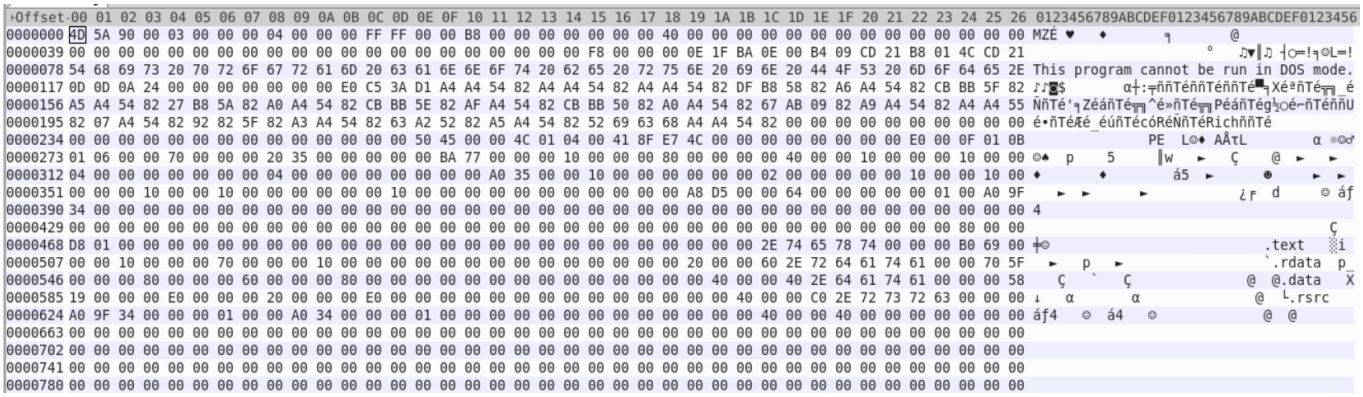
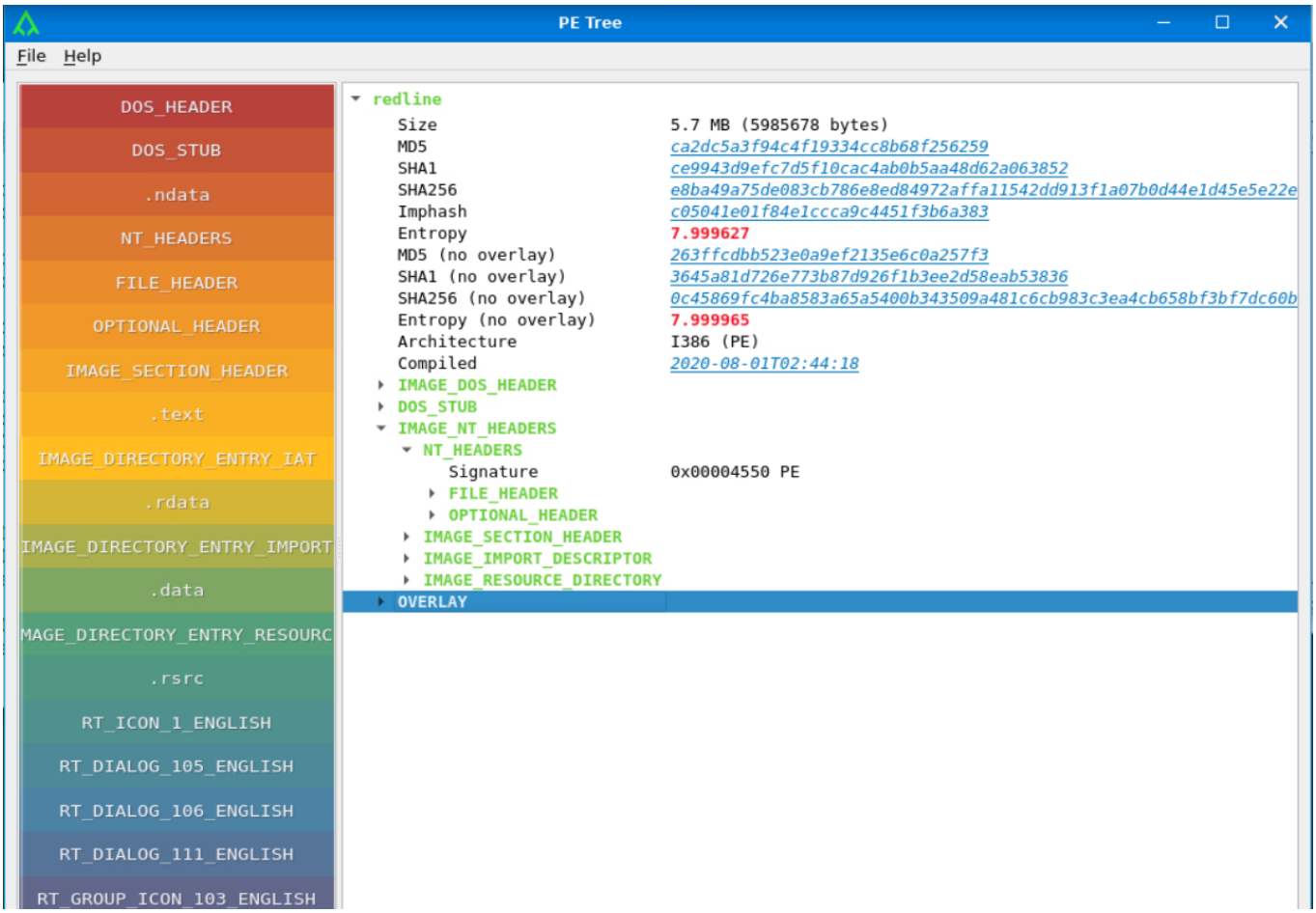
In the Windows Operating System, you might have often seen files with extension `.exe`. The `.exe` extension here stands for executable. As the name suggests, an executable file contains code that can be executed. Therefore, anything that needs to be run on a Windows Operating System is executed using 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.

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.



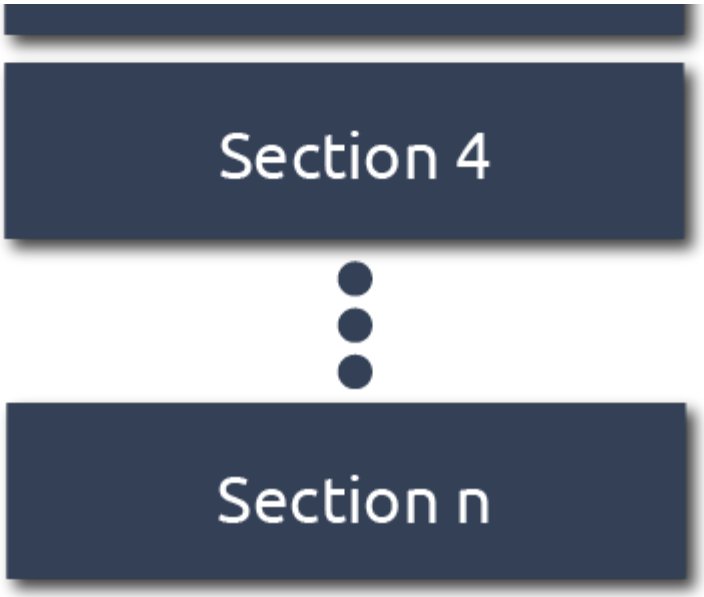
In the upcoming tasks, we will try to make sense of the Hex numbers we see in the above screenshot and describe them as Windows understands them. We will also explore how to leverage this information for malware analysis. We will use the `wxHexEditor` utility, present in the next task's attached VM to 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


DOS Header
DOS Stub
NT Headers
• PE signature
• File Header
• Optional Header
Section Table
Section 1
Section 2
Section 3



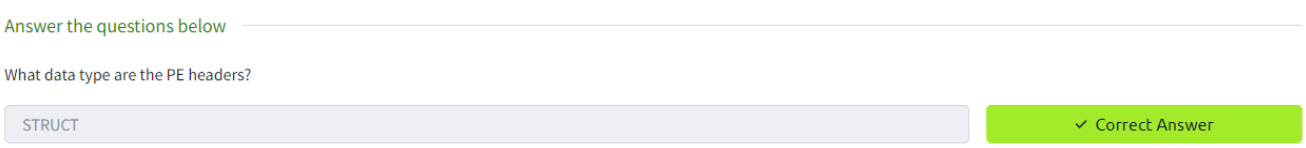
discuss in this room are:

- 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.



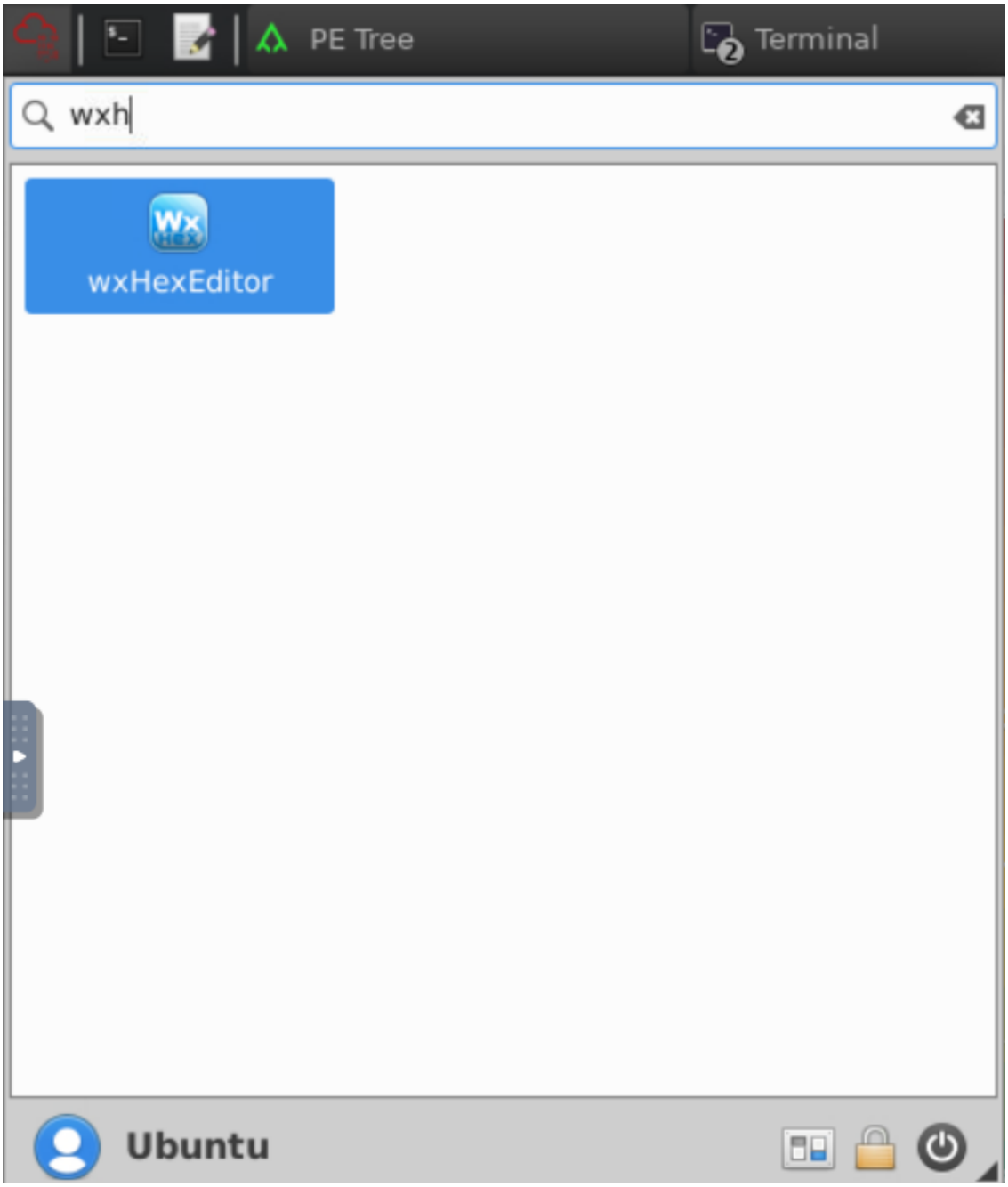
For this task, we will need to use the attached VM. For this purpose, press the 'Start Machine' button on the top-right corner of this task to start the attached machine. The machine will start in a split-screen view. In case the machine is not visible, use the blue Show Split View button at

the top-right of the page. Alternatively, you can log in to the machine using the following credentials:

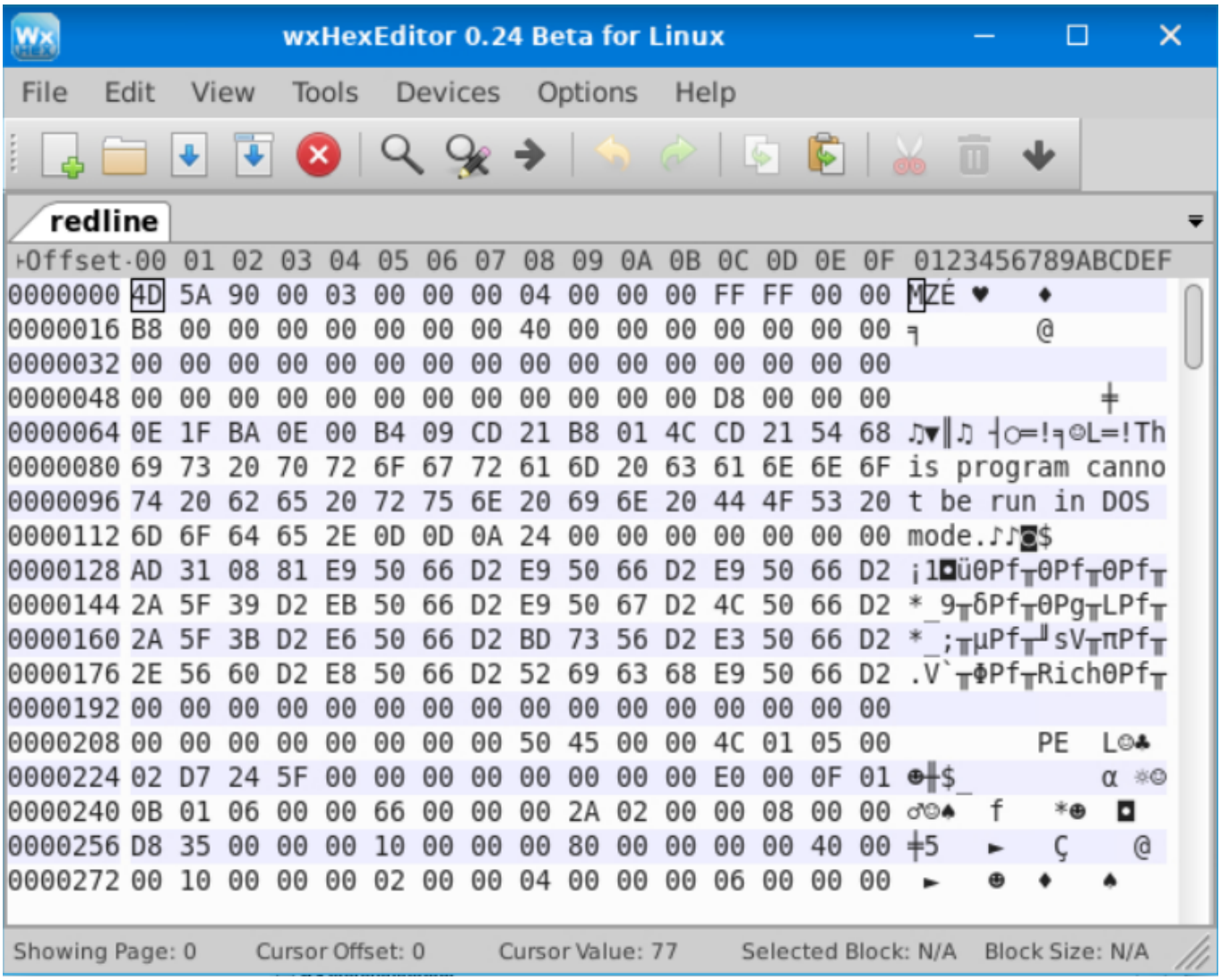
Username: ubuntu

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.



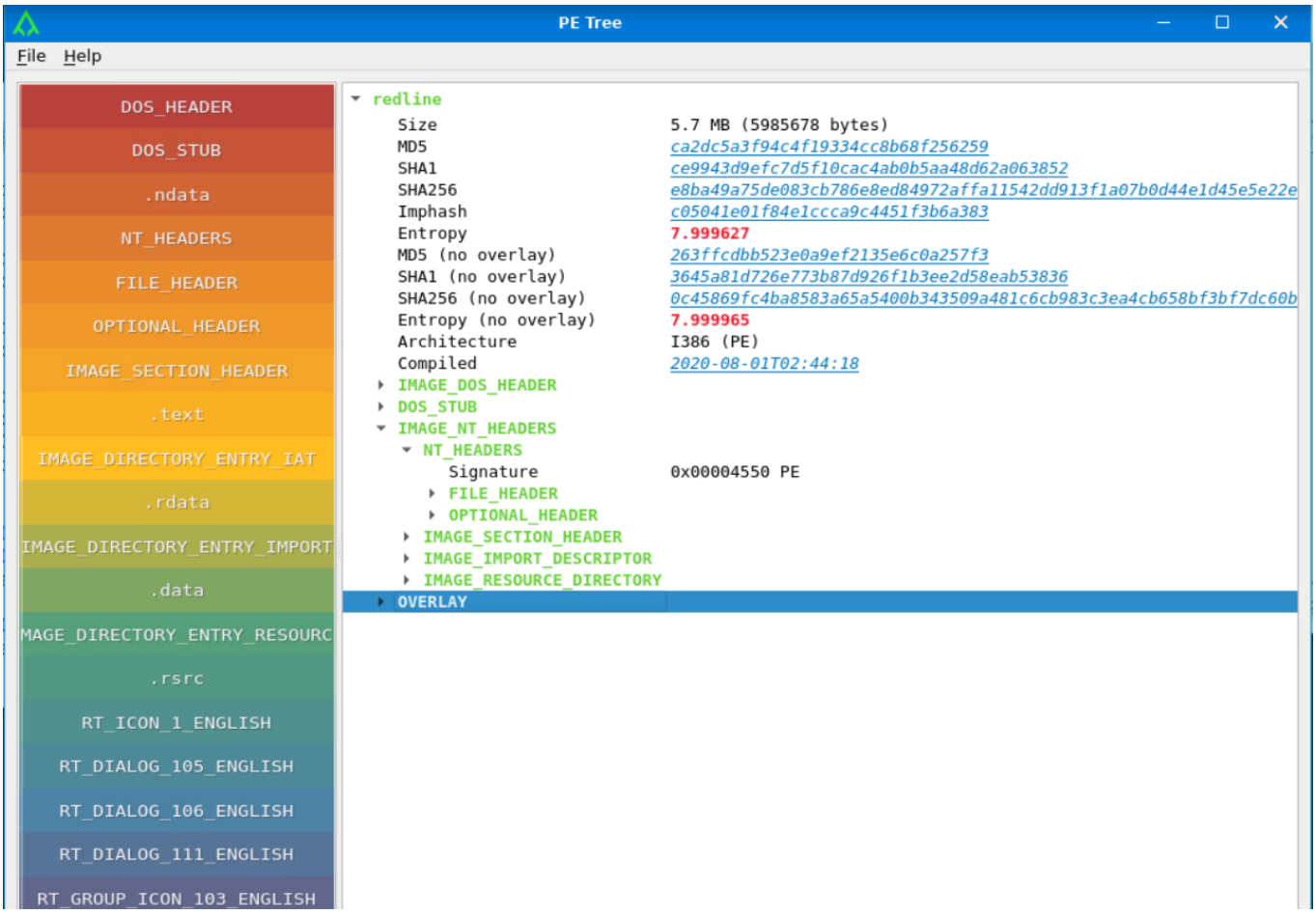
This is what the redline PE will look like in the Hex Editor.



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
```

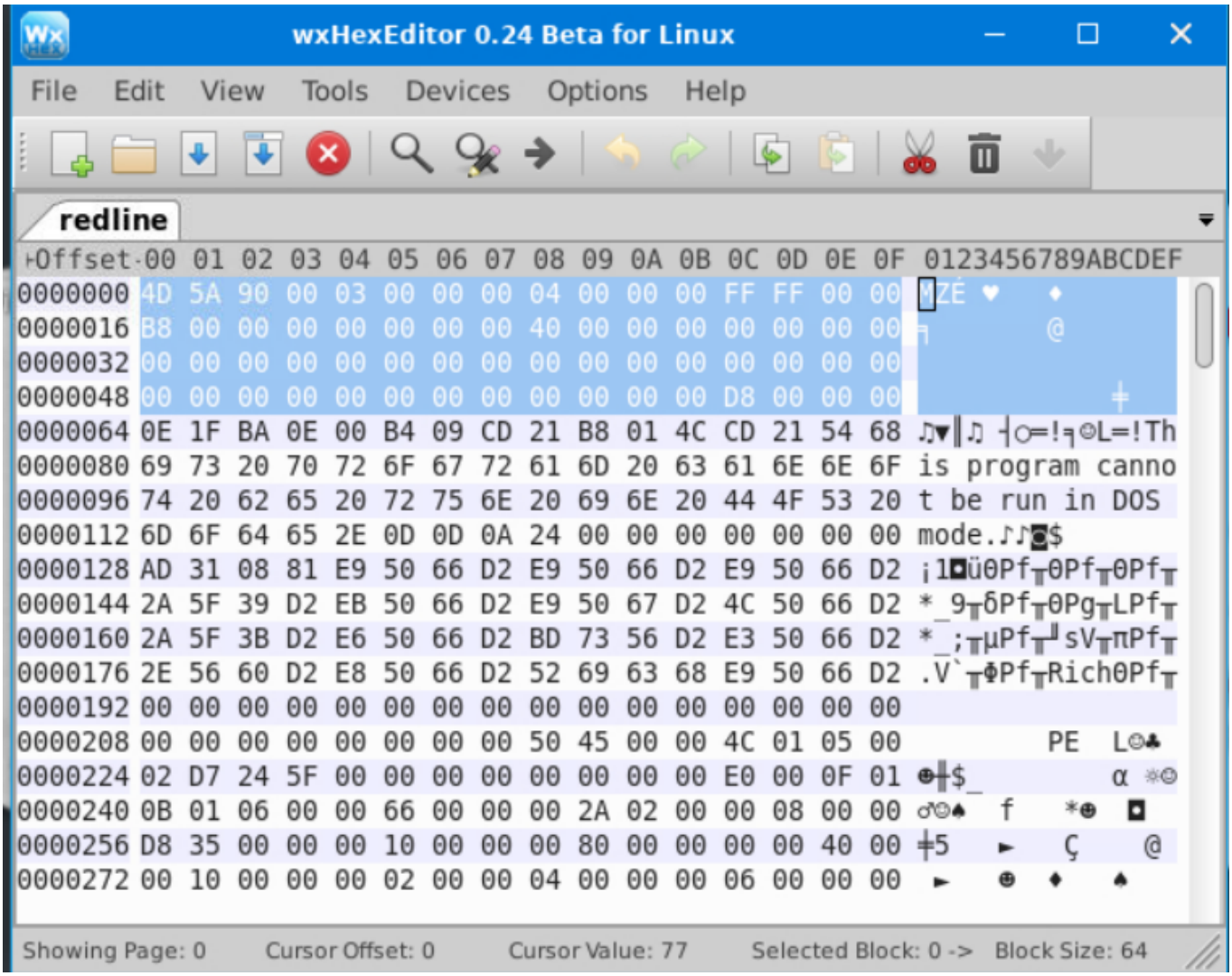
Please note that the pe-tree utility will take roughly 8 minutes to open. In the meanwhile, you can continue reading the text and come back once it has opened. This is what the `pe-tree` output will look like when we open the redline utility.



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:

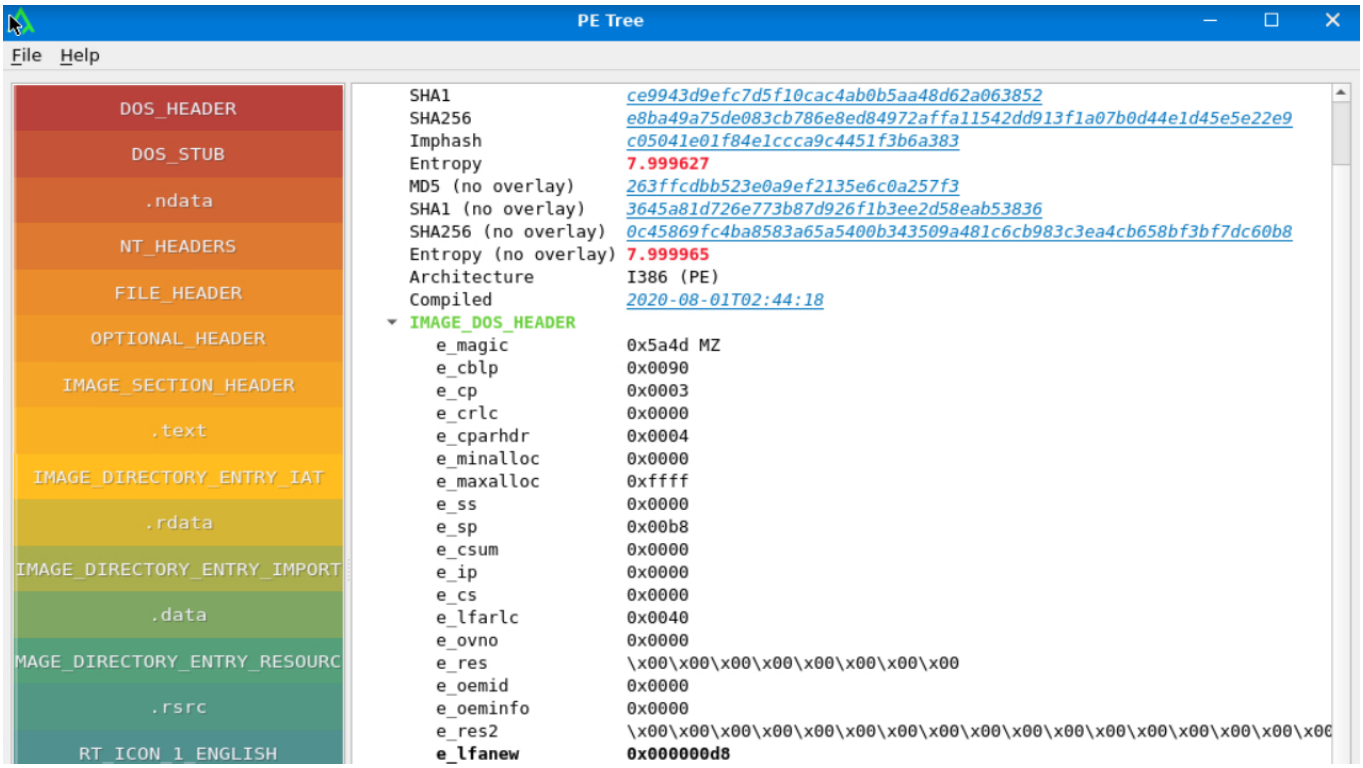
The IMAGE_DOS_HEADER consists of the first 64 bytes of the PE file. We will analyze some of the valuable information found in the IMAGE_DOS_HEADER below. The below screenshot has the IMAGE_DOS_HEADER highlighted in the Hex Editor.



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?

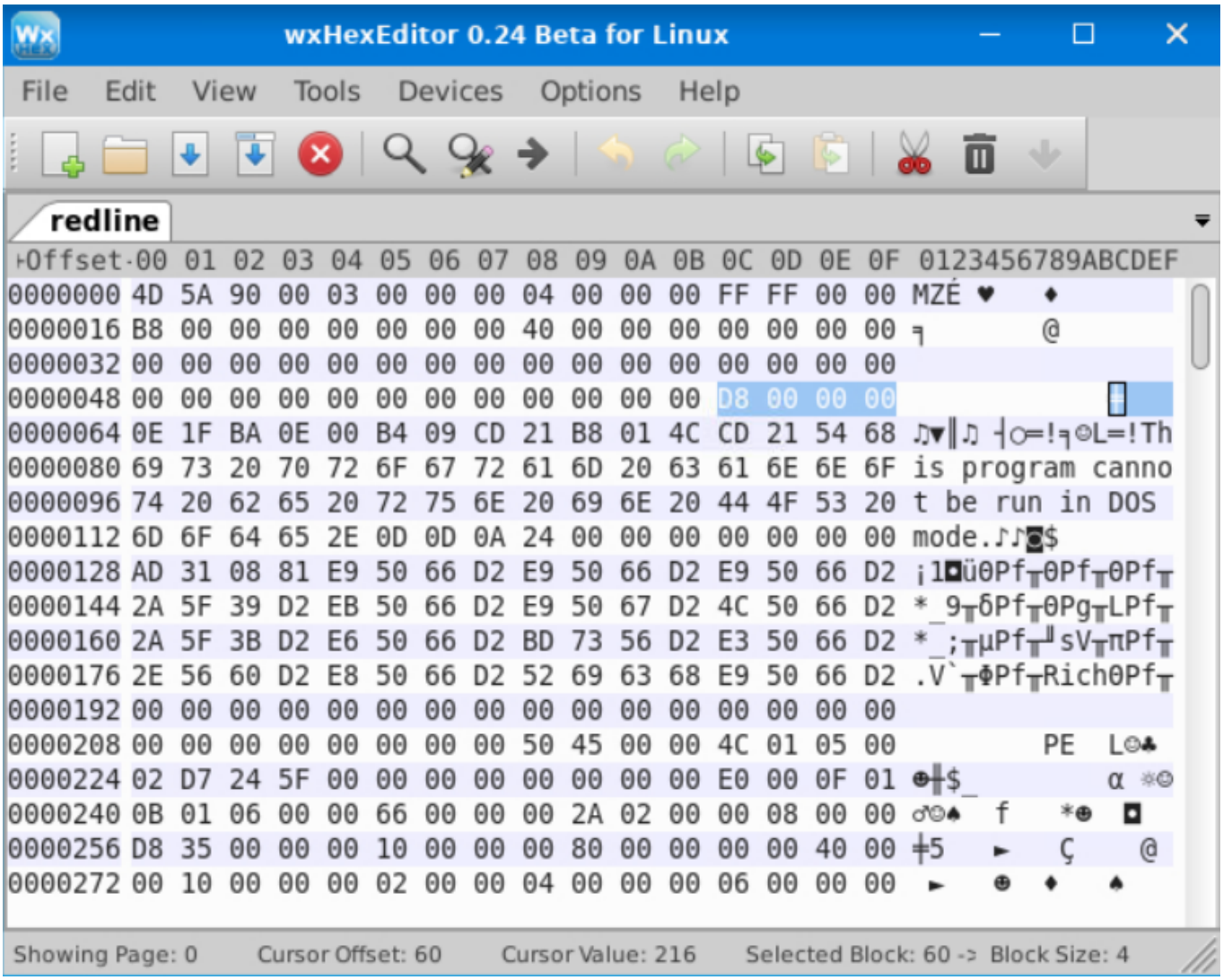
The MZ characters denote the initials of Mark Zbikowski, one of the Microsoft architects who created the MS-DOS file format. The MZ characters are an identifier of the Portable Executable format. When these two bytes are present at the start of a file, the Windows OS considers it a Portable Executable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

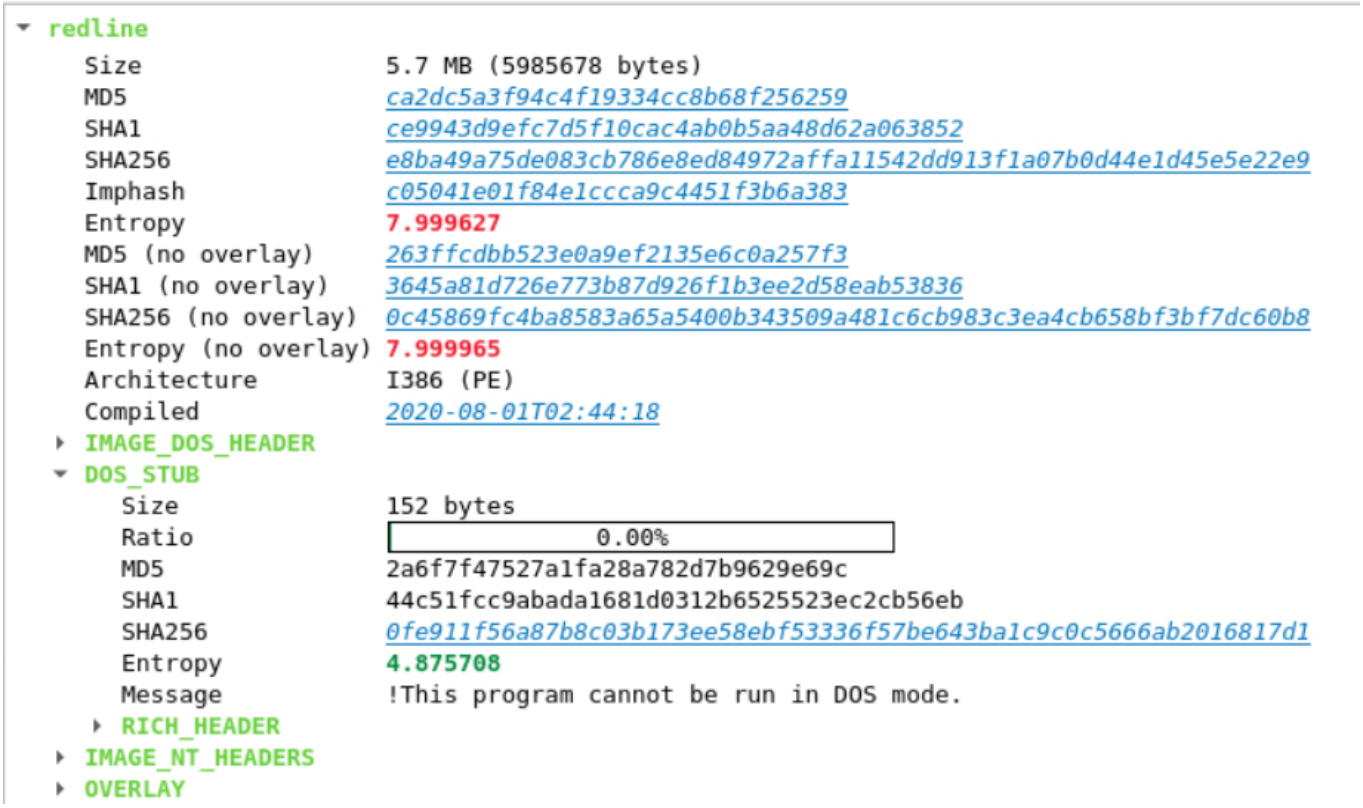


We have to remember that the byte order is switched when we compare those reported by the pe-tree utility and those we see in the Hex editor due to endianness.

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.

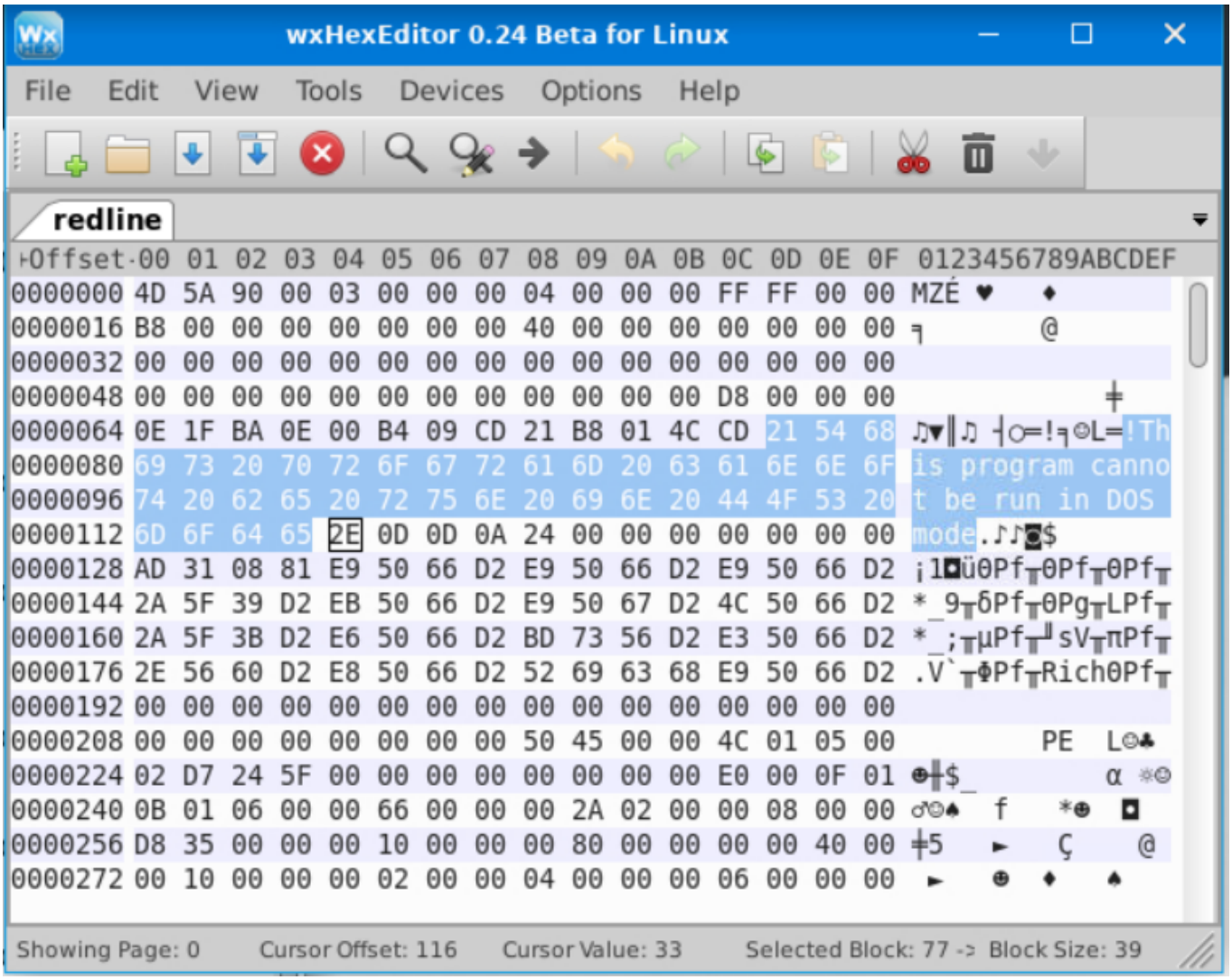
The DOS_STUB:

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.



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.

How many bytes are present in the IMAGE_DOS_HEADER?

64 ✓ Correct Answer

What does MZ stand for?

Mark Zbikowski ✓ Correct Answer

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

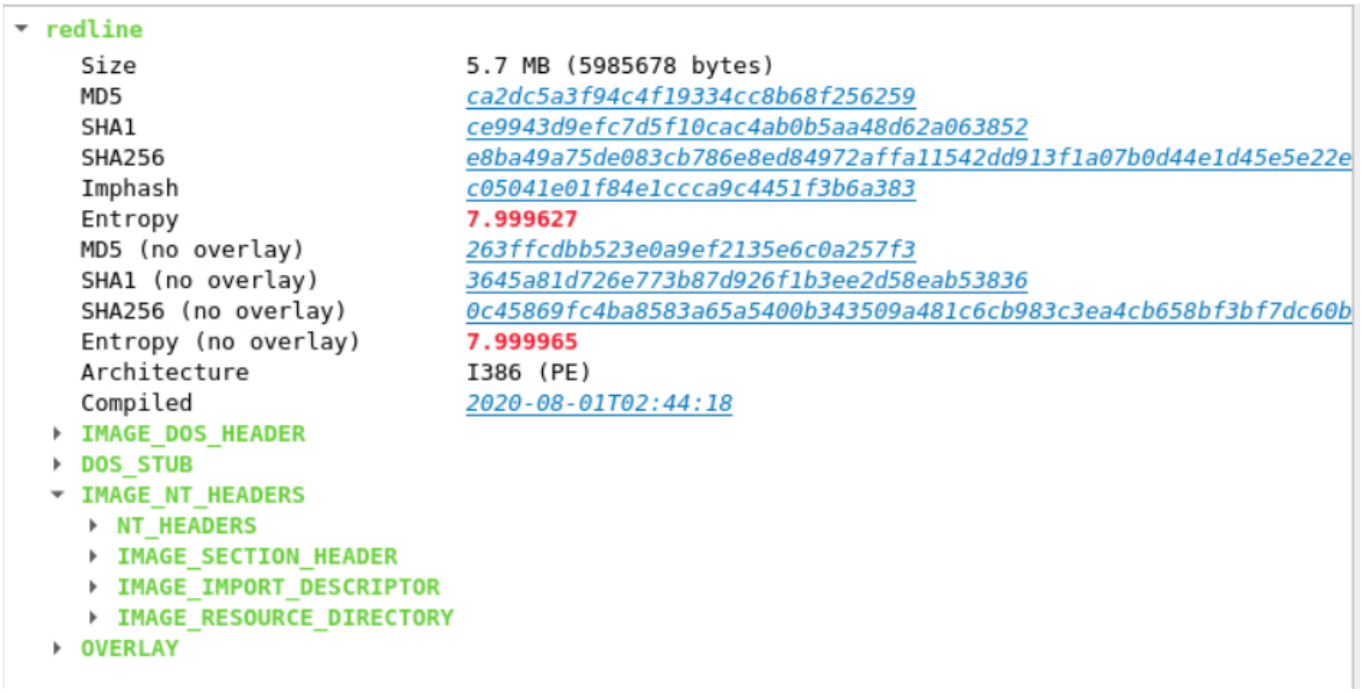
e_lfanew ✓ Correct Answer

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?

0x000000f8 ✓ Correct Answer Hint

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:

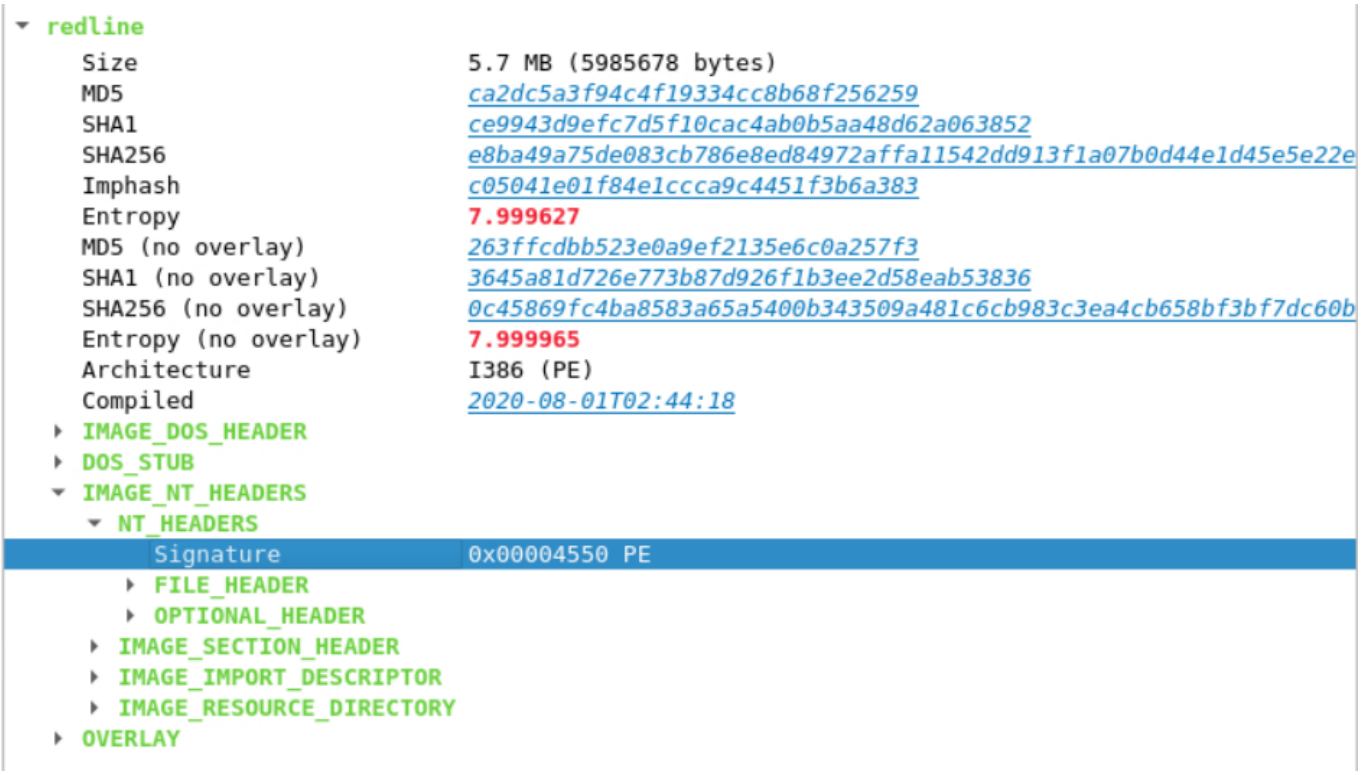


NT_HEADERS:

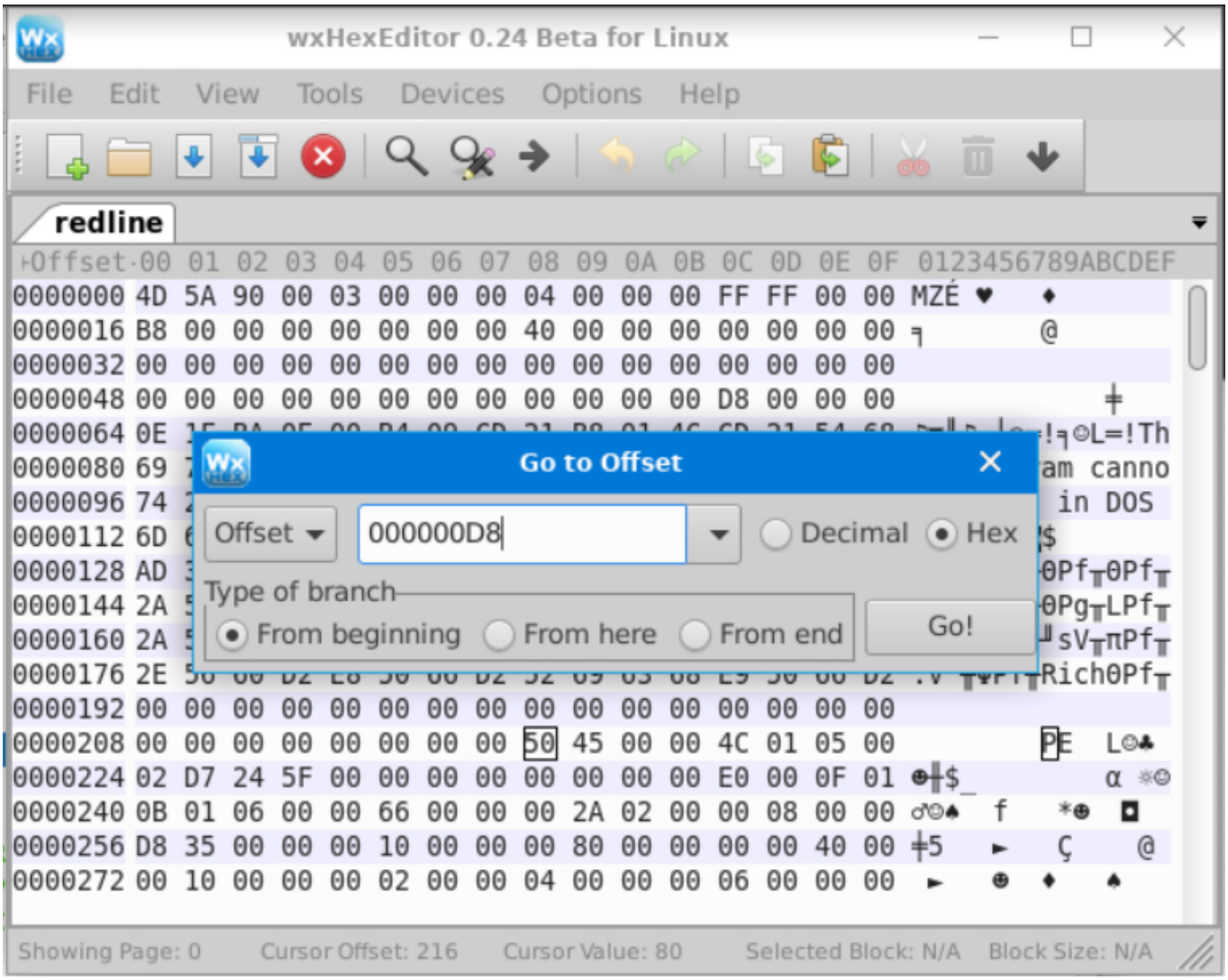
Before diving into the details of NT_HEADERS, let's get an overview of the NT_HEADERS. The NT_HEADERS consist of the following:

- Signature
- FILE_HEADER
- OPTIONAL_HEADER

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



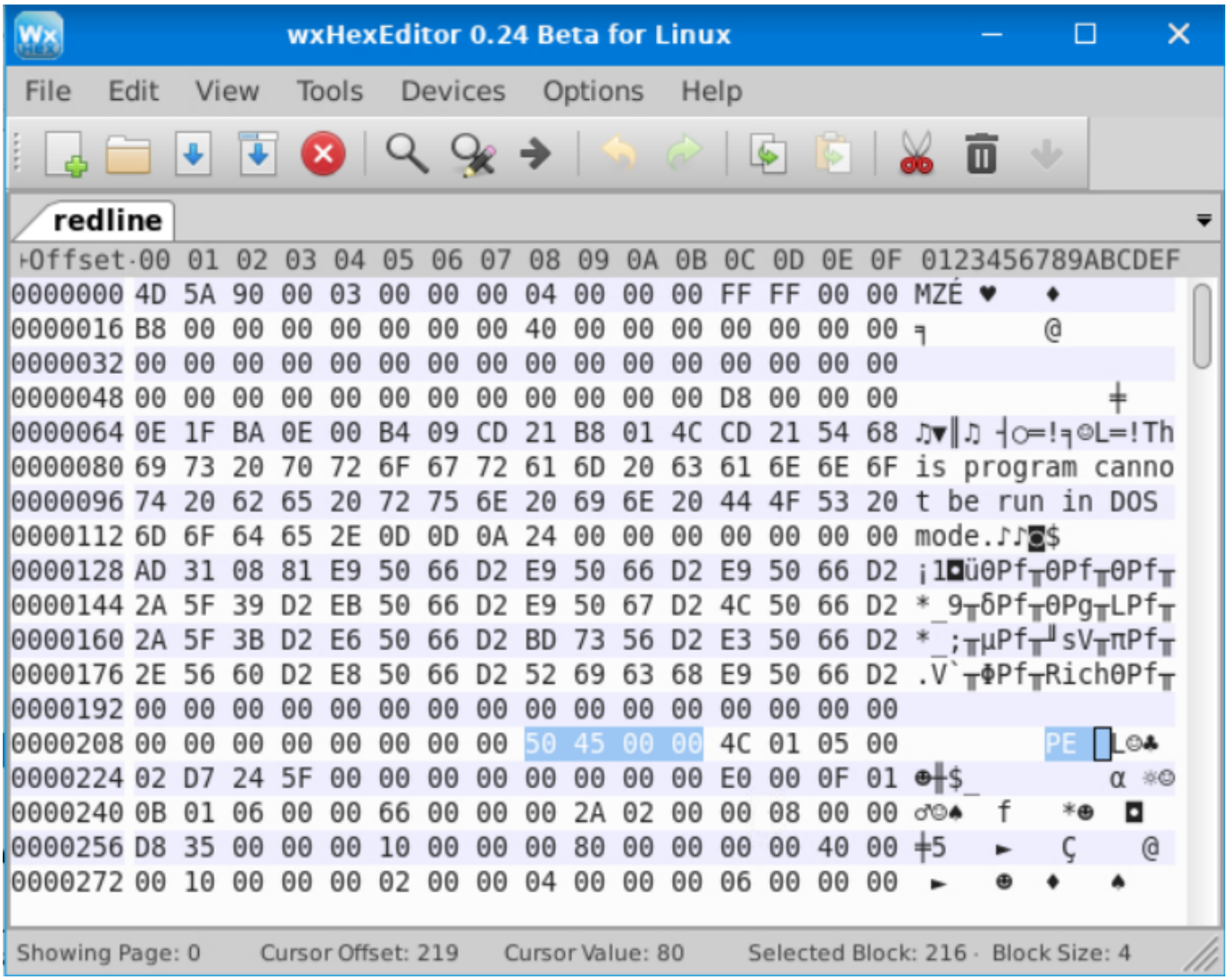
The starting address of IMAGE_NT_HEADERS is found in `e_lfanew` from the IMAGE_DOS_HEADER. In the redline binary, we saw that this address was `0x000000D8`. So let's start by going to this offset and see what we find there. We can do that by pressing `Ctrl+G` in the Hex Editor Window or going to Edit>Go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

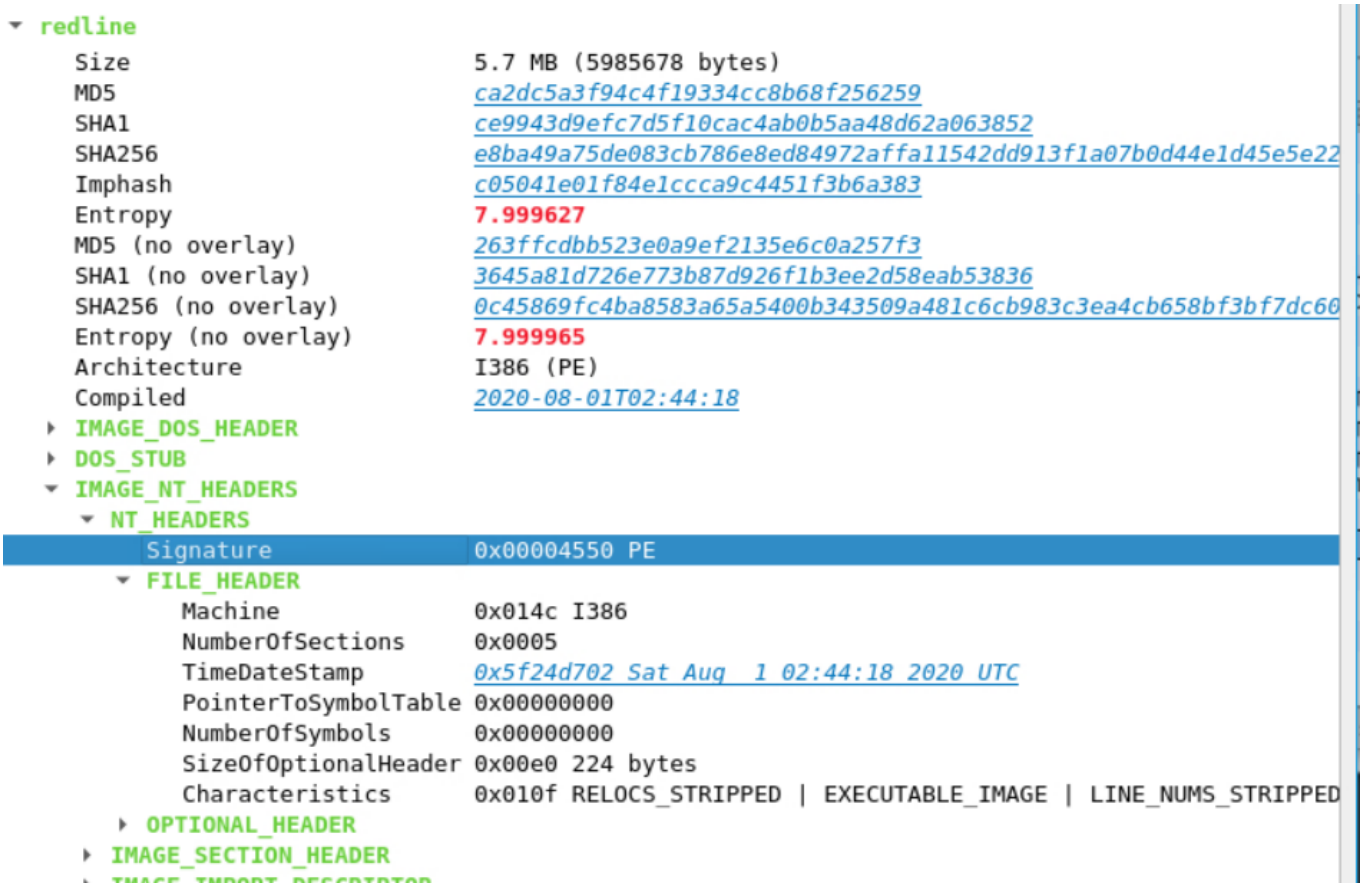
The first 4 bytes of the NT_HEADERS consist of the Signature. We can see this as the bytes `50 45 00 00` in Hex, or the characters `PE` in ASCII as shown in the Hex editor.



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.

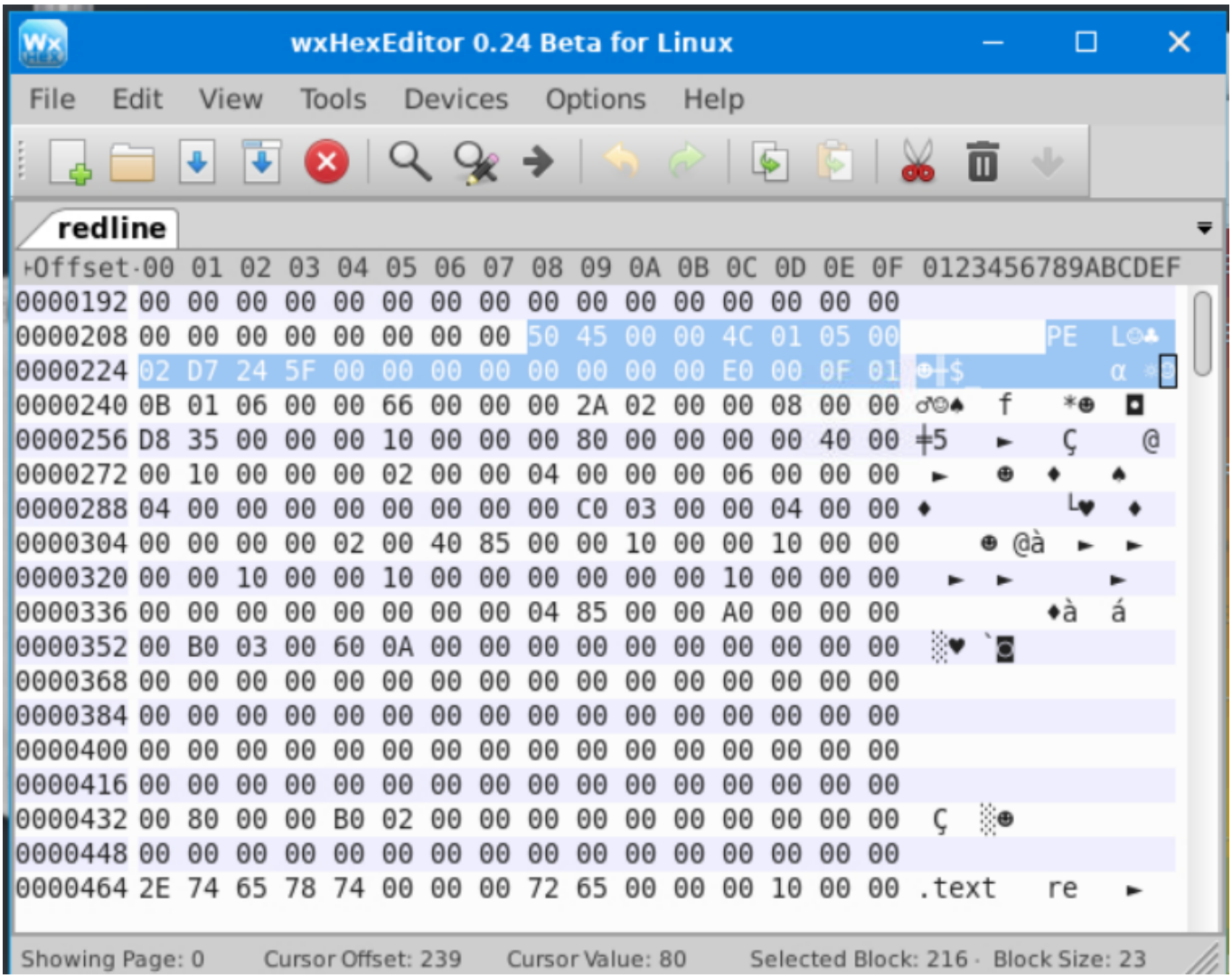


As we can see in the above screenshot, the FILE_HEADER has the following fields:

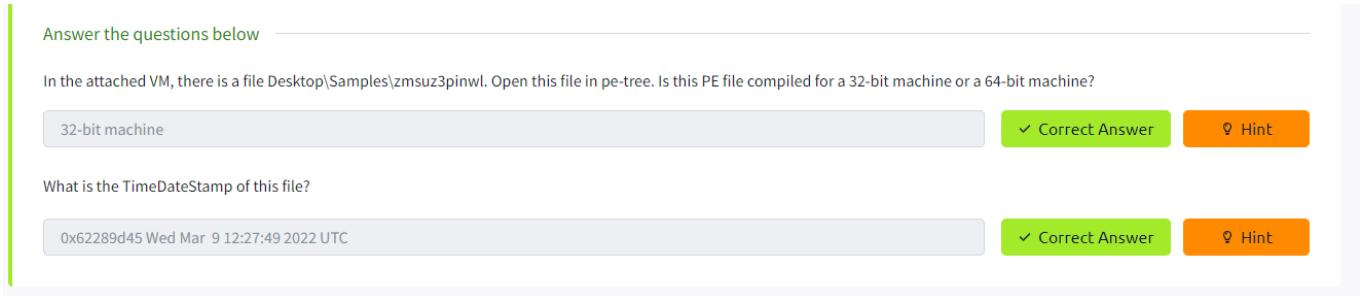
- *Machine:* This field mentions the type of architecture for which the PE file is written. In the above example, we can see that the architecture is i386 which means that this PE file is compatible with 32-bit Intel architecture.
- *NumberOfSections:* A PE file contains different sections where code, 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.
- *TimeDateStamp:* 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

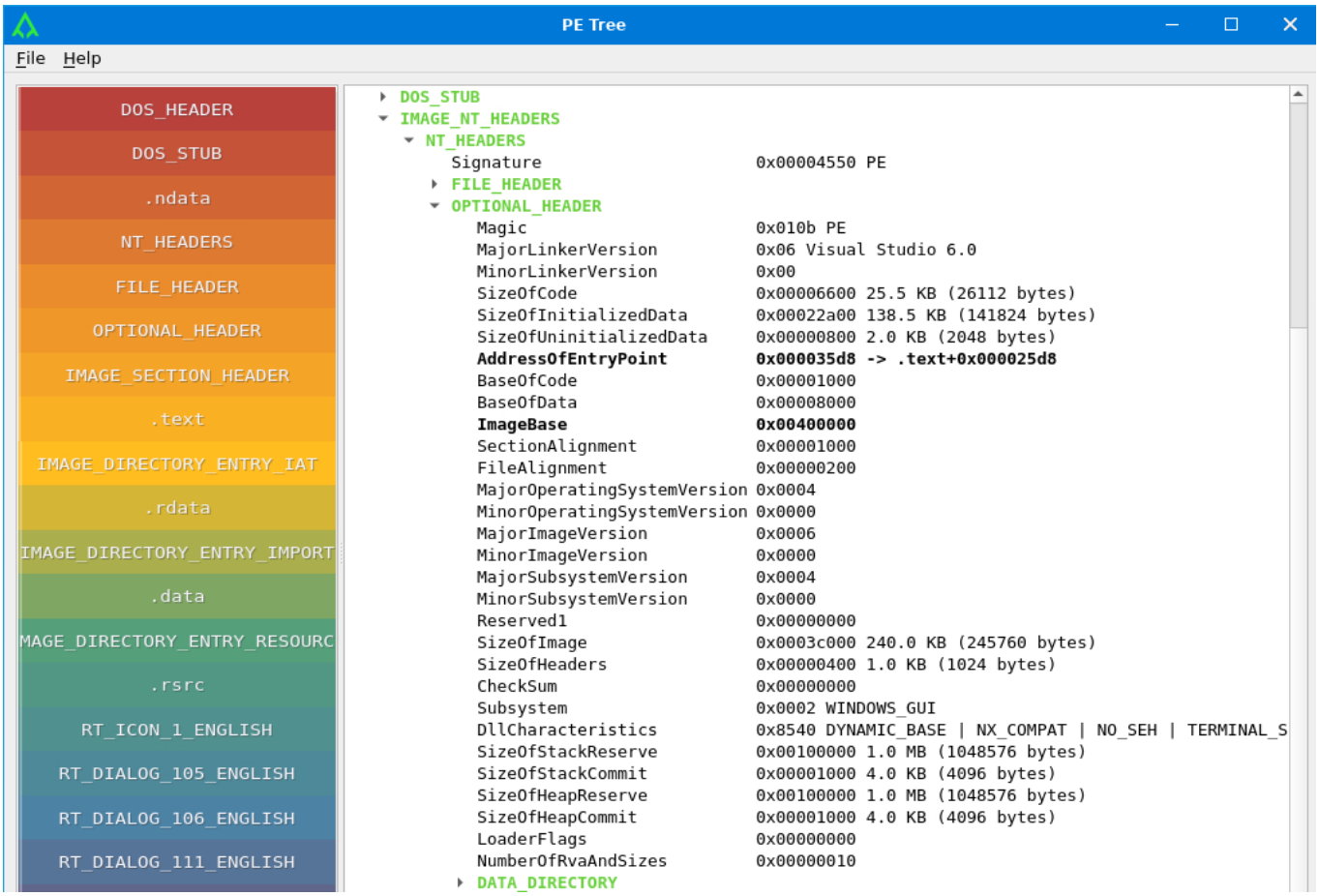
each value is located?



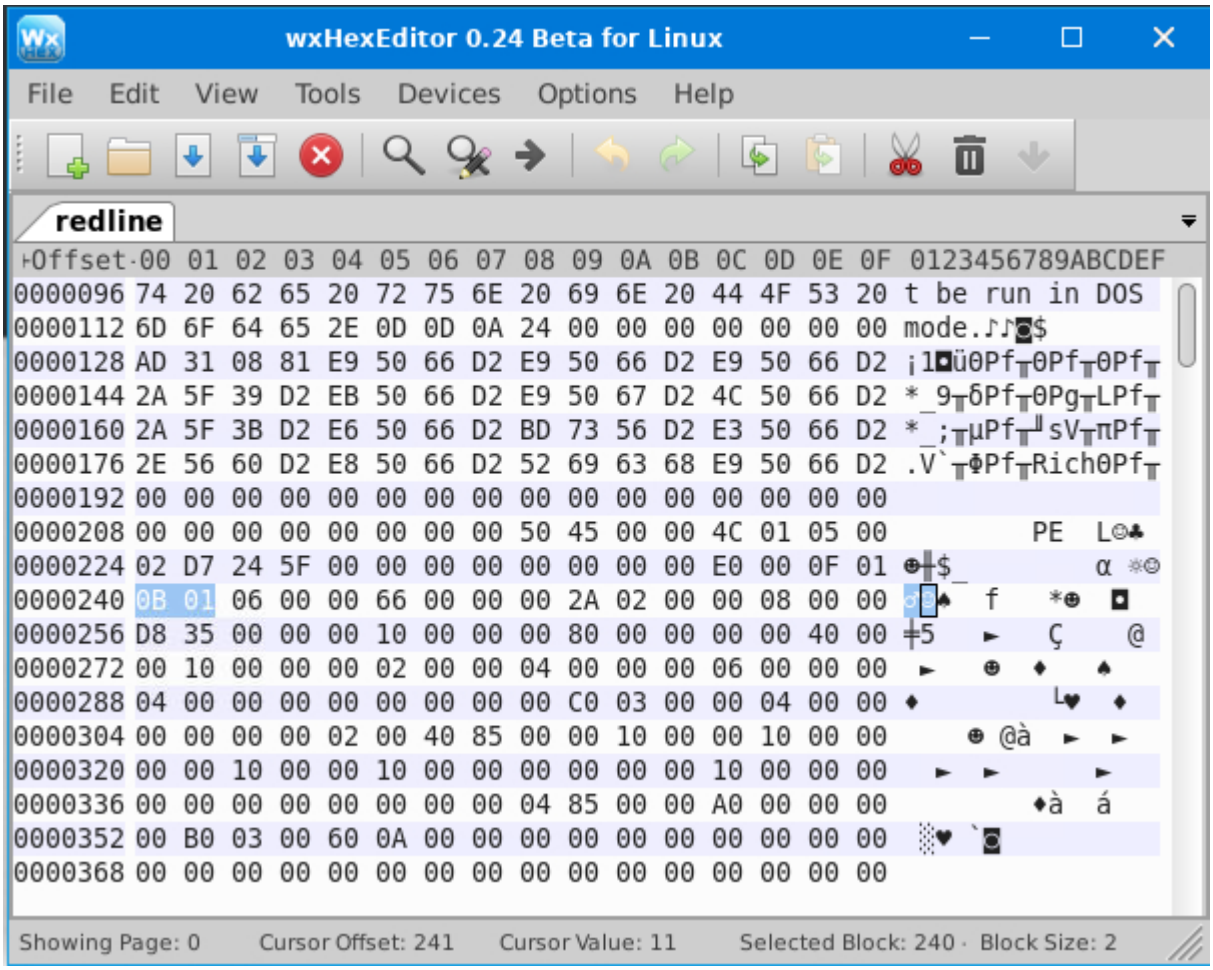
We are starting to learn to read Hex now, aren't we? To learn more about the FILE_HEADER, you can check out Microsoft Documentation for it.



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.

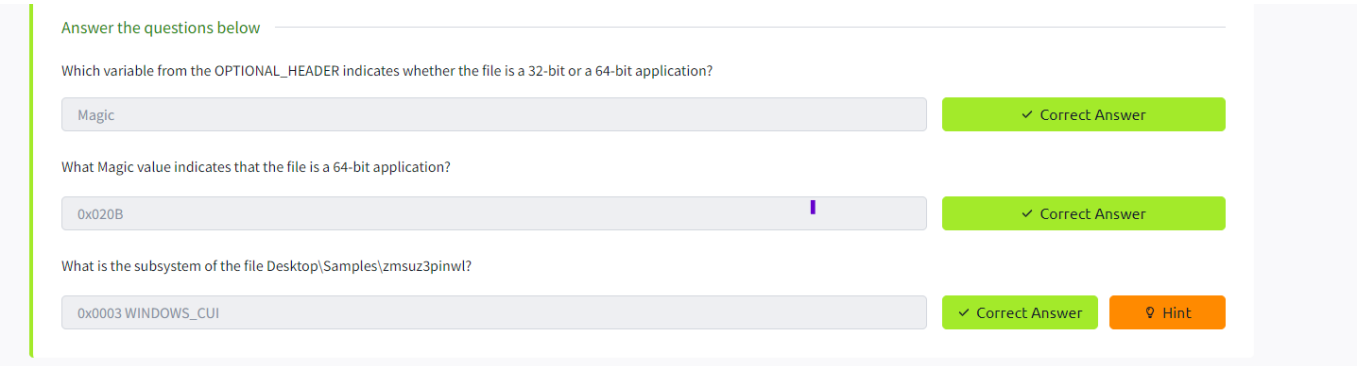


Let's learn about some of the critical fields in the OPTIONAL_HEADER.

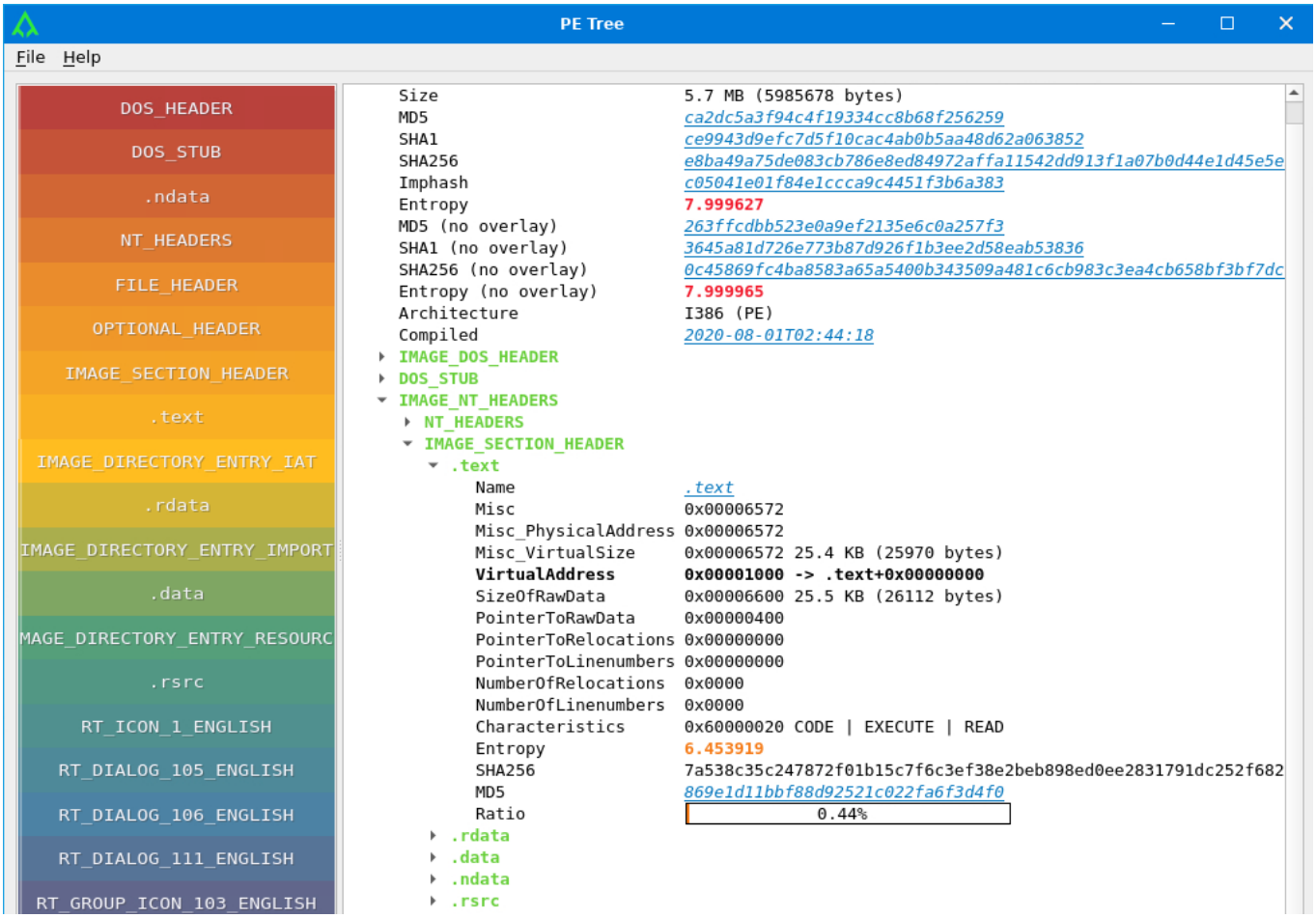
- *Magic:* The Magic number tells whether the PE file is a 32-bit or 64-bit application. If the value is 0x010B, it denotes a 32-bit application; if the value is 0x020B, it represents a 64-bit application. The above screenshot of the Hex Editor shows the highlighted bytes, which show the magic of the 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.



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.

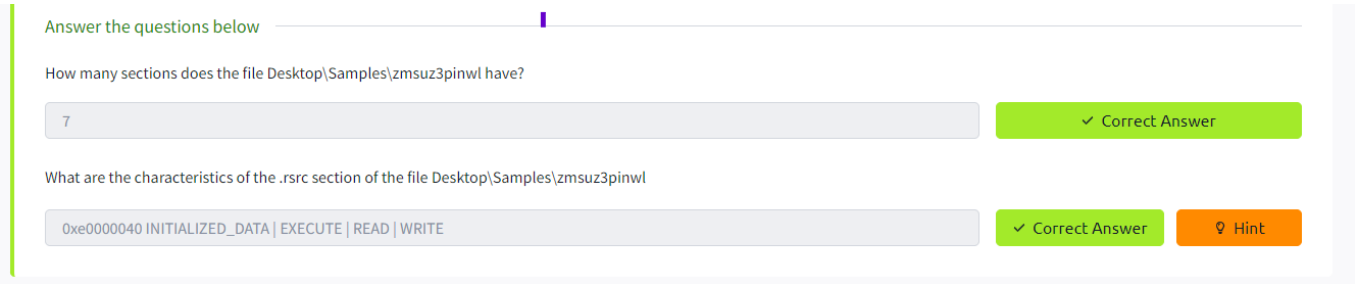


As we can see, the IMAGE_SECTION_HEADER has different sections, namely `.text`, `.rdata`, `.data`, `.ndata` and `.rsrc`. Before moving to the information present in the header of each section, let's learn about the commonly found sections in a PE file.

- *.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.

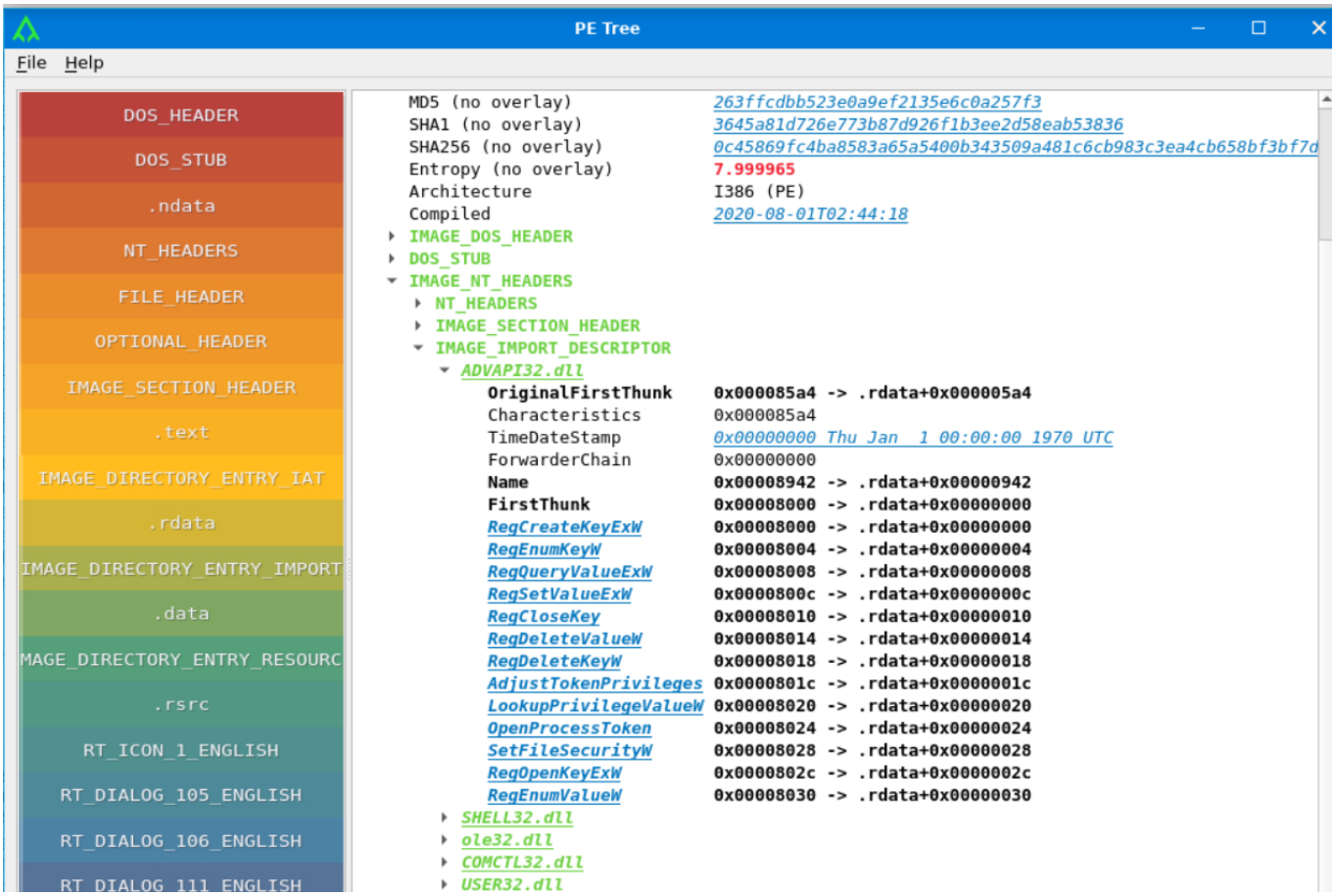
Now that we know what the different types of sections are commonly found in a PE file, let's see what important information the section headers for each section include:

- *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.



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.



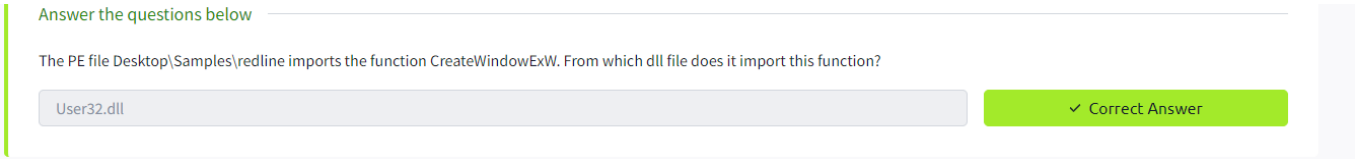
Here we can see that the PE file we are looking at imports functions from ADVAPI32.dll, SHELL32.dll, ole32.dll, COMCTL32.dll, and USER32.dll. These files are dynamically linked libraries that export Windows functions or APIs for other PE files. The above screenshot shows that the PE file imports some 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.

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

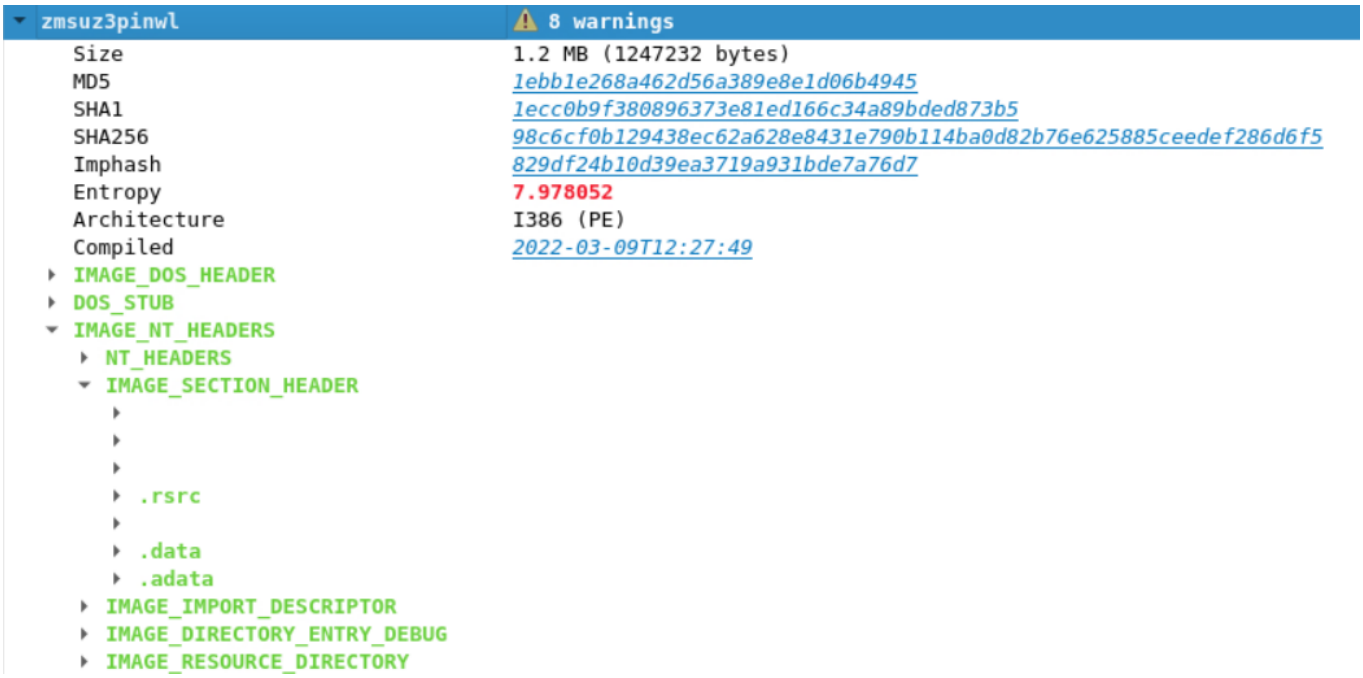
Take the redline binary from the attached VM as an example. Its IMAGE_IMPORT_DESCRIPTOR imports notable functions such as CreateProcessW, CreateDirectoryW, and WriteFile from kernel32.dll. This implies that this PE file intends to create a process, create a directory, and write some data to a file. Similarly,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



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).



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:

PECheck utility

```
user@machine$ pecheck zmsuz3pinwl
PE check for 'zmsuz3pinwl':
Entropy: 7.978052 (Min=0.0, Max=8.0)
MD5     hash: 1ebb1e268a462d56a389e8e1d06b4945
SHA-1   hash: 1ecc0b9f380896373e81ed166c34a89bded873b5
SHA-256 hash: 98c6cf0b129438ec62a628e8431e790b114ba0d82b76e625885ceedef286d6f5
SHA-512 hash:
6921532b4b5ed9514660eb408dfa5d28998f52aa206013546f9eb66e26861565f852ec7f04c85a
e9be89e7721c4f1a5c31d2fae49b0e7fdfd20451191146614a
 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: 24f922c1cd45811eb5f3ab6f29872cda11db7d2251b7a3f44713627ad3659ac9
SHA-512 hash:
e122e4600ea201058352c97bb7549163a0a5bcfb079630b197fe135ae732e64f5a6daff328f789
e7b2285c5f975bce69414e55adba7d59006a1f0280bf64971c
.
.
.
.
.
```

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

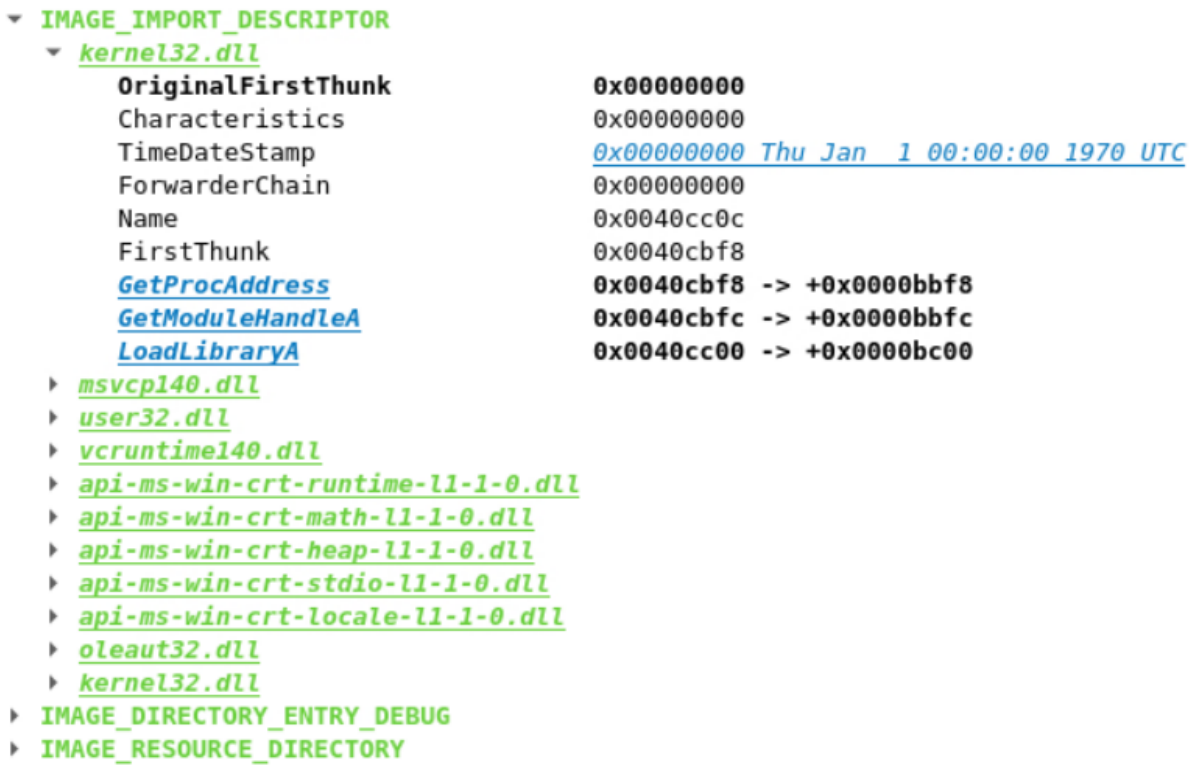
previous task, higher Entropy represents a higher level of randomness in data. Random data is generally generated when the original data is obfuscated, indicating that these values might indicate a packed executable.

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.



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