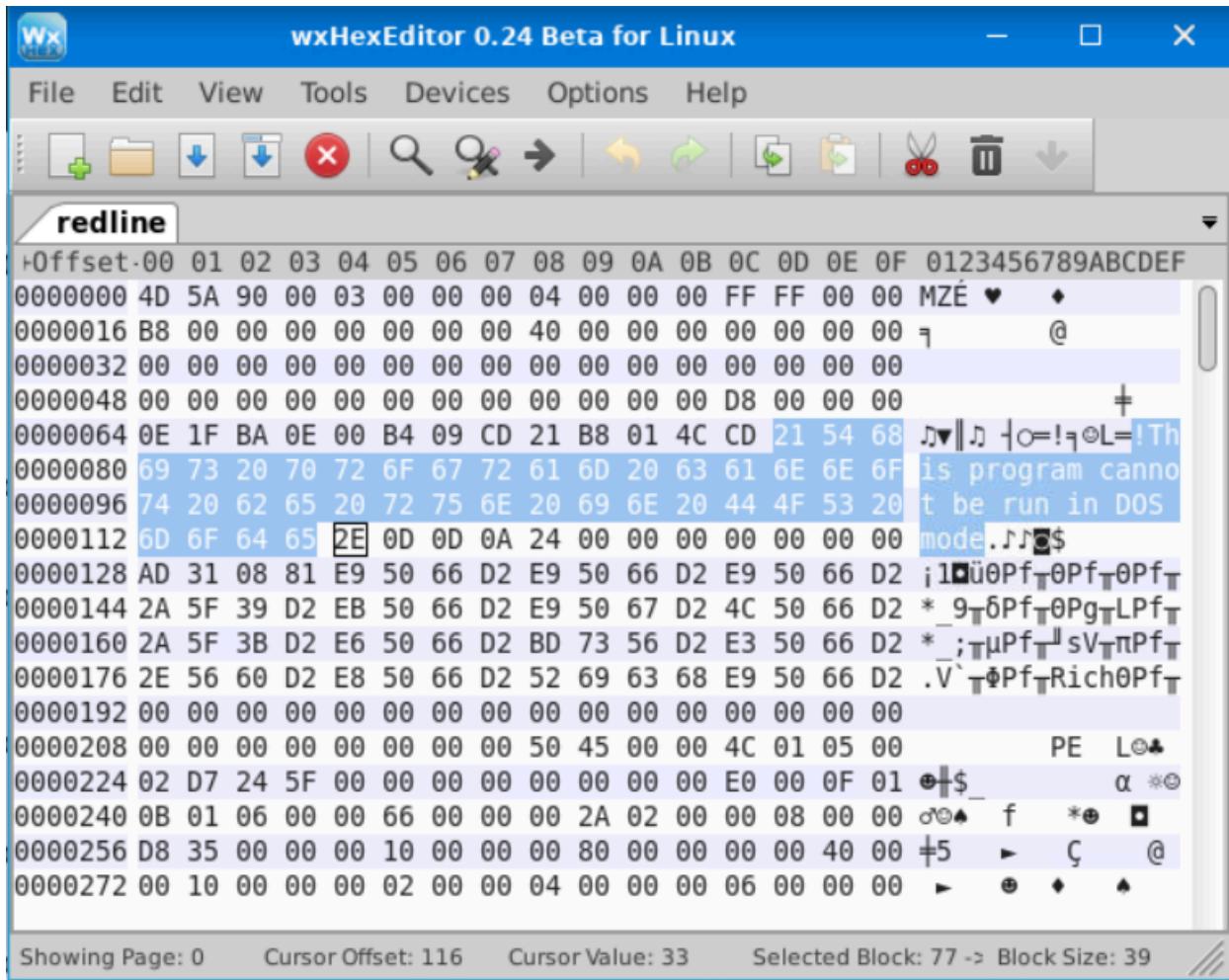
The DOS_STUB

In the pe-tree utility, we see that the following dropdown menu after IMAGE_DOS_HEADER is the DOS STUB. Let's expand that and see what we find in there.

▼ redline	
Size	5.7 MB (5985678 bytes)
MD5	ca2dc5a3f94c4f19334cc8b68f256259
SHA1	ce9943d9efc7d5f10cac4ab0b5aa48d62a063852
SHA256	e8ba49a75de083cb786e8ed84972affa11542dd913f1a07b0d44e1d45e5e22e9
Imphash	c05041e01f84e1ccca9c4451f3b6a383
Entropy	7.999627
MD5 (no overlay)	263ffcd8bb523e0a9ef2135e6c0a257f3
SHA1 (no overlay)	3645a81d726e773b87d926f1b3ee2d58eab53836
SHA256 (no overlay)	0c45869fc4ba8583a65a5400b343509a481c6cb983c3ea4cb658bf3bf7dc60b8
Entropy (no overlay)	7.999965
Architecture	I386 (PE)
Compiled	2020-08-01T02:44:18
► IMAGE_DOS_HEADER	
► DOS_STUB	
Size	152 bytes
Ratio	0.00%
MD5	2a6f7f47527a1fa28a782d7b9629e69c
SHA1	44c51fcc9abada1681d0312b6525523ec2cb56eb
SHA256	0fe911f56a87b8c03b173ee58ebf53336f57be643ba1c9c0c5666ab2016817d1
Entropy	4.875708
Message	!This program cannot be run in DOS mode.
► RICH_HEADER	
► IMAGE_NT_HEADERS	
► OVERLAY	

The DOS STUB contains the message that we also see in the Hex Editor !This program cannot be run in DOS mode, as seen in the screenshot below. Please note that the size, hashes, and Entropy shown here by pe-tree are not related to the PE file; instead, it is for the particular section we are analyzing. These values are calculated based on the data in a specific header and are not included.

The size value denotes the size of the section in bytes. Then we see different hashes for the section. We learned about hashes in the [Intro to Malware Analysis room](#). Entropy is the amount of randomness found in data. The higher the value of Entropy, the more random the data is. We will learn about the utility of Entropy as we learn more about malware analysis.



The DOS STUB is a small piece of code that only runs if the PE file is incompatible with the system it is being run on. It displays the message mentioned above. For example, since this PE file we are examining is a Windows executable, if it is run in MS-DOS, the PE file will exit after showing the message in the DOS STUB.

Answer the questions below:

How many bytes are present in the IMAGE_DOS_HEADER?

Answer: 64

What does MZ stand for?

Answer: Mark Zbikowski

In what variable of the IMAGE_DOS_HEADER is the address of IMAGE_NT_HEADERS saved?

Answer:

Answer: e Ifanew

In the attached VM, open the PE file Desktop/Samples/zmsuz3pinwl in pe-tree. What is the address of IMAGE NT HEADERS for this PE file?

Answer: 0x000000f8

IMAGE NT HEADERS

The rest of the room will focus on the different parts of IMAGE_NT_HEADERS. We can find details of IMAGE_NT_HEADERS in Microsoft Documentation. This header contains most of the vital information related to the PE file. In pe-tree, this is how the IMAGE_NT_HEADERS look like:

▼ redline	
Size	5.7 MB (5985678 bytes)
MD5	ca2dc5a3f94c4f19334cc8b68f256259
SHA1	ce9943d9efc7d5f10cac4ab0b5aa48d62a063852
SHA256	e8ba49a75de083cb786e8ed84972affa11542dd913f1a07b0d44e1d45e5e22e
Imphash	c05041e01f84e1ccca9c4451f3b6a383
Entropy	7.999627
MD5 (no overlay)	263ffcd8bb523e0a9ef2135e6c0a257f3
SHA1 (no overlay)	3645a81d726e773b87d926f1b3ee2d58eab53836
SHA256 (no overlay)	0c45869fc4ba8583a65a5400b343509a481c6cb983c3ea4cb658bf3bf7dc60b
Entropy (no overlay)	7.999965
Architecture	I386 (PE)
Compiled	2020-08-01T02:44:18
► IMAGE_DOS_HEADER	
► DOS_STUB	
▼ IMAGE_NT_HEADERS	
► NT_HEADERS	
► IMAGE_SECTION_HEADER	
► IMAGE_IMPORT_DESCRIPTOR	
► IMAGE_RESOURCE_DIRECTORY	
► OVERLAY	

NT_HEADERS

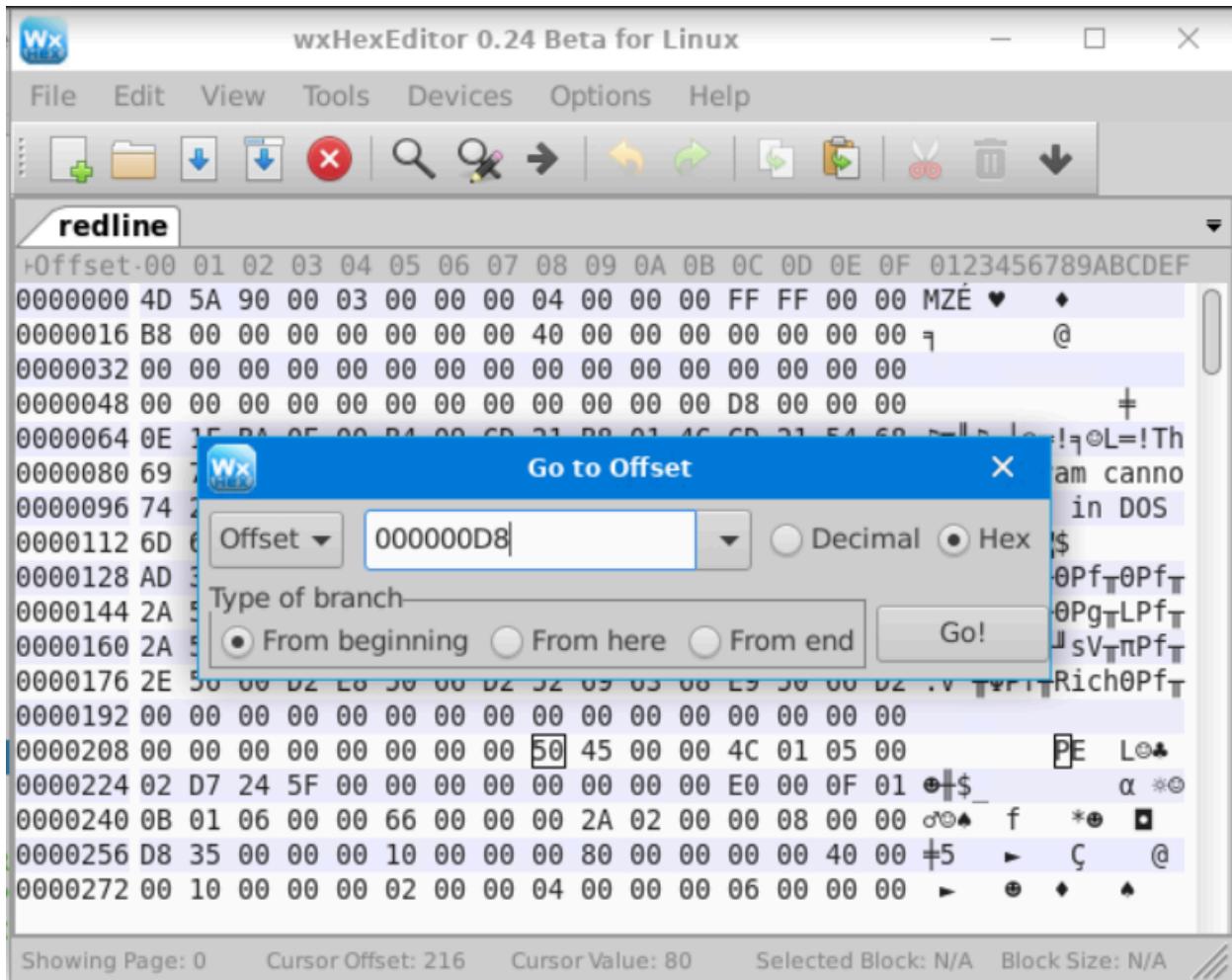
Before diving into the details of NT_HEADERS, let's get an overview of the NT_HEADERS. The NT_HEADERS consist of the following:

- Signature
- FILE_HEADER
- OPTIONAL_HEADER

We will cover the Signature and FILE_HEADER in this task but the OPTIONAL_HEADER in the next task.

▼ redline	
Size	5.7 MB (5985678 bytes)
MD5	ca2dc5a3f94c4f19334cc8b68f256259
SHA1	ce9943d9efc7d5f10cac4ab0b5aa48d62a063852
SHA256	e8ba49a75de083cb786e8ed84972affa11542dd913f1a07b0d44e1d45e5e22e
Imphash	c05041e01f84e1ccca9c4451f3b6a383
Entropy	7.999627
MD5 (no overlay)	263ffcd8bb523e0a9ef2135e6c0a257f3
SHA1 (no overlay)	3645a81d726e773b87d926f1b3ee2d58eab53836
SHA256 (no overlay)	0c45869fc4ba8583a65a5400b343509a481c6cb983c3ea4cb658bf3bf7dc60b
Entropy (no overlay)	7.999965
Architecture	I386 (PE)
Compiled	2020-08-01T02:44:18
► IMAGE_DOS_HEADER	
► DOS_STUB	
► IMAGE_NT_HEADERS	
▼ NT_HEADERS	
Signature	0x00004550 PE
► FILE_HEADER	
► OPTIONAL_HEADER	
► IMAGE_SECTION_HEADER	
► IMAGE_IMPORT_DESCRIPTOR	
► IMAGE_RESOURCE_DIRECTORY	
► OVERLAY	

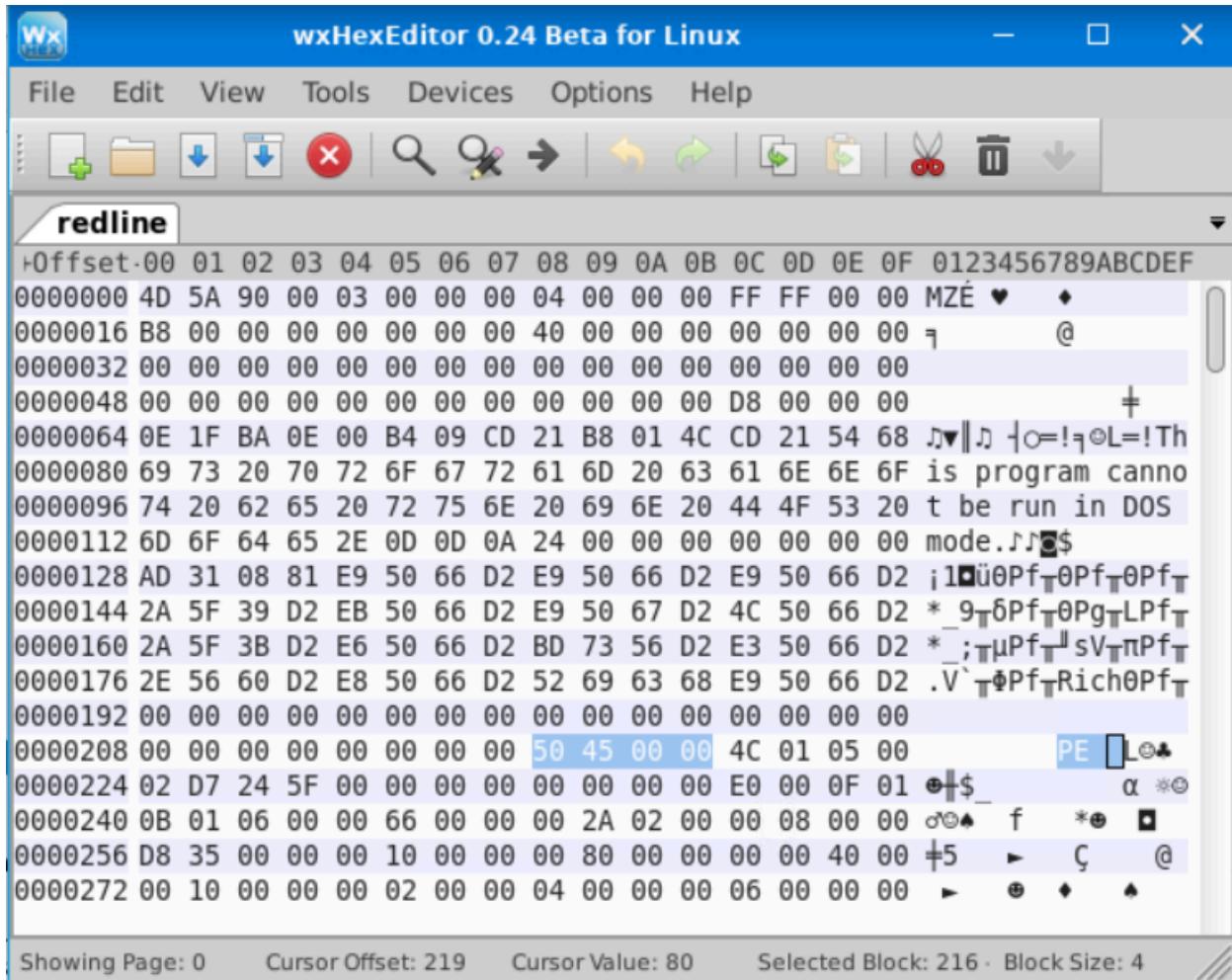
The starting address of IMAGE_NT_HEADERS is found in e_lfanew from the IMAGE_DOS_HEADER. In the redline binary, we saw that this address was 0x000000D8. So let's start by going to this offset and see what we find there. We can do that by pressing Ctrl+G in the Hex Editor Window or going to Edit>Go to offset from the GUI.



We have to make sure that we select From beginning in Type of branch option at the bottom and the data type is set to Hex for correct results.

Signature

The first 4 bytes of the NT_HEADERS consist of the Signature. We can see this as the bytes 50 45 00 00 in Hex, or the characters PE in ASCII as shown in the Hex editor.



The Signature denotes the start of the NT_HEADER. Apart from the Signature, the NT_HEADER contains the FILE_HEADER and the IMAGE_OPTIONAL_HEADER.

FILE_HEADER

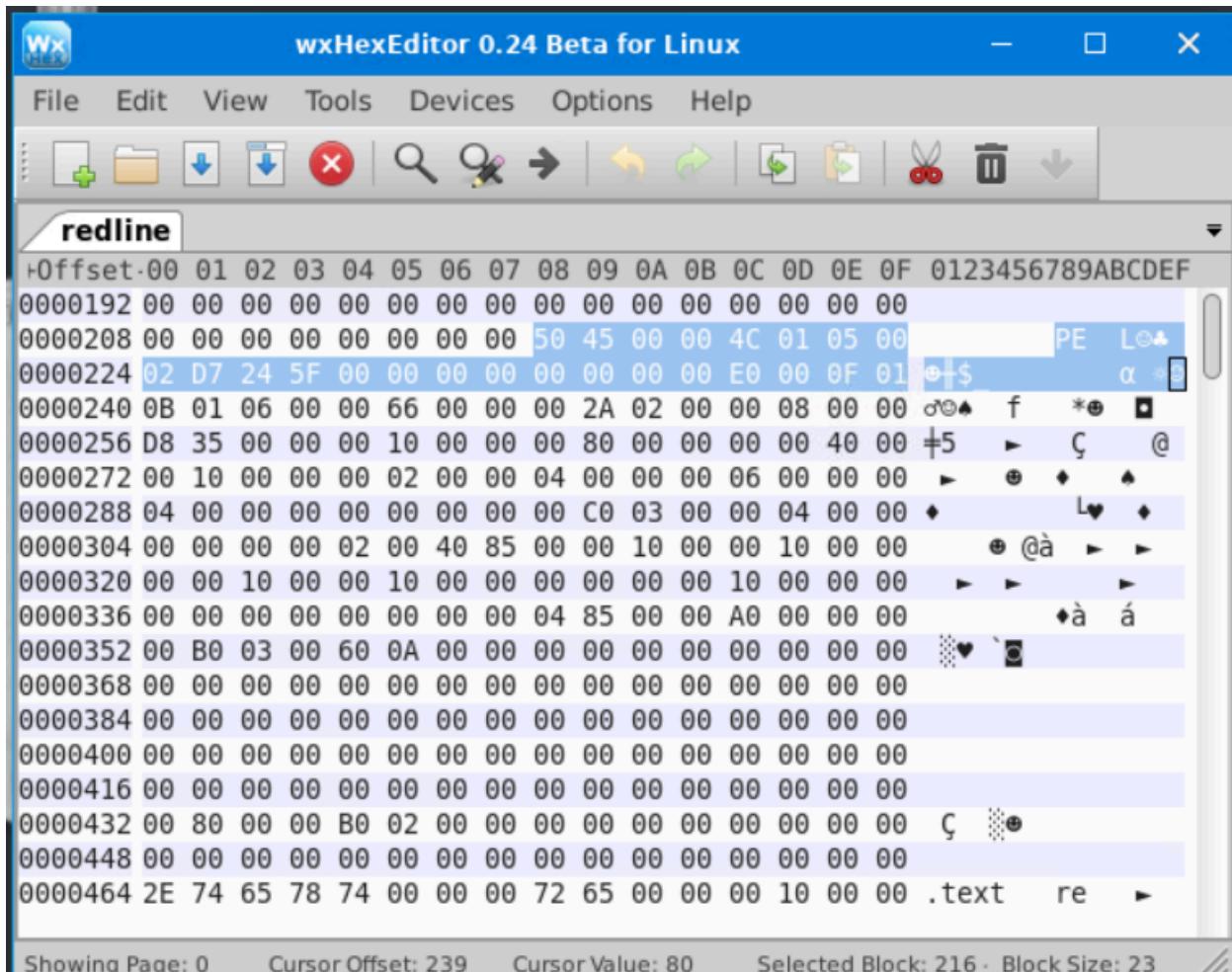
The FILE_HEADER contains some vital information. The following screenshot shows the FILE_HEADER as shown in the pe-tree utility.

Size	5.7 MB (5985678 bytes)
MD5	ca2dc5a3f94c4f19334cc8b68f256259
SHA1	ce9943d9efc7d5f10cac4ab0b5aa48d62a063852
SHA256	e8ba49a75de083cb786e8ed84972affa11542dd913f1a07b0d44e1d45e22
Imphash	c05041e01f84e1ccca9c4451f3b6a383
Entropy	7.999627
MD5 (no overlay)	263ffcd8b523e0a9ef2135e6c0a257f3
SHA1 (no overlay)	3645a81d726e773b87d926f1b3ee2d58eab53836
SHA256 (no overlay)	0c45869fc4ba8583a65a5400b343509a481c6cb983c3ea4cb658bf3bf7dc60
Entropy (no overlay)	7.999965
Architecture	I386 (PE)
Compiled	2020-08-01T02:44:18
► IMAGE_DOS_HEADER	
► DOS_STUB	
► IMAGE_NT_HEADERS	
► NT_HEADERS	
Signature	0x00004550 PE
► FILE_HEADER	
Machine	0x014c I386
NumberOfSections	0x0005
TimeStamp	0x5f24d702 Sat Aug 1 02:44:18 2020 UTC
PointerToSymbolTable	0x00000000
NumberOfSymbols	0x00000000
SizeOfOptionalHeader	0x00e0 224 bytes
Characteristics	0x010f RELOCS_STRIPPED EXECUTABLE_IMAGE LINE_NUMS_STRIPPED
► OPTIONAL_HEADER	
► IMAGE_SECTION_HEADER	
► IMAGE_DIRECTORY_ENTRY	

As we can see in the above screenshot, the FILE_HEADER has the following fields:

- Machine: This field mentions the type of architecture for which the PE file is written. In the above example, we can see that the architecture is i386 which means that this PE file is compatible with 32-bit Intel architecture.
- NumberOfSections: A PE file contains different sections where code, variables, and other resources are stored. This field of the IMAGE_FILE_HEADER mentions the number of sections the PE file has. In our case, the PE file has five sections. We will learn about sections later in the room.
- TimeStamp: This field contains the time and date of the binary compilation.
- PointerToSymbolTable and NumberOfSymbols: These fields are not generally related to PE files. Instead, they are here due to the COFF file headers.
- SizeOfOptionalHeader: This field contains the size of the optional header, which we will learn about in the next task. In our case, the size is 224 bytes.
- Characteristics: This is another critical field. This field mentions the different characteristics of a PE file. In our case, this field tells us that the PE file has stripped relocation information, line numbers, and local symbol information. It is an executable image and compatible with a 32-bit machine.

While we looked at the FILE_HEADER using the pe-tree utility, we can see that the hex values for each field are also shown in the pe-tree utility. Can you look at the Hex editor and find where each value is located?



We are starting to learn to read Hex now, aren't we? To learn more about the FILE_HEADER, you can check out [Microsoft Documentation](#) for it.

Answer the questions below:

In the attached VM, there is a file Desktop\Samples\zmsuz3pinwl. Open this file in pe-tree. Is this PE file compiled for a 32-bit machine or a 64-bit machine?

zmsuz3pinwl		⚠ 8 warnings
Size	1.2 MB (1247232 bytes)	
MD5	lebb1e268a462d56a389e8e1d06b4945	
SHA1	1ecc0b9f380896373e81ed166c34a89bded873b5	
SHA256	98c6cf0b129438ec62a628e8431e790b114ba0d8	
Imphash	829df24b10d39ea3719a931bde7a76d7	
Entropy	7.978052	
Architecture	I386 (PE)	
Compiled	2022-03-09T12:27:49	
▶ IMAGE_DOS_HEADER		
▶ DOS_STUB		
▼ IMAGE_NT_HEADERS		
▼ NT_HEADERS		
Signature	0x00004550 PE	
▼ FILE_HEADER		
Machine	0x014c I386	
NumberOfSections	0x0007	
TimeDateStamp	0x62289d45 Wed Mar 9 12:27:49 2022 UTC	
PointerToSymbolTable	0x00000000	
NumberOfSymbols	0x00000000	
SizeOfOptionalHeader	0x00e0 224 bytes	
Characteristics	0x0103 RELOCS_STRIPPED EXECUTABLE_IMAGE	
▶ OPTIONAL_HEADER		
▶ IMAGE_SECTION_HEADER		
▶ IMAGE_IMPORT_DESCRIPTOR		
▶ IMAGE_DIRECTORY_ENTRY_DEBUG		
▶ IMAGE_RESOURCE_DIRECTORY		

Answer: 32-bit Machine

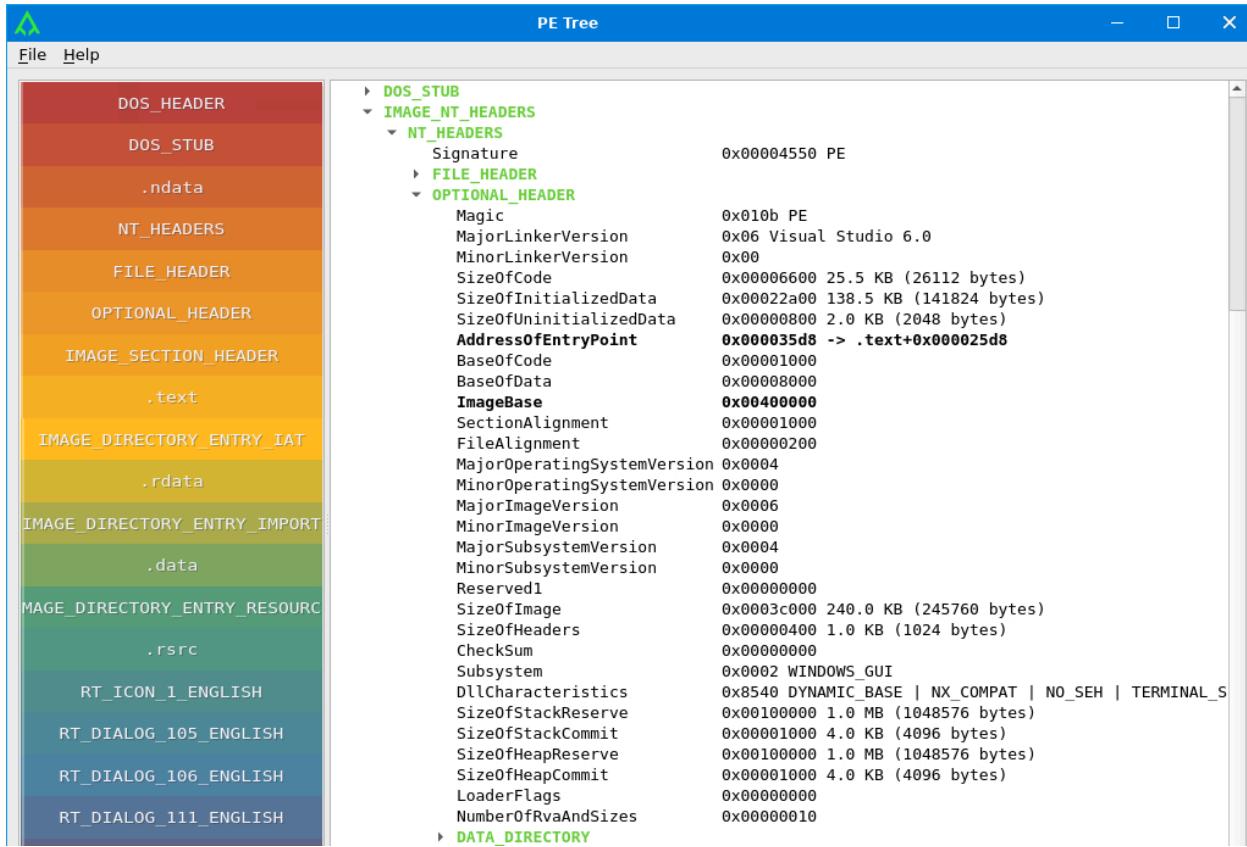
What is the TimeDateStamp of this file?

zmsuz3pinwl		8 warnings
Size	1.2 MB (1247232 bytes)	
MD5	1ebble268a462d56a389e8e1d06b4945	
SHA1	1ecc0b9f380896373e81ed166c34a89bded873b5	
SHA256	98c6cf0b129438ec62a628e8431e790b114ba0d8	
Imphash	829df24b10d39ea3719a931bde7a76d7	
Entropy	7.978052	
Architecture	I386 (PE)	
Compiled	2022-03-09T12:27:49	
► IMAGE_DOS_HEADER		
► DOS_STUB		
▼ IMAGE_NT_HEADERS		
▼ NT_HEADERS		
Signature	0x00004550 PE	
▼ FILE_HEADER		
Machine	0x014c I386	
NumberOfSections	0x0007	
TimeStamp	0x62289d45 Wed Mar 9 12:27:49 2022 UTC	
PointerToSymbolTable	0x00000000	
NumberOfSymbols	0x00000000	
SizeOfOptionalHeader	0x00e0 224 bytes	
Characteristics	0x0103 RELOCS_STRIPPED EXECUTABLE_IMAGE	
► OPTIONAL_HEADER		
► IMAGE_SECTION_HEADER		
► IMAGE_IMPORT_DESCRIPTOR		
► IMAGE_DIRECTORY_ENTRY_DEBUG		
► IMAGE_RESOURCE_DIRECTORY		

Answer: **0x62289d45 Wed Mar 9 12:27:49 2022 UTC**

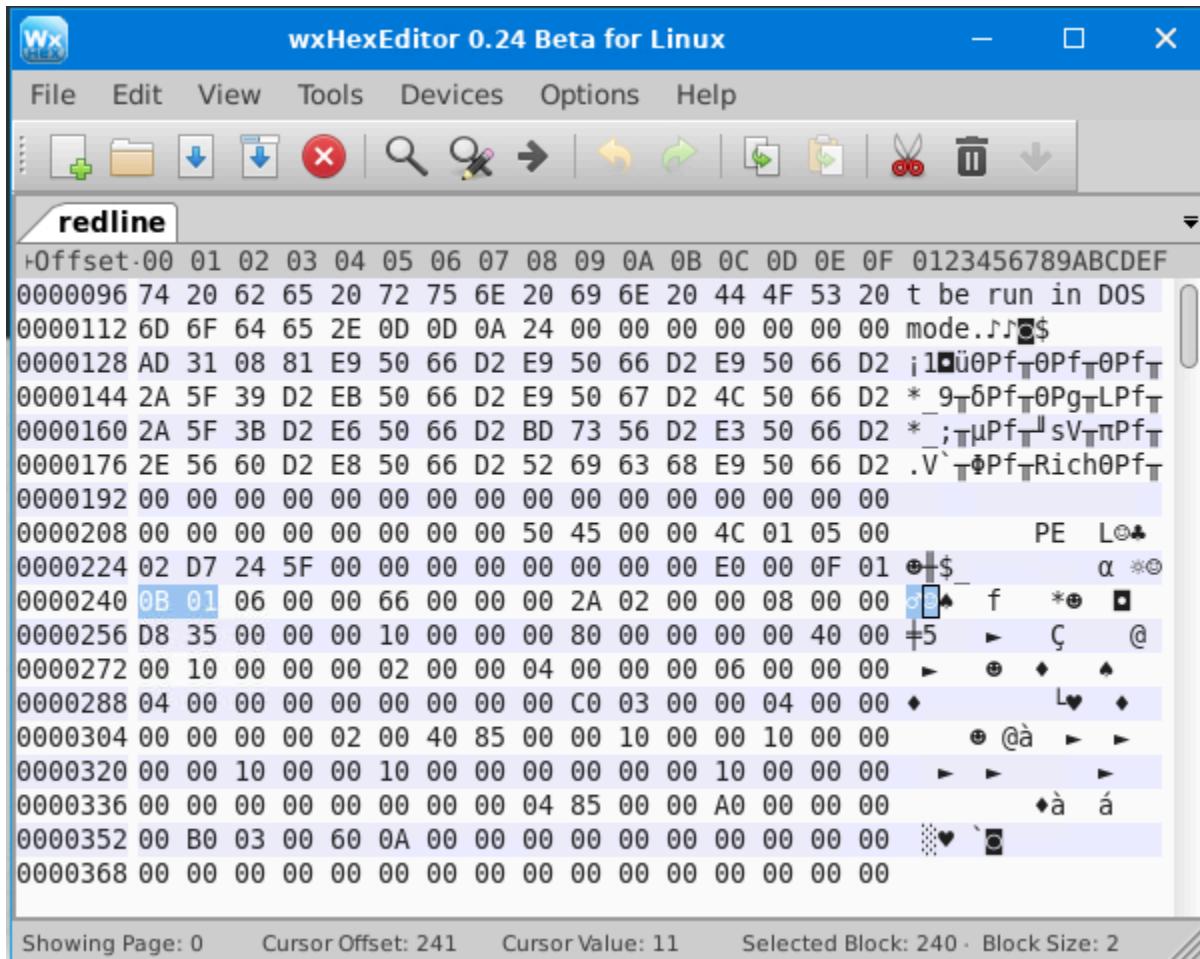
OPTIONAL_HEADER

The OPTIONAL_HEADER is also a part of the NT_HEADERS. It contains some of the most important information present in the PE headers. Let's see what it looks like in the pe-tree utility.



In the Hex editor, the OPTIONAL_HEADER starts right after the end of the FILE_HEADER. Below you can see the start of the OPTIONAL_HEADER of the redline

binary in the Hex Editor.



Let's learn about some of the critical fields in the OPTIONAL_HEADER.

- Magic: The Magic number tells whether the PE file is a 32-bit or 64-bit application. If the value is 0x010B, it denotes a 32-bit application; if the value is 0x020B, it represents a 64-bit application. The above screenshot of the Hex Editor shows the highlighted bytes, which show the magic of the loaded PE file. Since the value is 0x010B, it shows that it is a 32-bit application.
- AddressOfEntryPoint: This field is significant from a malware analysis/reverse-engineering point of view. This is the address from where Windows will begin execution. In other words, the first instruction to be executed is present at this address. This is a Relative Virtual Address (RVA), meaning it is at an offset relative to the base address of the image (ImageBase) once loaded into memory.
- BaseOfCode and BaseOfData: These are the addresses of the code and data sections, respectively, relative to ImageBase.
- ImageBase: The ImageBase is the preferred loading address of the PE file in memory. Generally, the ImageBase for .exe files is 0x00400000, which is also the

case for our PE file. Since Windows can't load all PE files at this preferred address, some relocations are in order when the file is loaded in memory. These relocations are then performed relative to the ImageBase.

- Subsystem: This represents the Subsystem required to run the image. The Subsystem can be Windows Native, GUI (Graphical User Interface), CUI (Commandline User Interface), or some other Subsystem. The screenshot above from the pe-tree utility shows that the Subsystem is 0x0002, representing Windows GUI Subsystem. We can find the complete list in [Microsoft Documentation](#).
- DataDirectory: The DataDirectory is a structure that contains import and export information of the PE file (called Import Address Table and Export Address Table). This information is handy as it gives a glimpse of what the PE file might be trying to do. We will expand on the import information later in this room.

Though there is more information in the OPTIONAL_HEADER, we will not go into those in this room. If you want to learn more about the OPTIONAL_HEADER, you can check out [Microsoft Documentation](#) about this header.

Answer the questions below:

Which variable from the OPTIONAL_HEADER indicates whether the file is a 32-bit or a 64-bit application?

Answer: **Magic**

What Magic value indicates that the file is a 64-bit application?

Answer: **0x020B**

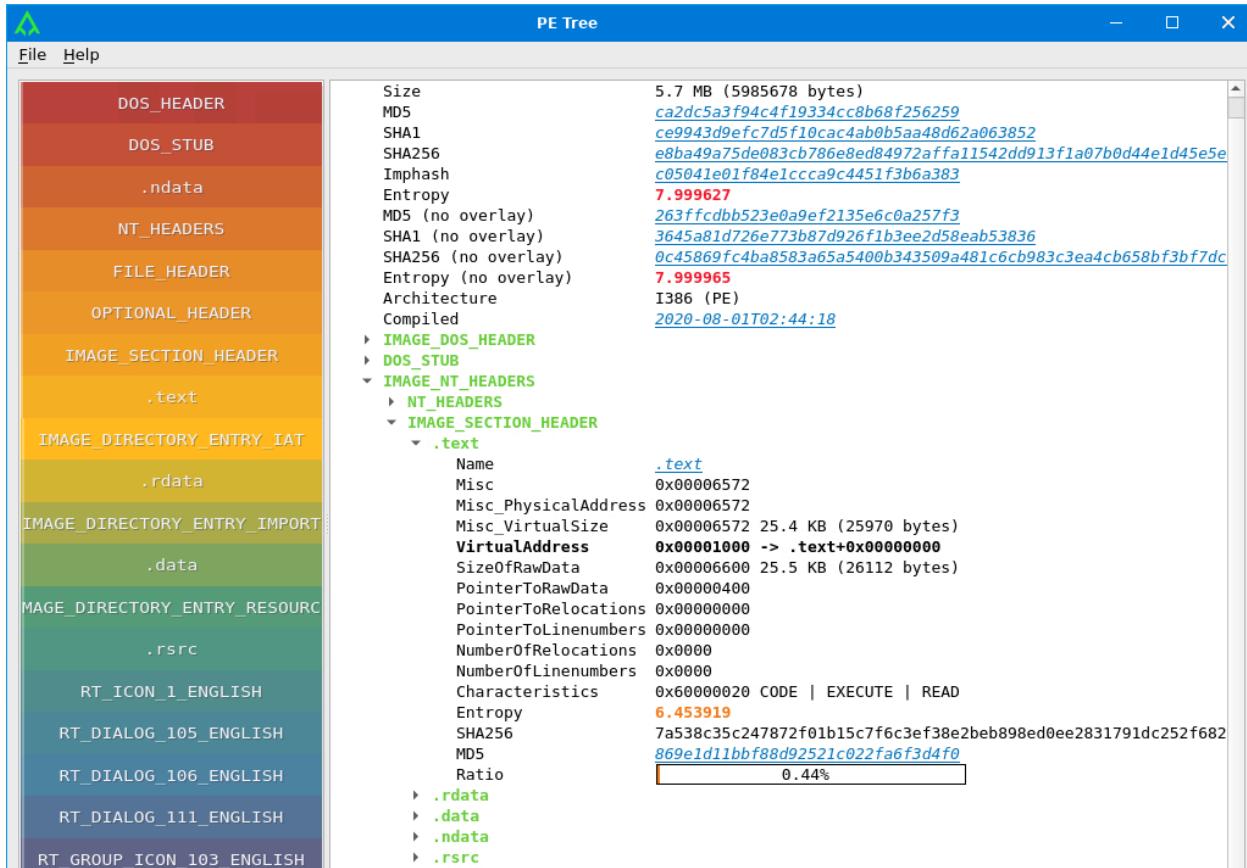
What is the subsystem of the file Desktop\Samples\zmsuz3pinwl?

► IMAGE_DOS_HEADER	
► DOS_STUB	
▼ IMAGE_NT_HEADERS	
▼ NT_HEADERS	
Signature	0x00004550 PE
► FILE_HEADER	
▼ OPTIONAL_HEADER	
Magic	0x010b PE
MajorLinkerVersion	0x0e 14.30
MinorLinkerVersion	0x1e
SizeOfCode	0x003f3800 4.0 MB (4143104 bytes)
SizeOfInitializedData	0x00014c00 83.0 KB (84992 bytes)
SizeOfUninitializedData	0x00000000 0 bytes
AddressOfEntryPoint	0x00001000 -> +0x00000000
BaseOfCode	0x00001000
BaseOfData	0x003f5000
ImageBase	0x00400000 -> .rsrc+0x00008000
SectionAlignment	0x00001000
FileAlignment	0x00000200
MajorOperatingSystemVersion	0x0006 Windows Vista/Windows Serv
MinorOperatingSystemVersion	0x0000
MajorImageVersion	0x0000
MinorImageVersion	0x0000
MajorSubsystemVersion	0x0006
MinorSubsystemVersion	0x0000
Reserved1	0x00000000
SizeOfImage	0x00469000 4.4 MB (4624384 bytes)
SizeOfHeaders	0x00000400 1.0 KB (1024 bytes)
CheckSum	0x0040991b
Subsystem	0x0003 WINDOWS_CUI
DllCharacteristics	0x8100 NX_COMPAT TERMINAL_SERVE
SizeOfStackReserve	0x00100000 1.0 MB (1048576 bytes)
SizeOfStackCommit	0x00001000 4.0 KB (4096 bytes)
SizeOfHeapReserve	0x00100000 1.0 MB (1048576 bytes)

Answer: **0x0003 WINDOWS_CUI**

IMAGE_SECTION_HEADER

The data that a PE file needs to perform its functions, like code, icons, images, User Interface elements, etc., are stored in different Sections. We can find information about these Sections in the IMAGE_SECTION_HEADER. In the pe-tree utility, the IMAGE_SECTION_HEADER is shown for each separate section, as can be seen in the below screenshot.



As we can see, the IMAGE_SECTION_HEADER has different sections, namely .text, .rdata, .data, .ndata and .rsrc. Before moving to the information present in the header of each section, let's learn about the commonly found sections in a PE file.

- .text: The .text section is generally the section that contains executable code for the application. We can see above that the Characteristics for this section include CODE, EXECUTE and READ, meaning that this section contains executable code, which can be read but can't be written to.
- .data: This section contains initialized data of the application. It has READ/WRITE permissions but doesn't have EXECUTE permissions.
- .rdata/.idata: These sections often contain the import information of the PE file. Import information helps a PE file import functions from other files or Windows API.
- .ndata: The .ndata section contains uninitialized data.
- .reloc: This section contains relocation information of the PE file.
- .rsrc: The resource section contains icons, images, or other resources required for the application UI.

Now that we know what the different types of sections are commonly found in a PE file, let's see what important information the section headers for each section include:

- VirtualAddress: This field indicates this section's Relative Virtual Address (RVA) in the memory.
 - VirtualSize: This field indicates the section's size once loaded into the memory.
 - SizeOfRawData: This field represents the section size as stored on the disk before the PE file is loaded in memory.
 - Characteristics: The characteristics field tells us the permissions that the section has. For example, if the section has READ permissions, WRITE permissions or EXECUTE permissions.
- *****

Answer the questions below:

How many sections does the file Desktop\Samples\zmsuz3pinwl have?

zmsuz3pinwl		⚠ 8 warnings
Size	1.2 MB (1247232 bytes)	
MD5	1ebb1e268a462d56a389e8e1d06b4945	
SHA1	1ecc0b9f380896373e81ed166c34a89bded873b	
SHA256	98c6cf0b129438ec62a628e8431e790b114ba0d	
Imphash	829df24b10d39ea3719a931bde7a76d7	
Entropy	7.978052	
Architecture	I386 (PE)	
Compiled	2022-03-09T12:27:49	
► IMAGE_DOS_HEADER		
► DOS_STUB		
▼ IMAGE_NT_HEADERS		
► NT_HEADERS		
▼ IMAGE_SECTION_HEADER		
►		
►		
►		
► .rsrc		
►		
► .data		
► .adata		
► IMAGE_IMPORT_DESCRIPTOR		
► IMAGE_DIRECTORY_ENTRY_DEBUG		
► IMAGE_RESOURCE_DIRECTORY		

Answer: 7

**What are the characteristics of the .rsrc section of the file
Desktop\Samples\zmsuz3pinwl**

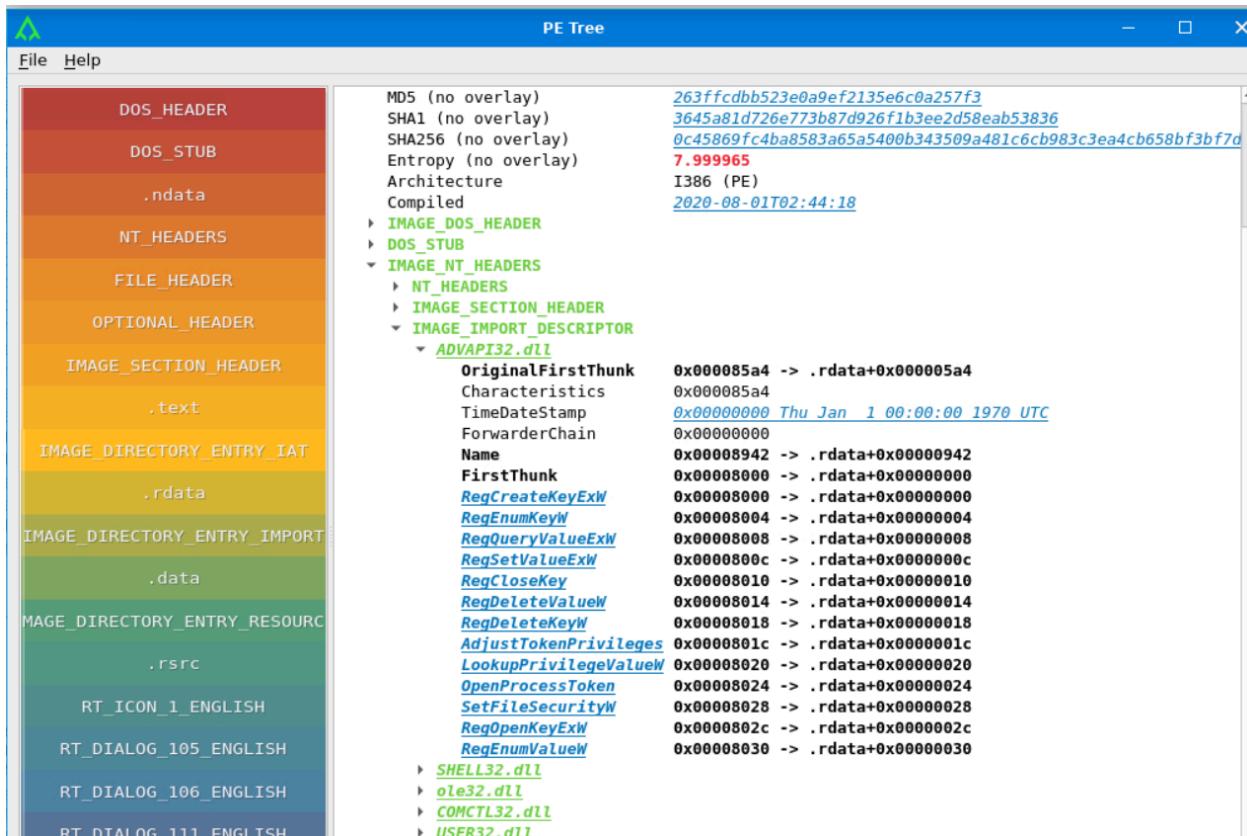
IMAGE_SECTION_HEADER	
Name	.rsrc
Misc	0x00013000
Misc_PhysicalAddress	0x00013000
Misc_VirtualSize	0x00013000 76.0 KB (77824 bytes)
VirtualAddress	0x003f8000 -> .rsrc+0x00000000
SizeOfRawData	0x00000400 1.0 KB (1024 bytes)
PointerToRawData	0x000d4e00
PointerToRelocations	0x00000000
PointerToLinenumbers	0x00000000
NumberOfRelocations	0x0000
NumberOfLinenumbers	0x0000
Characteristics	0xe0000040 INITIALIZED DATA EXECUTE READ WRITE
Entropy	6.938747
SHA256	3316da18e71713f02b2fe1e518eed352068a35387e649bac8b411eb524fcfd472
MD5	08c9e0fc9b79f886b9e3bcb2686d3887
Ratio	0.08%
.data	
.adata	

Answer: 0xe0000040 INITIALIZED DATA | EXECUTE | READ | WRITE

IMAGE IMPORT DESCRIPTOR

PE files don't contain all the code they need to perform their functions. In a Windows Operating System, PE files leverage code from the Windows API to perform many functions. The `IMAGE_IMPORT_DESCRIPTOR` structure contains information about the different Windows APIs that the PE file loads when executed. This information is handy in identifying the potential activity that a PE file might perform. For example, if a PE file imports `CreateFile` API, it indicates that it might create a file when executed.

This is what the IMAGE_IMPORT_DESCRIPTOR looks like in the pe-tree utility.



Here we can see that the PE file we are looking at imports functions from ADVAPI32.dll, SHELL32.dll, ole32.dll, COMCTL32.dll, and USER32.dll. These files are dynamically linked libraries that export Windows functions or APIs for other PE files. The above screenshot shows that the PE file imports some functions that perform some registry actions. To find more information about what the function does, we can check out Microsoft Documentation. For example, [this link](#) has details about the RegCreateKeyExW function.

In the above screenshot, we can see the values OriginalFirstThunk and FirstThunk. The Operating System uses these values to build the Import Address Table (IAT) of the PE file. We will learn more about these values in the coming rooms.

By studying the import functions of a PE file, we can identify some of the activities that the PE file might perform.

<u>FirstThunk</u>	0x00008070 -> .rdata+0x00000070
<u>GetExitCodeProcess</u>	0x00008070 -> .rdata+0x00000070
<u>WaitForSingleObject</u>	0x00008074 -> .rdata+0x00000074
<u>GetModuleHandleA</u>	0x00008078 -> .rdata+0x00000078
<u>GetProcAddress</u>	0x0000807c -> .rdata+0x0000007c
<u>GetSystemDirectoryW</u>	0x00008080 -> .rdata+0x00000080
<u>lstrcmpA</u>	0x00008084 -> .rdata+0x00000084
<u>Sleep</u>	0x00008088 -> .rdata+0x00000088
<u>lstrcpyA</u>	0x0000808c -> .rdata+0x0000008c
<u>WriteFile</u>	0x00008090 -> .rdata+0x00000090
<u>GetTempFileNameW</u>	0x00008094 -> .rdata+0x00000094
<u>lstrcmpiA</u>	0x00008098 -> .rdata+0x00000098
<u>RemoveDirectoryW</u>	0x0000809c -> .rdata+0x0000009c
<u>CreateProcessW</u>	0x000080a0 -> .rdata+0x000000a0
<u>CreateDirectoryW</u>	0x000080a4 -> .rdata+0x000000a4
<u>GetLastError</u>	0x000080a8 -> .rdata+0x000000a8
<u>CreateThread</u>	0x000080ac -> .rdata+0x000000ac
<u>GlobalLock</u>	0x000080b0 -> .rdata+0x000000b0
<u>GlobalUnlock</u>	0x000080b4 -> .rdata+0x000000b4
<u>GetDiskFreeSpaceW</u>	0x000080b8 -> .rdata+0x000000b8
<u>WideCharToMultiByte</u>	0x000080bc -> .rdata+0x000000bc

Take the redline binary from the attached VM as an example. Its IMAGE_IMPORT_DESCRIPTOR imports notable functions such as CreateProcessW, CreateDirectoryW, and WriteFile from kernel32.dll. This implies that this PE file intends to create a process, create a directory, and write some data to a file. Similarly, by studying the rest of the imports, we can potentially identify other activities that a PE file intends to perform.

Answer the questions below:

The PE file Desktop\Samples\redline imports the function CreateWindowExW. From which dll file does it import this function?

COMPILED		2020-09-01 10:44.10
►	IMAGE_DOS_HEADER	
►	DOS_STUB	
▼	IMAGE_NT_HEADERS	
►	NT_HEADERS	
►	IMAGE_SECTION_HEADER	
▼	IMAGE_IMPORT_DESCRIPTOR	
►	ADVAPI32.dll	
►	SHELL32.dll	
►	ole32.dll	
►	COMCTL32.dll	
▼	USER32.dll	
OriginalFirstThunk	0x00008738 -> .rdata+0x00000738	
Characteristics	0x00008738	
TimeDateStamp	0x00000000 Thu Jan 1 00:00:00 1970 UTC	
ForwarderChain	0x00000000	
Name	0x00008e98 -> .rdata+0x00000e98	
FirstThunk	0x00008194 -> .rdata+0x00000194	
GetClientRect	0x00008194 -> .rdata+0x00000194	
EndPaint	0x00008198 -> .rdata+0x00000198	
DrawTextW	0x0000819c -> .rdata+0x0000019c	
IsWindowEnabled	0x000081a0 -> .rdata+0x000001a0	
DispatchMessageW	0x000081a4 -> .rdata+0x000001a4	
wsprintfA	0x000081a8 -> .rdata+0x000001a8	
CharNextA	0x000081ac -> .rdata+0x000001ac	
CharPrevW	0x000081b0 -> .rdata+0x000001b0	
MessageBoxIndirectW	0x000081b4 -> .rdata+0x000001b4	
GetDlgItemTextW	0x000081b8 -> .rdata+0x000001b8	
SetDlgItemTextW	0x000081bc -> .rdata+0x000001bc	
GetSystemMetrics	0x000081c0 -> .rdata+0x000001c0	
FillRect	0x000081c4 -> .rdata+0x000001c4	
AppendMenuW	0x000081c8 -> .rdata+0x000001c8	
TrackPopupMenu	0x000081cc -> .rdata+0x000001cc	
OpenClipboard	0x000081d0 -> .rdata+0x000001d0	

DrawTextW	0x0000819c -> .rdata+0x0000019c
IsWindowEnabled	0x000081a0 -> .rdata+0x000001a0
DispatchMessageW	0x000081a4 -> .rdata+0x000001a4
wsprintfA	0x000081a8 -> .rdata+0x000001a8
CharNextA	0x000081ac -> .rdata+0x000001ac
CharPrevW	0x000081b0 -> .rdata+0x000001b0
MessageBoxIndirectW	0x000081b4 -> .rdata+0x000001b4
GetDlgItemTextW	0x000081b8 -> .rdata+0x000001b8
SetDlgItemTextW	0x000081bc -> .rdata+0x000001bc
GetSystemMetrics	0x000081c0 -> .rdata+0x000001c0
FillRect	0x000081c4 -> .rdata+0x000001c4
AppendMenuW	0x000081c8 -> .rdata+0x000001c8
TrackPopupMenu	0x000081cc -> .rdata+0x000001cc
OpenClipboard	0x000081d0 -> .rdata+0x000001d0
SetClipboardData	0x000081d4 -> .rdata+0x000001d4
CloseClipboard	0x000081d8 -> .rdata+0x000001d8
IsWindowVisible	0x000081dc -> .rdata+0x000001dc
CallWindowProcW	0x000081e0 -> .rdata+0x000001e0
GetMessagePos	0x000081e4 -> .rdata+0x000001e4
CheckDlgButton	0x000081e8 -> .rdata+0x000001e8
LoadCursorW	0x000081ec -> .rdata+0x000001ec
SetCursor	0x000081f0 -> .rdata+0x000001f0
GetWindowLongW	0x000081f4 -> .rdata+0x000001f4
GetSysColor	0x000081f8 -> .rdata+0x000001f8
SetWindowPos	0x000081fc -> .rdata+0x000001fc
PeekMessageW	0x00008200 -> .rdata+0x00000200
SetClassLongW	0x00008204 -> .rdata+0x00000204
GetSystemMenu	0x00008208 -> .rdata+0x00000208
EnableMenuItem	0x0000820c -> .rdata+0x0000020c
GetWindowRect	0x00008210 -> .rdata+0x00000210
ScreenToClient	0x00008214 -> .rdata+0x00000214
EndDialog	0x00008218 -> .rdata+0x00000218
RegisterClassW	0x0000821c -> .rdata+0x0000021c
SystemParametersInfoW	0x00008220 -> .rdata+0x00000220
CreateWindowExW	0x00008224 -> .rdata+0x00000224
GetClassInfoW	0x00008228 -> .rdata+0x00000228
DialogBoxParamW	0x0000822c -> .rdata+0x0000022c

Answer: **USER32.dll**

Packing and Identifying Packed Executables

Since a PE file's information can be easily extracted using a Hex editor or a tool like pe-tree, it becomes undesirable for people who don't want their code to be reverse-engineered. This is where the packers come in. A packer is a tool to obfuscate the data in a PE file so that it can't be read without unpacking it. In simple words, packers pack the PE file in a layer of obfuscation to avoid reverse engineering and render a PE file's static analysis useless. When the PE file is executed, it runs the unpacking routine to extract the original code and then executes it. Legitimate software developers use packing to address piracy concerns, and malware authors use it to avoid detection. So how do we identify packers?

From Section Headers

In the previous tasks, we learned that commonly, a PE file has a .text section, a .data section, and a .rsrc section, where only the .text section has the execute flag set because it contains the code. Now take the example of the file named zmsuz3pinwl. When we open this file in pe-tree, we find that it has unconventional section names (or no names, in this case).

zmsuz3pinwl		⚠ 8 warnings
Size	1.2 MB (1247232 bytes)	
MD5	1ebb1e268a462d56a389e8e1d06b4945	
SHA1	1ecc0b9f380896373e81ed166c34a89bded873b5	
SHA256	98c6cf0b129438ec62a628e8431e790b114ba0d82b76e625885ceedef286d6f5	
Imphash	829df24b10d39ea3719a931bde7a76d7	
Entropy	7.978052	
Architecture	I386 (PE)	
Compiled	2022-03-09T12:27:49	
► IMAGE_DOS_HEADER		
► DOS_STUB		
► IMAGE_NT_HEADERS		
► NT_HEADERS		
► IMAGE_SECTION_HEADER		
►		
►		
►		
► .rsrc		
►		
► .data		
► .adata		
► IMAGE_IMPORT_DESCRIPTOR		
► IMAGE_DIRECTORY_ENTRY_DEBUG		
► IMAGE_RESOURCE_DIRECTORY		

We might think this has something to do with the tool we use to analyze the file. Therefore, let's check it using another PE analysis tool called pecheck. The pecheck tool provides the same information we have been gathering from the pe-tree tool, but it is a command-line tool. We navigate to the Desktop\Samples directory in the terminal and give the following command to run the pecheck tool.

```
pecheck zmsuz3pinwl
```

Let's see the information in the PE Sections heading in the output:

```

PE Check utility

user@machine$ pecheck zmsuz3pinwl
PE check for 'zmsuz3pinwl':
Entropy: 7.978052 (Min=0.0, Max=8.0)
MD5 hash: 1eb81e268a462d56a389e8e1d06b4945
SHA-1 hash: 1ecc0b9f380896373e81ed166c34a89bbed873b5
SHA-256 hash: 98c6cf0b129438ec62a628e8431e790b114ba0d82b76e625885ceedef286d6f5
SHA-512 hash: 6921532b4b5ed9514660eb408dfa5d28998f52aa206013546f9eb66e26861565f852ec7f04c85ae9be89e7721c4f1a5c31d2fae49b0e7fdfd20451191146
entropy: 7.999788 (Min=0.0, Max=8.0)
entropy: 7.961048 (Min=0.0, Max=8.0)
entropy: 7.554513 (Min=0.0, Max=8.0)
.rsrc entropy: 6.938747 (Min=0.0, Max=8.0)
entropy: 0.000000 (Min=0.0, Max=8.0)
.data entropy: 7.866646 (Min=0.0, Max=8.0)
.adata entropy: 0.000000 (Min=0.0, Max=8.0)

.
.
.
.

-----PE Sections-----

[IMAGE_SECTION_HEADER]
0x1F0 0x0 Name:
0x1F8 0x8 Misc: 0x3F4000
0x1F8 0x8 Misc_PhysicalAddress: 0x3F4000
0x1F8 0x8 Misc_VirtualSize: 0x3F4000
0x1FC 0xC VirtualAddress: 0x1000
0x200 0x10 SizeOfRawData: 0xD3400
0x204 0x14 PointerToRawData: 0x400
0x208 0x18 PointerToRelocations: 0x0
0x20C 0x1C PointerToLinenumbers: 0x0
0x210 0x20 NumberOfRelocations: 0x0
0x212 0x22 NumberOfLinenumbers: 0x0
0x214 0x24 Characteristics: 0xE0000040
Flags: IMAGE_SCN_CNT_INITIALIZED_DATA, IMAGE_SCN_MEM_EXECUTE, IMAGE_SCN_MEM_READ, IMAGE_SCN_MEM_WRITE
Entropy: 7.999788 (Min=0.0, Max=8.0)
MD5 hash: fa9814d3aeb1fbfaa1557bac61136ba7
SHA-1 hash: 8db955c622c5bea3ec63bd917db9d41ce038c3f7
SHA-256 hash: 24f922c1cd45811eb5f3ab6f29872cda11dbd2251b7a3f44713627ad3659ac9
SHA-512 hash: e122e4600ea201058352c97bb7549163a0a5bcfb079630b197fe135ae732e64f5a6daff328f789e7b2285c5f975bce69414e55adba7d59006a1f0280bf64
.
.
.
.
```

We see here that the section name is empty, and it is not a glitch in the tool we used to analyze the PE file.

Another thing that we might notice here is that the Entropy of the .data section and three of the four unnamed sections is higher than seven and is approaching 8. As we discussed in a previous task, higher Entropy represents a higher level of randomness in data. Random data is generally generated when the original data is obfuscated, indicating that these values might indicate a packed executable.

Apart from the section names, another indicator of a packed executable is the permissions of each section. For the PE file in the above terminal, we can see that the section contains initialized data and has READ, WRITE and EXECUTE permissions.

Similarly, some other sections also have READ, WRITE and EXECUTE permissions. This is also not found in the ordinary unpacked PE file, where only the .text section has EXECUTE permissions, as we saw in the redline malware sample.

Another valuable piece of information from the section headers to identify a packed executable is the SizeOfRawData and Misc_VirtualSize. In a packed executable, the SizeOfRawData will always be significantly smaller than the Misc_VirtualSize in sections with WRITE and EXECUTE permissions. This is because when the PE file unpacks during execution, it writes data to this section, increasing its size in the memory compared to the size on disk, and then executes it.

From Import Functions

The last important indicator of a packed executable we discuss here is its import functions. The redline PE file we analyzed earlier imported lots of functions, indicating the activity it potentially performs. However, for the PE file zmsuz3pinwl, we will see only a handful of imports, especially the GetProcAddress, GetModuleHandleA, and LoadLibraryA. These functions are often some of the only few imports of a packed PE file because these functions provide the functionality to unpack the PE file during runtime.

```
▼ IMAGE_IMPORT_DESCRIPTOR
  ▼ kernel32.dll
    OriginalFirstThunk      0x00000000
    Characteristics        0x00000000
    TimeStamp               0x00000000 Thu Jan 1 00:00:00 1970 UTC
    ForwarderChain          0x00000000
    Name                    0x0040cc0c
    FirstThunk              0x0040cbf8
    GetProcAddress          0x0040cbf8 -> +0x0000bbf8
    GetModuleHandleA        0x0040cbfc -> +0x0000bbfc
    LoadLibraryA            0x0040cc00 -> +0x0000bc00
  ▷ msvcpi40.dll
  ▷ user32.dll
  ▷ vcruntime140.dll
  ▷ api-ms-win-crt-runtime-l1-1-0.dll
  ▷ api-ms-win-crt-math-l1-1-0.dll
  ▷ api-ms-win-crt-heap-l1-1-0.dll
  ▷ api-ms-win-crt-stdio-l1-1-0.dll
  ▷ api-ms-win-crt-locale-l1-1-0.dll
  ▷ oleaut32.dll
  ▷ kernel32.dll
  ▷ IMAGE_DIRECTORY_ENTRY_DEBUG
  ▷ IMAGE_RESOURCE_DIRECTORY
```

Summing up, the following indications point to a packed executable when we look at its PE header data:

- Unconventional section names
- EXECUTE permissions for multiple sections

- High Entropy, approaching 8, for some sections.
 - A significant difference between SizeOfRawData and Misc_VirtualSize of some PE sections
 - Very few import functions
-

Answer the questions below:

Which of the files in the attached VM in the directory Desktop\Samples seems to be a packed executable?

Answer: zmsuz3pinwl

Conclusion

That concludes this room about Dissecting PE headers. In this room, we learned:

- What is a PE header
- What are the different parts of the PE header
- How to read the information from the PE header
- Identify packed executables using the PE header